НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В. о. зав. кафедри
_____ Новотарський Михайло
(підпис)

“__” _____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Комп'ютерні системи та мережі”

спеціальності 123 “Комп'ютерна інженерія”

на тему: Веб-застосунок для організації та координації командної роботи

Виконав : студент __4__ курсу, групи _ІО-13
(шифр групи)

_____Кунчій Ростислав Олександрович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____ас. Валько В.В._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) ас. Гончаренко Олександр Олексійович___
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _к.т.н. ст. викл. каф. ІСТ Орленко Сергій Петрович_____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Комп'ютерні системи та мережі”

спеціальності 123 “Комп'ютерна інженерія”

ЗАТВЕРДЖУЮ
В. о. зав. кафедри
_____ Новотарський Михайло
(підпис)
“__” _____2025 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Кунчія Ростислава Олександровича

1. Тема проєкту *Веб-застосунок для організації та координації командної роботи*
керівник проєкту _____ас. Валько В.В._____
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

2. Термін здачі студентом закінченого проєкту __02.06.2025____

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)
Розділ 1. Аналіз Предметної Області.

Розділ 2. Аналіз та вибір інструментів для розробки.

Розділ 3. Розробка проєкту.

Розділ 4. Представлення роботи та графічного інтерфейсу проєкту.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структура системи, діаграма класів, алгоритм дій програмного забезпечення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «7» лютого 2025 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*02.02.2025-07.02.2025*	
2.	*Вивчення та аналіз завдання*	*15.03.2025-14.04.2025*	
3.	*Розробка архітектури та загальної структури системи*	*14.04.2025-28.04.2025*	
4.	*Розробка структур окремих підсистем*	*21.04.2025-28.04.2025*	
5.	*Програмна реалізація системи*	*24.04.2025-27.05.2025*	
6.	*Оформлення пояснювальної записки*	*05.05.2025-28.05.2025*	
7.	*Захист програмного продукту*	*31.05.2025*	
8.	*Передзахист*	*02.06.2025*	
9.	*Захист*	*16.06.2025*	

Студент-дипломник _____ Ростислав Кунчій
(підпис)

Керівник проєкту _____ Володимир Валько
(підпис)

АНОТАЦІЯ

У бакалаврському дипломному проекті створено веб-застосунок для організації та координації командної роботи.

Система орієнтована на ІТ-команди та дозволяє управляти проектами і задачами за методологіями Scrum/Kanban. Реалізовано канбан-дошки, таймлайни задач, функціонал управління доступністю співробітників, а також інтеграцію з Telegram для спрощення комунікації. Замовники мають доступ до інтерфейсу для перегляду прогресу проектів і створення нових запитів.

Проведено аналіз аналогів, таких як Notion, Jira, Worksection, Trello, для вдосконалення функціоналу.

Розглянуто сучасні технології (фреймворки, патерни проектування, методології розробки, мови програмування) для того щоб зробити застосунок продуктивним, оптимізованим та простим в додавані нових функцій.

Ключові слова: Scrum Kanban, управління проектами, командна робота.

ANNOTATION

The bachelor's diploma project involves the creation of a web application for organizing and coordinating teamwork.

The system is designed for IT teams and enables the management of projects and tasks using Scrum/Kanban methodologies. It includes features such as Kanban boards, task timelines, employee availability management, and Telegram integration to simplify communication. Clients have access to an interface for monitoring project progress and submitting new requests.

An analysis of existing solutions like Notion, Jira, Worksection, and Trello was conducted to improve the functionality.

Modern technologies (frameworks, design patterns, development methodologies, programming languages) were utilized to ensure the application is lightweight and easy to extend with new features.

Keywords: Scrum, Kanban, project management, teamwork.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.466500.002 ТЗ***	Веб-застосунок для	4		
			організації та координації			
			командної роботи			
			Технічне завдання			
	А4	***ІАЛЦ.466500.003 ПЗ***	Веб-застосунок для	97		
			організації та координації			
			командної роботи			
			Пояснювальна записка			
	А4	***ІАЛЦ.466500.004 Д1***	Веб-застосунок для	1		
			організації та координації			
			командної роботи			
			Структура системи			
	А4	***ІАЛЦ.466500.005 Д2***	Веб-застосунок для	1		
			організації та координації			
			командної роботи			
			Діаграма класів			
	А4	***ІАЛЦ.466500.006 Д3***	Веб-застосунок для	1		
			організації та координації			
			командної роботи			
			Алгоритм дій програмного			
			забезпечення			
	А4	***ІАЛЦ.466500.007 Д4***	Веб-застосунок для	15		
			організації та координації			
			командної роботи			
			Текст програмного коду			

					# *ІАЛЦ.466500.001 ОА*		
Зм	*Лист*	*№ докум.*	*Підп*	*Дата*			
Розроб		Кунчій Р.О.			*Веб-застосунок для організації та координації командної роботи* Опис альбому	Літ.	Аркуш 1 / Аркушів 1
Перев		Валько В.В.					
						НТУУ “КПІ” ФІОТ ІО-13	

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок для організації та координації командної роботи*»

Київ – 2025

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку веб-застосунку для організації та координації командної роботи. Система призначена для використання IT-командами, менеджерами проектів та замовниками для управління задачами, комунікації та координації роботи за методологією Scrum Kanban.

Сфера застосування включає управління IT-проектами, планування ресурсів, моніторинг прогресу виконання задач, а також інтеграцію з популярними інструментами для комунікації та звітності.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даної розробки є створення багатофункціонального веб-застосунку, що дозволяє IT-командам і менеджерам проектів ефективно організовувати робочі процеси, координувати задачі, планувати ресурси та відслідковувати прогрес. Основне призначення — полегшення управління проектами за допомогою інтеграції з Telegram, тайм-лайну задач та кастомізованих Kanban-дощок.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки даного проекту є:

- Публікації на тему управління проектами та програмним забезпеченням.
- Наукові статті та література з теорії програмування.
- Документація популярних веб-застосунків для управління проектами.
- Інтернет-ресурси, присвячені сучасним технологіям і методологіям розробки.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Простий і інтуїтивно-зрозумілий інтерфейс.
- Надати можливість користувачам створювати та управляти проектами, задачами та канбан-дошками.
- Забезпечити функціонал для управління доступністю членів команди.
- Надати інтеграцію з Telegram для комунікації.
- Забезпечити можливість перегляду та взаємодії із задачами для замовників проектів нарівні з іншими користувачами.
- Забезпечити можливість перегляду прогресу задач у вигляді таймлайну

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Будь-який сучасний веб-браузер (Google Chrome, Mozilla Firefox, Microsoft Edge).

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3-2100T.
- ROM не менше ніж 64 ГБ.
- RAM не менше ніж 4 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	02.02.2025-07.02.2025
Вивчення та аналіз завдання	15.03.2025-14.04.2025
Розробка архітектури та загальної структури системи	14.04.2025-28.04.2025
Розробка структур окремих частин системи	21.04.2025-28.04.2025
Програмна реалізація системи	24.04.2025-27.05.2025
Оформлення пояснювальної записки	05.05.2025-28.05.2025

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок для організації та координації командної роботи*»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

БД	База Даних
API	(Application Programming Interface) інтерфейс програмування додатків.
REST	(Representational State Transfer) архітектурний підхід до мережевих протоколів.
ORM	(Object Relational Mapping) Об’єктно-реляційне відображення.
JSON	(JavaScript Object Notation) Текстовий формат обміну даними.
JWT	(JSON Web Token) Стандарт токена доступу на основі JSON.
HTML	(HyperText Markup Language) Мова розмітки гіпертексту для перегляду вебсторінок у браузері.
CSS	(Cascading Style Sheets) Каскадні таблиці стилів.
JS	(JavaScript) Мова програмування
MVC	(Model-View-Controller) Архітектура
SSE	(Server Side Events) Технологія повідомлень клієнту від сервера

ВСТУП

Сьогодні існує багато різних методологій управління проектами, і кожна команда має можливість обрати ту, що найкраще підходить для її специфіки роботи. Можна користуватися широким вибором інструментів, таких як Notion, Jira, Worksection і Trello. Однак, для деяких команд ці інструменти можуть виявитися не дуже зручними або недостатньо гнучкими. Іноді хочеться мати більший контроль над функціоналом і оновленнями, щоб адаптувати їх до своїх потреб.

Метою цього проекту є створення веб-застосунку, який відповідатиме специфічним вимогам деяких команд розробників. Система буде продуктивною, оптимізованою, зручною у використанні та розробленою за допомогою сучасних інструментів. Це забезпечить продуктивність, масштабованість і легкість подальшого редагування та додавання нових функцій.

Система передбачає створення окремого інтерфейсу для замовників. Це дозволить їм спостерігати за процесом розробки, переглядати задачі та впливати на процес роботи, додавати свої коментарі чи пропозиції.

Проект дозволить командам ефективно організовувати свою роботу, створювати спеціальні чати для обговорень, управляти задачами та взаємодіяти із замовниками. Завдяки адаптованості та простоті редагування, цей інструмент стане зручним рішенням для команд із вузькими специфічними потребами.

Система буде спрямована на використання невеликими командами для яких не треба величезної кількості функцій. Вона буде легка в освоєнні, високу швидкість роботи та інтуїтивний інтерфейс як і для команди так і для замовника.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Гнучке управління проектами: роль Scrum і Kanban

Управління проектами є важливою складовою діяльності будь-якої IT-команди. Методи Scrum і Kanban стали одними з найпоширеніших методологій менеджменту, що використовуються для організації процесів. Вони дозволяють розбивати роботу на невеликі задачі, встановлювати пріоритети та ефективно відслідковувати прогрес. Постійне оновлення поставлених задач і контроль за їх виконанням є критично важливим для досягнення успіху проекту.

1.1.1 Scrum

Scrum — це фреймворк, який допомагає людям, командам та організаціям бути продуктивнішими завдяки адаптивним рішенням для вирішення складних проблем. [1].

Важливість Scrum:

- Scrum допомагає розбивати великі цілі на менші, яких легше досягнути.
- Дозволяє швидко реагувати на зміни.
- Забезпечує регулярну оцінку прогресу і підвищує якість кінцевого продукту.

1.1.2 Kanban

Kanban — це методологія управління задачами, яка спрямована на візуалізацію робочого процесу і оптимізацію його ефективності [2].

Канбан-дошка – це візуальний інструмент, який дозволяє відображати задачі у вигляді карток, розміщених у колонках, що відображають етапи виконання (наприклад, "Заплановано", "Виконується", "Завершено").

Важливість Kanban:

- Дозволяє легко відслідковувати стан задач у реальному часі
- Зменшує час, витрачений на перемикання між задачами
- Забезпечує гнучкість при розподілі пріоритетів

1.1.3 Важливість

Чому ці методології важливі:

- Покращення ефективності: Scrum і Kanban дозволяють оптимізувати процеси, скоротити час реалізації проектів і підвищити продуктивність команди
- Прозорість: Візуалізація задач та прозорий прогрес допомагають усім учасникам команди і замовникам бути в курсі справ
- Гнучкість: Обидві методології дозволяють швидко адаптуватися до змін, що особливо важливо в IT-проектах
- Фокус на якості: Постійне спостереження за прогресом і регулярні оцінки забезпечують високий рівень якості кінцевого продукту

Методології Scrum і Kanban є ключовими інструментами для сучасних IT-команд, які прагнуть досягти високої ефективності та якості роботи. Вони дозволяють організувати процеси таким чином, щоб забезпечити прозорість, гнучкість і продуктивність команди, що є важливими факторами для успішної реалізації проектів.

1.2 Аналіз існуючих інструментів

Для дотримання методологій Scrum/Kanban часто використовуються різні інструменти які дозволяють зручно слідкувати за всіма процесами. Розробка такого виду інструменту є метою цієї роботи.

Розглянемо найвідоміші інструменти, врахуємо плюси і мінуси кожного з них щоб врахувати це під час формування вимог до веб-застосунку.

1.2.1 Notion

Notion — це багатофункціональний інструмент, який поєднує функції менеджера задач, бази даних, текстового редактора та інструменту для створення нотаток. Його основна перевага полягає у високій гнучкості, яка дозволяє користувачам налаштовувати робочий простір під свої потреби. [3]

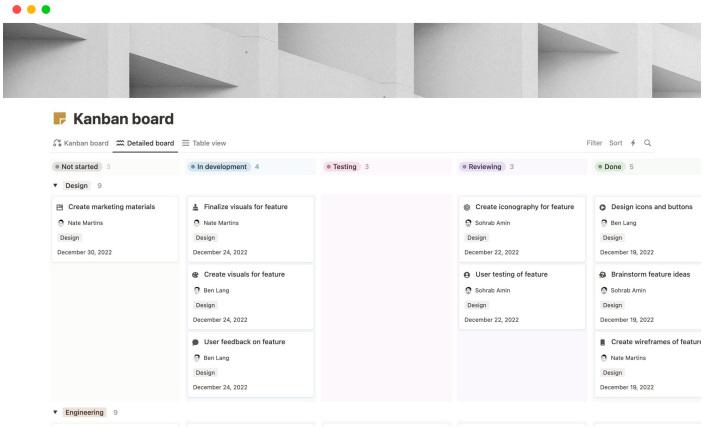


Рисунок 1.1 – Інтерфейс Kanban у Notion

Переваги:

- Гнучкість налаштувань: Користувачі можуть створювати власні шаблони, використовувати бази даних, канбан-дошки, таймлайни та інші інструменти для організації роботи.
- Колаборація в реальному часі: Notion дозволяє кільком користувачам працювати над одним документом одночасно.
- Інтеграція: Підтримує інтеграції з іншими популярними сервісами, такими як Slack, Google Drive і Zapier.
- Кросплатформеність: Може використовуватися на всіх основних платформах, включаючи Windows, macOS, Android і iOS.

Недоліки:

- Складність для новачків: Велика кількість функцій може створювати додатковий бар’єр для нових користувачів. Також замовнику може бути незручно відслідковувати прогрес через важкість в освоєнні.

- Обмеження інтеграцій: Хоча Notion підтримує інтеграції, їх кількість все ще обмежена у порівнянні з іншими інструментами.
- Відсутність спеціалізації: Notion є універсальним інструментом, але може не відповідати специфічним потребам команд, які працюють за методологіями Scrum або Kanban.
- Відсутність спеціалізованих інструментів для зворотного зв’язку
- Відсутність функціоналу командного менеджменту: Notion не має вбудованих інструментів для управління командою, таких як відстеження доступності співробітників чи розподіл ролей у команді

Аналіз:

Notion є чудовим інструментом для організації загальних процесів, але його універсальність та багатофункціональність стає недоліком для команд, які потребують вузькоспеціалізованого функціоналу.

Notion має величезну кількість функцій, серед яких варто виділити відображення задач у вигляді Timeline, яким користуються доволі часто для відстеження порядку виконання задач та термінів виконання задач. Timeline — це спрощений і досить інтуїтивний вигляд **діаграми Ґанта**.

Діаграма Ґанта (також стрічкова діаграма, графік Ґанта) — діаграма, яка використовується для ілюстрації плану, графіка робіт за будь-яким проєктом. Є одним з засобів планування та управління проєктами. [4]

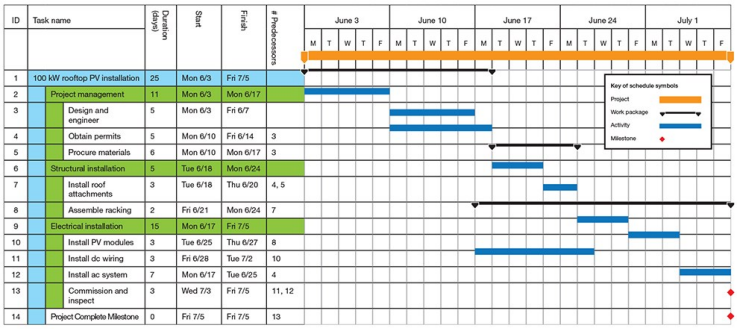


Рисунок 1.2 – Діаграма Ґанта

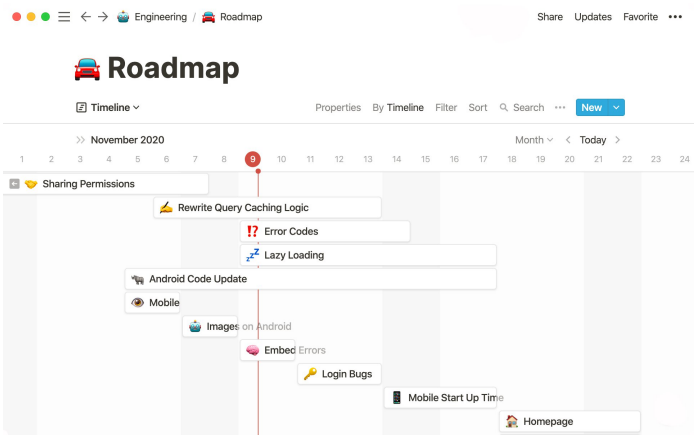


Рисунок 1.3 – Timeline у Notion

Діаграма Ґанта інколи може здаватися складною та не інтуїтивною у використанні, а така інтерпретація дозволяє швидко зрозуміти коли буде готова певна задача кожному хто на неї подивиться, навіть тим хто вперше її бачить, в тому числі і замовнику, тому це важливий інструмент який варто використати в проєкті.

Під час розробки власного веб-застосунку варто врахувати необхідність більшої адаптації до потреб IT-команд, так як велика гнучкість Notion стала його основним недоліком. Також реалізація Timeline є важливим етапом розробки, так як цей інструмент користується популярністю та є дуже важливим для менеджменту. Використання Timeline дозволяє команді замовнику швидко оцінити прогрес проєкту та вчасно внести корективи, що мінімізує ризики затримок.

1.2.2Jira

Jira — це потужний інструмент для управління проектами, який спеціально розроблений для команд, що працюють за методологіями Agile, такими як Scrum і Kanban. Він пропонує широкий набір функцій для планування, відстеження та випуску програмного забезпечення [5].

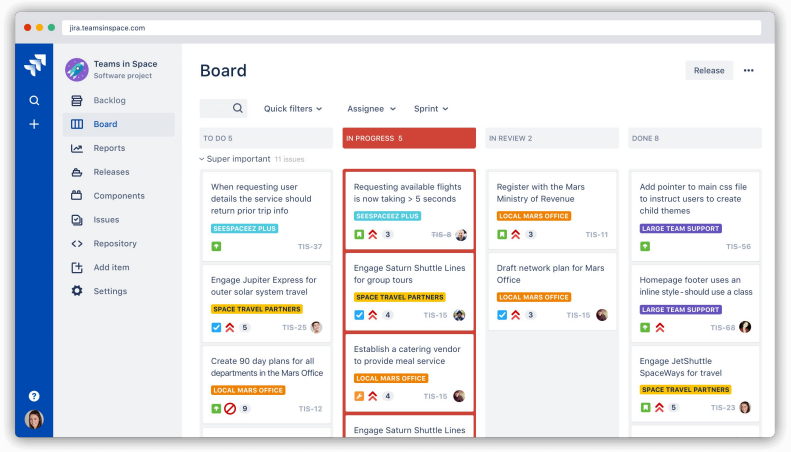


Рисунок 1.4 – Інтерфейс Jira

Переваги:

- Гнучкість у налаштуваннях: Jira дозволяє створювати власні робочі процеси, які можна адаптувати до специфіки проєкту.
- Інтеграція з Agile: Інструмент підтримує Scrum-дошки, Kanban-дошки, спринти, беклоги та інші елементи Agile.
- Прозорість: Дає змогу відстежувати прогрес задач у реальному часі, включаючи статуси, терміни виконання та відповідальних осіб.
- Зв’язки між задачами: Дозволяє встановлювати залежності між задачами (наприклад, одна задача не може початися, поки не завершиться інша).
- Аналітика та звітність: Пропонує потужні інструменти для створення звітів, графіків і діаграм (наприклад, Burndown Chart, Velocity Chart), що допомагають оцінити ефективність команди.
- Інтеграції: Підтримує інтеграцію з великою кількістю сторонніх сервісів, таких як Confluence, GitHub, Slack тощо.
- Кросплатформеність: Доступний як веб-додаток, а також має мобільні додатки для iOS і Android.
- Має вбудовані можливості для управління командами, які дозволяють ефективно організовувати роботу команд

Недоліки:

- Складність освоєння: Jira має високу складність для нових користувачів через велику кількість функцій і можливостей.
- Вартість: Вартість ліцензії Jira може бути високою для малих команд або стартапів.
- Перевантаження функціоналом: Інструмент може бути занадто складним для невеликих проєктів або команд, які не потребують усіх його можливостей.
- Вимоги до налаштувань: Для ефективного використання Jira може знадобитися багато часу на налаштування робочих процесів.
- Jira не має вбудованого функціоналу чатів, але інтегрується з іншими популярними інструментами для комунікації

Аналіз:

Jira є одним із найбільш популярних інструментів для управління проектами завдяки своїй гнучкості та потужному функціоналу. Це ідеальний вибір для команд, які працюють за методологіями Scrum або Kanban, особливо для великих проєктів із багатьма залежностями між задачами. Однак, для невеликих команд чи незначних проєктів використання Jira може бути недоцільним через її складність і високу вартість.

Головний недоліком цього існтрументу, як і у випадку з Notion є його перезавантаженість. Він складний і дорогий. Він ідеально підійде для великих команд і великий проєктів, але буде незручним для малих команд. Загалом це хороший приклад для наслідування, але треба відкинути неінтуїтивні та маловикористовувані функції.

1.2.3 Trello

Trello — це популярний інструмент для управління задачами, який ґрунтується на методології Kanban. Він пропонує простий і зрозумілий інтерфейс, що дозволяє створювати дошки, списки та картки для організації робочих процесів [6].

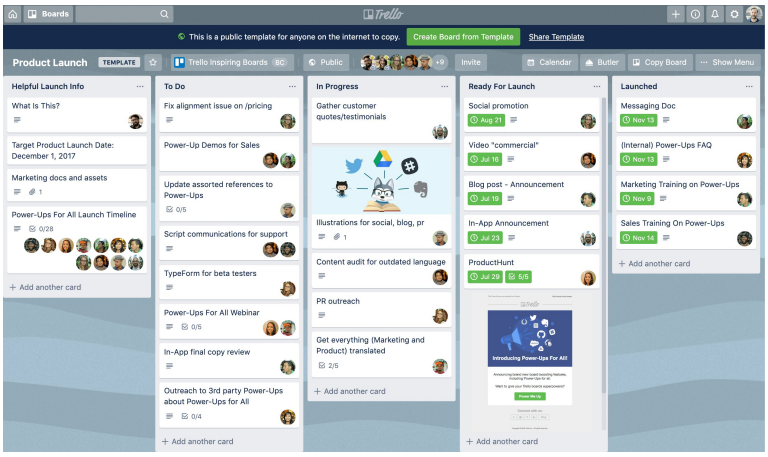


Рисунок 1.5 – Інтерфейс Trello

Переваги:

- Простота використання: Інтуїтивний інтерфейс робить Trello одним із найзручніших інструментів для новачків та невеликих команд.
- Мобільність: Trello доступний на різних платформах (веб, мобільні додатки для iOS та Android).
- Гнучкість: можна створювати різні дошки і колонки з задачами, які можна налаштовувати
- Інтеграції: Підтримує інтеграцію з популярними сервісами, такими як Slack, Google Drive, Dropbox, GitHub, Zapier.
- Безкоштовний тариф: Базовий функціонал доступний безкоштовно, що робить його привабливим для невеликих команд.
- Можливість додавати функціонал якого не вистачає за допомогою Power-Ups

Недоліки:

- Відсутність командного менеджменту: Trello не має розширених інструментів для управління командами (наприклад, відстеження завантаженості, розподіл ролей).
- Відсутність чатів: Trello не має вбудованих чатів для обговорення задач, комунікація обмежена коментарями до карток.

Аналіз:

Trello є чудовим прикладом продуктивного та оптимізованого інструменту з інтуїтивним інтерфейсом. Однак, він має кілька обмежень, таких як відсутність керування командою, відсутність комунікацій та інтерфейсу для замовника, що є важливими вимогами. Тому Trello це також чудова система для наслідування, але потрібно додати деякі функції.

1.2.4 WorkSection

Worksection — це популярна українська платформа для управління проєктами, яка орієнтована на малий і середній бізнес [7]. Вона надає інструменти для планування, співпраці та контролю за виконанням задач у командах.

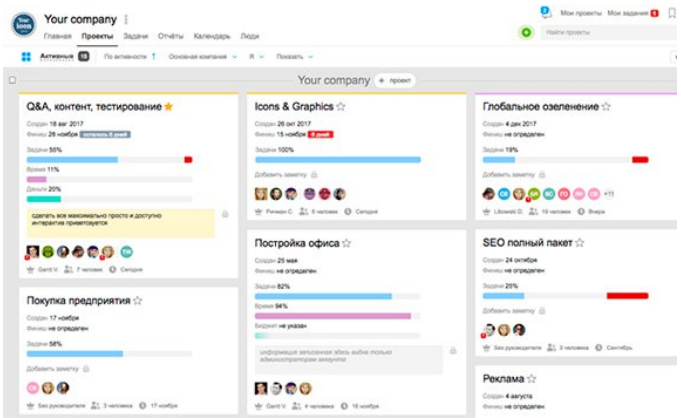


Рисунок 1.6 – Інтерфейс WorkSection

Переваги:

- Календар і діаграма Ґанта: Інтегрований календар та діаграма Ґанта дозволяють візуалізувати задачі та терміни їх виконання
- Командна співпраця: Можливість додавання членів команди до проєктів, обговорення задач через коментарі та спільний доступ до файлів
- Звіти та аналітика: Інструменти для автоматичного створення звітів про виконання задач і продуктивність команди
- Кросплатформеність: Платформа доступна на Windows, macOS, iOS та Android, що дозволяє працювати з будь-якого пристрою

Недоліки:

- Обмежені можливості кастомізації
- Недуже зручний та інтуїтивний інтерфейс
- Відсутність інтерфейсу для замовника
- Зроблений за допомогою JQuery

Аналіз:

Worksection є хорошим базовим інструментом для управління проєктами, але реалізація більш гнучкого, інтуїтивного і простого інтерфейсу, додавання чатів, більше налаштувань саме для менеджменту задач та урізання функціоналу який рідко використовується дозволить покращити продукт. До того ж він зроблений за допомогою застарілої бібліотеки JQuery, через яку в системі присутні помилки, її важко оновлювати та вдосконалювати.

Висновок і формування вимог

Отже, проаналізувавши декілька інструментів для управління проєктами, врахувавши їхні переваги та недоліки, можна зробити такі висновки:

- Notion і Jira — це потужні та багатофункціональні інструменти, які пропонують велику кількість функцій, інтеграцій і налаштувань. Вони ідеально підходять для великих проєктів і команд, які потребують гнучкості та глибокої кастомізації. Однак, через свою складність і багатофункціональність, ці інструменти можуть бути незручними для малих команд або простих проєктів.
- Trello і Worksection — це легкі та інтуїтивно зрозумілі інструменти, які орієнтовані на менші команди та проєкти. Вони прості у використанні, але їх функціонал місцями обмежений і місцями зайвий, що може створювати певні незручності для певних команд.
- Ідеальним рішенням може стати створення інструменту, який поєднує переваги Trello і Worksection, додаючи необхідні функції (наприклад, діаграми Ґанта, чати для команд, розширений тім-менеджмент) та усуваючи їхні недоліки. Це дозволить створити зручний і ефективний інструмент, який буде орієнтований на малі та середні команди, і залишатиметься простим у використанні.

На Основі аналізу можна зробити порівняльну таблицю:

Таблиця 1.1 – Порівняльна таблиця Інструментів

Характеристика	**Notion**	**Jira**	**Trello**	**WorkSection**
Важкість освоєння	Висока	Висока	Низька	Середня
Інтуїтивність	-	-	+	+/-
Масштаб команди/проєкту	Середній	Великий	Малий	Середній
Менеджмент команди	-	+	-	+/-
Чати	-	+/-	-	-
Діаграма Ґанта	+/- (спрощена)	+	-	+
Кастомізація дошок	+	+/-	+	-
Права доступу у користувачів	+/-	+	-	+/-
Доступність для замовника	+/-	+	-	+

Отже, метою проекту є реалізація проекту з низьким порогом входу, який буде продуктивний і швидкий та буде ідеально підходити для команд середнього розміру а також буде легко зрозумілим для замовників.

Для успішної розробки веб-застосунку важливо визначити основні вимоги, що забезпечать його ефективність, зручність і відповідність потребам кінцевих користувачів. Вимоги до проєкту можна розділити на функціональні та нефункціональні.

Функціональні вимоги:

1. Методології Scrum і Kanban
    a. Можливість Створювати проекти
    b. Можливість налаштовувати довільну кількість канбан-дошок в кожному з проектів відповідно до потреб команди
    c. Мати можливість обмежити доступ до певний дошок учасникам команди
2. Управління задачами
    a. Створення, редагування та видалення задач
    b. Встановлення пріоритетів для задач
    c. Відстеження статусу задач за допомогою канбан-дошок
    d. Можливість встановлення термінів виконання та перегляд задач у вигляді Timeline
3. Комунікація
    a. Створення довільної кількості чатів в кожному проекті з можливістю налаштовувати учасників чату
    b. Інтеграція Telegram бота
4. Користувачі
    a. Мати можливість створювати групи і підгрупи для користувачів
    b. Кожен користувач матиме свої права

c. Користувачем також може бути замовник, зі своїми рівнями доступу
d. Таблиця доступностей, для налаштування гнучкого графіку роботи

Нефункціональні вимоги:

1. Оптимізація та продуктивність системи
2. Висока швидкість роботи
3. Інтуїтивний інтерфейс
4. Простота у використанні
5. Безпека (обмеженість в діях кожного користувача)
6. Орієнтація на невеликі команди та проекти

ВИСНОВОК РОЗДІЛУ

У результаті проведеного аналізу предметної області було визначено, що методології Scrum і Kanban є ключовими підходами для ефективного управління проєктами в IT-сфері. Вони забезпечують гнучкість, прозорість та оптимізацію робочих процесів.

Аналіз існуючих інструментів, таких як Notion, Jira, Trello та Worksection, показав, що кожен із них має свої сильні та слабкі сторони. Notion і Jira вирізняються багатофункціональністю та гнучкістю, але є складними у використанні, особливо для малих команд. Trello і Worksection, навпаки, орієнтовані на простоту та легкість, але мають обмежений функціонал, який не завжди задовольняє потреби команд та замовників.

Для створення ефективного веб-застосунку необхідно поєднати переваги цих інструментів, усуваючи їхні недоліки. Орієнтиром має стати оптимізація, продуктивність, інтуїтивність та можливість адаптації під невеликі команди і середні проєкти. Важливими аспектами є реалізація гнучких Kanban-дошок та Timeline для візуалізації прогресу, додавання чіткої системи управління задачами, наявність чатів у проєктах для полегшення комунікації, інтеграція з Telegram ботом для сповіщень, запровадження таблиці доступностей для гнучкого планування роботи команди та забезпечення прав доступу для різних категорій користувачів, включаючи замовників.

Таким чином, метою проєкту є створення інструменту, який буде легким у використанні, але водночас функціональним, орієнтованим на потреби невеликих команд. Цей інструмент повинен забезпечувати зручність, швидкість роботи та відповідати сучасним вимогам безпеки. Розробка такого веб-застосунку дозволить поєднати простоту Trello та функціональність Jira, доповнивши їх важливими інструментами, такими як чати, Timeline, таблиця доступностей та розширений менеджмент задач.

РОЗДІЛ 2
АНАЛІЗ ТА ВИБІР ІНСТРУМЕНТІВ ДЛЯ РОЗРОБКИ

Для створення сучасного, оптимізованого, продуктивного та інтуїтивного веб-застосунку, який відповідає вимогам, визначеним у розділі 1, необхідно ретельно підібрати інструменти для розробки. У цьому розділі буде проведено аналіз інструментів для фронтенду та бекенду.

2.1 Вибір інструментів для фронтенду

Фронтенд є ключовим компонентом веб-застосунку, адже саме він забезпечує інтерактивність, зручність використання та візуальну частину інтерфейсу. Для розробки фронтенду необхідно обрати технології, які забезпечать:

- Швидкість роботи.і продуктивність
- Інтуїтивний і адаптивний дизайн.
- Простоту розробки та підтримки.

Оберемо один серед трьох найпопулярніших інструментів:

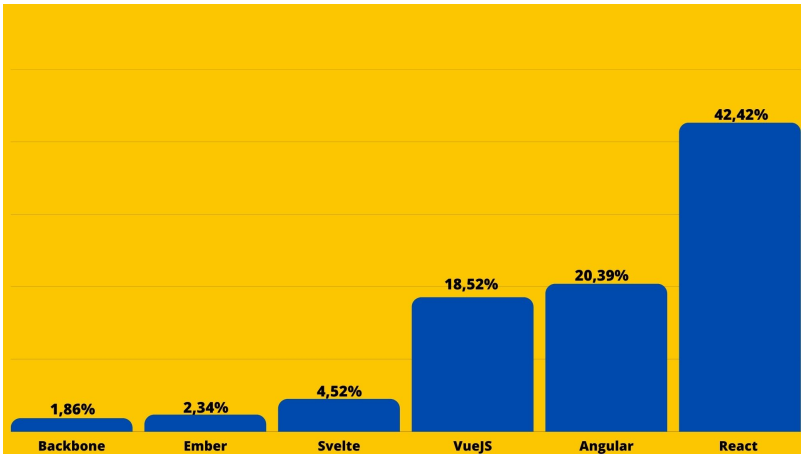


Рисунок 2.1 – Найпопулярніші фронтенд фреймворки [8]

2.1.1 React

Переваги:

- Підтримка компонентного підходу.
- Висока продуктивність завдяки віртуальному DOM.
- Велика кількість бібліотек та інструментів для розширення функціоналу.
- Активна спільнота та хороша документація.

Недоліки:

- Важкість в освоєнні для новачків.

Аналіз:

Загалом React виглядає як хороший вибір: швидкий, орієнтований на проєкти середніх розмірів, має хорошу документацію та багато бібліотек, проте є суттєвий недолік: він важкий в освоєнні. Основна проблема React полягає у складності для новачків і великої гнучкості, яка призводить до того що кожен проєкт має свою структуру і підхід, до якого важко звикнути новому розробнику.

“Недосвідчений розробник, який спробує побудувати додаток за допомогою цієї бібліотеки, швидше за все створить чудовисько Франкенштейна, яким складно керувати” [8] – цитата професійного розробника.

2.1.2 Vue

Переваги:

- Простий для вивчення та розробки.
- Підтримка компонентного підходу.
- Активна спільнота та хороша документація.

- Велика кількість бібліотек та інструментів для розширення функціоналу.

Недоліки:

- Хоч Vue і має велику спільноту і велику кількість бібліотек, проте вона менша ніж в React
- Менш гнучкий ніж React

Аналіз:

Vue виглядає як чудовий вибір, так само як і React швидкий, орієнтований на проєкти середніх розмірів, має хорошу документацію та багато бібліотек і не має суттєвого мінусу у вигляді складності для новачків і великої гнучкості. На відмінну від React, Vue має більш шаблонний підхід, тому проекти на ньому структуровані і новачку важче створити “Франкенштейна” [8]. Хоч Vue і не такий гнучкий як React і має меншу спільноту, він залишається чудовим вибором тому що нам не потрібна величезна гнучкість і величезна кількість бібліотек.

2.1.3 Angular

Переваги:

- Повний фреймворк із вбудованими інструментами для роботи з формами, маршрутизацією та сервісами.
- Висока продуктивність для великих проєктів.

Недоліки:

- Велика вага фреймворку.
- Складність для новачків.

Аналіз:

Angular не підходить під наші цілі. Він об’ємний, важкий в освоєнні і орієнтований на продуктивність у великих проєктах [9], а наша розробка невелика і не потребує тих плюсів які надає даний фреймворк.

2.1.4 Вибір

Серед проаналізованих фреймворків найбільше нам підходить Vue, адже:

- Орієнтований на малі-середні масштаби проєктів.
- Оптимізований і швидкий.
- Легкий в освоєнні
- Структурований
- Має хорошу документацію та спільноту.
- Активно розвивається.
- Має достатню кількість бібліотек та модулів.

2.1.5 Vite

Для збірки та розробки проєкту на основі Vue буде використано Vite. Цей інструмент забезпечує такі переваги:

- Швидкість: Завдяки ES-модулям забезпечується миттєве завантаження проєкту [10].
- Простота інтеграції: Готові шаблони для Vue прискорюють налаштування.
- Оптимізованість, продуктивність: Мінімальні вимоги до ресурсів і зменшення розміру фінального застосунку [10].

Vite є сучасним, легким і швидким інструментом, який підвищить швидкість розробки і оптимізує розміри збірки, він ідеально підходить для нашого проєкту.

2.2 Вибір інструментів для бекенду

Бекенд відповідає за обробку даних, логіку застосунку та інтеграцію з базами даних. Для вибору бекенд-інструментів необхідно врахувати простоту розробки та підтримки а також достатню продуктивність. Нам не потрібно щось дуже продуктивне і гнучке, потрібне щось шаблонне, з невеликим порогом входу, що легко підтримувати і водночас достатньо швидке.

Вимоги до бекенд-інструментів:

- Підтримка REST API
- Достатня продуктивність (підтримка великої кількості одночасних запитів)
- Підтримка SSE (для реалізації чатів у реальному часі)
- Підтримка бази даних

2.2.1 Express

Express — це мінімалістичний фреймворк для Node.js, який спрощує розробку API [11]. Node.js — це популярне середовище виконання JavaScript, що дозволяє створювати бекенд-застосунки.

Переваги:

- Продуктивність: Завдяки неблокуючій архітектурі Node.js добре підходить для обробки великої кількості одночасних запитів [12].
- JavaScript: Використання однієї мови (JavaScript) для фронтенду та бекенду спрощує розробку.
- Гнучкість: Express дозволяє легко створювати REST API та інтегрувати SEE для повідомлень.
- Екосистема: Велика кількість бібліотек та модулів у npm.
- Легка інтеграція з базами даних.

Недоліки:

- Ручне налаштування: express.js не є повноцінним фреймворком, тому багато елементів (безпека, структура проєкту) потрібно налаштовувати вручну
- Відсутність конвенцій: Немає чітких стандартів організації коду.

Аналіз:

Express ідеально підходить для малих і середніх проєктів. Він забезпечує достатню продуктивність, гнучкість і простоту інтеграції з фронтендом. Хоча Express не є повноцінним фреймворком, існують різні шаблони організації коду. Використання JavaScript для всього стеку є великим плюсом для швидкості розробки.

2.2.2 Flask

Flask — це фреймворк для Python, орієнтований на простоту та гнучкість при розробці веб-застосунків.

Переваги:

- Продуктивність: Flask є мінімалістичним, що дозволяє додавати лише потрібний функціонал.
- Гнучкість: Flask легко адаптувати під специфічні вимоги проєкту
- Легка інтеграція з базами даних.

Недоліки:

- Потребує знання Python окрім JavaScript для фронтенду.
- Менша продуктивність: У порівнянні з Node.js, Python може бути менш ефективним для обробки великої кількості одночасних запитів.

- Обмеження в асинхронності: Хоча Python підтримує асинхронність, вона менш ефективна, ніж у Node.js.

Аналіз:

Flask є чудовим вибором для швидкої розробки завдяки простоті Python. Цей фреймворк підходить для гнучких і мінімалістичних застосунків, але має обмеження в продуктивності та потребує використання різних мов програмування.

2.2.3 Spring Boot

Spring Boot — це фреймворк для Java, який спрощує створення веб-додатків на цій мові.

Переваги:

- Надійність: Java є однією з найбільш стабільних мов програмування.
- Підходить для великих проєктів: Spring Boot добре працює з комплексними системами [13].

Недоліки:

- Складність: Java та Spring Boot важкі в освоєнні.
- Велика вага: Фреймворк і мова потребують більше ресурсів.

Аналіз:

Spring Boot є відмінним вибором для великих проєктів, але його складність і вага роблять його менш підходящим для мінімалістичних застосунків. Тому це поганий вибір для нашої системи.

2.2.4 Вибір

Серед проаналізованих інструментів нам підходить Flask або Express. Вони обидва спрямовані на проекти малих-середніх розмірів, мають достатню продуктивність для рівня обчислень нашої системи, можуть реалізовувати Rest API та працювати з SSE та базами даних. Проте Express має перевагу в тому що він буде написаній на одній мові програмування з фронтендом, що пришвидшить розробку, тож вибором буде **Express.**

Express є оптимальним рішенням для створення мінімалістичного, швидкого та гнучкого бекенду:

- Він забезпечує високу продуктивність завдяки неблокуючій архітектурі.
- Дозволяє використовувати одну мову (JavaScript) для фронтенду та бекенду.
- Легко інтегрується з базами даних і підтримує SSE для реалізації чатів.
- Має велику кількість бібліотек, що спрощує розробку.

2.2.7 TypeScript

Для забезпечення більшої надійності коду та зручності розробки бекенду буде використано TypeScript — надбудову над JavaScript, яка додає статичну типізацію [14].

Переваги для проєкту:

- Зменшення часу на дебаг: Завдяки перевірці типів на етапі компіляції зменшується кількість помилок [14].
- Зручність командної роботи: TypeScript робить код більш зрозумілим і передбачуваним, що важливо при командній розробці.
- Масштабованість: Завдяки чіткій структурі коду проект стане легшим у підтримці та розширенні.

TypeScript забезпечить високу якість коду, стабільність та зручність розробки, що є важливим для нашого веб-застосунку.

2.2.5 База Даних

Для зберігання даних, таких як задачі, користувачі, чати, права доступу тощо, необхідно обрати базу даних, яка буде надійною, продуктивною та відповідатиме вимогам проєкту. Бази даний бувають реляцій та нереляційні, варто обрати який тип нам підходить краще. Оберемо найпопулярнішу реляційну та нереляційну бази які орієнтовані на проєкти середній розмірів та порівняємо їх. Серед найпоширеніших варіантів — MySQL (реляційна база даних) і MongoDB (документоорієнтована база даних).

MySQL — компактний багатопотоковий сервер баз даних. Характеризується високою швидкістю, стійкістю і простотою використання. [15].

Переваги:

- Реляційна структура: Відмінно підходить для даних, які мають чіткі зв’язки (наприклад, задачі, які належать до конкретного проєкту чи користувача).
- простота у встановленні та використанні [15]
- висока швидкість виконання команд [15]
- підтримується необмежена кількість користувачів, що одночасно працюють із БД [15]
- Надійність: наявність простої і ефективної системи безпеки [15].

Недоліки:

- Складність масштабування: Вертикальне масштабування потребує додаткових ресурсів.

- Складність налаштування: Може потребувати більше часу на налаштування порівняно з нереляційними базами.

MongoDB — це документоорієнтована база даних, яка зберігає дані у форматі JSON-подібних документів [16].

Переваги:

- Гнучкість: Дозволяє зберігати дані з динамічною структурою, що підходить для проєктів із мінливими вимогами.
- Легке масштабування: Широке горизонтальне масштабування [16].
- Швидкість: Висока продуктивність для простих операцій запису та читання [16].
- Простота роботи з JSON: Природна інтеграція з JavaScript та легкість у роботі з даними з динамічною структурою.

Недоліки:

- Відсутність складних запитів: Не підходить для проєктів, які потребують складних зв’язків між даними.
- Менша надійність: Не забезпечує таких високих гарантій цілісності даних, як MySQL

Отже, для нашого проєкту, де дані мають чіткі зв’язки (наприклад, задачі, які належать до конкретних проєктів і користувачів), а також потрібна надійність і підтримка складних запитів, **MySQL** є кращим вибором.

MySQL забезпечить стабільність, гнучкість та продуктивність, що повністю відповідає вимогам нашого проєкту.

Для взаємодії з базою даних можна використати бібліотеку Sequelize.js

Sequelize.js — це ORM бібліотека для Node.js для роботи з реляційними базами даних (наприклад, PostgreSQL, MySQL, SQLite, MariaDB) через JavaScript-код без написання SQL запитів.

2.2.6 MVC

Для організації бекенду буде використано архітектурний підхід MVC (Model-View-Controller). MVC є популярним підходом для структурування веб-додатків, який розділяє логіку програми на три основні компоненти: модель, представлення та контролер. Це архітектурний шаблон, який використовується під час проєктування та розробки програмного забезпечення [17].

Переваги використання MVC у проєкті:

- Чітка структура: MVC дозволить зберегти організованість коду, що особливо важливо для командної роботи.
- Гнучкість: Легко додавати новий функціонал, змінювати логіку чи формат даних без впливу на інші компоненти.
- Сумісність з Node.js Express: Express чудово підтримує реалізацію MVC завдяки своїй гнучкості та мінімалістичному підходу.
- Прозорість: Розділення логіки забезпечує чітке розуміння роботи кожного компонента [17].

ВИСНОВОК РОЗДІЛУ

У цьому розділі було проведено аналіз інструментів для фронтенду та бекенду, а також обрано оптимальні технології для розробки веб-застосунку.

Для фронтенду було обрано **Vue.js** у поєднанні з **Vite** та **Pinia**, що забезпечує продуктивність, мінімалістичність і достатню швидкість роботи, простоту використання завдяки легкості в освоєнні, а також структурованість і адаптивність для створення сучасного інтерфейсу.

Для бекенду (розробки **REST API**) було обрано **Express**, який надає достатньо високу продуктивність завдяки неблокуючій архітектурі, можливість використовувати одну мову програмування (JavaScript) для фронтенду та бекенду, а також гнучкість і сумісність із сучасними підходами, такими як MVC.

Для зберігання даних було обрано базу даних **MySQL**, яка забезпечує реляційну структуру для чітких зв'язків між даними та є гарним та швидким рішенням для малих і середніх застосунків. Для взаємодії з базою даних буде використано ORM-бібліотеку **Sequelize.js**, що спрощує роботу з MySQL.

Було обрано архітектурний підхід **MVC**, який забезпечить чітку структуру коду та легкість у підтримці та розширенні функціоналу.

Окрім цього, використання **TypeScript** дозволить виявляти помилки на етапі написання коду завдяки статичній типізації та забезпечити стабільність, масштабованість і зрозумілість коду, що особливо важливо для командної розробки.

Таким чином, обраний стек технологій забезпечує баланс між продуктивністю, простотою розробки та відповідністю вимогам проєкту, що дозволить створити сучасний, інтерактивний і надійний веб-застосунок.

РОЗДІЛ 3
РОЗРОБКА ПРОЄКТУ

3.1 Загальне проєктування

Робота складається з двох частин:

1. Створення REST-API (Back-end)
2. Створення Front-end частини (single-page-application), яка буде служити інтерфейсом для взаємодії з REST-API

Перше завдання буде виконане на мові програмування TypeScript - Node.js, також буде використовуватись бібліотека «express.js» для реалізації веб-сервера та бібліотека «Sequelize.js» для взаємодії з базою даних MySQL

Друге завдання буде виконуватись на фреймворку Vue.js (TypeScript | HTML | CSS)

Можна створити загальну схему роботи додатку:

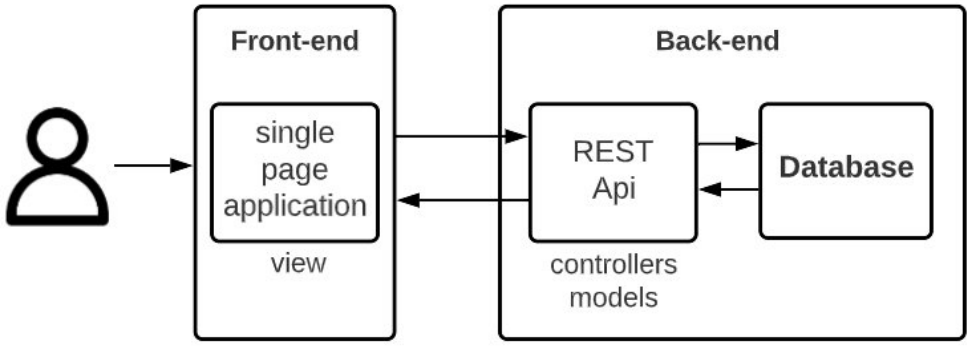


Рисунок 3.1 – загальна структура роботи додатку

Клієнт буде бачити перед собою front-end частину. При взаємодії з нею, вона, в свою чергу, буде спілкуватися з back-end частиною (REST-API).

Back-end частина повністю відділена і незалежна від front-end, що дає нам змогу, наприклад, в майбутньому створити декілька front-end частин зі своїм інтерфейсом, на зовсім різних мовах програмування, все що від них потрібно –

це логіка взаємодії з back-end, тобто отримання інформації та її відображення для клієнта.

3.1.1 Бекенд (серверна частина)

Спрощено роботу бекенду можна зобразити наступною схемою:

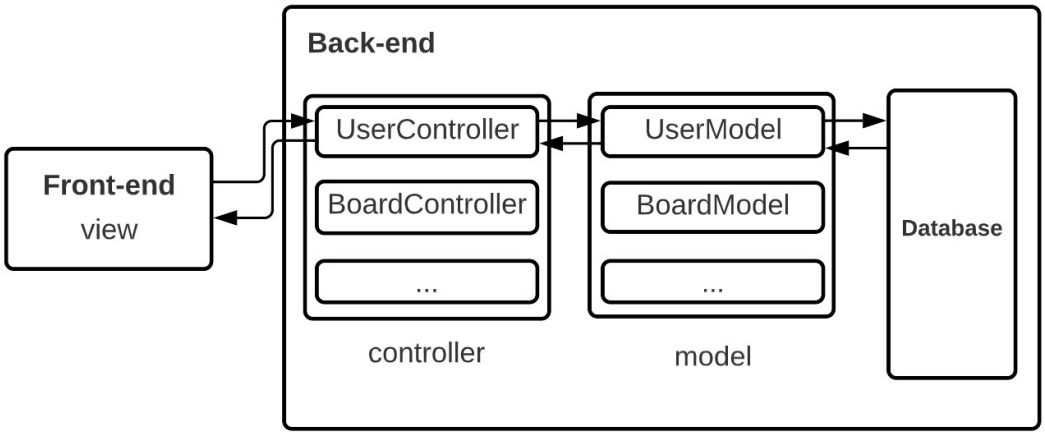


Рисунок 3.2 – загальна спрощена структура роботи бекенду

Ця структура будується за принципами архітектури MVC (Model – View - Controller). Ця архітектура відокремлює бізнес-логіку (модель) від її візуалізації (представлення). За рахунок цього підвищується гнучкість системи та покращуються можливості повторного використання та її підтримки (оновлення) [17].

Ця архітектура містить в собі наступні складові:

1. Models - визначають структуру та логіку використовуваних даних, використовуються для бізнес-логіки
2. Views - визначає візуальну частину, як відображатимуться дані
3. Controllers - обробляють вхідні http-запити, використовуючи для обробки Models, та надсилають у відповідь клієнту результат обробки

Back-end частина бере на себе обов’язки реалізації «controllers» та «models» з архітектури MVC.

Controllers (контроллери)

Ця частина має відповідати за прийняття http-запитів з «views» та використати «models» для отримання результатів і відсилання їх назад до «views». Тобто має бути налаштований роутинг та взаємодія з «models».

Роутинг буде реалізовано за правилами REST (Representational State Transfer).

Основна ідея REST API полягає в поділі різних операцій (найчастіше CRUD) при зверненні до одного і того ж URL за допомогою HTTP методів, основні з яких:

1. GET – використовується для отримання даних
2. POST – використовується для створення нових записів
3. PUT – використовується для редагування вже існуючих записів
4. DELETE – використовується для видалення записів

Реалізувати таку систему можна за допомогою бібліотеки express.js, вона надає інструменти для зручного роутингу, а також містить у собі всі методи REST. Express дозволяє організувати роути в логічні групи та використовувати middleware для обробки запитів до того, як вони надійдуть до контролерів, що дозволяє реалізувати такі функції, як аутентифікація, валідація, логування тощо.

Ось так виглядає більш детальна структура роботи бекенду з концентрацією на “Controllers”:

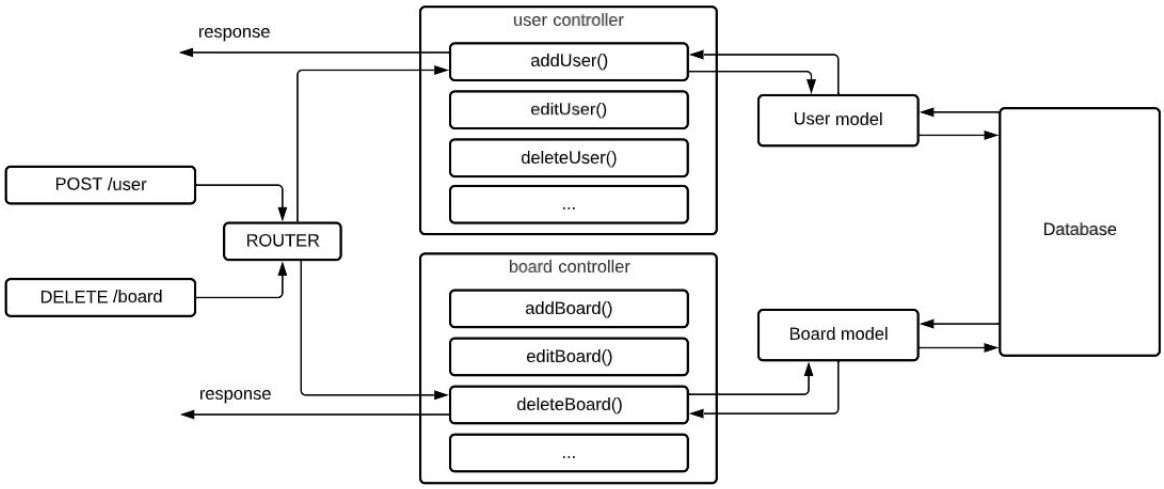


Рисунок 3.3 – загальна спрощена структура роботи “Controllers”

Models (моделі)

«Models» буде реалізовано у вигляді класів, які міститимуть різні методи. Саме в цих методах буде реалізовуватися отримання даних з бази даних та їх організація.

Services (Сервіси)

Взаємодія моделей з базою даних буде не пряма, буде організовано ще один шар "Services", який буде містити методи, що взаємодіють з базою даних через ORM (Object-Relational Mapping). Цей архітектурний підхід дозволяє:

- Відокремити бізнес-логіку від логіки доступу до даних
- Підвищити тестованість коду (можливість мокування сервісів)
- Спростити підтримку та рефакторинг
- Забезпечити єдиний підхід до обробки транзакцій та помилок
- Полегшити майбутні зміни в інфраструктурі (зміна бази даних чи ORM)

В якості ORM буде використано Sequelize.js, який надає потужний набір інструментів для роботи з реляційними базами даних через JavaScript. Sequelize дозволяє:

- Визначати моделі даних у вигляді JavaScript-класів
- Автоматично генерувати SQL-запити

- Керувати міграціями бази даних
- Забезпечувати стійкість до SQL-ін'єкцій
- Реалізовувати складні зв'язки між таблицями (one-to-one, one-to-many, many-to-many)
- Підтримувати транзакції та каскадні операції [18]

Потік даних

В контролері будуть робитися виклики певних методів з певної моделі або сервісу і результат їх роботи буде відправлятися назад клієнтові. Типовий потік даних у backend-частині виглядає так:

- HTTP-запит надходить на серверний маршрутизатор (Router)
- Маршрутизатор визначає відповідний контролер і його метод
- Middleware виконує попередню обробку запиту (аутентифікація, валідація)
- Контролер обробляє запит та викликає потрібні моделі або сервіси
- Моделі реалізують бізнес-логіку та звертаються до сервісів для роботи з даними
- Сервіси взаємодіють з базою даних через ORM
- ORM перетворює операції на SQL-запити до бази даних
- Результати запитів повертаються назад по ланцюжку до контролера
- Контролер формує HTTP-відповідь і відправляє її клієнту

Така архітектура забезпечує чітке розділення відповідальності, полегшує підтримку та розширення функціоналу, а також дозволяє ефективно працювати в команді, де різні розробники можуть зосередитись на різних компонентах системи.

3.1.2 Фронтенд (клієнтська частина)

Front-end частина буде реалізована як SPA (Single-Page Application) за допомогою фреймворку Vue.js. SPA-підхід дозволяє створювати динамічні веб-додатки, які не перезавантажують сторінку під час використання, а змінюють DOM сторінки у відповідь на дії користувача, що забезпечує плавну взаємодію аналогічну до настільних додатків.

Vue.js надає спеціальну структуру та підхід для розробки веб-додатків, використовуючи компонентний підхід: весь веб-додаток буде складатися з інкапсульованих компонентів, які взаємодіють один з одним, та містять в собі певну логіку, верстку та стилі. Кожен компонент є «Чорним ящиком», який працює зі своїми даними, який має свій життєвий цикл та стан, що дозволяє логічно розділити додаток на незалежні частини [19].

Компоненти Vue.js організовані в ієрархічну деревоподібну структуру, де батьківські компоненти передають дані дочірнім через властивості (props), а дочірні сповіщають батьківські про події за допомогою емітованих подій (events). Така структура забезпечує однонаправлений потік даних, що спрощує відлагодження та розуміння логіки програми.

Крім того, Vue.js надає ефективні інструменти для:

- Реактивної роботи з даними (Composition API, Options API)
- Маршрутизації сторінок без перезавантаження (Vue Router)
- Централізованого управління станом додатку (Pinia/Vuex)
- Створення переходів та анімацій між станами інтерфейсу
- Використання директив для прямих DOM-маніпуляцій

Така архітектура надає проекту значної гнучкості, дозволяє легко масштабувати систему, вдосконалювати окремі компоненти без впливу на інші частини програми та ефективно організовувати командну розробку, коли різні розробники можуть працювати над різними компонентами одночасно. Крім

того, компонентний підхід спрощує повторне використання коду та модульне тестування функціональності.

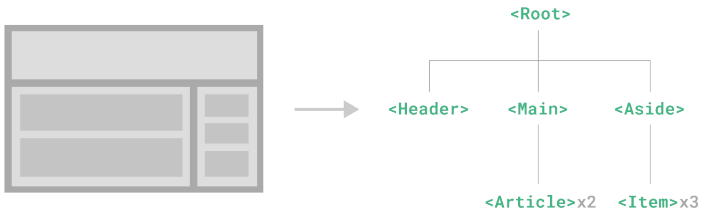


Рисунок 3.4 – представлення сторінки у вигляді дерева компонентів [19]

Також в проекті використовується глобальний «Store» (Pinia) - це об’єкт з даними, та різними методами для взаємодії з цими даними, доступ до якого можливий з кожного компоненту [20].

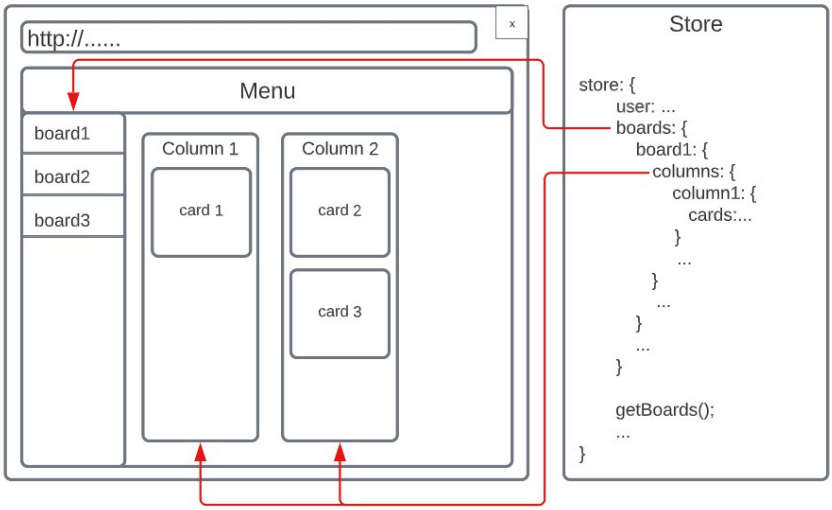


Рисунок 3.5 – відображення даних на сторінці з глобального Store.

Vue.js надає можливість працювати напряму з даними, не думаючи про те, як буде відбуватися їх рендер на екран, при зміні даних всі візуальні зміни реактивно відображаються на екрані. В даній схемі (див. рис. 3.5) метод getBoards() має робити запит на сервер і отримувати у відповідь список з дошок

і записувати його в поле boards, це поле автоматично відмалюється при отримані, та буде реактивно перемальовуватися при внесені змін в нього, нам не потрібно буде втручатися в це малювання, всю відповідальність за це несе фреймворк.

За допомогою Vue.js та такої структури можна будувати великі, гнучкі та швидкі веб-додатки. Він бере на себе частину «view» з архітектури MVC, тобто відображає клієнту дані, які приходять від «conroller».

3.1.3 Авторизація через JWT

JWT (JSON Web Token) – це відкритий стандарт токенів доступу, що забезпечує безпечну передачу інформації між сторонами. На відміну від традиційних сесійних механізмів, JWT дозволяє реалізувати безстатусну (stateless) авторизацію, коли сервер не зберігає сесійний стан.

У нашій системі реалізовано підхід з використанням "вічного" JWT-токена, який містить лише ідентифікатор користувача. Такий мінімалістичний підхід зменшує розмір токена і забезпечує високу продуктивність системи завдяки зменшенню обсягу даних, що передаються між клієнтом і сервером.

Процес авторизації працює наступним чином:

1. Користувач проходить автентифікацію, надаючи свої облікові дані (логін/пароль)
2. Система перевіряє облікові дані та, у випадку успіху, генерує JWT-токен з ID користувача
3. Клієнт зберігає цей токен у локальному сховищі (localStorage або cookies) і додає його до заголовка Authorization при кожному запиті
4. Спеціальний middleware на сервері перехоплює запити, вилучає та перевіряє токен
5. Після верифікації токена, система використовує ID користувача для отримання його повних даних з бази даних

6. На основі отриманих даних (включаючи права доступу та статус активності) приймається рішення про доступ до ресурсів

Хоча "вічний" токен може створювати певні ризики безпеки через відсутність терміну дії, в системі впроваджено кілька компенсаторних механізмів:

- Токен містить мінімум інформації (лише ID користувача), що зменшує ризики у випадку компрометації
- При кожному запиті система перевіряє актуальний статус блокування користувача в базі даних
- Права доступу не зберігаються в токені, а завжди отримуються з бази даних, що забезпечує актуальність дозволів
- У випадку компрометації токена, можливе швидке блокування облікового запису користувача

Такий підхід до авторизації значно спрощує взаємодію з системою для користувачів, усуваючи необхідність частого повторного входу, і одночасно підтримує достатній рівень безпеки для систем з неконфіденційними даними. Крім того, JWT-авторизація добре масштабується для розподілених систем та мікросервісної архітектури завдяки своїй безстатусності.

3.2 Авторизація та керування командою

Ця частина відповідатиме за авторизацію та керування користувачами, а саме:

- Можливість авторизуватися в додатку
- Редагувати свій профіль
- Бачити профілі інших користувачів
- Створювати інших користувачів
- Редагувати профілі інших користувачів
- Видаляти або блокувати інших користувачів

- Обмежувати доступ певним користувачам до певних дій за допомогою системи ролей, в кожного користувача може бути кілька ролей
- Доступні ролі: адміністратор, проджект менеджер, розробник та клієнт
- Створювати групи користувачів та підгрупи в цих групах (дерево груп)
- Редагувати групи та учасників груп

3.2.1 База даних

Для початку важливо організувати базу даних, створити таблиці та зв’язки які будуть відповідати нашим вимогам.

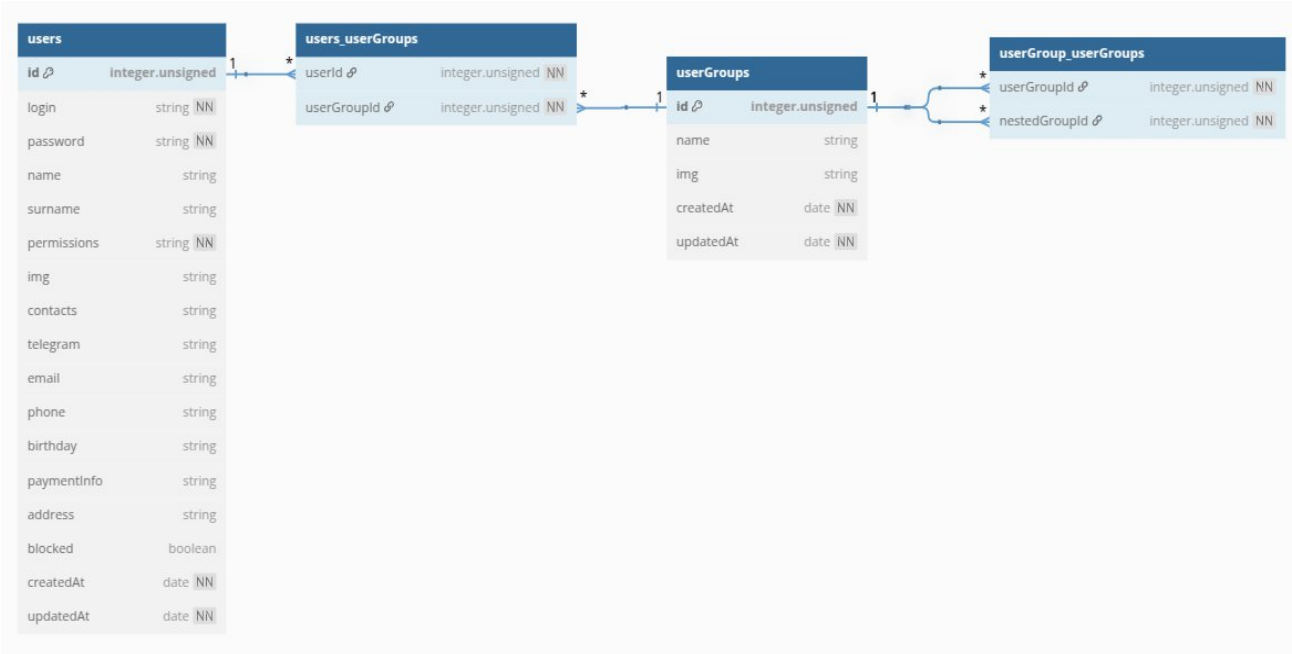


Рисунок 3.6 – діаграма бази даних для авторизації та керування користувачами.

Ця діаграма (див. Рис. 3.6) задовольняє наші вимоги. В таблиці users будуть зберігатися дані всіх користувачів, пароль буде у вигляді хешу, ролі будуть зберігатися в полі permissions розділені через кому. В табличці userGroups будуть зберігатися дані всіх груп. За допомогою таблички users_userGroups можна створити зв’язок між users та userGroups, щоб

користувач міг знаходитись одночасно в декількох групах. За допомогою таблички userGroup_userGroup можна створити зв’язок між групами щоб утворилось дерево груп.

Тепер цю схему потрібно відобразити в ORM:

```
export class UserGroup extends Model<InferAttributes<UserGroup>, InferCreationAttributes<UserGroup>>
  declare id: CreationOptional<number>;
  declare name?: string | null;
  declare img?: string | null;
  declare createdAt: CreationOptional<string>;
  declare updatedAt: CreationOptional<string>;
}

export const userGroupFields = {
  id: {
    type: DataTypes.INTEGER.UNSIGNED,
    primaryKey: true,
    autoIncrement: true,
  },
  name: {
    type: DataTypes.STRING,
    allowNull: true,
  },
  img: {
    type: DataTypes.STRING,
    allowNull: true
  },
  createdAt: {
    type: DataTypes.DATE,
    allowNull: false,
    defaultValue: DataTypes.NOW,
  },
  updatedAt: {
    type: DataTypes.DATE,
    allowNull: false,
    defaultValue: DataTypes.NOW,
  },
};
```

Рисунок 3.7 – відображення таблиці userGroups в ORM Sequelize.

Подібним чином буде реалізовано інші таблиці. Тобто для кожної таблиці буде створена модель за допомогою якої можна робити запити в базу даних без написання SQL запитів. Таблиця users буде реалізована як модель User, таблиця userGroups як модель userGroup, таблиці зв’язків будуть створені окремо і використані в налаштуванні зв’язків. Наступним кроком буде створення зв’язків між таблицями:

```
const user_userGroup = db.define('user_userGroup', {}, {
  tableName: 'users_userGroups'
});

const userGroup_userGroup = db.define('userGroup_userGroup', {}, {
  tableName: 'userGroup_userGroups'
});
```

Рисунок 3.8 – таблиці зв’язків для користувачів та груп.

```
User.belongsToMany(UserGroup, { through: user_userGroup });
UserGroup.belongsToMany(User, { through: user_userGroup });
UserGroup.belongsToMany(UserGroup, { through: userGroup_userGroup, as: 'nestedGroups' });
AvailabilityRecord.hasMany(User);
```

Рисунок 3.9 – створення зв’язків для користувачів та груп.

```
const db = new Sequelize(process.env.DB_NAME as string, process.env.DB_USER as string, process.env.DB_PASS, {
  host: process.env.DB_HOST,
  dialect: 'mysql'
});
```

Рисунок 3.10 – підключення до БД.

Отже база даних налаштована та підключена. Хост, пароль та користувач зберігаються в .env файлі, цей файл створений саме для зберігання подібних констант, щоб в подальшому можна було легко їх знаходити та в разі необхідності змінювати.

Також під час налаштування ORM було створено інтерфейси для цих типів даних, так як в проекті використовується TypeScript. Всі типи будуть записуватися в одному файлі і використовуватись по всьому проекту.

```typescript
export interface IUser {
  id: number,
  login: string,
  password: string,
  name?: string | null,
  surname?: string | null,
  permissions: string,
  img?: string | null,
  contacts?: string | null,
  telegram?: string | null,
  email?: string | null,
  phone?: string | null,
  birthday?: string | null,
  paymentInfo?: string | null,
  address?: string | null,
  blocked?: boolean | null,
  type?: string | null,
  createdAt: string,
  updatedAt: string,
}
```

Рисунок 3.11 – інтерфейс користувача TypeScript.

За таким же принципом були створені інтерфейси і для інших сутностей.

3.2.2 Модель та сервіс користувача

Наступним кроком буде створення сервісу та моделі для користувача щоб контролер міг використовувати їх методи для відсилання даних на клієнтську частину. Ці компоненти відіграють ключову роль в забезпеченні роботи REST API та дотриманні принципів MVC архітектури. Сервіс користувача буде інкапсулювати всю бізнес-логіку та операції з базою даних, а модель забезпечить структуровану роботу з даними користувача в системі.

Базові функції які мають бути присутні в контролері це отримання користувача за допомогою ідентифікатора, отримання всіх користувачів, створення користувача, редагування користувача, видалення користувача.

```
export const userService = {
  getUserById: (id: number):Promise<User> => {
    return new Promise((resolve, reject) => {
      if (id === -798) {
        resolve(new User(rootUser));
      } else {
        DB_User.findByPk(id)
        .then((user) => {
          if (user) {
            resolve(new User(user.toJSON()));
          } else {
            reject(new Error('User not found'));
          }
        })
        .catch((error) => {
          reject(error);
        });
      }
    });
  },

  getUserByLogin: (login: string):Promise<User> => {···
  },

  getUsers: (ids?: number[]):Promise<User[]> => {···
  },

  createUser: (userData: IUser):Promise<User> => {···
  },

  updateUser: (userId: number, userData: IUserUpdateData):Promise<void> => {···
  },

  deleteUser: (userId: number):Promise<void> => {···
  },
};
```

Рисунок 3.12 – сервіс користувача.

Отже було створено сервіс користувача який має набір базових методів для повної реалізації CRUD-операцій та автентифікації:

getUserById(id: number): Promise - метод для отримання інформації про користувача за його унікальним ідентифікатором.

getUserByLogin(login: string): Promise - метод пошуку користувача за логіном, необхідний для процесу автентифікації.

getUsers(ids?: number[]): Promise<User[]> - метод для отримання списку всіх користувачів або вибірки користувачів за їх ідентифікаторами.

createUser(userData: IUser): Promise - метод створення нового користувача. Перед збереженням у базу даних проводиться валідація даних, хешування паролю та перевірка на унікальність логіну.

updateUser(userId: number, userData: IUserUpdateData): Promise - метод для редагування інформації про користувача. Окремий тип IUserUpdateData забезпечує безпеку операції оновлення, дозволяючи змінювати лише дозволені поля.

deleteUser(userId: number): Promise - метод для видалення користувача із системи.

Для забезпечення безпеки та швидкості розробки реалізовано механізм імітації користувача з id=-798 для тестування функціоналу без необхідності постійних звернень до бази даних під час розробки а також для надзвичайних ситуацій.

Всі методи працюють асинхронно, використовуючи Promise для забезпечення неблокуючої роботи сервера. Кожен метод містить обробку помилок за допомогою try-catch блоків та контекстно-залежні повідомлення про помилки.

Сервіс взаємодіє з базою даних через ORM Sequelize, що дозволяє абстрагуватись від конкретної реалізації бази даних. Модель User представляє собою TypeScript-клас, який відображає структуру таблиці users у базі даних та інкапсулює бізнес-логіку роботи з користувачами.

Взаємодія між контролером та сервісом відбувається за принципами інверсії залежностей: контролер залежить від абстракції сервісу, а не від його конкретної реалізації, що спрощує тестування та підтримку коду.

```
export class User {
  private _userData: IUser;

  constructor(userData: IUser) {
    this._userData = userData;
  }

  get id(): number {···
  }

  get blocked(): boolean {···
  }

  get permissions(): UserPermission[] {···
  }

  get isAdmin(): boolean {···
  }

  get isProjectManager(): boolean {···
  }

  get isDeveloper(): boolean {···
  }

  get isClient(): boolean {···
  }

  get normalizedData(): IUserAuth {···
  }

  public generateToken(password: string): string | null {···
  }

  public update(update: IUserUpdateData):Promise<void> {···
  }

  public delete(password: string):Promise<void> {···
  }

  public static hashPassword(password: string):string {···
  }
}
```

Рисунок 3.13 – модель користувача.

На рисунку 3.13 представлено модель користувача, яка є фундаментальною частиною системи авторизації та керування користувачами.

Модель користувача тісно інтегрується з сервісом користувача (див. рис. 3.12), утворюючи повноцінний шар бізнес-логіки для керування користувачами системи. Контролери використовують цю модель через сервісний шар, забезпечуючи надійне розділення відповідальності згідно з принципами MVC.

Ця модель містить різні гетери та методи, які спрощують взаємодію з користувачем. Важливим методом є generateToken, який приймає пароль, зрівнює його хеш з хешом в базі даних та повертає JWT токен, який використовується для авторизації.

Також було створено модель та сервіс для груп

```
export class UserGroup {
  private _groupData: IUserGroup;

  constructor(groupData: IUserGroup) {
    this._groupData = groupData;
  }

  get id(): number {···
  }

  get normalizedData(): IUserGroup {···
  }

  public update(update: IUserGroupUpdateData):Promise<void> {···
  }

  public delete():Promise<void> {···
  }
}
```

Рисунок 3.14 – модель групи користувачів

```
export const userGroupService = {
  getUserGroupById: (id: number):Promise<UserGroup> => {···
  },

  getUserGroups: ():Promise<UserGroup[]> => {···
  },

  createUserGroup: (groupData: IUserGroupUpdateData):Promise<UserGroup> => {···
  },

  updateUserGroup: (groupId: number, groupData: IUserGroupUpdateData):Promise<void> => {···
  },

  deleteUserGroup: (groupId: number):Promise<void> => {···
  },

  getUserGroups_UsersRel: ():Promise<UserGroup[]> => {···
  },

  createUserGroups_UsersRel: (userGroupId: number, userId: number):Promise<void> => {···
  },

  deleteUserGroups_UsersRel: (userGroupId: number, userId: number):Promise<void> => {···
  },

  getUserGroups_UserGroupsRel: ():Promise<UserGroup[]> => {···
  },

  createUserGroups_UserGroupsRel: (userGroupId: number, nestedGroupId: number):Promise<void> => {···
  },
};
```

Рисунок 3.15 – сервіс групи користувачів.

Ця модель і сервіс мають схожі методи що і для користувача, сервіс відрізняється тим що він має додаткові методи для створення зв’язків, тобто додавання користувача до групи, та додавання підгрупи до групи.

3.2.3 Роутинг та контроллери

Наступний крок це створення контролерів та роутів, які будуть передавати запит певному контролеру в залежності від URL-шляху та HTTP-методу. Реалізація роутингу базується на принципах REST API та забезпечує чітку структуру взаємодії клієнта із сервером.

```
class UsersController {
  public async login(req:Request, res:Response):Promise<void> {···
  }

  public async auth(req:Request, res:Response):Promise<void> {···
  }

  public async getUsers(_req:Request, res:Response) {···
  }

  public async createUser(req:Request, res:Response) {···
  }

  public async editUser(req:Request, res:Response) {···
  }

  public async deleteUser(req:Request, res:Response) {···
  }
}
```

Рисунок 3.16 – контроллер користувача.

На рисунку 3.16 зображено реалізацію контролера користувача, який забезпечує обробку всіх запитів, пов'язаних з користувачами системи. Контролер реалізовано як клас з асинхронними методами, кожен з яких відповідає за певну функціональність:

- login(req: Request, res: Response): Promise - метод для обробки запитів на вхід користувача в систему. Цей метод приймає облікові дані (логін та

пароль) з тіла запиту, перевіряє їх коректність через сервіс користувача та, у випадку успішної автентифікації, генерує JWT-токен, який повертається клієнту.

- auth(req: Request, res: Response): Promise - метод для перевірки автентифікації користувача. Він аналізує отриманий токен, декодує його та повертає дані поточного користувача.
- getUsers(req: Request, res: Response) - метод для отримання списку користувачів.
- createUser(req: Request, res: Response) - метод для створення нового користувача.
- editUser(req: Request, res: Response) - метод для оновлення інформації про користувача.
- deleteUser(req: Request, res: Response) - метод для видалення користувача.

```
public async deleteUser(req:Request, res:Response) {
  const user:User = req.body.reqUser;
  const userToDeleteId = +(req.params.userId || 0);

  if (userToDeleteId && !isNaN(userToDeleteId) && user.isAdmin) {
    userService.getUserById(userToDeleteId)
      .then(userToDelete => {
        userToDelete.delete(req.body.code)
          .then(() => {
            res.json('success');
          })
          .catch(() => {
            res.status(400).json(Errors.Database);
          })
      })
      .catch(() => {
        res.status(404).json(Errors.NotFound);
      });
  } else {
    res.status(400).json(Errors.NoPermission);
  }
}
```

Рисунок 3.17 – приклад методу контроллера користувача.

На рисунку 3.17 представлено приклад реалізації одного із методів контроллера. Всі методи мають схожу логіку. Контролер фокусується на:

- Обробці HTTP-запитів
- Валідації вхідних даних
- Перевірці прав доступу
- Форматуванні відповідей
- Обробці помилок

Такий розподіл відповідальності відповідає принципам чистої архітектури та забезпечує модульність системи, де кожен компонент виконує свою чітко визначену роль.

Усі методи контролера реалізовано як асинхронні (async) для забезпечення неблокуючої обробки запитів, що особливо важливо для операцій, які включають взаємодію з базою даних або зовнішніми API.

```
class UserGroupsController {
  public async getUserGroups(_req:Request, res:Response) {···
  }

  public async createUserGroup(req:Request, res:Response) {···
  }

  public async editUserGroup(req:Request, res:Response) {···
  }

  public async addUserToGroup(req:Request, res:Response) {···
  }

  public async removeUserFromGroup(req:Request, res:Response) {···
  }

  public async deleteUserGroup(req:Request, res:Response) {···
  }
}
```

Рисунок 3.18 – контроллер групи користувачів.

Цей контроллер схожий на контроллер користувача, відмінність полягає у відсутності методів авторизації та додаткових методів для додавання користувача до групи та підгрупи до групи.

```
usersRouter.get('/',
  authMiddleware,
  usersController.getUsers
);

usersRouter.post('/login',
  validateBody(['login', 'password']).notEmpty().trim(),
  checkValidation,
  usersController.login
);

usersRouter.get('/auth',
  authMiddleware,
  usersController.auth
);

usersRouter.post('/',
  authMiddleware,
  validateBody(['password']).notEmpty(),
  validateBody(['permissions']).custom((value) => /^[a-zA-Z,]*$/.test(value) || value === ''),
  checkValidation,
  usersController.createUser,
);

usersRouter.put('/:userId',
  authMiddleware,
  validateBody(['permissions']).custom((value) => /^[a-zA-Z,]*$/.test(value) || value === ''),
  checkValidation,
  usersController.editUser
);

usersRouter.delete('/:userId',
  authMiddleware,
  validateBody(['code']).notEmpty().trim(),
  checkValidation,
  usersController.deleteUser,
);
```

Рисунок 3.19 – роутер користувачів.

На рисунку 3.19 представлено реалізацію маршрутизації для користувачів у системі. Це важлива частина архітектури, яка забезпечує зв'язок між клієнтськими HTTP-запитами та методами контролера, що їх обробляють. Маршрутизація реалізована за допомогою Express Router з використанням різних проміжних обробників (middleware) для валідації і перевірки автентифікації. Тобто в кожному роуті визначено “шлях” яким має пройти запит, кінцевою точкою якого є контроллер. Кожна частинка цього шляху обробляє запит і вирішує чи пропускати його до наступного кроку, чи завершити запит, також кожна частинка може модифікувати запит.

Структура роутера груп користувачів відповідає структурі його контролера, де кожен маршрут пов'язаний з відповідним методом обробки.

```
export const authMiddleware = async (req:Request, res:Response, next:NextFunction) => {
  try {
    if (req.headers.authorization) {
      const token:string = req.headers.authorization.split(' ')[1];

      if (!token) {
        res.status(403).json(Errors.NotAuthorized);
      } else {
        const userInfo = jwt.verify(token, jwtKey);

        if (typeof userInfo != 'string' && !!userInfo.id) {
          const user = await userService.getUserById(userInfo.id);

          if ((user && !user.blocked)) {
            req.body.reqUser = user;
            next();
          } else {
            res.status(403).json(Errors.NotAuthorized);
          }
        } else {
          res.status(400).json(Errors.NotAuthorized);
        }
      }
    } else {
      res.status(403).json(Errors.NotAuthorized);
    }
  } catch(err:any) {
    console.error(err);
    res.status(403).json(Errors.NotAuthorized);
  }
}
```

Рисунок 3.20 – авторизація.

На рисунку 3.20 зображено реалізацію проміжного обробника (middleware) авторизації. Цей компонент перевіряє JWT-токени та встановлює авторизованого користувача для кожного запиту.

authMiddleware працює за таким алгоритмом:

1. Отримує токен з заголовка Authorization запиту
2. Перевіряє наявність токена і повертає код помилки 403, якщо токен відсутній
3. Верифікує JWT-токен за допомогою секретного ключа та отримує декодовану інформацію
4. Отримує актуальні дані користувача з бази даних за ідентифікатором з токена

5. Перевіряє статус користувача - якщо користувач існує і не заблокований, додає його дані до запиту і передає керування далі; інакше повертає помилку авторизації
6. Обробляє помилки в процесі верифікації, повертаючи код 403 при будь-яких проблемах

Цей підхід до авторизації забезпечує безстатусну авторизацію без необхідності зберігання сесій на сервері, своєчасне оновлення прав доступу завдяки отриманню свіжих даних користувача та надійний захист API від несанкціонованого доступу.

3.2.4 Фронтенд

Після реалізації всіх необхідних функцій бекенду починаємо робити фронтенд. Почнемо з авторизації. Оголосимо глобальне сховище (store) для того щоб зберігати там дані поточного користувача і потім відображати їх на сторінці браузера.

```
export const useAuthStore = defineStore('auth', () => {
  const user = ref<IUser>(emptyUser);
  const userLoading = ref<boolean>(false);
  const accessToken = ref<string>(String(localStorage.getItem('token') || ''));

  const isAdmin = computed<boolean>(() => user.value.permissions.includes('admin'));

  const auth = (): void => {···
  };

  const loginLoading = ref<boolean>(false);
  const login = (login: string, password: string): Promise<void> => {···
  };

  const logout = (): void => {···
  };

  return {···
  };
});
```

Рисунок 3.21 – централізоване сховище для управління станом автентифікації.

Це сховище містить поточний стан автентифікації та функції які дозволяють на нього впливати.

```
export const usersApi = {
  auth(token: string): Promise<IUser> {
    return new Promise<IUser>((resolve, reject) => {
      axios<IUserRaw>({
        method: 'GET',
        url: base + `/users/auth`,
        headers: {
          Authorization: 'Bearer ' + token,
        },
      })
        .then((res) => {
          resolve(userConverter(res.data));
        })
        .catch((err) => {
          reject(err);
        });
    });
  },

  login(login: string, password: string): Promise<string> {···
  },

  getUsers(token: string): Promise<IUser[]> {···
  },

  createUser(token: string, user: IUser): Promise<IUser> {···
  },

  editUser(token: string, user: IUserUpdateData): Promise<void> {···
  },

  deleteUser(token: string, user: IUser, code: string): Promise<void> {···
  },
};
```

Рисунок 3.22 – відокремлені функції для API-запитів.

Запити до бекенду відбуваються не напряму зі сховища, а через додаткові функції (див. Рис. 3.22) для відокремлення логіки, що допомагає що допомагає підтримувати чистоту коду.

Тепер оголосимо головний компонент, який буде початком дерева компонентів. В ньому буде відбуватись логіка відображення сторінки для користувача.

```
<script lang="ts" setup>
import DashboardLayout from '@/components/templates/dashboard.layout.vue';
import DefaultLoader from '@/components/atoms/DefaultLoader.vue';
import NotificationWidget from '@/components/organisms/NotificationWidget.vue';
import { useAuthStore } from '@/store/auth.store';
import { defineAsyncComponent } from 'vue';

const authStore = useAuthStore();

const AuthLayout = defineAsyncComponent({
  loader: () => import('@/components/templates/auth.layout.vue'),
  loadingComponent: DefaultLoader,
});

authStore.auth();
</script>

<template>
  <DefaultLoader class="loader" v-if="authStore.userLoading" />
  <template v-else>
    <DashboardLayout v-if="authStore.user.id" />
    <AuthLayout v-else="!authStore.user.id" />
  </template>
  <NotificationWidget />
</template>

<style lang="scss">···
</style>
```

Рисунок 3.23 – початок дерева компонентів фронтенду.

На рисунку 3.23 можна побачити структуру компонентів Vue.js, він складається з трьох частин:

1. Script – тут міститься “логіка” компоненту, тобто всі обчислення, робота з даними і т.д. (JS)
2. Template - визначає структуру та вміст компонента, який буде відображатися користувачу, його зовнішній вигляд. (HTML + JS)
3. Style – стилі компоненту (CSS)

В структурі можна побачити логіку автентифікації: якщо в сховищі є дані про користувача – відображаємо “Дешборд” інакше відображаємо “Авторизацію”. Авторизація в свою чергу це сторінка яка показує форму з логіном і паролем, при відправці форми виконується валідація і виклик функції login() з глобального сховища, ця функція в свою чергу робить запит на сервер, який уже налаштований. На сервері запит потрапить в маршрутизатор, який переправить його на відповідний контроллер, де

згенерується токен і відправиться назад користувачеві. Потім цей токен збережеться в сховищі на клієнтській частині буде використовуватись в подальших запитах, таких як auth() (автентифікація). Таким же чином цей запит потрапляє від клієнтської частини до серверної, де маршрутизатором він переходить через авторизацію токена і потрапляє в контроллер який повертає потрібну інформацію, яка записується в сховищі в змінну user і тоді користувачеві відобразиться не “Авторизація”, а “Дешборд”. Токен збережеться в localStorage браузера, тому при перезавантаженні сторінки знову викличеться функція auth() яка автентифікує токен і знову відобразить користувачеві “Дешборд” на якому він зможе працювати.

```
<template>
  <div class="admin-panel">
    <Transition name="sidebar" appear>
      <Sidebar class="admin-panel__sidebar" />
    </Transition>
    <main class="main">
      <router-view v-slot="{ Component }">
        <DefaultLoader v-if="pageLoading" class="main__loader" />
        <Transition v-show="!pageLoading" name="page" appear>
          <component class="page" :is="Component" @loaded="pageLoading = false" />
        </Transition>
      </router-view>
    </main>
  </div>
</template>
```

Рисунок 3.24 – структура “Дешборд”.

Це структура того що користувач буде бачити після авторизації, тут відбувається відображення всіх сторінок веб додатку, які керуються за допомогою **Vue Router.** Ця технологія допомагає відображати сторінки і переходити по них без перезавантаження сторінки. Кожна сторінка має свій компонент.

```
const routes: RouteRecordRaw[] = [
  {
    path: '/',
    name: 'home',
    component: () => import('@/components/pages/main.page.vue'),
    meta: {
      title: 'Головна',
    },
  },
  {
    path: '/account',
    name: 'account',
    component: () => import('@/components/pages/account.page.vue'),
    meta: {
      title: 'Акаунт',
    },
  },
  {
    path: '/users/',
    name: 'users',
    children: [
      {
        path: '',
        name: 'users-list',
        component: () => import('@/components/pages/users.page.vue'),
      },
      {
        path: ':id',
        name: 'user',
        component: () => import('@/components/pages/account.page.vue'),
      },
    ],
    meta: {
      title: 'Користувачі',
    },
  },
  {
    path: '/groups',
    name: 'groups',
    component: () => import('@/components/pages/groups.page.vue'),
    meta: {
      title: 'Групи',
    },
  },
```

Рисунок 3.25 – налаштування Vue Router.

На рисунку 3.25 продемонстровано налаштування маршрутизатора, тобто який компонент потрібно відобразити в залежності від того де зараз знаходиться користувач.

У нас уже є все необхідне щоб зробити сторінки для керування користувачами та керування групами, а також сторінки акаунта певного користувача. Цих сторінок буде достатньо для того щоб зручно керувати командою.

```vue
<template>
  <div class="users-page">
    <h1>Користувачі</h1>
    <div class="users">
      <div class="users__list" v-if="usersStore.users">
        <router-link v-for="user in usersStore.users" :key="user.id" class="users__user-link"
          <UserItem class="users__user" :user="user"></UserItem>
        </router-link>
      </div>
      <DefaultButton @click="togglePopup" text="Додати користувача" />
      <AddUserPopup v-model="popupActive" />
    </div>
  </div>
</template>
```

Рисунок 3.26 – структура сторінки користувачів.

```ts
export const useUsersStore = defineStore('users', () => {
  const users = ref<IUser[]>([]);
  const authStore = useAuthStore();

  const usersMap = computed<Map<number, IUser>>(() => {···
  });

  const fetchUsers = (): Promise<void> => {···
  };

  const createUser = (user: IUser): Promise<void> => {···
  };

  const editUser = (user: IUser): Promise<void> => {···
  };

  const setUserBlocked = (user: IUser, value: boolean): Promise<void> => {···
  };

  const setUserPassword = (user: IUser, password: string): Promise<void> => {···
  };

  const deleteUser = (user: IUser, code: string): Promise<void> => {···
  };

  return {···
  };
});
```

Рисунок 3.27 – сховище користувачів.

Отже так відбувається відображення користувачів. Ми отримуємо список користувачів із сервера і відображаємо їх користувачеві на певній сторінці. Також тут присутній компонент створення користувача.

```
<template>
  <div class="user-page">
    <div style="cursor: pointer" @click="router.back()">
      <img class="user-page__back" src="/back-icon.svg" alt="" />
    </div>
    <UserEditor class="user-page__user-editor" :userId="+route.params.id ||
  </div>
</template>
```

Рисунок 3.28 – структура сторінки користувача.

На цій сторінці відображається акаунт певного користувача, компонент UserEditor дозволяє редагувати, блокувати, видаляти цього користувача. Проте не всі користувачі можуть це робити, система ролей не дасть користувачеві без певної ролі видалити іншого користувача на стороні сервера, навіть якщо він натисне кнопку яка буде прихована.

```
export const useUserGroupsStore = defineStore('userGroups', () => {
  const authStore = useAuthStore();

  const groups = ref<IUserGroup[]>([]);
  const userGroup_users = ref<{ userGroupId: number; userId: number }[]>([]);
  const userGroup_nestedGroups = ref<{ userGroupId: number; nestedGroupId: number }[]>([]);

  const userGroupsMap = computed<Map<number, IUserGroup>>(() => {…
  });

  const userGroupsNestedMap = computed<Map<number, number[]>>(() => {…
  });

  const userGroupsUsersMap = computed<Map<number, number[]>>(() => {…
  });

  const fetchGroups = (): Promise<void> => {…
  };

  const createGroup = (group: IUserGroup, parent?: number): Promise<void> => {…
  };

  const editGroup = (group: IUserGroup): Promise<void> => {…
  };

  const deleteGroup = (groupId: number): Promise<void> => {…
  };

  const addUserToGroup = (userId: number, groupId: number): Promise<void> => {…
  };

  const removeUserFromGroup = (userId: number, groupId: number): Promise<void> => {…
  };

  return {…
  };
});
```

Рисунок 3.29 – сховище груп користувачів.

```
<template>
  <div class="user-groups-page">
    <h1>Групи</h1>
    <br />
    <UserGroup
      class="user-groups-page__user-group"
      v-for="group in headGroups"
      :key="group.id"
      :createButton="authStore.isAdmin"
      :editButton="authStore.isAdmin"
      :groupId="group.id"
      @edit="toggleGroupEditor"
      @createInnerGroup="openUserGroupCreator"
    />
    <DefaultButton @click="openUserGroupCreator" text="Додати групу" />

    <SidePopup v-model="addPopupActive" title="Створити групу">···
    </SidePopup>
    <SidePopup v-model="editPopupActive" title="Керування групою">···
    </SidePopup>

    <YesNoPopup v-model="deleteGroupPopupActive" @confirm="deleteGroup(activeGroup?.id || 0)" />
  </div>
</template>
```

Рисунок 3.30 – структура сторінки груп користувачів.

На цій сторінці відображаються групи зі сховища. Компонент UserGroup відображає групу, а всередині групи користувачів які знаходяться в цій групі, а також дочірні групи, які в свою чергу так само мають користувачів і підгрупи. Також на цій сторінці присутні компоненти для додавання та редагування груп, тому окремих сторінок для цього не буде.

3.3 Таблиця Доступності

Цей модуль призначений для відслідковування доступності користувачів. Він буде реалізований у вигляді таблиці, яку користувачі заповнюватимуть вручну. Це дозволить проджект менеджерам бачити коли певний розробник буде доступний а коли ні, що допоможе вирішити кому краще дати новий проект. Це дозволить мати гнучкий графік. Також можна буде переглянути скільки днів відпрацював той чи інший член команди.

Отже складемо список вимог:

- Кожен користувач може відкрити таблицю доступності
- Кожен користувач може редагувати свою доступність на майбутні дні, лише адміністратор не має обмежень. Редагування виглядає так: користувач обирає день і позначає чи доступний він в цей день а також може залишити коментар.
- Зручне редагування та перегляд будь якого часового проміжку таблиці.

3.3.1 База даних

Додамо до бази даних ще одну таблицю availabilityRecords, яка буде містити запис доступності на певний день для певного користувача, ця таблиця буде зв’язана з таблицею users:

Рисунок 3.31 – діаграма users і availabilityRecords.

3.3.2 Сервіс та контроллер

```
export const availabilityRecordService = {
  getAvailabilityRecords: (startPoint: number, endPoint: number):Promise<IAvailabilityRecord[]> => {···
  },

  getAvailabilityRecord: (userId: number, date: string):Promise<IAvailabilityRecord | null> => {···
  },

  updateAvailabilityRecord: (userId: number, date: string, record: IAvailabilityRecordUpdateData):Promise<void> => {···
  },

  createAvailabilityRecord: (userId: number, date: string, record: IAvailabilityRecordUpdateData):Promise<void> => {···
  },
};
```

Рисунок 3.32 – сервіс таблиці доступностей.

Сервіс для таблиці доступностей досить простий, він дає можливість, читати, додавати та редагувати записи доступності.

```
class AvailabilityRecordsController {

  public async setRecord(req:Request, res:Response) {···
  }

  public async getAvailabilityTable(req:Request, res:Response) {···
  }
}
```

Рисунок 3.33 – контроллер таблиці доступностей.

Контроллер має всього два методи: отримати таблицю та записати (створити або редагувати в залежності від того чи запис існує). Таблиця доступності не потребує моделі так як вона досить проста, а модель лише додасть складності.

```
availabilityRecordsRouter.post('/',
  authMiddleware,
  validateBody(['userId', 'date']).notEmpty(),
  validateBody(['userId']).isNumeric(),
  validateBody(['date']).custom((value) => /^[0-9]{4}-[0-9]{2}-[0-9]{2}$/.test(value)),
  checkValidation,
  availabilityRecordsController.setRecord
);

availabilityRecordsRouter.get('/',
  authMiddleware,
  validateQuery(['startPoint', 'endPoint']).trim().notEmpty().custom((value) => /^[0-9]{4}-[0-9]{2}-[0-9]{2}$/.test(value)),
  validateQuery(['users']).trim().custom((value) => /^[0-9,]*$/.test(value) || !value),
  checkValidation,
  availabilityRecordsController.getAvailabilityTable,
);
```

Рисунок 3.34 – роутер таблиці доступностей.

3.3.3 Фронтенд

Для таблиці доступностей як і для інших сутностей буде створено файл для API-запитів, сховище для керування станом, та компонент відображення. Компонент відображення важливо зробити зручним, щоб можна було переглядати таблицю за будь-який часовий проміжок та бачити скільки днів кожен користувач був доступний або відсутній. Також заповнення таблиці має бути зручним, щоб це не займало багато часу для користувачів. Прикладом зручної таблиці є звичайна ексель таблиця, де можна фарбувати колонки та писати в них текст, тож таблиця доступностей буде виконана у такому вигляді, можна буде вибирати декілька комірок, фарбувати їх, писати там текст, копіювати інші комірки, робити відміну останніх змін.

```
export const useAvailabilityRecordstore = defineStore('availabilityRecords', () => {
  const authStore = useAuthStore();
  const records = ref<IAvailabilityTable>([]);
  const dates = ref<string[]>([]);

  const getRecords = (···
  };

  const updateRecords = (···
  };

  return {···
  };
});
```

Рисунок 3.35 – сховище таблиці доступностей.

```
<div class="availability-table__body">
  <div class="availability-table__row availability-table__row--days">
    <div
      v-for="day in availabilityRecordstore.dates.length"
      class="availability-table__cell availability-table__cell--day"
      :class="[
        'availability-table__cell--' + ['mon', 'tues', 'wed', 'thurs', 'fri', 'sat', 'sun'][(day - 1 + new Date(tableStartDate).getUTCDay() - 1) % 7],
        new Date().setHours(0,0,0,0) == new Date(availabilityRecordstore.dates[day - 1]).setHours(0,0,0,0) && 'availability-table__cell--today',
        focusedDay == day && 'availability-table__cell--focused'
      ]"
      @click="focuseDay(day)"
      >
        {{ availabilityRecordstore.dates[day - 1].slice(8, 10) + '-' + availabilityRecordstore.dates[day - 1].slice(5, 7) }} <br/>
        {{ ['Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб', 'Нд'][(day - 1 + new Date(tableStartDate).getUTCDay() - 1) % 7] }}
    </div>
  </div>
  <div v-for="(row, i) in availabilityRecordstore.records" v-show="!usersStore.users?.find(u => u.id == row[0])?.blocked" class="availability-table__r
    <div
      v-for="(record, i) in row[1]"
      :key="row[0] + availabilityRecordstore.dates[i]"
      class="availability-table__cell availability-table__status-cell status-cell"
      :class="[ 'status-cell--' + record.status, (selection && selection.dates.has(new Date(availabilityRecordstore.dates[i]).getTime())) && selection
      @pointerdown="toggleSelection(row[0], new Date(availabilityRecordstore.dates[i]).getTime(), record.comment || '', record.status)"
      @mouseenter="selectionUnion(new Date(availabilityRecordstore.dates[i]).getTime())"
    >
      {{ record.comment || ' ' }}
    </div>
  </div>
</div>
```

Рисунок 3.36 – “тіло” таблиці доступностей.

3.4 Робота з файлами

Цей модуль присвячений роботі з файлами. Файли будуть зберігатися у файловій системі сервера. Клієнтська частина має мати можливість завантажити будь-який файл на сервер і отримати унікальне посилання на цей файл. Поки що файли будуть використовуватись лише для того щоб можна було завантажити картинку користувача, в майбутньому цей самий модуль буде використовуватись для того щоб завантажувати файли в різні проекти, чати та завдання.

Так як збереження буде відбуватись у файловій системі нам не потрібно налаштовувати базу даних. В ній ми будемо зберігати лише посилання, яке може бути будь-яким, навіть на зовнішні ресурси. Наприклад в таблиці users є поле img, в яке можна записати посилання на картинку користувача.

Завантаження файлів на сервер буде відбуватись так: користувач обирає файл для завантаження, клієнтська частина відправляє вміст цього файлу у форматі base64 на сервер, а сервер генерує унікальне посилання на файл і зберігає цей файл у файловій системі, потім при зверненні на це посилання сервер просто відправляє файл клієнтові.

```
class FileController {
  public async uploadFile(req:Request, res:Response):Promise<void> {
    try {
      const splitedFileBase64:string[] = req.body.file.split(';base64,');

      if (base64regex.test(splitedFileBase64[1])) {
        const fileName = 'f' + Date.now() + (req.body.fileName || Date.now());
        fs.writeFileSync(process.env.PUBLIC_DIR + path.sep + fileName, splitedFileBase64[1], 'base64');
        res.json({ url: fileName });
      } else {
        res.status(400).json(Errors.InvalidRequest);
      }
    } catch(err:any) {
      res.status(500).json(Errors.Unknown);
    }
  }
}
```

Рисунок 3.36 – контроллер завантаження файлів.

Контроллер (див. Рис. 3.36) виглядає досить просто, тут немає сервісу і моделі, цей контроллер виконує всю вищеописану логіку.

3.5 Клієнти

Цей модуль призначений для роботи з клієнтами. Клієнти – це теж користувачі, проте вони відокремлені для зручності. Для цього модуля не буде зроблено окремого роутера чи контролера, для нього буде використовуватись той самий бекенд що і для користувача, відмінність користувача і клієнта полягає в полі “type” в базі даних, щоб було зрозуміло чи цей користувач є клієнтом чи звичайним користувачем.

На фронтенді буде зроблена окрема сторінка, яка буде копією сторінки користувачів, але відображатиме лише користувачів з типом клієнт, а сторінка користувачів буде відображати лише користувачів з типом користувач.

3.6 Проєкти

Цей модуль присвячений роботі з проєктами, а саме: створення, редагування та видалення проєктів, дошок, колонок, завдань. Адміністратори та менеджери можуть бачити і редагувати всі проєкти і все що в них є, лише адміністратор може видалити проєкт, інші учасники можуть бачити лише видимі їм проєкти і працювати в них на виділених для них дошках.

Маємо наступний список вимог:

- Проєкт це робоче місце розробників
- Створення проєктів: адміністратор та менеджер
- Редагування проєктів та учасників проєкту: адміністратор та менеджер
- Видалення проєктів: адміністратор
- Створення дошок в проєкті
- Редагування дошок та учасників дошок: адміністратор та менеджер
- Видалення дошок : адміністратор та менеджер
- Створення/редагування/видалення колонок: адміністратор, менеджер та учасники дошки

- Створення/редагування/видалення завдань: адміністратор, менеджер та учасники дошки

3.6.1 База Даних

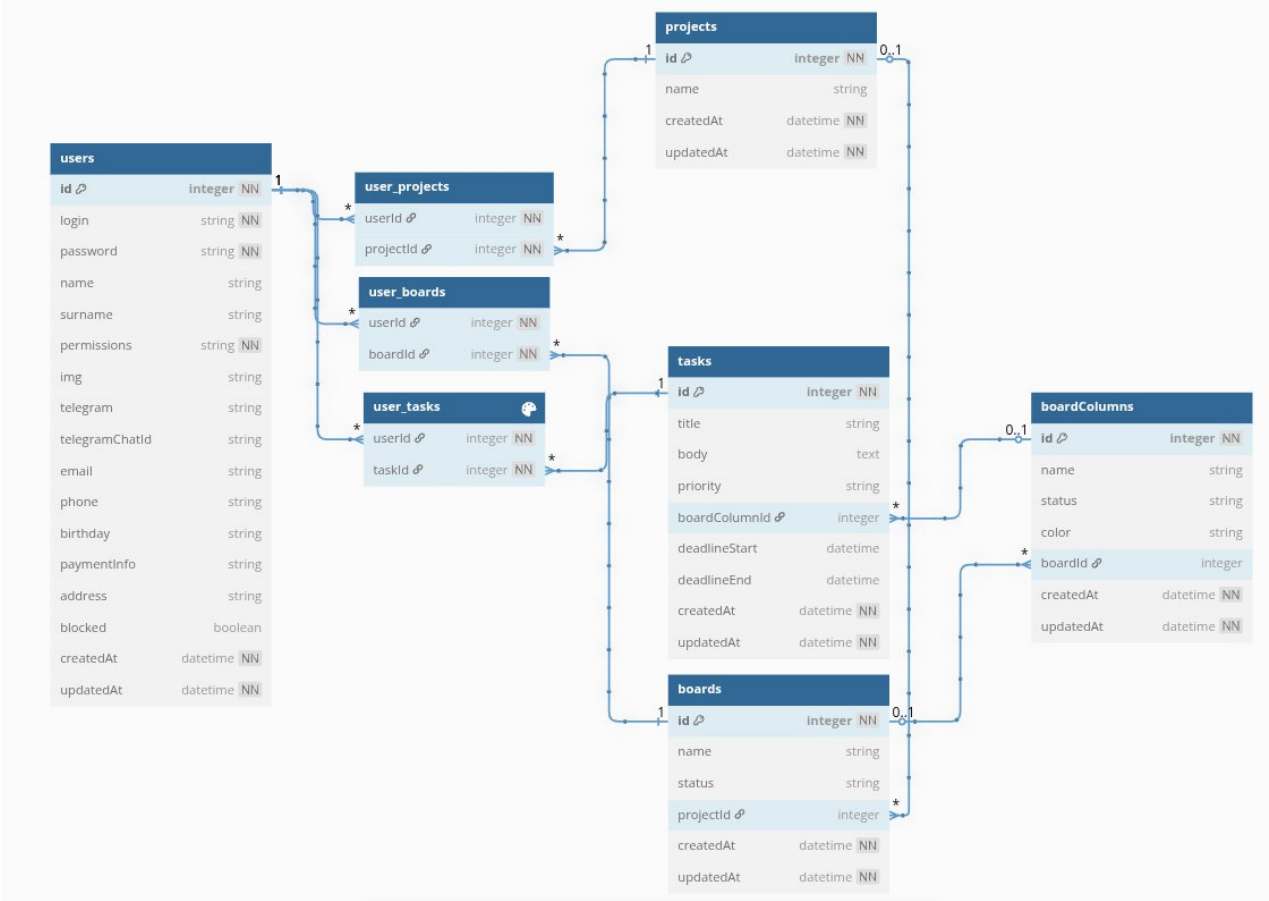


Рисунок 3.37 – структура бази даних для проєктів.

На рисунку 3.37 зображена структура бази даних, яка задовільняє вимоги. Маємо наступну ієрархію: проєкт може мати багато користувачів і багато дошок. Кожна дошка також може мати багато користувачів та колонок. Кожна колонка може мати багато завдань. Кожне завдання може мати багато користувачів.

3.6.2 Бекенд

На бекенді було створено моделі для ORM ідентично до моделей користувачів або груп. Також було створено контролери, моделі, сервіси для цих чотирьох сутностей. Всі вони зроблені ідентично до випадку з користувачами і групами. Єдина відмінність полягає в спільному роутері.

```
projectsRouter.get('/',
  authMiddleware,
  projectsController.getProjects
);

projectsRouter.post('/',
  authMiddleware,
  projectsController.createProject
);

projectsRouter.get('/:projectId',
  authMiddleware,
  projectsController.getFullProject
);

projectsRouter.put('/:projectId',
  authMiddleware,
  projectsController.editProject
);

projectsRouter.delete('/:projectId',
  authMiddleware,
  projectsController.deleteProject
);

projectsRouter.post('/:projectId/users',
  authMiddleware,
  projectsController.addUserToProject
);

projectsRouter.delete('/:projectId/users',
  authMiddleware,
  projectsController.removeUserFromProject
);

projectsRouter.post('/:projectId/boards',
  authMiddleware,
  projectBoardsController.createProjectBoard
);

projectsRouter.put('/:projectId/boards/:boardId',
  authMiddleware,
  projectBoardsController.editProjectBoard
);

projectsRouter.delete('/:projectId/boards/:boardId',
  authMiddleware,
  projectBoardsController.deleteProjectBoard
);

projectsRouter.post('/:projectId/boards/:boardId/users',
  authMiddleware,
  projectBoardsController.addUserToBoard
);

projectsRouter.delete('/:projectId/boards/:boardId/users',
  authMiddleware,
  projectBoardsController.removeUserFromBoard
);
```

Рисунок 3.38 – роутер для проєктів (частина 1)

```
projectsRouter.post('/:projectId/boards/:boardId/columns',
  authMiddleware,
  projectBoardColumnsController.createProjectBoardColumn
);

projectsRouter.put('/:projectId/boards/:boardId/columns/:columnId',
  authMiddleware,
  projectBoardColumnsController.editProjectBoardColumn
);

projectsRouter.delete('/:projectId/boards/:boardId/columns/:columnId',
  authMiddleware,
  projectBoardColumnsController.deleteProjectBoardColumn
);

projectsRouter.post('/:projectId/boards/:boardId/columns/:columnId/tasks',
  authMiddleware,
  taskController.createTask
);

projectsRouter.put('/:projectId/boards/:boardId/columns/:columnId/tasks/:taskId',
  authMiddleware,
  taskController.editTask
);

projectsRouter.delete('/:projectId/boards/:boardId/columns/:columnId/tasks/:taskId',
  authMiddleware,
  taskController.deleteTask
);

projectsRouter.post('/:projectId/boards/:boardId/columns/:columnId/tasks/:taskId/users',
  authMiddleware,
  taskController.addUserToTask
);

projectsRouter.delete('/:projectId/boards/:boardId/columns/:columnId/tasks/:taskId/users',
  authMiddleware,
  taskController.removeUserFromTask
);
```

Рисунок 3.39 – роутер для проєктів (частина 2)

На рисунках 3.38 та 3.39 зображений роутер для проєктів. В ньому присутні вкладеності, які перенаправляють на різні контролери вкладених в проект сутностей.

3.6.3 Фронтенд

Для фронтенду також буде використано ідентичний підхід як для попередніх сутностей. Для кожної сутності буде створено компонент відображення, також будуть створені компоненти для канбану та таймлайну, які будуть відображати завдання з певної дошки. Для проєкту потрібен головний компонент, який відображатиме дошки. Дошка має два відображення: канбан та таймлайн. Компонент дошки відображатиме завдання, які можна буде переміщати між колонками.

3.7 Чати

Цей модуль призначений для роботи з чатами, а саме: створення, редагування та видалення чатів, керування учасниками чатів, відправка повідомлень та отримання їх клієнтами, або надсилання повідомлення через бота Telegram.

3.7.1 База Даних

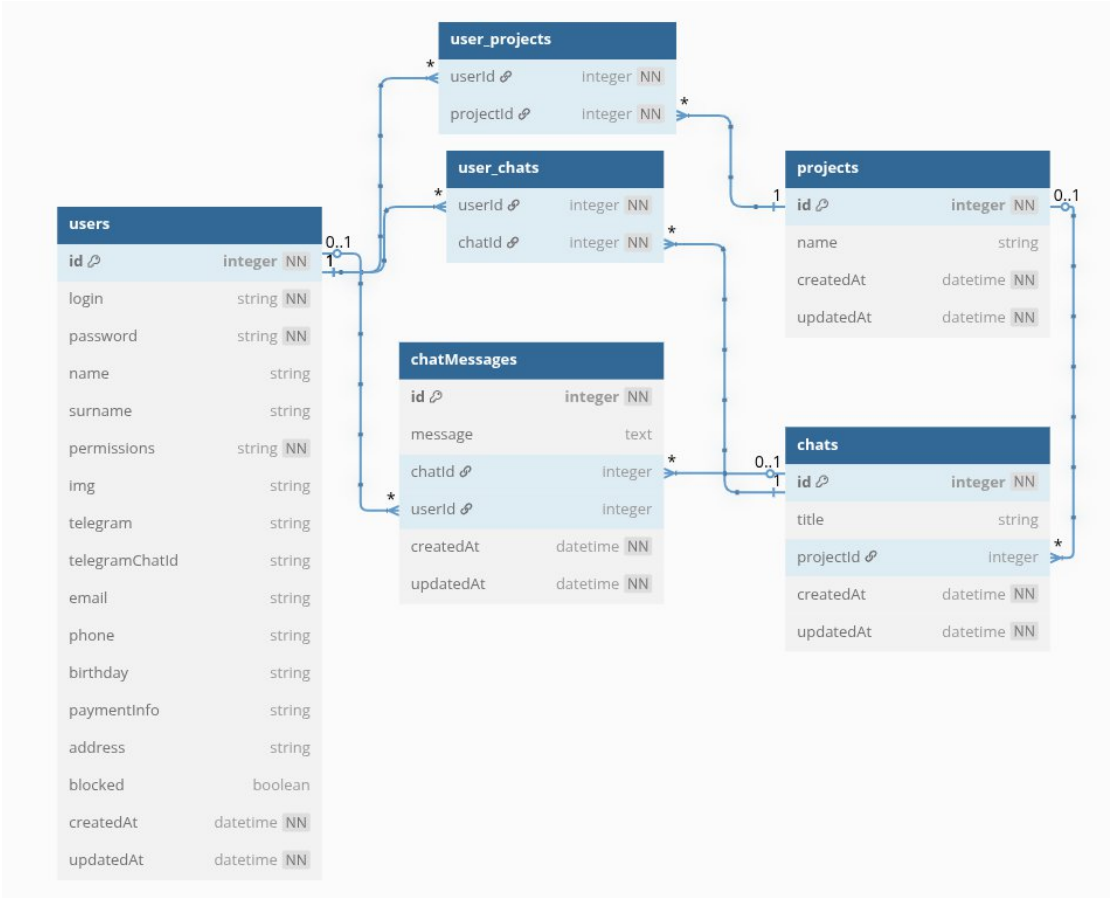


Рисунок 3.40 – структура бази даних чатів

Ця структура є розширенням структури з рисунку 3.37, до проєкту додано ще одну сутність: чат. Чат має повідомлення і багато користувачів, повідомлення прив’язане до користувача (автора).

3.7.2 Бекенд

Для чату і повідомлень так само як і для попередній сутностей були створені ORM-моделі, сервіси, моделі, контролер. Роутер проєкту був розширений додаванням до нього чатів.

```
projectsRouter.post('/:projectId/chats',
  authMiddleware,
  chatsController.createChat
);

projectsRouter.put('/:projectId/chats/:chatId',
  authMiddleware,
  chatsController.editChat
);

projectsRouter.delete('/:projectId/chats/:chatId',
  authMiddleware,
  chatsController.deleteChat
);

projectsRouter.post('/:projectId/chats/:chatId/users',
  authMiddleware,
  chatsController.addUserToChat
);

projectsRouter.delete('/:projectId/chats/:chatId/users',
  authMiddleware,
  chatsController.removeUserFromChat
);

projectsRouter.post('/:projectId/chats/:chatId/messages',
  authMiddleware,
  chatsController.createChatMessage
);

projectsRouter.get('/:projectId/chats/:chatId',
  authMiddleware,
  chatsController.getFullChat
);
```

Рисунок 3.41 – роутер проєктів з чатами

Крім звичайного створення записів у базі даних та відображенні їх на стороні клієнта важливою частиною чатів є SSE.

Server Side Events (SSE) — це технологія, яка дозволяє серверу надсилати до клієнта повідомлення в режимі реального часу. На відміну від Long- та Short-polling, клієнт та сервер одноразово встановлюють з’єднання (часто це відбувається під час ініціалізації застосунку), і клієнт підписується на оновлення з цього каналу. Після цього сервер може відправляти оновлення без зайвих запитів з боку клієнта. [21]

Для реалізації SSE було створено сервіс та контроллер з роутером.

```ts
type TStream = { send: (payload: any) => void };

export class SEE {
  private _streams:{[key:string]: TStream} = {};

  public getMessagesStreamByUserId(userId:number):TStream | undefined {
    return this._streams['user'+ userId];
  }

  public setMessagesStreamByUserId(userId:number, stream:TStream):void {
    this._streams['user'+ userId] = stream;
  }

  public removeMessagesStreamByUserId(userId:number):void {
    delete this._streams['user'+ userId];
  }
}

export const sseService = new SEE();
```

Рисунок 3.42 – сервіс SSE

```ts
class _SSEController {

  public subscribeMessagesStream(req:Request, res:Response) {
    const user:User = req.body.reqUser;

    if (user.id) {
      res.writeHead(200, {
        'Content-Type': 'text/event-stream',
        'Connection': 'keep-alive',
        'Cache-Control': 'no-cache',
        'X-Accel-Buffering': 'no'
      });

      sseService.setMessagesStreamByUserId(user.id, {
        send: (payload:any) => {
          res.write(`data: ${JSON.stringify({payload})}\n\n`);
        }
      });

      req.on('close', () => {
        sseService.removeMessagesStreamByUserId(user.id);
        res.end();
      });
    } else {
      res.status(400).json(Errors.NotAuthorized);
    }
  }
}
```

Рисунок 3.43 – контроллер SSE

```ts
generalRouter.get('/sse/subscribe',
  authMiddleware,
  SSEController.subscribeMessagesStream
);
```

Рисунок 3.44 – роутер SSE

Це (див. Рис. 3.42-3.43) дозволить нам зберігати активні підключення та відправляти на них будь-які повідомлення які будуть оброблятись клієнтською частиною.

Крім SSE потрібно щоб користувачі знали про нові повідомлення навіть коли вони не мають відкритий веб-додаток, для цього чудово підходить Telegram бот, якого можна легко налаштувати та використовувати для сповіщень про нові повідомлення. Для бота було створено сервіс.

```
bot.onText(/\/start/, (msg) => {
  const chatId = msg.chat.id;
  const userName = msg.from?.username || 'User';

  if (userName) {
    userService.getUserByTelegram(userName)
      .then(user => {
        user.update({ telegramChatId: chatId.toString() })
        bot.sendMessage(chatId, 'Welcome!');
      })
      .catch(err => {
        console.error('Error updating user with Telegram chat ID:', err);
        bot.sendMessage(chatId, 'An error occurred while linking your account.');
      })
  }
});

export const telegramService = {
  sendMessage: (chatId: string | number, message: string) => {
    return bot.sendMessage(chatId, message);
  }
};
```

Рисунок 3.45 – сервіс Telegram

Цей сервіс дозволить нам давати користувачам можливість підключитись до телеграм бота та отримувати сповіщення.

```
chat.createMessage({ message: messageContent, userId: user.id })
  .then((message) => {

    chat.users.forEach((userId) => {
      if (userId !== user.id) {
        const userStream = sseService.getMessagesStreamByUserId(userId);
        if (userStream) {
          userStream.send({
            event: 'newMessage',
            data: {
              chatId: chat.id,
              message: message,
              projectName: project.name,
              chatName: chat.name,
            },
          });
        } else {
          userService.getUserById(userId)
          .then((dbUser) => {
            if (dbUser.telegramChatID) {
              telegramService.sendMessage(dbUser.telegramChatID, `New message in chat (Project: ${project.name} Chat: ${chat.name})
            }
          })
          .catch((err) => {
            console.error(`Error sending Telegram message: ${err}`);
          });
        }
      }
    });

    res.json(message);
  })
```

Рисунок 3.46 – процес сповіщення користувачів про нове повідомлення

Такий алгоритм (див. Рис. 3.46) задовільняє наші вимоги. Якщо користувач онлайн і може прийняти SSE, тоді сервер відправляє йому повідомлення через з’єднання, якщо з’єднання немає, тобто користувач закрив веб-додаток, то сервер відправляє повідомлення в Telegram якщо в користувача він налаштований.

3.7.3 Фронтенд

Для фронтенду буде створений компонент чату, який буде відображатись в компоненті проєкту, та компонент повідомлення, який буде відображатись в компоненті чату.

Також буде створено обробник SSE, який буде створювати зв’язок з сервером та обробляти вхідні повідомлення і відображати оновлення чату без перезавантаження сторінки.

```
let messagesStream: EventSource;

const subscribeMessageStream = (): void => {
  messagesStream = new EventSource(import.meta.env.VITE_API_URL + '/general/sse/subscribe?au=' + authStore.accessToken);
  messagesStream.addEventListener('message', (message) => {
    if (message.data) {
      const data = JSON.parse(message.data);

      if (data?.payload?.event === 'newMessage' && data?.payload?.data && data.payload.data.chatId) {
        const newMessage = chatMessageConverter(data.payload.data.message);
        if (newMessage && currentChat.value && currentChat.value.id === newMessage.chatId && newMessage.userId !== authStore.user.id) {
          currentChat.value.messages = currentChat.value.messages || [];
          currentChat.value.messages.push(newMessage);
          currentChat.value = currentChat.value;
        } else if (newMessage.userId !== authStore.user.id) {
          notificationStore.success(`Нове повідомлення в чаті "${data.payload.data.chatName}" проєкту "${data.payload.data.projectName}"`);
        }
      }
    }
  });
}
```

Рисунок 3.47 – обробник SSE на клієнтській частині

ВИСНОВОК РОЗДІЛУ

У даному розділі було розглянуто практичну реалізацію веб-застосунку на основі технологічних рішень, обраних у попередньому розділі.

Серверна частина системи реалізована з використанням архітектури MVC з додатковим сервісним шаром. Контролери відповідають за обробку HTTP-запитів та взаємодію з клієнтом, сервіси містять бізнес-логіку системи, а моделі забезпечують структуровану роботу з даними. Такий підхід дозволяє ефективно розділити відповідальність між компонентами та спрощує подальше розширення функціоналу.

Реалізація REST API з використанням Express.js забезпечила створення повноцінного серверного додатку з підтримкою всіх необхідних операцій. Middleware-компоненти забезпечують автентифікацію користувачів та валідацію вхідних даних, що гарантує безпеку системи.

Використання ORM Sequelize.js значно спростило роботу з базою даних MySQL. Завдяки об'єктно-реляційному відображенню вдалося створити складну структуру з множинними зв'язками між таблицями без необхідності написання прямих SQL-запитів.

Клієнтська частина побудована на Vue.js з використанням компонентного підходу та Composition API. Кожен компонент інкапсулює власну логіку, стилі та шаблон, що забезпечує модульність коду. Централізоване управління станом через Pinia дозволяє ефективно синхронізувати дані між різними компонентами додатку.

Система авторизації на основі JWT-токенів забезпечує безстатусну автентифікацію, що є важливим для масштабування системи.

Впровадження технології Server-Sent Events для реалізації чатів дозволило створити систему обміну повідомленнями в реальному часі без

значного навантаження на сервер. Інтеграція з Telegram Bot API розширює можливості сповіщень за межі веб-додатку.

Модуль роботи з файлами реалізовано з використанням файлової системи сервера, що забезпечує ефективне зберігання файлів без додаткового навантаження на базу даних.

Використання TypeScript на всіх рівнях системи забезпечило статичну типізацію та раннє виявлення помилок під час розробки. Створені інтерфейси гарантують сумісність типів даних між клієнтською та серверною частинами.

Реалізована система повністю підтверджує ефективність обраних технологічних рішень та демонструє готовність до практичного використання для організації командної роботи в ІТ-проєктах.

РОЗДІЛ 4

ПРЕДСТАВЛЕННЯ РОБОТИ ТА ГРАФІЧНОГО ІНТЕРФЕЙСУ ПРОЄКТУ

4.1 Авторизація

Першим що буде бачити користувач при запуску веб додатку це екран авторизації. Авторизуватися і продовжити роботу він зможу тільки якщо він був зареєстрований у системі адміністратором, сам користувач виконати процедуру реєстрації не зможе.

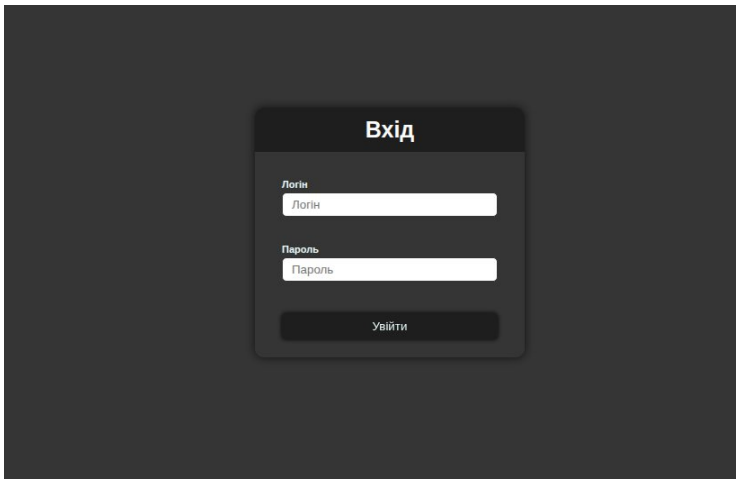


Рисунок 4.1 – екран авторизації

На рисунку 4.1 зображено екран авторизації. Користувач має ввести логін і пароль надані адміністратором.

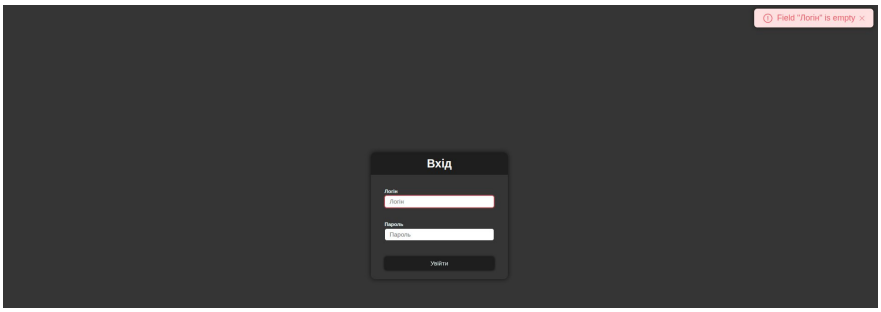


Рисунок 4.2 – валідація форми

На рисунку 4.2 зображено приклад роботи валідації. Користувач не може відправити форму авторизації поки всі поля в ній не відповідатимуть певним інструкціям. Тобто дані не будуть відправлятись на сервер і оброблятись щоб зменшити навантаження на сервер і перенести його на клієнтську частинку. Проте серверна частина також має валідацію і не буде приймати необроблювані дані.

4.2 Головна сторінка

Після введення правильного логіну і паролю користувач потрапить на головну сторінку, де він буде бачити список своїх проєктів. Він матиме змогу відкрити проєкт і почати роботу над ним, а користувачі з більшим рівнем доступу (адміністратори і менеджери) зможуть також редагувати ці проекти (змінювати назву або статус) і видаляти їх.

Рисунок 4.3 – головна сторінка

На рисунку 4.3 зображено головну сторінку, на якій відображаються проєкти, і кнопки для дій: відкрити проєкт, редагувати, створити проєкт.

Зліва (див. Рис. 4.3) можна побачити панель навігації, на ній є кнопки які відкриватимуть інші сторінки додатку. Зараз користувач знаходиться на головній тому там підсвічується іконка головної сторінки.

Для користувачів з нижчим рівнем доступу не буде відображатись кнопка щоб редагувати проєкт або додати проєкт. А також вони будуть бачити лише проєкти в яких вони є учасниками, а адміністратор і менеджер бачитимуть всі проєкти

Рисунок 4.4 – головна сторінка розробника

При натисканні кнопки додати проєкт буде відображена форма додавання проєкту, де треба ввести назву проєкту і картинку.

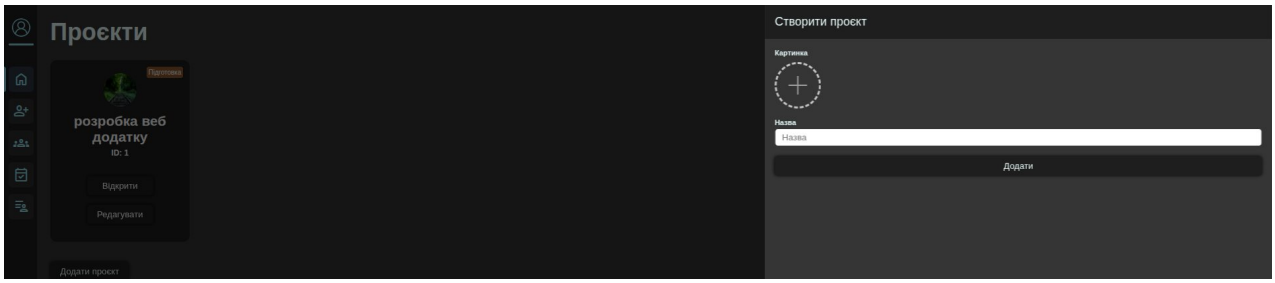


Рисунок 4.5 – створення нового проєкту

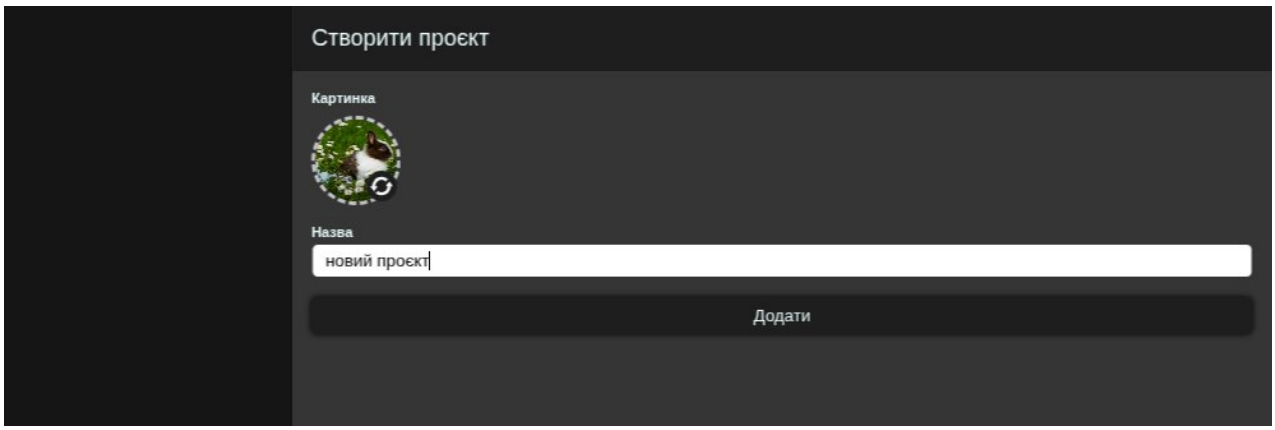


Рисунок 4.6 – заповнена форма створення нового проєкту

Рисунок 4.7 – новий проєкт створено

На рисунках 4.6, 4.6, 4.7 зображено процес створення проєкту, користувач натискає кнопку, бачить форму, заповнює її, натискає кнопку “додати” та бачить повідомлення про успіх у правому верхньому кутку екрану.

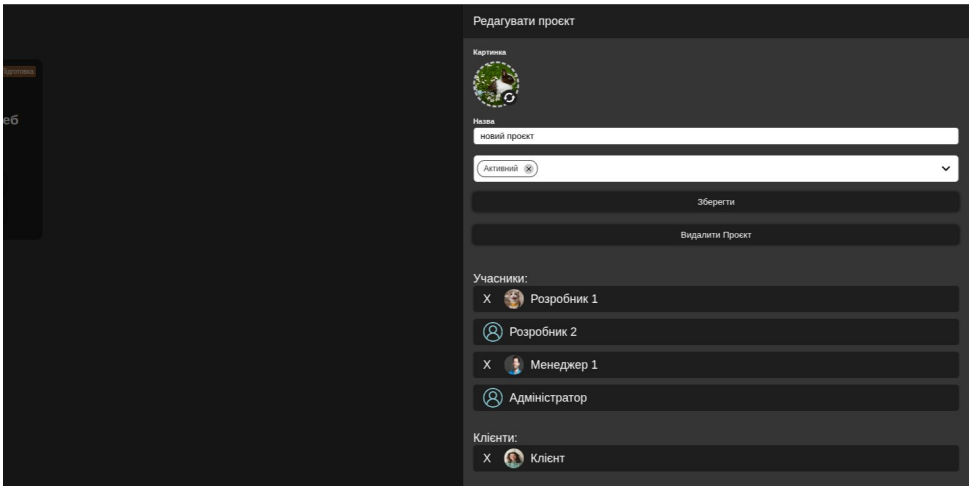


Рисунок 4.8 – редагування проєкту

На рисунку 4.8 зображено редагування проєкту. В цій форму можна змінити статус проєкту, назву, картинку, обрати користувачів які будуть учасниками цього проєкту, в тому числі і клієнта. Якщо не додати учасників, то цей проєкту буде видимий лише для адміністраторів і менеджерів. Також тут присутня кнопка видалення проєкту, яка видима лише адміністратору.

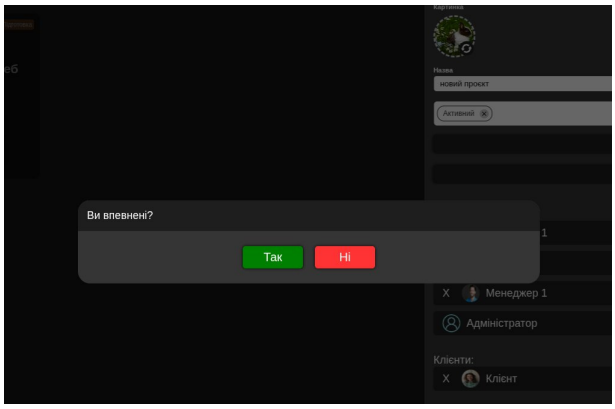


Рисунок 4.9 – видалення проєкту

На рисунку 4.9 зображено видалення проєкту, користувач має натиснути кнопку чи дійсно він впевнений чи хоче він видалити проєкт. Таке ж попередження буде відображатись при видаленні будь чого іншого, щоб уникнути випадкових видалень.

4.3 Сторінка “Користувачі”

Для керування користувачами було створено сторінку де будуть відображенні всі користувачі. Ця сторінка буде доступна тільки для адміністратора. Там можна буде створювати користувачів, відкривати їхні профілі, де можна редагувати інформацію користувача, блокувати його або видаляти.

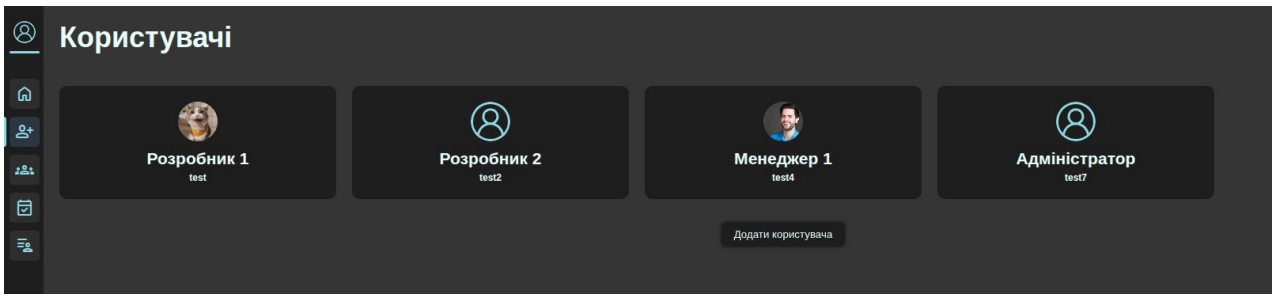


Рисунок 4.9 – сторінка “користувачі”

Рисунок 4.10 – форма додавання користувача

Рисунок 4.11 – профіль користувача

На рисунках 4.9, 4.10, 4.11 зображено екрани для керування користувачами, на них присутній весь необхідний функціонал: створення, багато різних полів, редагування, блокування, видалення, зміна паролю. Також в профілі присутня кнопка “Телеграм бот” яка відкриває бота, в якому користувач підключає його. Важливо щоб в полі телеграм був вказати правильний нікнейм з телеграму щоб бот запрацював.

Свій профіль користувача також можна відкрити натиснувши на свою іконку в лівому верхньому курку екрану. Користувач може редагувати свою інформацію незалежно від прав, чужі профілі може редагувати лише адміністратор та менеджер.

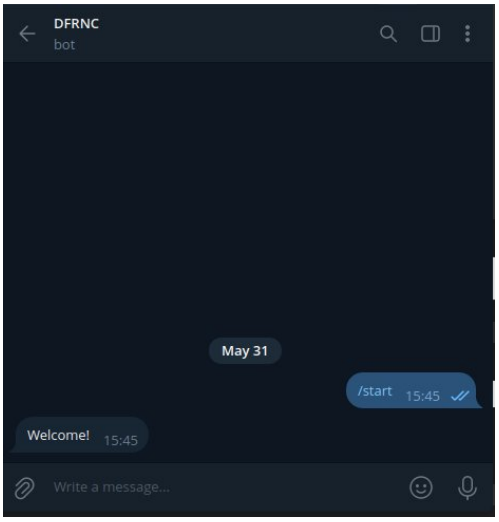


Рисунок 4.12 – підключення телеграм бота

4.4 Сторінка “Клієнти”

Для керування клієнтами створена окрема сторінка, вона ідентична до сторінки “користувачі”, адже клієнти це теж користувачі, проте для них є окрема сторінка для зручності. В клієнта також є профіль користувача і ролі.

4.5 Сторінка “Групи”

Для керування групами була створена окрема сторінка, на якій можна буде побачити всю ієрархію груп,створювати та редагувати кожну групу і її учасників.

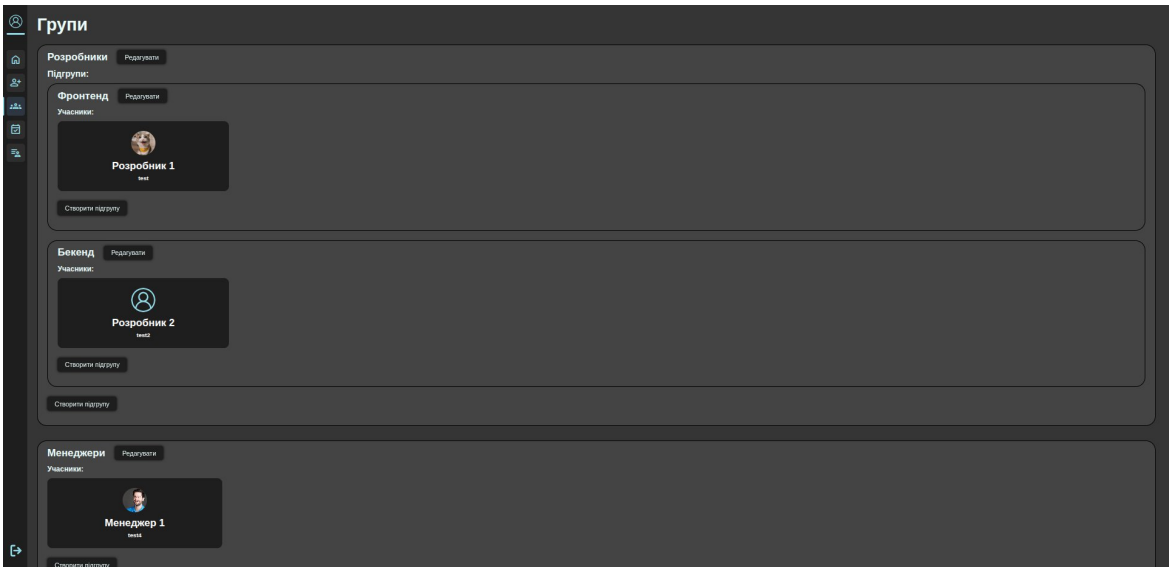


Рисунок 4.13 – сторінка “Групи”

На рисунку 4.13 зображено сторінку для керування групами. Ця сторінка доступна всім для перегляду, проте для редагування вона доступна лише адміністратору, тобто інші користувачі не бачитимуть жодної кнопки. На цій сторінці можна побачити що зараз існує дві групи: Розробники і Менеджери, група Розробники має дві підгрупи: Фронтед та Бекенд.

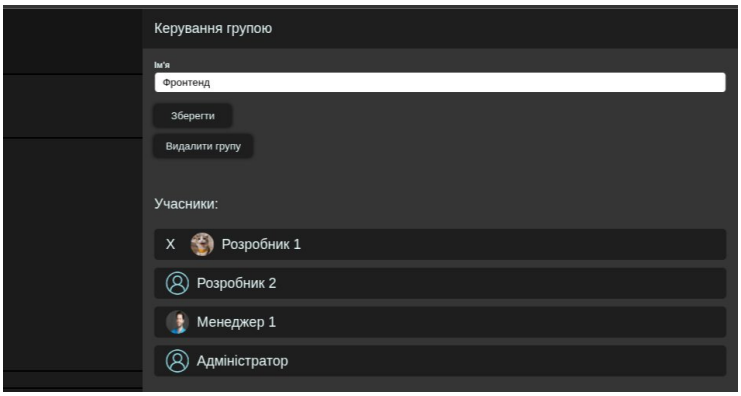


Рисунок 4.14 – редагування групи

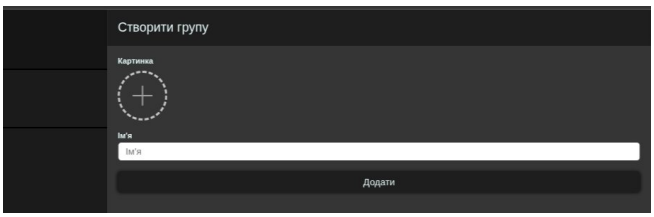


Рисунок 4.15 – створення групи

4.5 Сторінка “Доступність”

На цій сторінці буде відображатись таблиця доступності користувачів. Кожен може вказати чи буде він доступний для роботи в той чи інший день.

Рисунок 4.16 – сторінка “Доступність”

На рисунку 4.16 можна побачити таблицю доступності, на ній чітко видно коли і хто недоступний або доступний. Ця таблиця зроблена по прикладу Excel таблиці, тут можна виділяти декілька комірок і редагувати всі зразу, копіювати комірки і вставляти комірки, відміняти останні зміни. Цю сторінку можуть бачити всі користувачі, проте всі можуть редагувати лише свій рядок і лише майбутні дати, лише адміністратор і менеджер може міняти все без обмежень. Також тут можна обрати будь яку дату відображення.

4.5 Сторінка проєкту

Сторінка проєкту є основною робочою сторінкою, на ній буде відображатись канбан проєкту, де будуть відображатись завдання, також тут буде присутній таймлайн і чати. Відкрити її можуть лише адміністратори, менеджери, чи учасники або клієнти проєкту.

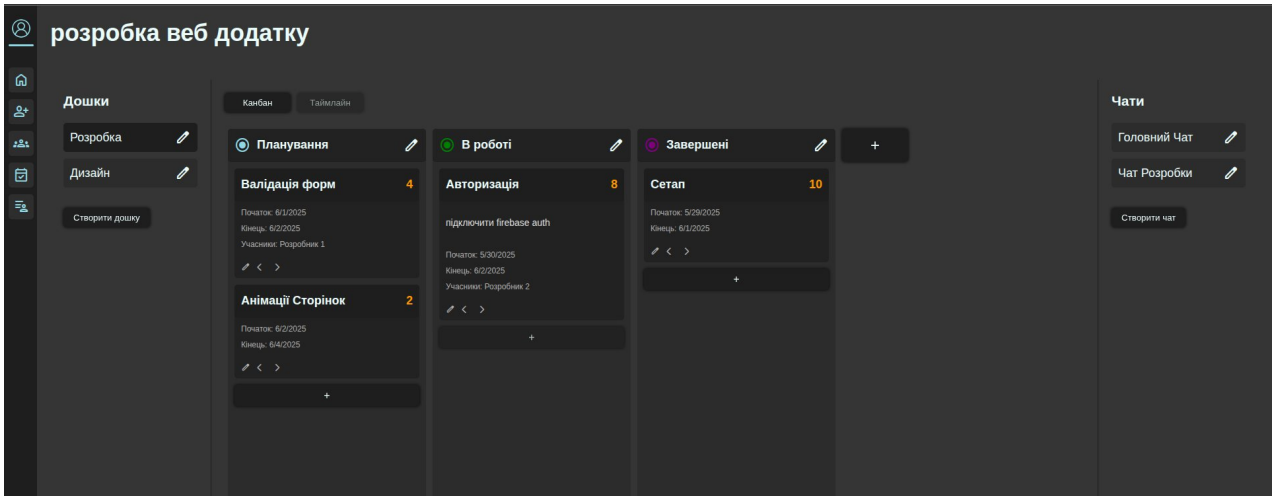


Рисунок 4.17 – сторінка проєкту

На рисунку 4.17 можна побачити сторінку проєкту. На ній можна побачити канбан, на якому відображаються завдання, які розташовані в різних колонках. Зверху відображається назва проєкту. Зліва є дошки, при натисканні на дошку відобразиться відповідний канбан цієї дошки. Для кожної дошки можна налаштувати учасників, щоб зробити індивідуальний інтерфейс для кожного і приховати певні дошки від інших користувачів. Наприклад зробити дошку з головними завданнями для всіх і точковими лише для розробників, щоб не нагромаджувати основну і не показувати клієнту лишньої інформації. Зліва відображаються чати. Чати, так само як дошки, мають своїх учасників і можуть бути приховані від когось. Посередині відображається активна область: канбан таймлайн або чат в залежності від дій користувача.

Біля дошок, чатів, завдань, чатів, колонок є кнопки редагування, редагувати чати та дошки можуть лише адміністратори чи менеджери, а от колонки і завдання – всі учасники дошки.

Дошка складається з колонок, ці колонки мають свій статус та колір, які можна налаштовувати при створенні чи редагуванні. Всі форми створення і редагування виглядають як попередні форми для користувачів чи груп. У кожній колонці є завдання, які зараз знаходяться на цьому етапі.

В карточці завдання видно назву завдання, пріоритет виконання, опис завдання, учасників завдання та час виконання. Завдання можна створити навіть пустим. Час виконання в завдання використовується для таймлайну, якщо його немає, то в таймлайні завдання не відобразиться. В цій карточці також є кнопки: редагування, перемістити в наступну колонку, перемістити в попередню колонку.

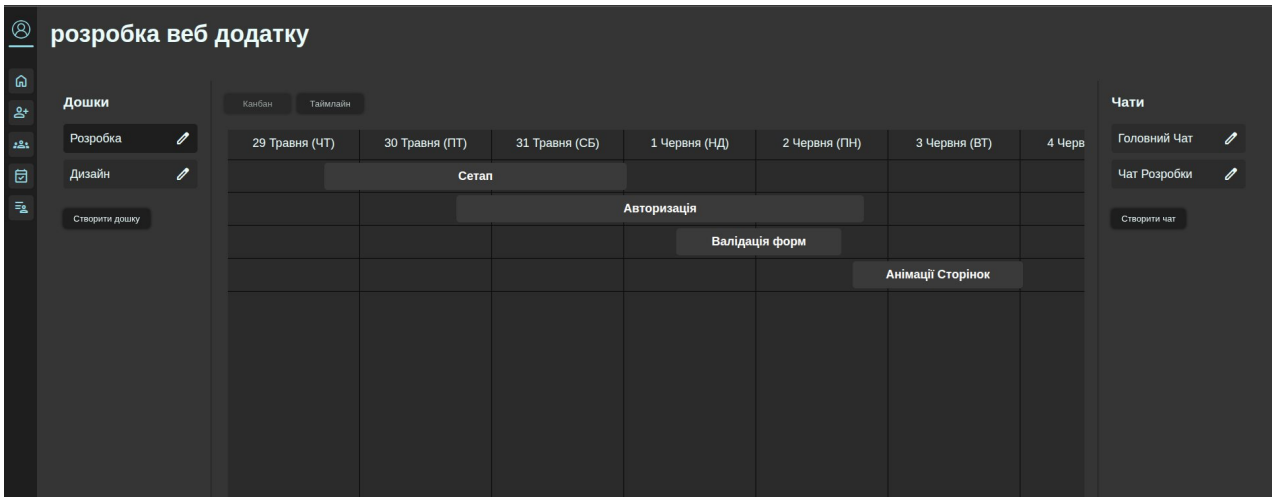


Рисунок 4.18 – таймлайн

На рисунку 4.18 зображено таймлайн, тут чітко видно коли приблизно очікувати виконання певних завдань, що є дуже зручним для менеджера чи клієнта.

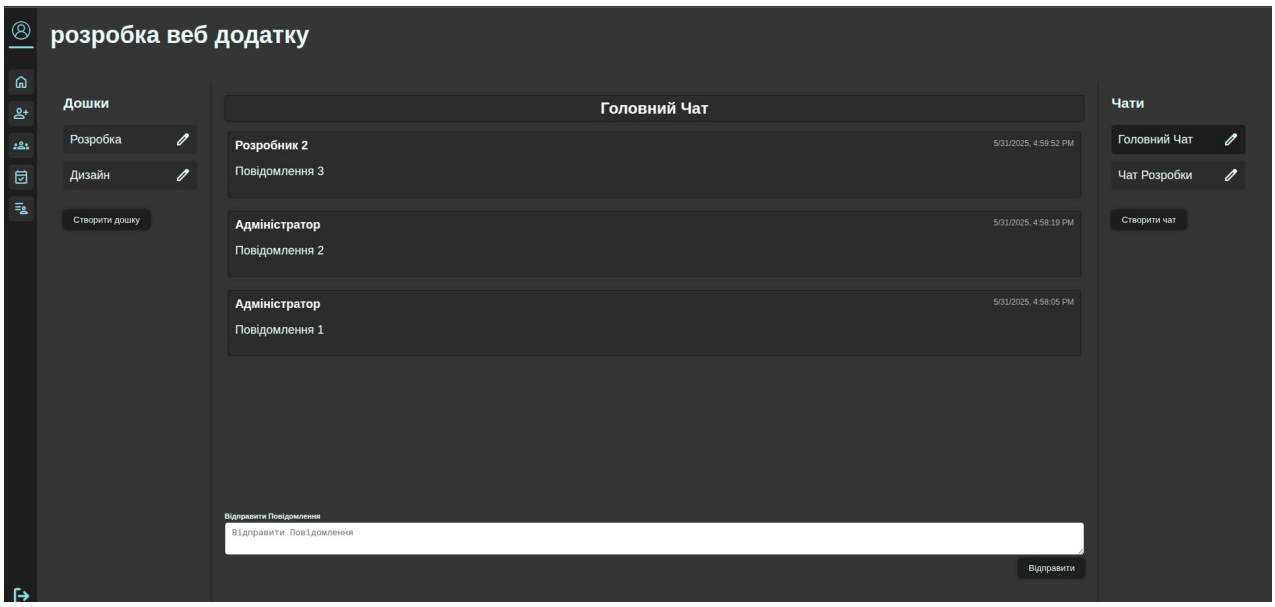


Рисунок 4.19 – чат

На рисунку 4.19 зображено чат. Знизу знаходиться поле для введення повідомлення, зверху назва чату. Посередині - список повідомлень, в кожному повідомленні є текст, автор та дата. Цей чат працює реактивно, якщо хтось відправить повідомлення воно відобразиться без перезавантаження за допомогою SSE, якщо користувач зараз не в чаті, повідомлення відобразиться страва зверху, якщо користувач закриє додаток, то повідомлення приходитимуть в телеграм.

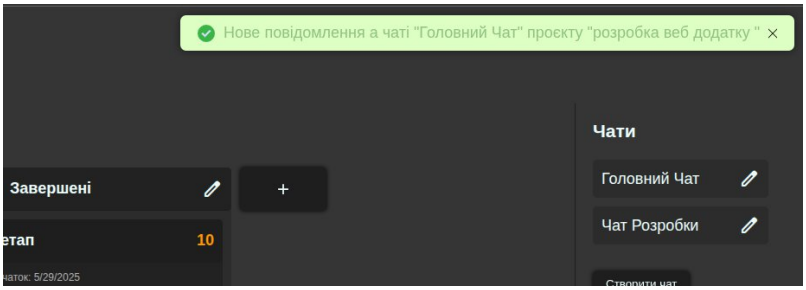


Рисунок 4.20 – нове повідомлення в чаті

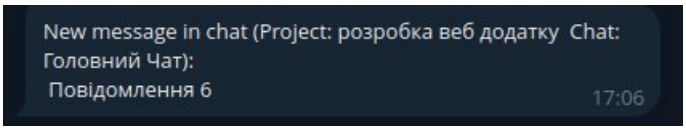


Рисунок 4.21 – нове повідомлення в чаті (телеграм)

ВИСНОВОК РОЗДІЛУ

У даному розділі було представлено графічний інтерфейс та функціональні можливості розробленого веб-застосунку.

Реалізований інтерфейс авторизації забезпечує безпечний доступ до системи. Система ролей ефективно обмежує доступ до функціоналу відповідно до прав користувача.

Головна сторінка надає зручний огляд проєктів. Розробники та клієнти бачать лише свої проєкти, тоді як адміністратори та менеджери мають доступ до всіх проєктів системи.

Модуль управління користувачами та клієнтами забезпечує повноцінне адміністрування системи з можливостями створення, редагування, блокування та видалення облікових записів. Інтеграція з Telegram-ботом розширює можливості сповіщень поза межами веб-додатку.

Система управління групами реалізована з підтримкою ієрархічної структури, що дозволяє створювати складні організаційні схеми команд з підгрупами та чіткою візуалізацією структури.

Таблиця доступності користувачів - зручний інструмент для планування робочого часу команди з інтуїтивним інтерфейсом, що нагадує Excel-таблицю, яка візуалізує зайнятість користувачів.

Сторінка проєкту є головним робочим простором, що поєднує канбан-дошки, таймлайн та чати. Гнучка система прав доступу до дошок та чатів дозволяє створювати персоналізовані робочі простори для різних учасників проєкту.

Реалізація чатів з технологією SEE забезпечує миттєвий обмін повідомленнями, а інтеграція з Telegram гарантує отримання сповіщень навіть при закритому додатку, що значно підвищує ефективність командної комунікації.

ВИСНОВКИ

У результаті виконання дипломної роботи було успішно розроблено веб-застосунок для організації та координації командної роботи, який повністю відповідає поставленим вимогам та цілям дослідження.

На основі аналізу предметної області було визначено, що існуючі інструменти мають недоліки: складність використання для малих команд (Notion, Jira) або обмежений функціонал (Trello, Worksection). На основі аналізу цих аналогів було складено вимоги до веб додатку.

Технічна реалізація системи продемонструвала ефективність прийнятих архітектурних рішень. Система авторизації забезпечила безпечний доступ з гнучкою системою ролей. Впровадження SSE дозволило створити систему комунікації в реальному часі, а інтеграція з Telegram-ботом розширила можливості сповіщень. Компонентна архітектура Vue.js з централізованим управлінням станом через Pinia забезпечила створення інтуїтивного та швидкого користувацького інтерфейсу.

Розроблений веб-застосунок повністю виконує всі поставлені завдання:

- Реалізовано гнучкі Kanban-дошки та Timeline для ефективної візуалізації прогресу проєктів з можливістю переміщення завдань між етапами та планування термінів виконання.
- Створено систему управління завданнями з повним функціоналом створення, редагування, призначення виконавців та встановлення пріоритетів.
- Впроваджено чати в проєктах з підтримкою комунікації в реальному часі та можливістю створення окремих чатів для різних учасників.
- Інтегровано Telegram-бот для отримання сповіщень про нові повідомлення та події в проєктах навіть при закритому веб-додатку.

- Розроблено таблицю доступності з Excel-подібним інтерфейсом для гнучкого планування робочого часу команди та відстеження зайнятості користувачів.
- Забезпечено систему прав доступу з чотирма ролями (адміністратор, проєкт-менеджер, розробник, клієнт) та можливістю обмеження доступу до певних проєктів і дошок.
- Реалізовано управління командою з можливістю створення ієрархічних груп, управління користувачами та клієнтами.

Створений інструмент поєднує простоту використання з необхідною функціональністю, орієнтуючись на потреби невеликих та середніх команд. Система забезпечує зручність роботи, високу швидкодію та відповідає сучасним вимогам безпеки.

Розроблений веб-застосунок ефективно вирішує поставлені завдання та може бути успішно використаний для організації командної роботи в ІТ-проєктах. Модульна архітектура системи забезпечує можливості для подальшого розвитку та додавання нового функціоналу відповідно до потреб користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Scrum [Електронний ресурс] – Режим доступу до ресурсу: https://scrumguides.org/scrum-guide.html
2. Kanban [Електронний ресурс] – Режим доступу до ресурсу: https://kanban.university/kanban-guide/
3. Notion [Електронний ресурс] – Режим доступу до ресурсу: https://www.notion.com/help/guides/category/documentation
4. Діаграма Ґанта [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%94%D1%96%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%B0_%D2%90%D0%B0%D0%BD%D1%82%D0%B0
5. Jira [Електронний ресурс] – Режим доступу до ресурсу: https://confluence.atlassian.com/jira
6. Trello [Електронний ресурс] – Режим доступу до ресурсу: https://trello.com/guide
7. WorkSection [Електронний ресурс] – Режим доступу до ресурсу: https://worksection.com/en/faq/api-start.html
8. 5 фреймворків та бібліотек для Front-end розробки [Електронний ресурс] – Режим доступу до ресурсу: https://www.digest.pro/news/5-freimvorkiv-ta-bibliotek-dlia-front-end-rozrobky/
9. Angular [Електронний ресурс] – Режим доступу до ресурсу: https://angular.dev/overview
10. Vite [Електронний ресурс] – Режим доступу до ресурсу: https://vite.dev/guide/
11. Express [Електронний ресурс] – Режим доступу до ресурсу: https://expressjs.com/en/guide/routing.html
12. Node Js [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/docs/latest/api/

13. Spring Boot [Електронний ресурс] – Режим доступу до ресурсу: https://docs.spring.io/spring-boot/documentation.html
14. TypeScript [Електронний ресурс] – Режим доступу до ресурсу: https://www.typescriptlang.org/docs/
15. MySQL [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/MySQL
16. MongoDB [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/docs/
17. MVC [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
18. Sequelize [Електронний ресурс] – Режим доступу до ресурсу: https://sequelize.org/docs/v6/getting-started/
19. Vue [Електронний ресурс] – Режим доступу до ресурсу: https://vuejs.org/guide/introduction.html
20. Pinia [Електронний ресурс] – Режим доступу до ресурсу: https://pinia.vuejs.org/introduction.html
21. SSE [Електронний ресурс] – Режим доступу до ресурсу: https://dou.ua/forums/topic/50234/

ДОДАТОК 1

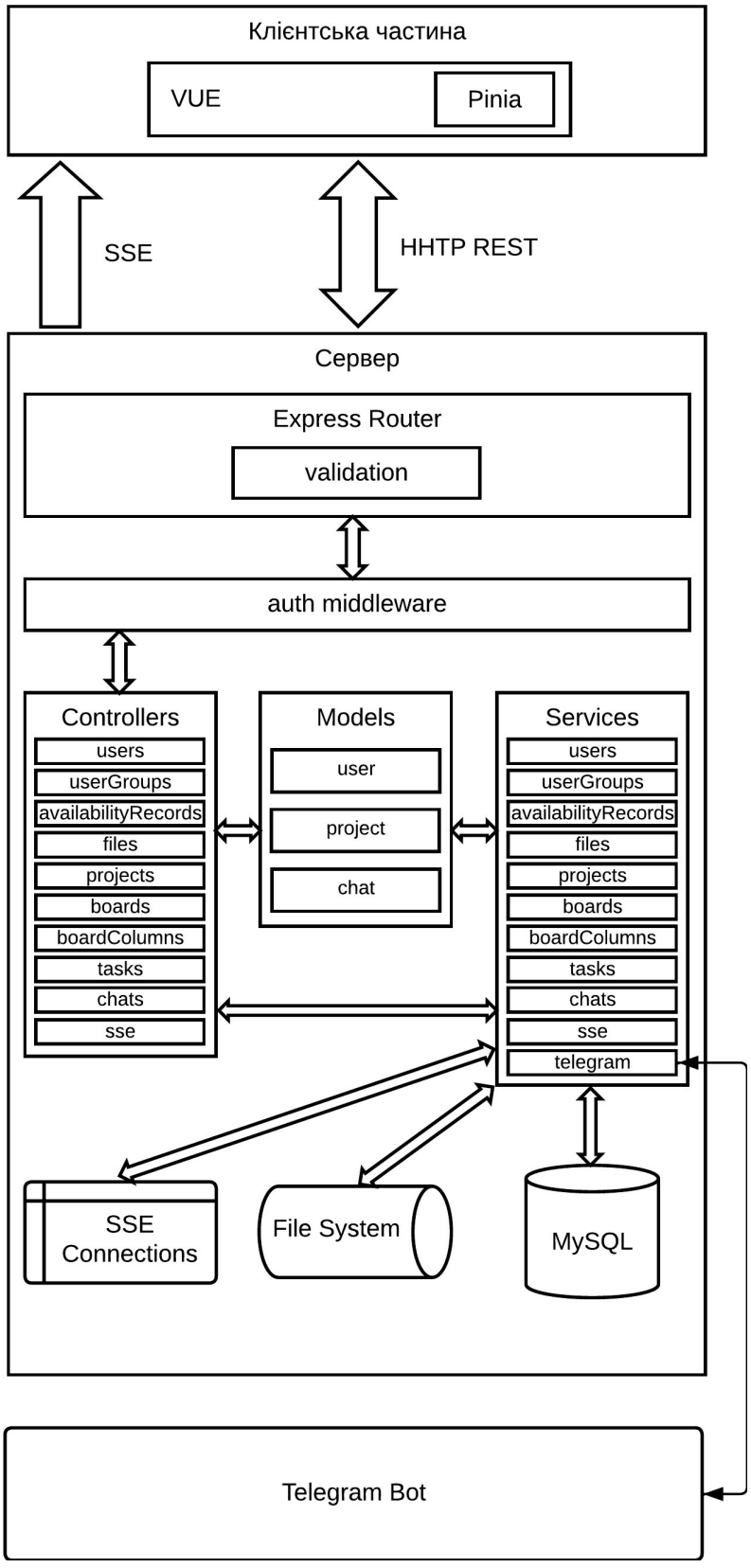
Веб-застосунок для організації та координації командної роботи

Структура системи

ІАЛЦ.466500.004 Д1

Аркушів 1

Київ 2025 р



					ІАЛЦ.466500.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Кунчій Р. О.			**Веб-застосунок для організації та координації командної роботи** Структура системи	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Валько В. В.				
						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-13
Н. Контр.						
Затвердив						

ДОДАТОК 2

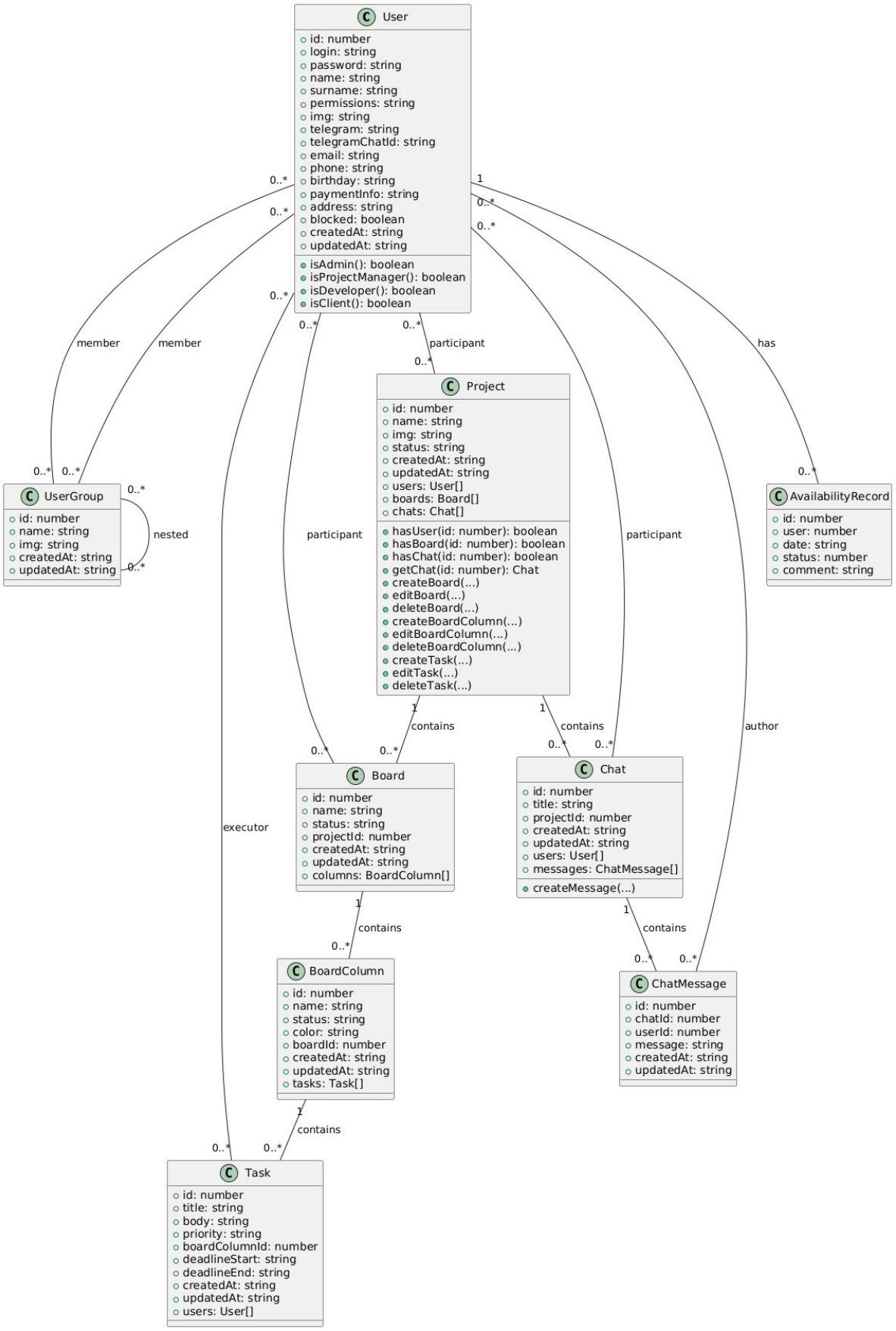
Веб-застосунок для організації та координації командної роботи

Діаграма класів

ІАЛЦ.466500.005 Д2

Аркушів 1

Київ 2025 р



					ІАЛЦ.466500.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Кунчій Р. О.			**Веб-застосунок для організації та координації командної роботи** Діаграма класів	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Валько В. В.				НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-13
Н. Контр.						
Затвердив						

ДОДАТОК 3

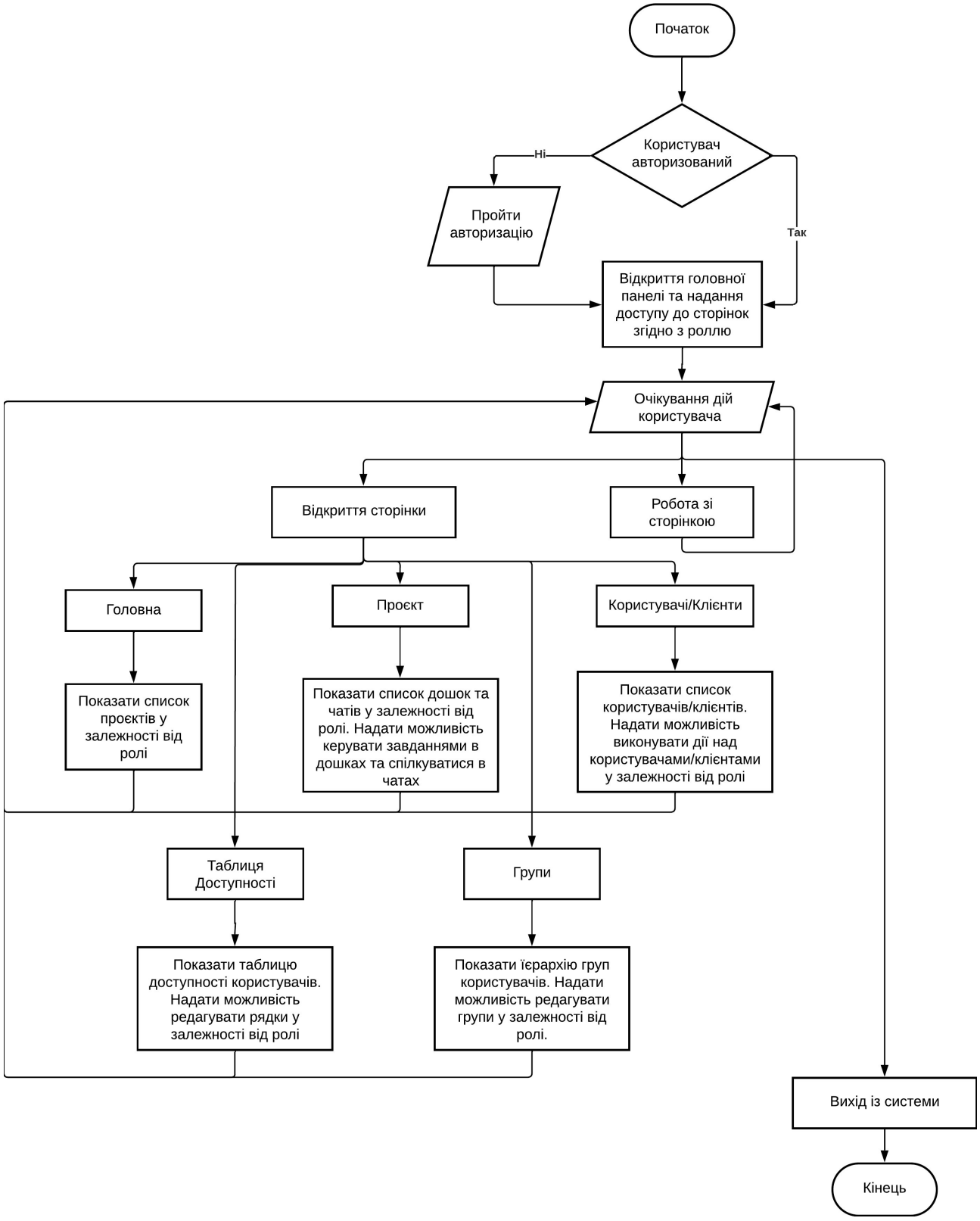
Веб-застосунок для організації та координації командної роботи

Алгоритм дій програмного забезпечення

ІАЛЦ.466500.006 ДЗ

Аркушів 1

Київ 2025 р



					ІАЛЦ.466500.006 ДЗ	
		№ докум.	Підпис	Дата		
Розробив		Кунчій Р. О.			**Веб-застосунок для організації та координації командної роботи** Алгоритм дій програмного забезпечення	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Валько В. В.				
Н. Контр.						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІО-13
Затвердив						

ДОДАТОК 4

Веб-застосунок для організації та координації командної роботи

Текст програмного коду

ІАЛЦ.466500.007 Д4

Аркушів 15

Київ 2025 р

server.ts

```
import express from 'express';
import router from './routes/router';
import cors from 'cors';
import path from 'path';
import fs from 'fs';

require('dotenv').config();
const port = process.env.PORT;

const main = async ():Promise<void> => {
  try {
    const app = express();

    if (process.env.PUBLIC_DIR && !fs.existsSync(process.env.PUBLIC_DIR)){
      fs.mkdirSync(process.env.PUBLIC_DIR, { recursive: true, mode: 0o777 });
    }

    app.use(express.json({ limit: 200_000_000 }));
    app.use('/documents', express.static(path.join(__dirname, 'public')));

    app.use(express.json());
    app.use(cors());
    app.use('/', router);
    app.listen(port, () => {
      console.log(`Success, port ${port}`)
    });
  } catch (error) {
    console.error(error);
    process.exit(1);
  }
};

void main();
```

types.ts

```
export interface IUser {
  id: number,
  login: string,
  password: string,
  name?: string | null,
  surname?: string | null,
  permissions: string,
  img?: string | null,
  contacts?: string | null,
  telegram?: string | null,
  email?: string | null,
```

```
export interface IProject {
  id: number,
  name?: string | null,
  status?: string | null,
}

export interface IKanbanBoard {
  id: number,
  projectId: number,
  name?: string | null,
  status?: string | null,
}

export interface IKanbanBoardColumn {
  id: number,
  boardId: number,
  name?: string | null,
  color?: string | null,
  status?: string | null,
}

export interface ITask {
  id: number,
  columnId: number,
  title?: string | null,
  priority?: number | null,
}

export interface IChat {
  id: number,
  projectId?: number | null,
  title?: string | null,
}



export enum Errors {
  Unknown = 'unknown',
  Exist = 'exist',
  WrongData = 'wrongData',
  InvalidRequest = 'invalidRequest',
  NotAuthorized = 'notAuthorized',
  NoPermission = 'permissionDenied',
  NotFound = 'notFound',
  Database = 'database',
  ServerError = 'serverError',
}




export enum UserPermission {
  Admin = 'admin',
  Developer = 'developer',
  ProjectManager = 'projectManager',
  Client = 'client',
}
```

router.ts

```
import express from 'express';
import { usersRouter } from './user.router';
import { filesRouter } from './files.router';
import { userGroupsRouter } from './userGroups.router';
import { availabilityRecordsRouter } from './availabilityRecords.router';
import { projectsRouter } from './projects.router';
import { generalRouter } from './general.router';
const router = express.Router();

router.use('/users', usersRouter);
router.use('/userGroups', userGroupsRouter);
router.use('/files', filesRouter);
router.use('/availabilityRecords', availabilityRecordsRouter);
router.use('/projects', projectsRouter);
router.use('/general', generalRouter);
export default router;
```

user.router.ts

```
import express from 'express';
import { body as validateBody, query as validateQuery } from 'express-validator';
const usersRouter = express.Router();

import { usersController } from './../controllers/users.controller';
import { authMiddleware } from '../middleware/auth.middleware';
import { checkValidation } from '../middleware/checkValidation.middleware';

usersRouter.get('/',
  authMiddleware,
  usersController.getUsers
);

usersRouter.post('/login',
  validateBody(['login', 'password']).notEmpty().trim(),
  checkValidation,
  usersController.login
);

usersRouter.get('/auth',
  authMiddleware,
  usersController.auth
);

usersRouter.post('/',
  authMiddleware,
  validateBody(['password']).notEmpty(),
  validateBody(['permissions']).custom((value) => /^[a-zA-Z,]*$/.test(value) || value === ''),
  checkValidation,
  usersController.createUser,
);

usersRouter.put('/:userId',
  authMiddleware,
  validateBody(['permissions']).custom((value) => /^[a-zA-Z,]*$/.test(value) || value === ''),
  checkValidation,
  usersController.editUser
);
```

```
usersRouter.delete('/:userId',
  authMiddleware,
  validateBody(['code']).notEmpty().trim(),
  checkValidation,
  usersController.deleteUser,
);

export { usersRouter };
```

files.router.ts

```
import { fileController } from './../controllers/file.controller';
import express from 'express';
import { body as validateBody, query as validateQuery } from 'express-validator';
import { authMiddleware } from '../middleware/auth.middleware';

const filesRouter = express.Router();



filesRouter.post('/',
  authMiddleware,
  validateBody(['fileName']).notEmpty(),
  validateBody(['fileName']).isLength({min: 1, max: 50}),
  fileController.uploadFile
);

export { filesRouter };
```

availabilityRecords.router.ts

```
import express from 'express';
import { body as validateBody, query as validateQuery } from 'express-validator';
import { authMiddleware } from '../middleware/auth.middleware';
import { checkValidation } from '../middleware/checkValidation.middleware';
import { availabilityRecordsController } from '../controllers/availabilityRecords.controller';


const availabilityRecordsRouter = express.Router();


availabilityRecordsRouter.post('/',
  authMiddleware,
  validateBody(['userId', 'date']).notEmpty(),
  validateBody(['userId']).isNumeric(),
  validateBody(['date']).custom((value) => /^[0-9]{4}-[0-9]{2}-[0-9]{2}$/.test(value)),
  checkValidation,
  availabilityRecordsController.setRecord
);


availabilityRecordsRouter.get('/',
  authMiddleware,
  validateQuery(['startPoint', 'endPoint']).trim().notEmpty().custom((value) => /^[0-9]{4}-[0-9]{2}-[0-9]{2}$/.test(value)),
  validateQuery(['users']).trim().custom((value) => /^[0-9,]*$/.test(value) || !value),
  checkValidation,
  availabilityRecordsController.getAvailabilityTable,
);


export { availabilityRecordsRouter };
```

auth.middleware.ts

```
require('dotenv').config();
```

```
import { Errors } from '../types';
import { Request, Response, NextFunction } from 'express';
import jwt from 'jsonwebtoken';
import { userService } from '../services/user.service';
const jwtKey:string = String(process.env.JWT_KEY);

export const authMiddleware = async (req:Request, res:Response, next:NextFunction) => {
  try {
    if (req.headers.authorization || req.query.au) {
      const token:string = req.headers?.authorization?.split(' ')?.[1] || String(req.query.au);

      if (!token) {
        res.status(403).json(Errors.NotAuthorized);
      } else {
        const userInfo = jwt.verify(token, jwtKey);

        if (typeof userInfo != 'string' && !!userInfo.id) {
          const user = await userService.getUserById(userInfo.id);

          if ((user && !user.blocked)) {
            req.body.reqUser = user;
            next();
          } else {
            res.status(403).json(Errors.NotAuthorized);
          }
        } else {
          res.status(400).json(Errors.NotAuthorized);
        }
      }
    } else {
      res.status(403).json(Errors.NotAuthorized);
    }
  } catch(err:any) {
    console.error(err);
    res.status(403).json(Errors.NotAuthorized);
  }
}
```

checkValidation.middleware.ts

```
require('dotenv').config();

import { Request, Response, NextFunction } from 'express';
import { validationResult } from 'express-validator/src/validation-result';
import { Errors } from '../types';

export const checkValidation = (req:Request, res:Response, next:NextFunction) => {
  try {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      res.status(400).json(Errors.InvalidRequest);
    } else {
      next();
    }
  } catch(err:any) {
    res.status(500).json(Errors.Unknown);
  }
}
```

users.controller.ts

```
require('dotenv').config();

import e, { Request, Response } from 'express';
import { Errors, IUser, IUserUpdateData } from '../types';
import { User } from '../models/user.model';
import { userService } from '../services/user.service';

class UsersController {
 public async login(req:Request, res:Response):Promise<void> {
  userService.getUserByLogin(req.body.login)
   .then(user => {
    const token = user.generateToken(req.body.password);

    if (token) {
     res.json({token});
    } else {
     res.status(400).json(Errors.WrongData);
    }
   })
   .catch(err => {
    console.error(err);
    res.status(404).json(Errors.NotFound);
   });
 }

 public async auth(req:Request, res:Response):Promise<void> {
  const user:User = req.body.reqUser;
  res.json(user.normalizedData);
 }

 public async getUsers(req:Request, res:Response) {
  userService.getUsers(undefined, String(req.query.type || '') === 'c' ? 'c' : 'u')
   .then((users) => {
    res.json(users.map((user) => user.normalizedData));
   })
   .catch(() => {
    res.status(400).json(Errors.Database);
   });
 }

 public async createUser(req:Request, res:Response) {
  const user:User = req.body.reqUser;

  const userToCreate:IUser = {
   id: 0,
   login: req.body.login,
   password: User.hashPassword(req.body.password),
   name: req.body.name,
   surname: req.body.surname,
   permissions: (req.body.permissions && req.body.permissions.join(',')) || '',
   img: req.body.img,
   telegram: req.body.telegram,
   email: req.body.email,
   phone: req.body.phone,
   birthday: req.body.birthday,
   paymentInfo: req.body.paymentInfo,
   address: req.body.address,
   type: req.body.type,
```

```
    createdAt: '',
    updatedAt: '',
  }

  if (user.isAdmin) {
    userService.createUser(userToCreate)
      .then((newUser) => {
        res.json(newUser.normalizedData);
      })
      .catch((err) => {
        console.error(err);
        res.status(400).json(Errors.Database);
      });
  } else {
    res.status(403).json(Errors.NoPermission);
  }
}

public async editUser(req:Request, res:Response) {
  const user:User = req.body.reqUser;
  const userToEditId = +(req.params.userId || 0);

  if (userToEditId && !isNaN(userToEditId) && (user.isAdmin || user.id == userToEditId)) {
    const newInfo:IUserUpdateData = {};
    if (req.body.name !== undefined) newInfo.name = String(req.body.name || '') || '';
    if (req.body.surname !== undefined) newInfo.surname = String(req.body.surname || '') || '';
    if (req.body.img !== undefined) newInfo.img = String(req.body.img || '') || '';
    if (req.body.telegram !== undefined) newInfo.telegram = String(req.body.telegram || '') || '';
    if (req.body.email !== undefined) newInfo.email = String(req.body.email || '') || '';
    if (req.body.phone !== undefined) newInfo.phone = String(req.body.phone || '') || '';
    if (req.body.birthday !== undefined) newInfo.birthday = String(req.body.birthday || '') || null;
    if (req.body.paymentInfo !== undefined) newInfo.paymentInfo = String(req.body.paymentInfo || '') || '';
    if (req.body.address !== undefined) newInfo.address = String(req.body.address || '') || '';

    if (user.isAdmin) {
      if (req.body.permissions !== undefined) newInfo.permissions = String(req.body.permissions || '') || '';
      if (req.body.blocked !== undefined) newInfo.blocked = Boolean(req.body.blocked);
    }

    userService.getUserById(userToEditId)
      .then(userToEdit => {
        userToEdit.update(newInfo)
          .then(() => {
            res.json(userToEdit.normalizedData);
          })
          .catch((err) => {
            console.error(err);
            res.status(400).json(Errors.Database);
          })
      })
      .catch(() => {
        res.status(404).json(Errors.NotFound);
      });

    } else {
    res.status(400).json(Errors.NoPermission);
  }
}

public async deleteUser(req:Request, res:Response) {
  const user:User = req.body.reqUser;
```

```
    const userToDeleteId = +(req.params.userId || 0);

    if (userToDeleteId && !isNaN(userToDeleteId) && user.isAdmin) {
      userService.getUserById(userToDeleteId)
        .then(userToDelete => {
          userToDelete.delete(req.body.code)
            .then(() => {
              res.json('success');
            })
            .catch(() => {
              res.status(400).json(Errors.Database);
            })
        })
        .catch(() => {
          res.status(404).json(Errors.NotFound);
        });
    } else {
      res.status(400).json(Errors.NoPermission);
    }
  }
}

export const usersController = new UsersController();
```

file.controller.ts

```
import { Request, Response } from 'express';
import { Errors } from '../types';
import fs from 'fs';
import path from 'path';

const base64regex:RegExp = /^([0-9a-zA-Z+/]{4})*(([0-9a-zA-Z+/]{2}==)|([0-9a-zA-Z+/]{3}=))?$/;

class FileController {
  public async uploadFile(req:Request, res:Response):Promise<void> {
    try {
      const splitedFileBase64:string[] = req.body.file.split(';base64,');

      if (base64regex.test(splitedFileBase64[1])) {
        const fileName = 'f' + Date.now() + (req.body.fileName || Date.now());
        fs.writeFileSync(process.env.PUBLIC_DIR + path.sep + fileName, splitedFileBase64[1], 'base64');

        console.log(splitedFileBase64[1])

        res.json({ url: fileName });
      } else {
        res.status(400).json(Errors.InvalidRequest);
      }
    } catch(err:any) {
      res.status(500).json(Errors.Unknown);
    }
  }
}

export const fileController = new FileController()
```

availabilityRecords.controller.ts

```
require('dotenv').config();
```

```
// types
import { Request, Response } from 'express';
import { IAvailabilityRecordUpdateData } from '../types';
import { Errors } from '../types';
import { User } from '../models/user.model';
import { availabilityRecordService } from '../services/availabilityRecords.service';
import { userService } from '../services/user.service';


class AvailabilityRecordsController {


  public async setRecord(req:Request, res:Response) {
    const user:User = req.body.reqUser;
    const date = new Date(new Date(req.body.date).setUTCHours(0, 0, 0, 0)).toUTCString();
    const userId = +req.body.userId;
    const status = +req.body.status;
    const comment = String(req.body.comment || '') || '';


    if (!user.isAdmin && !isNaN(userId) && !isNaN(status) && (userId != user.id || new Date(date).setUTCHours(0,0,0,0) <
new Date().setUTCHours(0,0,0,0)) ) {
      res.status(400).json(Errors.NoPermission);
    } else {
      availabilityRecordService.getAvailabilityRecord(userId, date)
      .then((record) => {
        if (record) {
          const newRecord:IAvailabilityRecordUpdateData = {};
          newRecord.status = status;


          if (comment) {
            newRecord.comment = comment;
          } if (comment === '') {
            newRecord.comment = null;
          }


          availabilityRecordService.updateAvailabilityRecord(userId, date, newRecord).then(() => {
            res.json('success');
          }).catch((err) => {
            console.error(err);
            res.status(400).json(Errors.Database);
          });


        } else {
          availabilityRecordService.createAvailabilityRecord(userId, date, { status, comment }).then(() => {
            res.json('success');
          }).catch((err) => {
            console.error(err);
            res.status(400).json(Errors.Database);
          });
        }
      })
      .catch((err) => {
        console.error(err);
        res.status(400).json(Errors.Database);
      });
    }
  }


  public async getAvailabilityTable(req:Request, res:Response) {


    const startDate:Date = new Date(req.query.startPoint as string);
```

```
    const endDate:Date = new Date(req.query.endPoint as string);

    if (startDate.getFullYear() < 2022 || startDate.getFullYear() - endDate.getFullYear() > 20) {
      res.status(400).json(Errors.Database);
    } else {
      const startPoint:number = startDate.setUTCHours(0, 0, 0, 0);
      const endPoint:number = endDate.setUTCHours(0, 0, 0, 0);
      const users:string = req.query.users as string;

      let userIds:number[] = [];

      if (users) {
        userIds = users.split(',').filter((e) => !isNaN(+e)).map((e) => +e);
      }

      availabilityRecordService.getAvailabilityRecords(startPoint, endPoint)
        .then((records) => {
          userService.getUsers(userIds, 'u').then((users) => {
            const result:[number, IAvailabilityRecordUpdateData[]][] = [];
            const dates:string[] = [];
            users.forEach(u => result.push([u.id as number, []]));

            for (let i = 0; i < (endPoint - startPoint) / 86400000 + 1; i++) {
              const dayTimestamp:number = startPoint + 86400000 * i;
              const date:Date = new Date(dayTimestamp);
              const year:number = date.getFullYear();
              const month:string = ('0' + (date.getMonth() + 1)).slice(-2);
              const day:string = ('0' + date.getDate()).slice(-2);
              dates.push(`${year}-${month}-${day}`);

              result.forEach(e => {
                const record = records.find(r => new Date(r.date).getTime() == dayTimestamp && r.user == e[0]);
                e[1].push({
                  status: record?.status,
                  comment: record?.comment
                });
              });
            }

            res.json({records: result, dates});
          }).catch(() => {
            res.status(400).json(Errors.Database);
          });
        })
        .catch(() => {
          res.status(400).json(Errors.Database);
        });
    }
  }
}

export const availabilityRecordsController = new AvailabilityRecordsController();
```

userGroups.controller.ts

```
require('dotenv').config();

import { Request, Response } from 'express';
```

```
import { Errors, IUserGroupUpdateData } from '../types';
import { userGroupService } from '../services/userGroup.service';
import { UserGroup } from '../models/userGroup.model';
import { User } from 'models/user.model';

class UserGroupsController {
  public async getUserGroups(_req:Request, res:Response) {

    let groups:UserGroup[] = [];
    let userGroup_users:any[] = [];
    let userGroup_userGroups:any[] = [];

    Promise.all([
      userGroupService.getUserGroups().then(_groups => groups = _groups),
      userGroupService.getUserGroups_UsersRel().then(_userGroup_users => userGroup_users = _userGroup_users),
      userGroupService.getUserGroups_UserGroupsRel().then(_userGroup_userGroups => userGroup_userGroups =
_userGroup_userGroups)
    ])
      .then(() => {
        res.json({
          groups: groups.map((group) => group.normalizedData),
          userGroup_users,
          userGroup_userGroups
        });
      })
      .catch(() => {
        res.status(400).json(Errors.Database);
      });
  }

  public async createUserGroup(req:Request, res:Response) {
    const user:User = req.body.reqUser;
    const parentId = req.body.parent || null;

    if (user.isAdmin) {
      const userGroupToCreate:IUserGroupUpdateData = {
        img: req.body.img,
        name: req.body.name,
      }

      userGroupService.createUserGroup(userGroupToCreate)
        .then((newUserGroup) => {

          if (parentId && parentId !== newUserGroup.id) {
            userGroupService.createUserGroups_UserGroupsRel(parentId, newUserGroup.id)
              .then(() => {
                res.json(newUserGroup.normalizedData);
              })
              .catch((err) => {
                console.error(err);
                res.status(400).json(Errors.Database);
              });
          } else {
            res.json(newUserGroup.normalizedData);
          }

        })
        .catch((err) => {
          console.error(err);
          res.status(400).json(Errors.Database);
        });
```

```
  } else {
    res.status(403).json(Errors.NoPermission);
  }
}

public async editUserGroup(req:Request, res:Response) {
  const user:User = req.body.reqUser;
  const groupId = +req.params.groupId;

  if (user.isAdmin && !isNaN(groupId)) {
    userGroupService.getUserGroupById(groupId)
      .then((userGroup) => {
        const userGroupToEdit:IUserGroupUpdateData = {
          img: req.body.img,
          name: req.body.name,
        }

        userGroup.update(userGroupToEdit)
          .then(() => {
            res.json('success');
          })
          .catch((err) => {
            console.error(err);
            res.status(400).json(Errors.Database);
          });
      })
      .catch(() => {
        res.status(404).json(Errors.NotFound);
      });
  } else {
    res.status(403).json(Errors.NoPermission);
  }
}

public async addUserToGroup(req:Request, res:Response) {
  const user:User = req.body.reqUser;
  const userId = +req.body.userId;
  const groupId = +req.params.groupId;

  if (user.isAdmin && !isNaN(groupId) && !isNaN(userId)) {
    userGroupService.createUserGroups_UsersRel(groupId, userId)
      .then(() => {
        res.json('success');
      })
      .catch((err) => {
        console.error(err);
        res.status(400).json(Errors.Database);
      });
  } else {
    res.status(403).json(Errors.NoPermission);
  }
}

public async removeUserFromGroup(req:Request, res:Response) {
  const user:User = req.body.reqUser;
  const userId = +req.body.userId;
  const groupId = +req.params.groupId;

  if (user.isAdmin && !isNaN(groupId) && !isNaN(userId)) {
    userGroupService.deleteUserGroups_UsersRel(groupId, userId)
      .then(() => {
```

```
        res.json('success');
       })
       .catch((err) => {
        console.error(err);
        res.status(400).json(Errors.Database);
       });
    } else {
     res.status(403).json(Errors.NoPermission);
    }
  }

  public async deleteUserGroup(req:Request, res:Response) {
    const user:User = req.body.reqUser;
    const groupId = +req.params.groupId;

    if (user.isAdmin && !isNaN(groupId)) {
     userGroupService.getUserGroupById(groupId)
      .then((userGroup) => {

        userGroup.delete()
         .then(() => {
          res.json('success');
         })
         .catch((err) => {
          console.error(err);
          res.status(400).json(Errors.Database);
         });
       })
       .catch(() => {
        res.status(404).json(Errors.NotFound);
       });
    } else {
     res.status(403).json(Errors.NoPermission);
    }
  }
}

export const userGroupsController = new UserGroupsController();
```

projects.controller.ts

```
require('dotenv').config();

import { Request, Response } from 'express';
import { Errors, IProjectUpdateData } from '../types';
import { projectService } from '../services/project.service';
import { Project } from 'models/project.model';
import { User } from 'models/user.model';

class ProjectsController {
  public async getProjects(req:Request, res:Response) {
    const user:User = req.body.reqUser;
    let projects:Project[] = [];

    Promise.all([
     projectService.getProjects().then(_projects => projects = _projects),
    ])
     .then(() => {
      res.json({
       projects: projects.filter((project) => {
        return user.isAdmin || user.isProjectManager || project.hasUser(user.id);
```

```
      }).map((project) => project.normalizedData)
    });
  })
  .catch(() => {
    res.status(400).json(Errors.Database);
  });
}

public async getFullProject(req:Request, res:Response) {
  const user:User = req.body.reqUser;
  const projectId = +req.params.projectId;

  projectService.getFullProjectById(projectId)
    .then((project) => {
      if (user.isAdmin || user.isProjectManager || project.hasUser(user.id)) {
        if (user.isAdmin || user.isProjectManager) {
          res.json(project.normalizedData);
        } else {
          res.json(project.getFilteredDataForUser(user.id));
        }
      } else {
        res.status(403).json(Errors.NoPermission);
      }
    })
    .catch((err) => {
      res.status(400).json(Errors.Database);
    });
}

public async createProject(req:Request, res:Response) {
  const user:User = req.body.reqUser;

  if (user.isAdmin || user.isProjectManager) {
    const projectToCreate:IProjectUpdateData = {
      img: req.body.img,
      name: req.body.name,
      status: req.body.status,
    }

    projectService.createProject(projectToCreate)
      .then((newProject) => {
        res.json(newProject.normalizedData);
      })
      .catch((err) => {
        console.error(err);
        res.status(400).json(Errors.Database);
      });
  } else {
    res.status(403).json(Errors.NoPermission);
  }
}

public async editProject(req:Request, res:Response) {
  const user:User = req.body.reqUser;
  const projectId = +req.params.projectId;

  if ((user.isAdmin || user.isProjectManager) && !isNaN(projectId)) {
    projectService.getProjectById(projectId)
      .then((project) => {
        const projectToEdit:IProjectUpdateData = {
          img: req.body.img,
```

```
        name: req.body.name,
        status: req.body.status,
       }

       project.update(projectToEdit)
        .then(() => {
         res.json('success');
        })
        .catch((err) => {
         console.error(err);
         res.status(400).json(Errors.Database);
        });
      })
      .catch(() => {
       res.status(404).json(Errors.NotFound);
      });
  } else {
    res.status(403).json(Errors.NoPermission);
  }
}

public async addUserToProject(req:Request, res:Response) {
  const user:User = req.body.reqUser;
  const userId = +req.body.userId;
  const projectId = +req.params.projectId;

  if ((user.isAdmin || user.isProjectManager) && !isNaN(projectId) && !isNaN(userId)) {
    projectService.createUser_ProjectRel(projectId, userId)
      .then(() => {
       res.json('success');
      })
      .catch((err) => {
       console.error(err);
       res.status(400).json(Errors.Database);
      });
  } else {
    res.status(403).json(Errors.NoPermission);
  }
}
```

Репозиторій (фронтенд): https://github.com/Rostyslav27/bachelor_frontend

Репозиторій (бекенд): https://github.com/Rostyslav27/bachelor_backend