**Форма № 24**

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
**Національний університет «Запорізька політехніка»**

Комп'ютерних наук і технологій
(повне найменування факультету)

Комп'ютерні системи та мережі
(повне найменування кафедри )

# Пояснювальна записка

до дипломного проєкту (роботи)
бакалаврський
(ступінь вищої освіти)

на тему РОЗРОБКА ВЕБСИСТЕМИ КЕРУВАННЯ ПРОЄКТАМИ
(назва теми)

Виконав(ла): студент(ка) _4_курсу,
групи КНТ-512сп

Спеціальності 123 Комп'ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп'ютерна інженерія

ЧЕРНОВ А.О.
(ПРІЗВИЩЕ та ініціали)

Керівник СКРУПСЬКИЙ С.Ю.
(ПРІЗВИЩЕ та ініціали)
Рецензент КОЗІНА Г. Л.
(ПРІЗВИЩЕ та ініціали)

2025

**Форма № 25**

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**Національний університет «Запорізька політехніка»**

Факультет Комп'ютерних наук і технологій

Кафедра Комп'ютерних систем та мереж

Ступінь вищої освіти бакалаврський

Спеціальність 123 Комп'ютерна інженерія

(код і найменування)

Освітня програма (спеціалізація) Комп'ютерна інженерія

(назва освітньої програми (спеціалізації)

**ЗАТВЕРДЖУЮ**

**Завідувач кафедри** КУДЕРМЕТОВ Р.К.

____________________

« 14 » квітня 2025 року

# З А В Д А Н Н Я

## НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ЧЕРНОВА Андрія Олексійовича

(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка вебсистеми керування проєктами

керівник проєкту (роботи) к.т.н., доцент, СКРУПСЬКИЙ С.Ю.,

(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від « 08 » квітня 2025 року №151

2. Строк подання студентом проєкту (роботи) 01.06.2025 р.

3. Вихідні дані до проєкту (роботи) опис предметної області, технології розробки клієнтської та серверної частини,

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Аналіз технічних вимог до системи, проєктування вебсистеми керування проєктами, реалізація вебсистеми керування проєктами, демонстрація роботи вебсистеми

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів) Слайди презентації

6. Консультанти розділів проєкту (роботи)

| Розділ | ПРІЗВИЩЕ, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | прийняв виконане завдання |
| 1-4 | СКРУПСЬКИЙ С.Ю. | | |
| Нормоконтроль | ЩЕРБАК Н.В, | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

7. Дата видачі завдання «__14__» ____квітня______2025_року.

## 1 КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів дипломного проєкту (роботи) | Строк виконання етапів проєкту ( роботи ) | Примітка |
|---|---|---|---|
| 1 | Визначення технологій розробки | до 19.04.2025 | |
| 2 | Визначення та аналіз вимог до програмного забезпечення | до 21.04.2025 | |
| 3 | Розробка специфікації вимог SRS | до 25.04.2025 | |
| 4 | Розробка діаграми варіантів використання | до 26.04.2025 | |
| 5 | Розробка діаграми класів | до 27.04.2025 | |
| 6 | Розробка серверної частини | до 14.05.2025 | |
| 7 | Розробка користувацького інтерфейсу | до 21.05.2025 | |
| 8 | Випробування комп'ютерної системи | до 23.05.2025 | |
| 9 | Оформлення пояснювальної записки | до 25.05.2025 | |
| 10 | Проходження нормоконтролю | до 01.06.2025 | |
| 11 | Перевірка на наявність академічного плагіату | до 03.06.2025 | |
| 12 | Проходження рецензування | до 09.06.2025 | |
| | | | |
| | | | |
| | | | |

**Студент(ка)** _____________ ____Андрій ЧЕРНОВ___
( підпис ) (Ім'я ПРІЗВИЩЕ)

**Керівник проєкту (роботи)** _____________ _Степан СКРУПСЬКИЙ_
( підпис ) (Ім'я ПРІЗВИЩЕ)

# РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 84 с., 4 табл., 1 додаток, 15 рис., 20 джерел.

ВЕБСИСТЕМА, КЕРУВАННЯ ПРОЄКТАМИ, NESTJS, NEXT.JS, TYPESCRIPT, PRISMA, MYSQL, AWS S3, GMAIL API, TWILIO, JWT, REST API.

У дипломній роботі розглянуто процес проєктування та реалізації сучасної вебсистеми керування проєктами, призначеної для організації командної роботи, ефективного розподілу завдань, контролю їх виконання, обміну повідомленнями та спільного доступу до файлів. Актуальність теми зумовлена зростаючими потребами команд у динамічному цифровому середовищі, де необхідні універсальні інструменти для гнучкого управління проєктами.

Для реалізації серверної частини було використано фреймворк NestJS, ORM Prisma, а також СКБД MySQL. На стороні клієнта застосовано Next.js, що забезпечило ефективну побудову SPA-інтерфейсу з підтримкою SSR. Система типізована за допомогою TypeScript.

Серед функціональних можливостей системи: реєстрація та авторизація, створення та редагування проєктів, додавання учасників, створення та переміщення завдань, коментування, фільтрація й сортування завдань.

Для безпечного доступу до ресурсів реалізовано систему ролей, аутентифікацію через JWT, контроль доступу на основі ролей (RBAC), а також захист API через middleware. Було створено REST API з повною документацією.

На етапі проєктування побудовано UML-діаграми, діаграми послідовностей, use-case схеми. Система протестована через Postman, перевірено авторизацію, виконання CRUD-операцій, роботу з файлами та відправлення повідомлень.

У результаті розробки було створено повнофункціональний вебзастосунок, який може бути використаний як внутрішній інструмент керування проєктами для IT-компаній, команд фрилансерів або стартапів.

## ABSTRACT


Explanatory note to the bachelor's thesis: 84 p., 4 tables, 1 appendix, 15 figures, 20 sources.

WEB SYSTEM, PROJECT MANAGEMENT, NESTJS, NEXT.JS, TYPESCRIPT, PRISMA, MYSQL, AWS S3, GMAIL API, TWILIO, JWT, REST API.

The thesis examines the process of designing and implementing a modern web-based project management system designed to organize teamwork, efficiently distribute tasks, monitor their implementation, exchange messages, and share files. The relevance of the topic is driven by the growing needs of teams in a dynamic digital environment, where universal tools for flexible project management are needed.

The NestJS framework, Prisma ORM, and MySQL database were used to implement the server side. Next.js was used on the client side, which ensured the efficient construction of the SPA interface with SSR support. The system is typed using TypeScript.

The system's functionality includes registration and authorization, creating and editing projects, adding participants, creating and moving tasks, commenting, filtering, and sorting tasks.

To ensure secure access to resources, we implemented a role-based system, JWT authentication, role-based access control (RBAC), and API protection through middleware. A REST API with full documentation was created.

At the design stage, we built UML diagrams, sequence diagrams, and use-case diagrams. The system was tested using Postman, authorization, CRUD operations, file handling, and message sending were checked.

The development resulted in a fully functional web application that can be used as an internal project management tool for IT companies, freelance teams, or startups.

# ЗМІСТ

Скорочення та умовні позначки ..... 7
Вступ ..... 8
1 Аналіз технічних вимог до системи ..... 9
1.1 Постановка задачі ..... 9
1.2 Аналіз предметної області ..... 10
1.3 Обґрунтування вибору технологій ..... 13
1.4 Визначення загальних та функціональних вимог ..... 17
1.5 Висновки ..... 22
2 Проєктування вебсистеми керування проєктами ..... 23
2.1 Архітектура програмного забезпечення ..... 23
2.2 Моделювання системи ..... 26
3 Реалізація вебсистеми керування проєктами ..... 33
3.1 Огляд технологічного стеку ..... 33
3.2 Опис серверної логіки ..... 39
3.3 Опис клієнтської логіки ..... 44
3.4 Інтеграції та сторонні сервіси ..... 47
4 Випробування роботи вебсистеми ..... 56
4.1 Авторизація/реєстрація ..... 56
4.2 Сторінка проєктів ..... 59
4.3 Список завдань ..... 61
4.4 Коментування та перегляд історії коментарів ..... 63
4.5 Перегляд, додавання та редагування файлів ..... 64
4.6 Чати та комунікація між учасниками ..... 66
Висновки ..... 68
Перелік джерел посилання ..... 69

# СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API – Application Programming Interface (інтерфейс прикладного програмування)

AWS – Amazon Web Services (хмарна платформа Amazon)

БД – База даних

JWT – JSON Web Token (токен для безпечної передачі даних)

OAuth – Open Authorization (відкритий протокол авторизації)

S3 – Amazon Simple Storage Service (сервіс зберігання даних від AWS)

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту)

URL – Uniform Resource Locator (уніфікований локатор ресурсу)

DTO – Data Transfer Object (об'єкт передачі даних)

GUI – Graphical User Interface (графічний інтерфейс користувача)

REST – Representational State Transfer (архітектурний стиль для API)

JWT – JSON Web Token (формат токенів авторизації)

NestJS – фреймворк для створення серверних додатків на Node.js

Next.js – фреймворк для React-додатків з SSR та SSG

CRUD – Create, Read, Update, Delete (основні операції з даними)

TLS – Transport Layer Security (протокол захищеного з’єднання)

ID – Identifier (унікальний ідентифікатор)

# ВСТУП

У сучасних умовах швидкого розвитку цифрових технологій та активного використання віддаленої форми роботи зростає потреба в ефективних інструментах для організації командної роботи та управління проєктами. У цьому контексті вебсистеми керування проєктами відіграють важливу роль, оскільки дозволяють централізовано планувати, контролювати й координувати виконання завдань у межах проєкту.

Актуальність теми зумовлена необхідністю створення доступних, гнучких та масштабованих рішень, які б задовольняли потреби як малих команд, так і великих організацій. Особливу увагу приділяють вебзастосункам, які працюють у браузері без потреби встановлення додаткового програмного забезпечення.

Метою переддипломної практики є отримання практичних навичок у розробці вебзастосунків, а також створення функціональної вебсистеми для управління проєктами, яка дозволяє створювати проєкти, призначати завдання, контролювати їх виконання та комунікувати між учасниками команди.

Завдання практики:

- ознайомлення з роботою IT-компанії/відділу;
- аналіз предметної області та потреб користувачів;
- проєктування архітектури вебсистеми;
- розробка основного функціоналу;
- проведення тестування та оцінка результатів.

Об’єктом дослідження є процес організації командної роботи над проєктами.

Предметом дослідження виступає програмне забезпечення для управління проєктами, розроблене як вебзастосунок.

# 1 АНАЛІЗ ТЕХНІЧНИХ ВИМОГ ДО СИСТЕМИ

## 1.1 Постановка задачі

У межах виконання дипломної роботи було розроблено індивідуальний проєкт – вебсистему керування проєктами. Основна мета цього застосунку – надати користувачам зручний та ефективний інструмент для планування, призначення і моніторингу завдань, а також забезпечення комунікації в рамках командної роботи над проєктами.

Система реалізується як повнофункціональний вебзастосунок зі зручною клієнтською частиною та надійним серверним API. Вона дозволяє автоматизувати ключові аспекти організації командної діяльності:

- створення проєктів із гнучкою структурою: кожен проєкт може мати власну назву, опис, дедлайни, список учасників, канали зв'язку та етапи реалізації;

- керування користувачами з підтримкою ролевої моделі;

- створення завдань із гнучкими параметрами;

- користувач може додавати кастомні поля (наприклад, пріоритет, вартість, оцінка годин), встановлювати дедлайни, додавати підзавдання;

- призначення виконавців, відповідальних осіб та перегляд завантаженості;

- відстеження статусів виконання з можливістю адаптації до методології (наприклад, Kanban або Scrum): система дозволяє налаштовувати статуси завдань (на кшталт "заплановано", "в роботі", "на перевірці", "завершено") під кожен проєкт окремо;

- обговорення завдань у вигляді коментарів у режимі реального часу;

- прикріплення файлів до завдань – документи, зображення, технічні специфікації тощо;

- пошук, фільтрація та сортування завдань за різними критеріями: статус, виконавець, дедлайн, пріоритет тощо;

- автентифікація та авторизація з використанням JWT (access/refresh токени) та підтримкою входу через сторонні сервіси (OAuth 2.0, наприклад Google);

- електронні сповіщення про призначення нових завдань або зміну статусу – за допомогою підключення Google Mail API;

- ініціювання відео та аудіо зустрічей у рамках проєктів, інтегроване через Twilio;

- адаптивний інтерфейс, який працює на десктопах і мобільних пристроях, що дозволяє зручно взаємодіяти з системою у будь-яких умовах.

Серед ролей передбачені: адміністратор, який має повний доступ до всіх проєктів і користувачів, менеджер проєкту, який керує власними проєктами, учасник проєкту, який має доступ лише до призначених йому проєктів і завдань, переглядач, який має доступ лише для перегляду, гість, який має обмежений тимчасовий доступ за запрошенням до базового функціоналу такого як чати, коментарі, перегляд задач).

Розробка велась з дотриманням сучасних підходів до фронтенду й бекенду: зокрема, повна типізація всіх частин системи за допомогою TypeScript, архітектура на основі NestJS для серверної логіки та Next.js + React для клієнтської частини. Створено надійний REST API з чітким розмежуванням прав доступу, а також використано Prisma ORM для зручної та безпечної роботи з базою даних MySQL.

Уся система була реалізована мною самостійно, з урахуванням практик розробки, які використовуються в індустрії. Проєкт виконує роль не лише навчального завдання, а й практично застосовного рішення, яке може бути використане в малих командах або як основа для розширеної системи управління проєктами в компаніях.

## 1.2 Аналіз предметної області

У сучасному цифровому середовищі управління проєктами є однією з ключових дисциплін для досягнення цілей організацій та команд. Особливо актуальним є впровадження вебзасобів для керування завданнями, планування,

контролю термінів і комунікації в рамках командної роботи. Цей розділ містить аналіз концепцій управління проєктами, проблем, які вирішує автоматизація, приклади популярних рішень, а також опис технологій, що застосовуються для створення подібних систем.

### 1.2.1 Управління проєктами: сутність та основні принципи

Управління проєктами – це процес планування, організації, мотивації та контролю ресурсів для досягнення конкретних цілей і задач в межах визначених обмежень, таких як час, бюджет і обсяг робіт.

Основними принципами управління проєктами є:

- чітке визначення цілей проєкту – цілі мають бути конкретними, вимірюваними, досяжними, релевантними та обмеженими в часі (SMART);
- планування ресурсів і завдань – поділ роботи на етапи, визначення залежностей, оцінка тривалості завдань;
- управління командою – призначення відповідальних осіб, організація комунікації, підтримка мотивації;
- контроль і моніторинг – відстеження ходу виконання, коригування планів, виявлення відхилень;
- завершення проєкту – оцінка результатів, підбиття підсумків, оформлення звітної документації.

У малих командах ці процеси часто виконуються вручну або за допомогою непрофільних інструментів, однак зі зростанням обсягів робіт зростає потреба у спеціалізованому програмному забезпеченні.

### 1.2.2 Проблеми ручного керування завданнями та їх автоматизація

Управління проєктами без використання спеціалізованих інструментів часто супроводжується численними проблемами, особливо при роботі в команді або над масштабними завданнями. Ручне керування, наприклад, за допомогою електронних таблиць чи месенджерів, може бути достатнім лише на початкових етапах, але з ростом обсягів роботи воно стає неефективним.

Серед основних проблем ручного керування завданнями можна виділити:

- відсутність централізованого доступу до актуальної інформації про проєкт;

- складність у відстеженні стану завдань та термінів виконання;
- труднощі з розподілом відповідальності між учасниками команди;
- підвищена ймовірність дублювання завдань або втрати даних;
- складність у веденні історії змін і фіксації результатів роботи.

Автоматизація управління за допомогою вебсистем дозволяє:

- зберігати всі дані про проєкти, завдання та учасників у єдиному цифровому середовищі;
- забезпечити прозору та керовану взаємодію між учасниками команди;
- підвищити ефективність планування та моніторингу;
- зменшити кількість помилок, пов’язаних з людським фактором;
- прискорити адаптацію нових учасників до робочого процесу.

Таким чином, впровадження автоматизованих систем управління проєктами сприяє покращенню організації командної роботи, підвищенню продуктивності та досягненню проєктних цілей у встановлені терміни.

### 1.2.3 Огляд існуючих систем управління проєктами

На ринку представлено багато готових рішень для управління проєктами, які мають різні функціональні можливості та орієнтовані на команди різного розміру. Більшість таких систем працюють як вебзастосунки, що забезпечує доступність з будь-якого пристрою та не вимагає встановлення додаткового програмного забезпечення.

Серед найбільш відомих систем управління проєктами можна виділити: Trello, Jira, Asana, ClickUp, Monday.com.

Trello – проста та візуально зручна система, побудована за принципом канбан-дошки. Користувачі можуть створювати дошки, списки та картки, які представляють завдання. Trello підходить для невеликих команд або персонального користування.

Jira – потужна платформа від компанії Atlassian, орієнтована на розробників програмного забезпечення. Підтримує роботу за методологіями Scrum і Kanban, має гнучкі налаштування, можливість створення кастомних робочих процесів, розширену систему звітності.

Asana – сервіс для командної роботи, який дозволяє створювати проєкти, ділити їх на завдання та підзавдання, призначати виконавців, слідкувати за дедлайнами. Відзначається зручним інтерфейсом і великою кількістю інтеграцій з іншими сервісами.

ClickUp – універсальний інструмент для управління завданнями, що поєднує функціональність Trello, Jira та Asana. Дає змогу користувачам налаштовувати вигляд завдань, створювати документи, використовувати тайм-трекінг.

Monday.com – платформа з широкими можливостями для візуалізації проєктів, побудови автоматизацій та інтеграцій. Активно використовується у великих організаціях.

Кожна з вищезазначених систем має свої переваги та недоліки, а також певний рівень складності освоєння. Вибір конкретного рішення залежить від потреб команди, бюджету та специфіки проєктної діяльності.

Проте для багатьох команд, особливо тих, які мають нестандартні вимоги або прагнуть гнучкості у розширенні функціоналу, доцільним є створення власної вебсистеми управління проєктами.

## 1.3 Обґрунтування вибору технологій

Для розробки вебсистеми управління проєктами було обрано сучасний технологічний стек, що поєднує в собі надійність, масштабованість, типобезпеку та зручність розробки. Проєкт реалізується як класичний вебзастосунок із поділом на фронтенд і бекенд, що взаємодіють через REST API. У цьому підрозділі наведено докладний опис використаних технологій.

### 1.3.1 Backend: NestJS + Prisma + MySQL

Основою серверної частини є NestJS [1]– прогресивний фреймворк для Node.js, побудований на базі TypeScript та архітектурних принципів Angular. NestJS підтримує модульну структуру, що спрощує масштабування проєкту та дозволяє

легко керувати залежностями. Крім того, фреймворк має вбудовану підтримку middleware, декораторів, фільтрів, guard'ів та інших інструментів для створення безпечного й контрольованого API.

Для взаємодії з базою даних використовується Prisma ORM [2], що надає зручний і типобезпечний спосіб роботи з SQL-базами. У проєкті використано MySQL[3] як систему керування реляційною базою даних. Prisma дозволяє генерувати типізовані клієнти на основі схеми, що значно спрощує доступ до даних, зменшує кількість помилок на етапі компіляції та пришвидшує розробку.

Бекенд реалізує REST API, через яке здійснюється обмін даними з клієнтською частиною. Усі маршрути API типізовані та організовані відповідно до принципів чистої архітектури.

### 1.3.2 Frontend: Next.js + React + Tailwind CSS

Клієнтська частина реалізована за допомогою Next.js[4] – фреймворку на основі React[5], що підтримує серверний рендеринг, статичну генерацію сторінок і оптимізовану маршрутизацію. Це дозволяє досягти високої швидкодії, SEO-оптимізації та кращого UX.

UI-логіка реалізована за допомогою React, що забезпечує декларативний підхід до побудови інтерфейсів. Для стилізації використовується Tailwind CSS – утилітарний CSS-фреймворк, який дозволяє швидко створювати адаптивні та сучасні дизайни без потреби писати окремі стилі [6].

Код на фронтенді повністю написаний з використанням TypeScript[7], що гарантує статичну перевірку типів, підвищену надійність і зручність рефакторингу.

### 1.3.3 Авторизація та безпека

Для контролю доступу використовується JWT (JSON Web Token) [8] – сучасний механізм токен-авторизації, який дозволяє захищати маршрути та перевіряти особу користувача без потреби в постійній перевірці сесій. Після автентифікації користувач отримує токен, який додається до заголовків усіх подальших запитів.

Після автентифікації користувача (через email/пароль або OAuth) сервер генерує два токени access та refresh.

Access token – короткочасний токен (зазвичай з терміном дії 5–15 хвилин), який клієнт додає до заголовків HTTP-запитів (Authorization: Bearer <token>). Він містить інформацію про користувача (наприклад, ID, роль) і використовується для доступу до захищених маршрутів API.

Refresh token – довготривалий токен (може діяти дні чи тижні), який зберігається на клієнті та використовується для автоматичного отримання нового access token, коли той протерміновується.

Цей підхід дає змогу уникнути повторної авторизації користувача при кожному запиті або після короткочасного завершення сесії. Коли access token стає недійсним (через термін дії), клієнт надсилає refresh token на окремий endpoint (/auth/refresh) і отримує новий access token, без участі користувача.

Переваги схеми access/refresh:

- безпека – access token має короткий термін дії, що мінімізує ризики при його викраденні. Навіть якщо токен потрапить до зловмисника, він швидко втратить чинність;

- зручність для користувача – забезпечується безперервна сесія без необхідності повторно вводити логін/пароль;

- контроль і відкликання – refresh токени можуть зберігатися у базі даних і мати механізми відкликання, що дозволяє адміністраторам обмежити доступ у разі виявлення підозрілої активності;

- масштабованість – токени не зберігаються на сервері в сесії, тому систему легко масштабувати горизонтально без синхронізації стану між інстансами.

У системі керування проєктами цей механізм реалізовано відповідно до сучасних стандартів безпеки. Refresh токен надсилається тільки через httpOnly cookie (не доступний через JavaScript), що захищає його від XSS-атак. Access токен використовується для всіх операцій з API, таких як створення/редагування завдань, робота з файлами, участь у дзвінках тощо.

**1.3.4 Інтеграції та сервіси**

У проєкті планується інтеграція кількох сторонніх сервісів, а саме: Gmail API, AWS S3, Twilio API, OAuth.

Gmail API (Google Mail Service) [9] – для надсилання автоматичних email-повідомлень користувачам про зміни в проєктах, додавання нових завдань, запрошення до участі тощо.

AWS S3 [10] – для зберігання файлів, прикріплених до завдань. Це дозволяє масштабовано й безпечно зберігати документи, зображення або інші типи файлів, пов’язаних з проєктами.

Twilio Video/Audio API [11] – для реалізації функціональності відео та аудіо зв’язку між учасниками проєкту. Це дозволяє проводити онлайн-наради прямо в межах системи без потреби у сторонніх додатках.

OAuth 2.0 [12] – для автентифікації користувачів через зовнішні сервіси, такі як Google. Це спрощує реєстрацію й вхід до системи, дозволяє не створювати окремі паролі та підвищує безпеку шляхом делегування автентифікації перевіреним провайдерам.

### 1.3.5 Інструменти розробки та тестування

Для тестування REST API використовується Postman[13] – популярний інструмент, що дозволяє створювати HTTP-запити, зберігати колекції, додавати токени авторизації, моделювати різні сценарії взаємодії з сервером та перевіряти правильність роботи ендпоінтів. Це значно пришвидшує процес налагодження та тестування бекенд-частини застосунку.

Керування версіями коду здійснюється за допомогою Git, що є стандартом у сучасній розробці. Для хостингу репозиторію використовується платформа GitHub[14], яка забезпечує:

- надійне зберігання коду;
- можливість створення гілок (branches) для розробки нових функцій;
- систему pull request'ів для перевірки змін;
- засоби контролю якості коду;
- зручну командну співпрацю та історію змін.

Основним середовищем розробки є WebStorm – потужний IDE від компанії JetBrains, оптимізований для JavaScript, TypeScript, React, Node.js та супутніх технологій. WebStorm має вбудовані засоби автодоповнення, інтеграцію з Git,

підтримку дебагінгу, форматування коду, перевірку типів, а також інструменти для зручної навігації в проєкті. Завдяки цьому середовище значно підвищує продуктивність розробника, забезпечує дотримання стилістики коду та мінімізує кількість помилок.

Таким чином, поєднання WebStorm, Postman, Git та GitHub забезпечує повний цикл ефективної розробки, тестування й контролю якості програмного забезпечення на всіх етапах створення вебсистеми.

У ході аналізу було визначено основні функціональні та нефункціональні вимоги до вебсистеми керування проєктами. Було вивчено потреби користувачів, сформовано основні сценарії використання та встановлено вимоги до безпеки, масштабованості й продуктивності. На основі отриманих даних були сформульовані технічні специфікації, які стали основою для наступних етапів проєктування.

## 1.4 Визначення загальних та функціональних вимог

### 1.4.1 Загальні вимоги до системи

Розроблювана вебсистема управління проєктами повинна відповідати низці загальних вимог, що забезпечують її зручність, стабільність і масштабованість. Вимоги сформульовані з урахуванням потреб потенційних користувачів, типових сценаріїв використання, а також сучасних стандартів веброзробки.

До загальних вимог відносяться:

- кросплатформеність – система повинна коректно працювати у всіх сучасних браузерах (Chrome, Firefox, Edge, Safari) та бути доступною як на ПК, так і на мобільних пристроях;

- модульність – архітектура має передбачати можливість легкого розширення функціональності без значного впливу на вже реалізовані частини;

- безпека – система повинна забезпечувати захист персональних даних

користувачів, автентифікацію через JWT [8], контроль доступу до проєктів, захист API від несанкціонованого доступу;

- швидкодія – інтерфейс має бути швидким і реагувати без затримок навіть при великій кількості завдань чи учасників у проєкті;

- масштабованість – система повинна підтримувати роботу як невеликих команд, так і великих організацій з багатьма проєктами одночасно;

- надійність – система повинна стабільно працювати протягом тривалого часу без збоїв, із забезпеченням збереження даних;

- доступність API – усі основні функції мають бути доступні через REST API для можливості подальшої інтеграції з мобільними додатками або іншими сервісами.

Загальні вимоги створюють основу для формування конкретних функціональних та нефункціональних вимог, які детальніше розглядаються у наступних підрозділах.

### 1.4.2 Функціональні вимоги до системи

Функціональні вимоги визначають, які дії повинна виконувати система для досягнення поставлених цілей. У контексті системи управління проєктами ці вимоги охоплюють як базову, так і розширену функціональність, що забезпечує повноцінне керування проєктною діяльністю користувачів.

Система повинна забезпечувати такі основні функції:

- реєстрація та авторизація користувачів за допомогою email, пароля або через OAuth [12] (Google-акаунт);

- створення, редагування та видалення проєктів з можливістю додавання обкладинки, опису, дедлайну;

- запрошення інших користувачів до участі в проєкті з визначенням їхньої ролі (адміністратор, менеджер, учасник, спостерігач);

- створення та редагування завдань у межах проєкту з можливістю встановлення статусу, пріоритету, дедлайну, відповідального;

- коментування завдань усіма учасниками проєкту з фіксацією часу та авторства;

- зберігання файлів, прикріплених до завдань, у сховищі AWS S3 [10];

- відправлення email-сповіщень користувачам при певних подіях (запрошення, зміна статусу завдання тощо);

- можливість проведення відео чи аудіо зв'язку між учасниками проєкту за допомогою Twilio [11];

- фільтрація, пошук і сортування завдань за різними критеріями (статус, виконавець, дата тощо);

- захист маршруту та ресурсів системи відповідно до прав доступу користувача.

Перераховані функції є критично важливими для повноцінного використання системи у командній роботі над проєктами.

### 1.4.3 Користувацькі вимоги

Для ефективного використання вебсистеми управління проєктами користувач повинен мати базові знання в галузі користування сучасними вебзастосунками, а також доступ до необхідних технічних засобів. Система проєктувалась таким чином, щоб бути доступною широкому колу користувачів без потреби у спеціальній підготовці.

Основні вимоги до користувача:

- наявність стабільного інтернет-з'єднання, що забезпечує завантаження вебінтерфейсу, синхронізацію даних у реальному часі та використання відео/аудіо дзвінків;

- сучасний браузер з підтримкою JavaScript та HTML5 (Google Chrome, Mozilla Firefox, Microsoft Edge, Safari);

- при використанні функцій авторизації через сторонні сервіси (OAuth [12] ) – наявність активного Google-акаунта.

Для користувачів з роллю адміністратора або керівника проєкту бажано мати базове уявлення про принципи організації командної роботи та ведення проєктної документації.

#### 1.4.5 Вимоги до безпеки

Зважаючи на те, що вебсистема керування проєктами працює з персональними даними користувачів, включаючи облікову інформацію, дані про завдання, файли, а також комунікаційні повідомлення, питання безпеки є одним із ключових аспектів під час розробки. У системі реалізовано низку механізмів, що забезпечують її захист на різних рівнях – від автентифікації до контролю доступу, шифрування, а також взаємодії з API сторонніх сервісів.

##### 1.4.5.1 Автентифікація та авторизація

Для автентифікації користувачів у системі використовується механізм JWT (JSON Web Token), що дозволяє створювати безсесійні авторизаційні системи. Користувач після успішного входу в систему отримує два типи токенів access та refresh.

Access Token – короткоживучий токен, який додається до заголовка кожного запиту, що потребує авторизації.

Refresh Token – довгоживучий токен, який зберігається на клієнті (наприклад, у HTTP-only cookie) та використовується для оновлення access token без необхідності повторного входу.

Така система забезпечує як безпеку, так і зручність для користувача. Access токени мають обмежений термін дії (наприклад, 15 хвилин), що знижує ризик використання викраденого токена, а refresh токени дозволяють продовжити сесію без повторної автентифікації.

##### 1.4.5.2 Ролі та контроль доступу (RBAC)

Усі маршрути системи захищені відповідно до ролі користувача. Використовується модель Role-Based Access Control, що дозволяє:

- обмежити доступ до певних функцій залежно від ролі (гість, учасник, менеджер, адміністратор);

- реалізувати на рівні бекенду перевірки прав доступу до ресурсів (наприклад, доступ до певного проєкту або завдання);

- уникати несанкціонованих дій, наприклад, редагування проєктів іншими учасниками.

#### 1.4.5.3 Захист даних у транзиті та на зберіганні

Вся взаємодія між клієнтом і сервером здійснюється через захищене з’єднання (HTTPS) із використанням SSL/TLS.

Паролі користувачів зберігаються у базі даних у зашифрованому вигляді з використанням сучасних алгоритмів хешування, таких як bcrypt.

Користувацькі файли зберігаються в AWS S3 з обмеженням доступу за токеном або підписаним URL, що виключає публічний доступ до них.

#### 1.4.5.4 Захист API та перевірка запитів

Усі API-ендпоінти вимагають наявності access токена в заголовку запиту;

Реалізовано перевірку CSRF (Cross-Site Request Forgery) для чутливих операцій, у випадках, коли refresh токени використовуються через cookie;

Забезпечено обмеження кількості запитів (rate limiting) для запобігання атакам типу brute force.

#### 1.4.5.5 Інші заходи безпеки

Використовується санітизація вхідних даних для запобігання SQL-ін’єкціям та XSS-атакам;

Обробка помилок здійснюється без виводу зайвої технічної інформації, яка могла б бути використана зловмисниками;

Здійснюється логування дій користувачів для виявлення підозрілої активності (аудит).

#### 1.4.5.6 Безпечна інтеграція зі сторонніми сервісами

Інтеграція з Google OAuth, Google Mail API, AWS S3 та Twilio відбувається через захищені API з використанням токенів доступу;

Усі сторонні ключі (API keys, secrets) зберігаються в змінних середовища та ніколи не потрапляють у відкритий код або репозиторій.

## 1.5 Висновки

На основі проведеного аналізу предметної області, існуючих рішень, а також поставленої задачі щодо розробки вебсистеми керування проєктами, сформовано чіткі вимоги до майбутньої системи та обґрунтовано вибір програмних засобів, які будуть використані під час розробки.

Було визначено, що основною метою проєкту є створення повнофункціональної вебплатформи, яка дозволяє організовувати проєктну діяльність, створювати та керувати завданнями, спілкуватися в межах проєктів, відстежувати статуси, а також забезпечувати безпечний доступ користувачів відповідно до їх ролей. Особливу увагу приділено забезпеченню зручності користувача та швидкодії системи при одночасному збереженні масштабованості.

Для реалізації серверної частини обрано NestJS – прогресивний фреймворк для створення ефективних та надійних Node.js-застосунків. У якості ORM використовується Prisma, що дозволяє швидко і безпечно взаємодіяти з базою даних MySQL, яка забезпечує реляційну структуру збереження інформації.

Фронтенд реалізується за допомогою Next.js у поєднанні з React – це дає змогу створити реактивний, SEO-оптимізований інтерфейс із можливістю серверного рендерингу. Для стилізації обрано Tailwind CSS, що забезпечує швидке створення адаптивного й сучасного дизайну інтерфейсу.

Система авторизації побудована на основі JWT (JSON Web Token), а також реалізована підтримка OAuth через Google-акаунти. Це забезпечує надійний захист даних користувача та дозволяє швидко проходити реєстрацію або вхід.

Для реалізації зберігання файлів використано Amazon S3, що дозволяє зручно завантажувати та переглядати файли в межах проєктів. Відправку повідомлень (наприклад, email-повідомлень про оновлення в проєктах) реалізовано через Gmail API. Для відеозв’язку між учасниками проєктів інтегровано Twilio Video API.

Таким чином, технічне завдання сформовано з урахуванням актуальних вимог ринку до систем управління проєктами, а вибір архітектури, технологій та

інструментів забезпечить реалізацію надійної, масштабованої та функціонально багатої вебсистеми. У наступних розділах буде детально розглянуто етапи проєктування, реалізації та тестування системи.

# 2 ПРОЄКТУВАННЯ ВЕБСИСТЕМИ КЕРУВАННЯ ПРОЄКТАМИ

## 2.1 Архітектура програмного забезпечення

Архітектура вебсистеми керування проєктами побудована з урахуванням принципів модульності, масштабованості та розділення відповідальностей. Система реалізована у вигляді клієнт-серверної моделі з чітким поділом між фронтендом і бекендом. Основу складає стек технологій NestJS + Prisma (MySQL) на серверній стороні та Next.js + React + Tailwind CSS на клієнтській, а загальна схема наведена на рисунку 2.1.

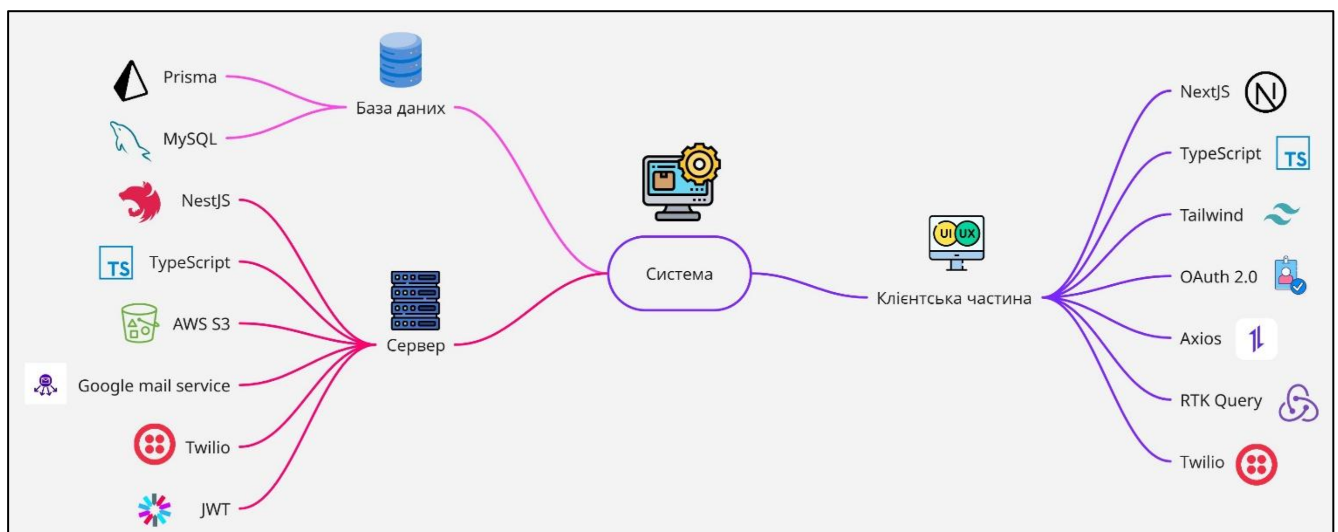



Рисунок 2.1 – Архітектура вебсистеми

### 2.1.1 Загальна структура

Загальна архітектура включає три основні рівні: клієнтська частина, серверна частина та база даних або декілька баз даних наприклад SQL та Redis для

оптимізації.

Клієнтська частина (Frontend) – реалізований на базі Next.js (React), що дозволяє будувати як серверний рендеринг, так і клієнтську SPA-архітектуру. Для стилізації інтерфейсу використовується Tailwind CSS, що забезпечує швидку розробку адаптивного, доступного UI.

Серверна частина (Backend) – реалізований на платформі NestJS з повною типізацією через TypeScript. Архітектура бекенду побудована за принципом модульності, з чітким розділенням логіки на контролери, сервіси, DTO (Data Transfer Object) та middleware.

База даних – використовується MySQL, робота з якою здійснюється через ORM Prisma. Це дає можливість типізовано взаємодіяти з моделями даних, реалізувати складні запити, а також підтримувати міграції схем.

Уся взаємодія між клієнтом і сервером відбувається через RESTful API, що реалізовано на стороні NestJS. Сервер відповідає за авторизацію, обробку бізнес-логіки, роботу з базою даних та зовнішніми сервісами (пошта, хмарне зберігання, відео дзвінки тощо).

### 2.1.2 Основні модулі системи

У системі реалізовано низку ключових модулів:

- модуль автентифікації – відповідає за реєстрацію, вхід користувача, генерацію access/refresh токенів, а також інтеграцію з OAuth (Google, GitHub);
- модуль користувача – забезпечує управління профілем, зміною паролю, ролями та правами доступу;
- модуль проєктів – дозволяє створювати, редагувати, видаляти проєкти, додавати учасників та керувати їхніми правами;
- модуль задач (Tasks) – реалізує CRUD-операції над завданнями, з можливістю фільтрації, коментування, зміни статусу та додавання вкладень;
- модуль комунікації – реалізує текстове спілкування через коментарі, а також відео/аудіо зустрічі через Twilio API;
- модуль повідомлень – автоматично надсилає сповіщення через Gmail SMTP про нові події в проєкті;

- модуль зберігання файлів – відповідає за завантаження, зберігання та видачу файлів через AWS S3.

Таке розділення дозволяє масштабувати проєкт, тестувати окремі частини логіки, та підключати нові функції без зміни існуючих модулів.

### 2.1.3 Структура бази даних

Уся інформація зберігається у базі даних, структура якої відповідає вимогам реляційної моделі. Основні таблиці:

- users – інформація про користувачів;
- projects – дані про створені проєкти;
- project_participants – зв'язок між користувачами і проєктами з ролями;
- tasks – завдання в межах проєктів;
- comments – коментарі до завдань;
- attachments – прикріплені файли;
- tokens – refresh токени;
- meetings – інформація про відео/аудіо зустрічі.

ORM Prisma дозволяє типізувати схему, автоматично мігрувати зміни та генерувати типи TypeScript для кожної моделі.

### 2.1.4 Інтеграції з зовнішніми сервісами

У межах архітектури реалізовано кілька зовнішніх інтеграцій:

- Google OAuth2 – забезпечує авторизацію через обліковий запис Google;
- Gmail API – використовується для надсилання email-повідомлень;
- AWS S3 – зберігання завантажених файлів (прикріплення до завдань);
- Twilio – ініціювання відео та аудіо зустрічей у рамках проєкту.

Кожна з інтеграцій інкапсульована в окремий сервіс, що взаємодіє через API ключі, зберігаючи логіку безпеки (ключі зберігаються у .env).

### 2.1.5 Взаємодія компонентів

Компоненти системи взаємодіють за допомогою стандартного REST API. Наприклад:

- при створенні завдання користувач надсилає POST-запит до /tasks, сервер перевіряє токен, валідує дані, створює запис у БД та повертає JSON з новим

завданням;

- при зміні статусу завдання PATCH-запит передає новий статус, сервер оновлює відповідне поле у базі даних;

- при вході в систему сервер повертає access/refresh токени, які зберігаються на клієнті;

- файли передаються на сервер, який зберігає їх на AWS S3 і повертає URL для завантаження.

### 2.1.6 Тестування та розгортання

Тестування REST API здійснюється через Postman, де створено колекції запитів до всіх основних ендпоінтів. Проєкт підтримує локальний запуск через Docker або Node.js, а також може бути розгорнутий на хмарному хостингу. Репозиторій з кодом розміщено на GitHub, що дозволяє використовувати CI/CD при потребі.

## 2.2 Моделювання системи

Моделювання є критично важливим етапом проєктування програмного забезпечення, що дозволяє заздалегідь визначити структуру системи, взаємозв'язки між її компонентами, логіку взаємодії та сценарії використання. У процесі моделювання вебсистеми керування проєктами були використані нотації UML (Unified Modeling Language) для створення діаграм, які відображають як статичні, так і динамічні аспекти системи.

### 2.2.1 Діаграма варіантів використання (Use Case Diagram)

Дана діаграма(рис. 2.2) ілюструє основні сценарії взаємодії користувача із системою. Учасники (Actors), такі як гість, учасник проєкту, менеджер, адміністратор, мають доступ до певних дій відповідно до своїх прав. Ця діаграма дозволяє швидко зрозуміти, як саме користувачі взаємодіють із системою.

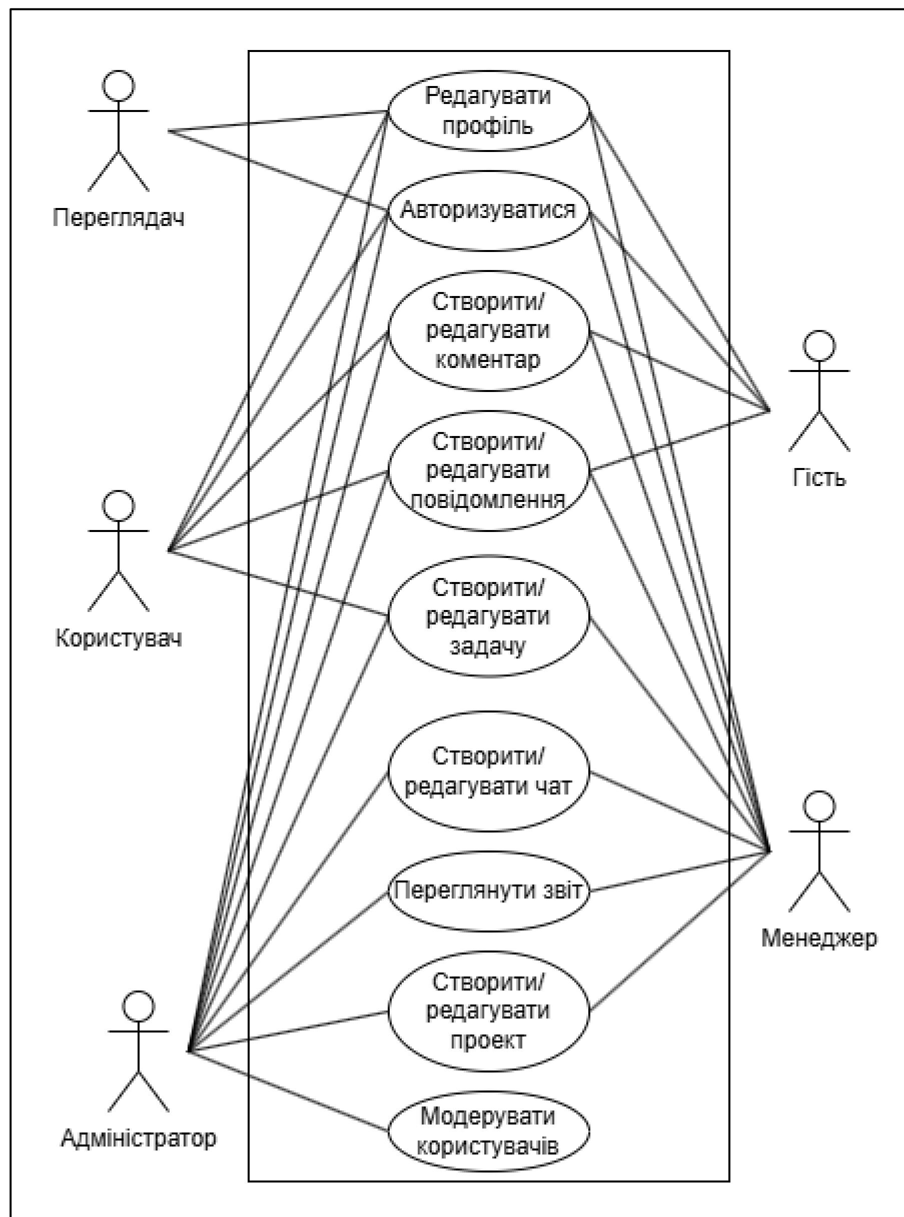



Рисунок 2.2 – Діаграма варіантів використання

### 2.2.2 Діаграма класів (Class Diagram)

Діаграма класів(рис. 2.3) демонструє логічну структуру системи на рівні об’єктів і їхніх взаємозв’язків. Основні класи:

- User – з полями id, email, passwordHash, role;
- Project – з назвою, описом, статусом, датою створення;
- Task – з полями назви, опису, дедлайну, статусу, виконавця;
- Chat – з полями назви, опису;
- Message – з полями текст, автора, дату створення;
- Comment – містить текст, автора, дату створення;

- Attachment – з іменем файлу, URL та зв’язком із конкретним завданням;
- Meeting – з інформацією про учасників і тип (аудіо/відео).

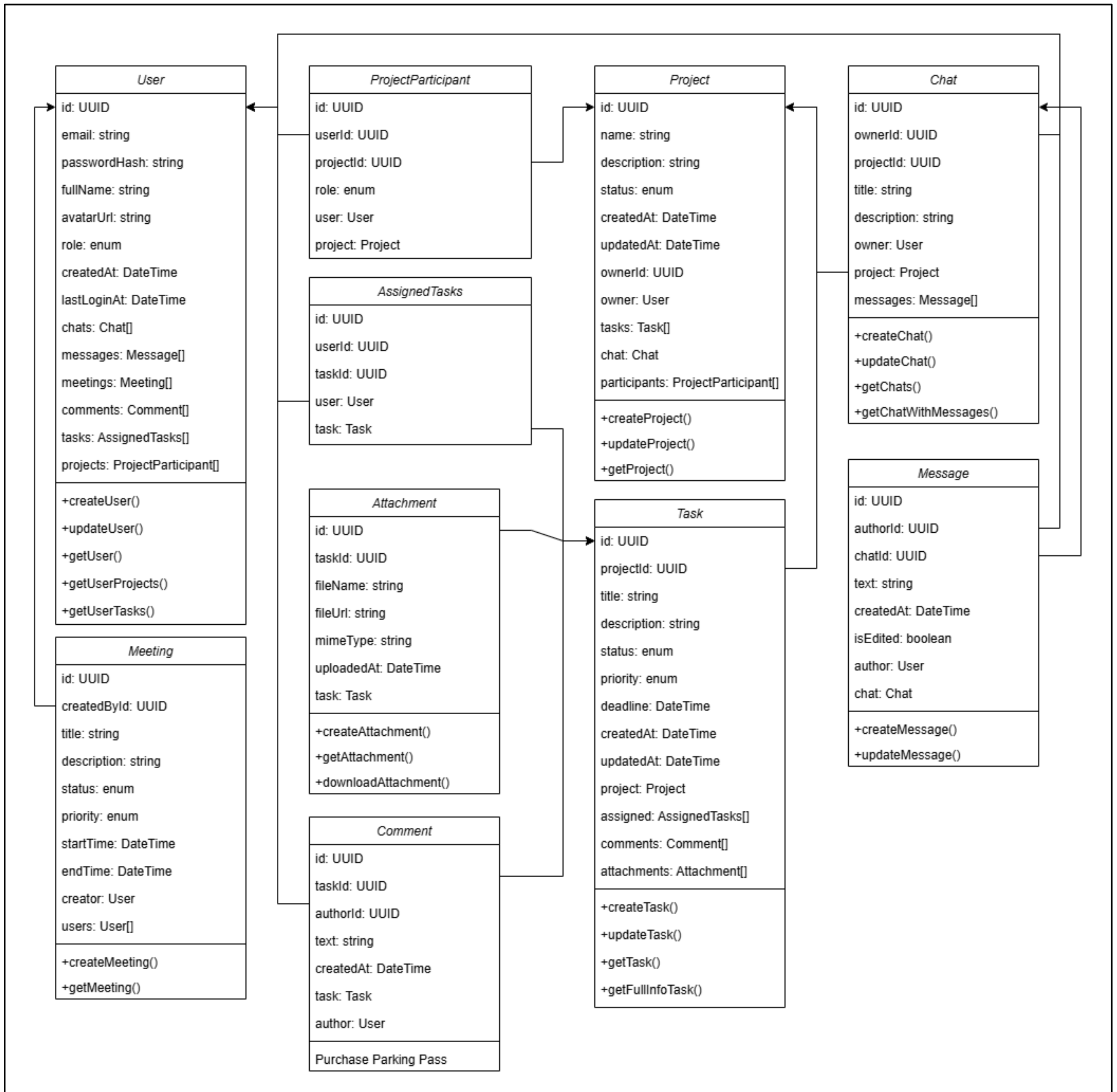



Рисунок 2.3 – Діаграма класів

### 2.2.3 Діаграми послідовностей (Sequence Diagram)

Цей тип діаграми показує послідовність взаємодій між об’єктами у часі.

Надана діаграма(рис. 2.4) послідовності демонструє процес авторизації користувача у вебсистемі. Користувач вводить дані на інтерфейсі, які

надсилаються на сервер для перевірки. Сервер проводить валідацію, здійснює пошук або створення користувача через Prisma, генерує JWT токен і повертає його клієнту, після чого користувача перенаправляють на сторінку завдань.

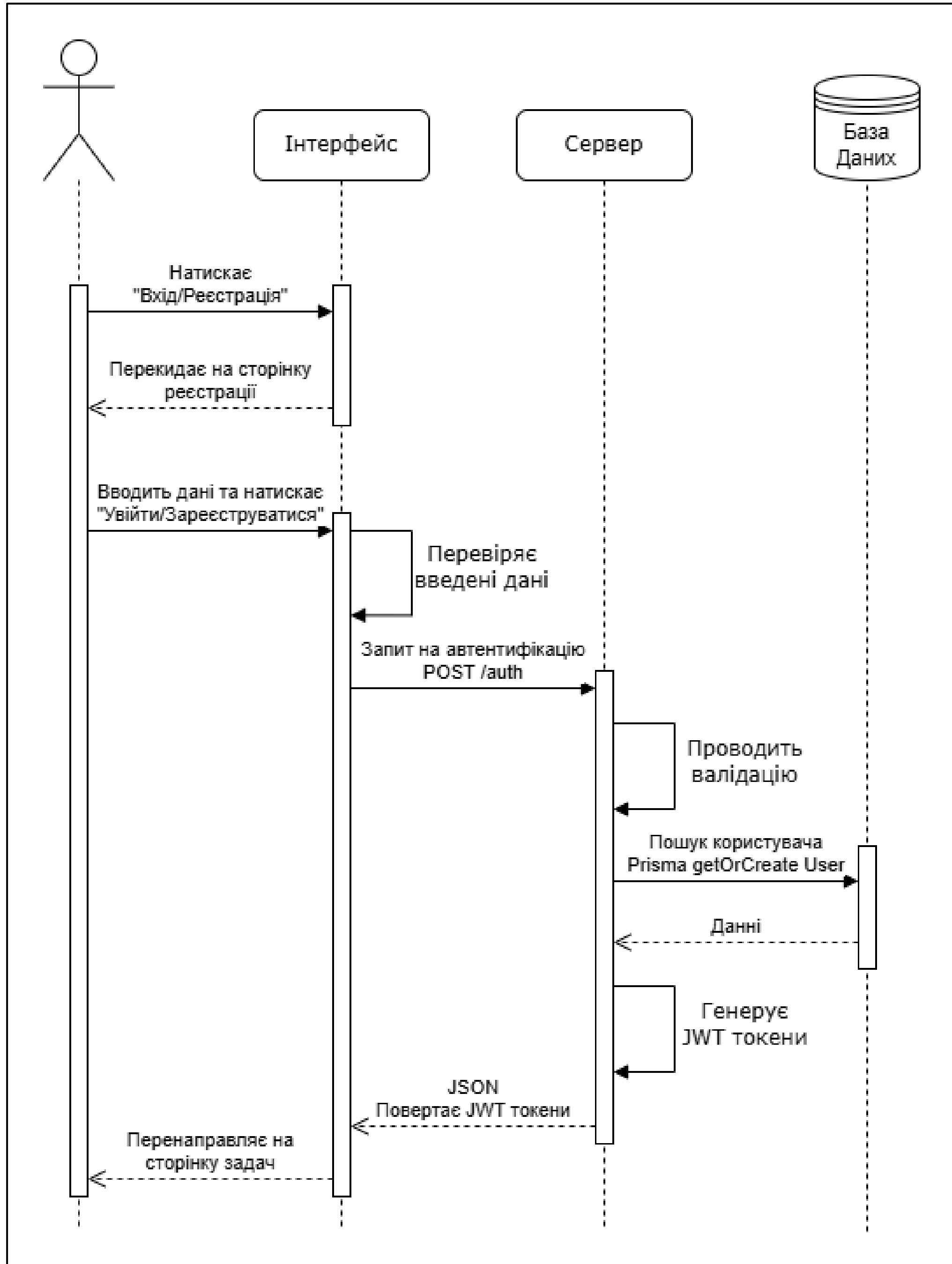



Рисунок 2.4 – Діаграма послідовності авторизації

Наступна діаграма(рис. 2.5) демонструє процес створення нового проєкту. Користувач ініціює дію через інтерфейс, заповнюючи форму з назвою, описом та іншими даними. Сервер приймає POST-запит, проводить валідацію, створює запис

у базі даних за допомогою Prisma і повертає відповідь. Після успішного створення користувача перенаправляють на сторінку проєкту.

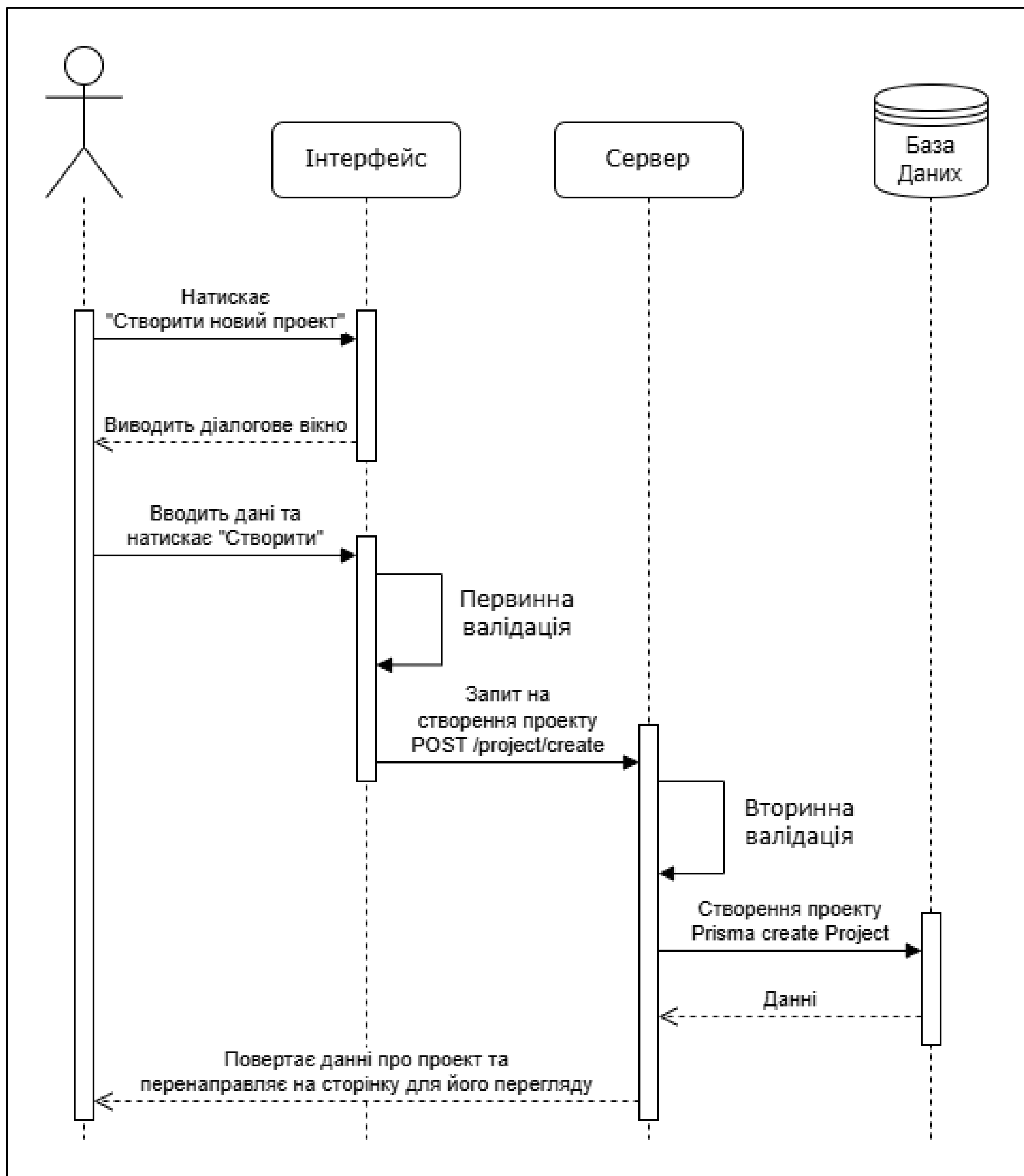



Рисунок 2.5 – Діаграма послідовності створення проєкту

Наступна діаграма(рис. 2.6) ілюструє процес перегляду звітності проєкту або завдань. Користувач натискає на відповідну опцію в інтерфейсі, система формує GET-запит до сервера. Сервер отримує дані з бази, виконує агрегації (наприклад, кількість завдань, статуси, прогрес), формує відповідь у форматі JSON і повертає її

клієнту для візуалізації у вигляді графіків або таблиць.

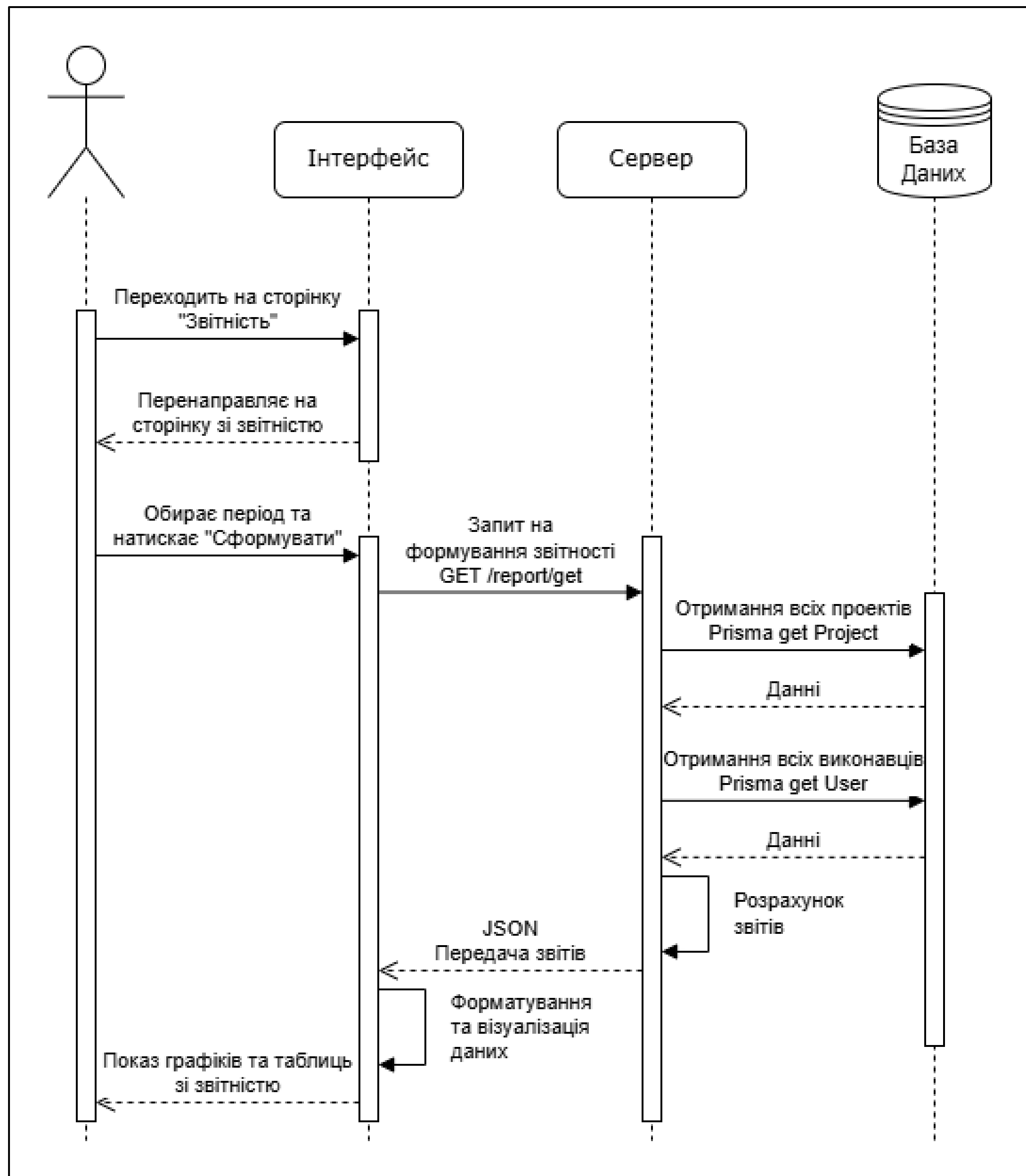



Рисунок 2.6 – Діаграма послідовності перегляду звітності

Такі діаграми дозволяють проаналізувати логіку системи, виявити потенційні проблеми або неефективності у процесах.

Розроблено архітектуру вебсистеми, яка враховує багаторівневий підхід до обробки даних та взаємодії з користувачем. Спроєктовано структуру бази даних, реалізовано діаграми класів, компонентів і послідовності, що описують основну логіку взаємодії системи. Було враховано принципи модульності, повторного

використання коду та розширюваності, що дозволяє легко адаптувати систему до нових вимог.

## 2.3 Системні вимоги

Перед реалізацією вебсистеми було визначено базові технічні вимоги до середовища її функціонування. Враховано як мінімальні параметри, необхідні для коректної роботи, так і рекомендовані — для стабільної роботи в умовах реального навантаження. Системні вимоги охоплюють як програмне, так і апаратне забезпечення для клієнтської та серверної частини, а також специфіку зовнішніх інтеграцій із сервісами AWS, Gmail API та Twilio. Нижче у таблицях 2.1 та 2.2 наведено відповідні вимоги для розгортання й тестування проєкту.

Таблиця 2.1 – Апаратні вимоги

| **Параметр** | **Мінімальні вимоги** | **Рекомендовані вимоги** |
| --- | --- | --- |
| Процесор | Двоядерний 1.8 ГГц | Чотириядерний 2.4 ГГц і вище |
| Оперативна пам’ять | 2 ГБ | 4–8 ГБ |
| Розширення екрану | 1280×720 | 1920×1080 |
| Відеокамера/Мікрофон | Вебкамера і мікрофон (опціонально) | HD-камера і шумопоглинаючий мікрофон |
| Інтернет | 5 Мбіт/с стабільне з’єднання | 20+ Мбіт/с для відеозустрічей |

Таблиця 2.2 – Програмні вимоги

| **Параметр** | **Мінімальні вимоги** | **Рекомендовані вимоги** |
| --- | --- | --- |
| Операційна система | Windows 8.1 / macOS 10.13 / Linux | Windows 10+, macOS 12+, Ubuntu 22.04 |
| Браузер | Google Chrome 90+, Firefox 88+, Edge | Остання стабільна версія Chrome або Brave |
| Підтримка JavaScript | Увімкнено | Увімкнено |
| Підтримка WebRTC (дзвінки) | Так | Так |

# 3 РЕАЛІЗАЦІЯ ВЕБСИСТЕМИ КЕРУВАННЯ ПРОЄКТАМИ

## 3.1 Огляд технологічного стеку

### 3.1.1 Backend: NestJS + Prisma (MySQL)

Сервер використовує NestJS – фреймворк для побудови серверних застосунків, який підтримує TypeScript з максимальною інтеграцією з різними технологіями. Для взаємодії з базою даних ми використовуємо Prisma ORM, що забезпечує зручне управління даними через типізовані моделі.

Структура серверу:

- /src – основний каталог з кодом;
- /auth – компоненти для авторизації та автентифікації;
- /user – логіка роботи з користувачами;
- /project – функціональність для роботи з проєктами;
- /chat – логіка функціонування чату;
- /task – завдання, їх створення, редагування;
- /comment – управління коментарями;
- /file – обробка файлів та їх завантаження;
- /meeting – інтеграція з Twilio для відео-зустрічей.

Його основні переваги:

- повна підтримка TypeScript.
- модульна архітектура, що дозволяє легко розширювати проєкт.
- зручна інтеграція з ORM та базами даних.
- вбудована підтримка Dependency Injection (DI).
- можливість створення REST або GraphQL API.

### 3.1.2 Frontend: Next.js + React + TailwindCSS

Клієнтська частина побудована на базі Next.js та React для створення динамічного інтерфейсу, що підтримує серверний рендеринг та швидку взаємодію з користувачем. TailwindCSS використовується для стилізації, що дозволяє ефективно розробляти адаптивні інтерфейси з мінімальним часом на стилізацію.

Структура клієнту:

- /pages – основні сторінки програми;

- /components – компоненти UI для різних частин застосунку;

- /context – зберігання глобальних станів;

- /utils – утиліти, що обробляють дані та взаємодіють з сервером.

Основні переваги:

- компонентна архітектура;

- велика спільнота та екосистема;

- підтримка хуків для зручної роботи з життєвим циклом компонентів і станом.

Для стилізації застосовано TailwindCSS – утилітарний CSS-фреймворк, що дозволяє швидко створювати адаптивний, сучасний та зручний інтерфейс без потреби у написанні власного CSS.

### 3.1.3 Типізація за допомогою TypeScript

Однією з ключових переваг сучасної веброзробки є використання мови TypeScript, яка розширює JavaScript за рахунок статичної типізації. У межах реалізації вебсистеми керування проєктами TypeScript було використано як у клієнтській (Next.js + React), так і у серверній (NestJS) частинах проєкту. Це забезпечило:

- безпечнішу розробку за рахунок раннього виявлення помилок;

- кращу інтеграцію з IDE (особливо в редакторі WebStorm), автозаповнення та перевірку типів під час написання коду;

- уніфіковану модель даних, що повторно використовується між клієнтом і сервером;

- легше масштабування проєкту у майбутньому.

У системі реалізовано чітке розмежування типів:

- інтерфейси (interface) – використовуються для опису форм об'єктів, що приходять із зовнішніх джерел (наприклад, відповіді з API);

- типи (type) – застосовуються для створення похідних або об'єднаних типів, коли необхідна гнучкість;

- DTO (Data Transfer Objects) – окремі класи, що слугують як контракт для вхідних даних, використовуються переважно в NestJS.

Переваги застосування TypeScript у проєкті:

- зменшення кількості runtime-помилок;

- спрощення підтримки та масштабування коду;

- зручність автодоповнення в редакторах;

- полегшення командної роботи (єдині правила типів);

- можливість рефакторингу без втрати стабільності.

Приклад використовуваних DTO наведено нижче в лістингу 3.1.

Лістинг 3.1 – приклад DTO реєстрації

```
export class CreateUserDto {
  @IsString()
  @IsNotEmpty({ message: "Password is required." })
  @MinLength(8, { message: "Password must be at least 8 characters long." })
  @MaxLength(32, { message: "Password cannot exceed 32 characters." })
  password: string;

  @IsString()
  @IsNotEmpty({ message: "Email is required." })
  @IsEmail({}, { message: "Email must be a valid email address." })
  email: string;

  emailVerificationToken?: string;
}
```

TypeScript став незамінним інструментом у процесі створення даної системи, підвищуючи надійність та читабельність коду на всіх етапах розробки.

### 3.1.4 Система авторизації (JWT, OAuth)

У вебсистемі керування проєктами реалізовано гнучку та безпечну систему авторизації, яка забезпечує контроль доступу до ресурсів, автентифікацію користувачів та збереження сесії. Основу авторизаційного механізму складають JWT-токени (JSON Web Token) для класичного логіну через email/пароль, а також OAuth 2.0 для авторизації через сторонні сервіси, зокрема Google.

JWT (JSON Web Token) – це безсесійний механізм автентифікації, який

дозволяє передавати дані про користувача у вигляді зашифрованого токена. Він складається з трьох частин: header, payload та signature, і є самодостатнім – не потребує постійного збереження сесій на сервері.

Принцип роботи JWT: Користувач надсилає запит на авторизацію з email та паролем.

Якщо дані коректні, сервер генерує пару токенів:

- access token (короткоживучий, на 15 хв);

- refresh token (довготривалий, зберігається в httpOnly cookies).

Access token додається до заголовку кожного наступного запиту (Authorization: Bearer <token>). Якщо access token закінчується, клієнт автоматично запитує новий, використовуючи refresh token. Токени зберігаються в безпечних cookie або локальному сховищі. Refresh-токени оновлюються автоматично та можуть бути відкликані при виході користувача. Всі критичні маршрути захищено middleware(код яких надано в лістингах 3.2 та 3.3), що перевіряє валідність токена.

Переваги OAuth:

- пришвидшення реєстрації (без введення пароля);

- підвищення довіри користувачів;

- безпека (пароль не зберігається на стороні сервера);

- підтримка декількох провайдерів (GitHub, Google тощо).

Лістинг 3.2 – Код access guard

```
import { ExecutionContext, Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class AtGuard extends AuthGuard('jwt') {
  constructor(private reflector: Reflector) {
    super();
  }

  canActivate(context: ExecutionContext) {
    const isPublic = this.reflector.getAllAndOverride('isPublic', [
      context.getHandler(),
      context.getClass(),
    ]);
    if (isPublic) return true;
```

```
    return super.canActivate(context);
  }
}
```

Лістинг 3.3 – Код refresh guard

```
import { AuthGuard } from '@nestjs/passport';
import { ExecutionContext, Injectable, UnauthorizedException } from
'@nestjs/common';
import { Reflector } from '@nestjs/core';
@Injectable()
export class RtGuard extends AuthGuard('jwt-refresh') {
  constructor(private reflector: Reflector) {
    super();
  }
  canActivate(context: ExecutionContext) {
    const isPublic = this.reflector.getAllAndOverride('isPublic', [
      context.getHandler(),
      context.getClass(),
    ]);
    if (isPublic) return true;

    return super.canActivate(context);
  }
}
```

Авторизаційна система пов'язана з ролями, які визначають рівень доступу в системі:

- admin – повний доступ до керування користувачами, проєктами, завданнями;
- manager – створення та редагування проєктів, призначення учасників;
- member – доступ до завдань, коментування, зміна статусу;
- viewer – перегляд проєктів без змін;
- guest – тимчасовий доступ або обмежений перегляд.

Відповідно реалізована перевірка ролі для запитів, роль зберігається так само у JWT й перевіряється при виконанні запиту на захищений відповідним способом API, реалізацію захисту по ролям наведено в лістингу 3.4.

Лістинг 3.4 – Код role guard

```
import { CanActivate, ExecutionContext, HttpException, HttpStatus,
Injectable, UnauthorizedException } from "@nestjs/common";
import { Reflector } from "@nestjs/core";
```

```
import { JwtService } from '@nestjs/jwt';
import { Observable } from "rxjs";
import { ROLES_KEY } from "../decorators/role-auth.decorator";
import { ConfigService } from "@nestjs/config";
@Injectable()
export class RoleGuard implements CanActivate {
  constructor(
    private jwtService:JwtService,
    private reflector: Reflector,
    private config: ConfigService,
  ) {}
  canActivate(context: ExecutionContext):
  boolean | Promise<boolean> | Observable<boolean> {
    try {
      const                         reqRoles                         =
this.reflector.getAllAndOverride<string[]>(ROLES_KEY, [
        context.getHandler(),
        context.getClass()
      ])
      if(!reqRoles){
        return true;
      }
      const req = context.switchToHttp().getRequest()
      const authHeader = req.headers.authorization;
      const bearer = authHeader.split(' ')[0]
      const token = authHeader.split(' ')[1]
      if(bearer !== 'Bearer' || !token){
        throw   new   UnauthorizedException({message:   'User   is   not
authorized.'})
      }
      const    user    =    this.jwtService.verify(token,    {secret:
this.config.get<string>('AT_SECRET')});
      req.user = user;
      const roleMatched = reqRoles.some(role => user.role === role);
      return roleMatched;
    } catch (error) {
      throw    new    HttpException('User    have    not    access.',
HttpStatus.FORBIDDEN)
    }}
}
```

Система авторизації реалізована з урахуванням сучасних вимог до безпеки, зручності користування та масштабованості. Комбінація JWT та OAuth дозволяє підтримувати як традиційні логіни, так і соціальні методи входу, що підвищує UX та відкриває можливість інтеграції з іншими сервісами.

## 3.2 Опис серверної логіки

Серверна частина вебсистеми керування проєктами розроблена з використанням NestJS – прогресивного Node.js фреймворку, що базується на архітектурних принципах Angular. Усі модулі реалізовано з урахуванням принципів SOLID, модульності, інверсії залежностей та повної типізації через TypeScript. Взаємодія з базою даних здійснюється через Prisma ORM, яка надає зручний типізований інтерфейс.

### 3.2.1 Структура основних класів і сервісів

Структура основних класів і сервісів описує основні компоненти серверної частини вебсистеми керування проєктами, побудованої з використанням NestJS та Prisma. У цьому розділі представлено таблицю з ключовими сервісами, що реалізують бізнес-логіку, а також короткий опис методів і функціонального призначення кожного з них:

- AuthService є центральним компонентом у механізмі авторизації: перевіряє облікові дані, генерує токени, обробляє їхнє оновлення через refresh-токен;

- UsersService управляє всією інформацією про користувачів системи;

- ProjectsService відповідає за логіку створення проєктів, призначення ролей, запрошення користувачів;

- TasksService реалізує розширене керування завданнями – включно з кастомним фільтруванням і ролями;

- ChatsService керує логікою групових та індивідуальних чатів;

- FilesService інтегрований з AWS S3 для зберігання завантажених користувачами документів;

- CallsService використовує Twilio API для створення "кімнат" і видачі доступів до відеозв’язку;

- JwtStrategy та GoogleStrategy – частини @nestjs/passport, які реалізують авторизаційні механізми відповідно для токенів і OAuth;

- RolesGuard забезпечує захист маршрутів залежно від ролі користувача, що

реалізує контроль доступу;

- PrismaService – технічний сервіс, який конфігурує та надає PrismaClient, що використовується в усіх сервісах.

Таблиця 3.1 – Сервіси, їх методи та призначення

| **Назва класу/ сервісу** | **Методи** | **Призначення** |
|---|---|---|
| AuthService | validateUser(), login(), refreshToken(), register() | Обробка автентифікації користувачів, створення JWT токенів, генерація та оновлення access/refresh токенів |
| UsersService | findAll(), findById(), updateProfile(), remove() | Робота з обліковими записами користувачів, оновлення профілю, видалення |
| ProjectsService | create(), findAll(), findById(), update(), delete() | CRUD-операції над проєктами, управління учасниками проєктів |
| TasksService | create(), findAll(), findById(), update(), delete() | Створення, редагування, видалення завдань, фільтрація, сортування, прив'язка до проєкту |
| ChatsService | create(), findAll(), findById(), update(), delete() | Створення, редагування та видалення чатів, надсилання та прийом повідомлень. |
| CommentsService | addComment(), getCommentsForTask(), deleteComment() | Додавання, перегляд та видалення коментарів до завдань |
| FilesService | uploadFile(), getFileUrl(), deleteFile() | Завантаження файлів на AWS S3, доступ за URL, видалення файлів |
| CallsService | createRoom(), getAccessToken() | Ініціація відео/аудіозустрічей через Twilio, отримання токенів для доступу |
| JwtStrategy | validate() | Перевірка валідності access токена, витягування користувача з токена |
| GoogleStrategy | validate() | Авторизація користувача через Google OAuth |
| RolesGuard | canActivate() | Захист маршрутів на основі ролі користувача (admin, manager, member тощо) |
| PrismaService | providePrismaClient() | Базовий доступ до бази даних через Prisma ORM, інкапсуляція взаємодії з моделями |

### 3.2.2 Фрагменти реалізації

*Аутентифікація користувачів*

У процесі реєстрації користувач вводить ім’я, email та пароль. Пароль перед збереженням хешується з використанням бібліотеки bcrypt, що забезпечує безпечне зберігання чутливих даних.

Для кожного запиту до захищеного API додається заголовок Authorization: Bearer <access_token>. Усі такі запити проходять через JwtAuthGuard, який перевіряє дійсність токена та витягує з нього дані користувача (лістинг 3.5).

Лістинг 3.5 – Частина коду авторизації

```
async signup(dto: CreateUserDto): Promise<Tokens> {
  const emailExists = await this.userService.getUserByField({ email:
dto.email });
  if (emailExists) throw new ForbiddenException('Email already in
use.');

  dto.password = await this.hashData(dto.password);
  const verificationToken = crypto.randomUUID();
  const newUser = await this.userService.createUser({
    ...dto,
    emailVerificationToken: verificationToken,
  });
  await         this.mailService.sendEmailConfirmation(newUser.email,
verificationToken);
  const { passwordHash, ...jwtPayload } = newUser;
  const tokens = await this.getTokens(jwtPayload);
  return tokens;
}
async confirmEmail(token: string) {
  const    user    =    await    this.userService.getUserByField({
emailVerificationToken: token });
  if (!user) throw new ForbiddenException('Invalid or expired token');
  await this.userService.updateUser(user.id, {
    emailVerified: true,
    emailVerificationToken: null,
  });
}
  async signin(dto: AuthDto): Promise<Tokens> {
    const  user  =  await  this.userService.getUserByField({  email:
dto.email });
    const  passwordMatches  =  await  bcrypt.compare(dto.password,
user.passwordHash);

    if  (!passwordMatches)  throw  new  ForbiddenException('Incorrect
password.');
```

```
    const { passwordHash, ...jwtPayload } = user;
    const tokens = await this.getTokens(jwtPayload);
    return tokens;
  }
```

Також підтримується вхід через OAuth (наприклад, Google), що спрощує доступ для користувачів, які не хочуть створювати локальний акаунт у системі(лістинг 3.6).

Лістинг 3.6 – OAuth авторизація

```
async oauthSignin(email: string): Promise<Tokens> {
  let user = await this.userService.getUserByField({ email });
  if (!user) {
    user = await this.userService.createUser({
      email,
      password: null,
    });
  }
  const { passwordHash, ...jwtPayload } = user;
  const tokens = await this.getTokens(jwtPayload);
  return tokens;
}
```

*Створення та редагування проєктів*

Основна функціональність вебсистеми – це керування проєктами. Кожен користувач може створювати проєкти, редагувати їх, додавати учасників і призначати завдання(лістинг 3.7).

Редагування можливе лише для власника або учасника з відповідними правами (наприклад, MANAGER). Доступ контролюється на рівні сервісів через перевірку ролей.

Лістинг 3.7 – Функції сервісу керування проєктами

```
async getUserProjects(userId: string) {
  return this.prisma.project.findMany({
    where: {
      participants: {
        some: {
          userId: userId,
        },
      },
```

```
      },
      include: {
        participants: {
          include: {
            user: {
              select: {
                id: true,
                email: true,
                role: true,
              },
            },
          },
        },
      },})}
  async createProject(data: CreateProjectDto, userId: string) {
    return this.prisma.project.create({
      data: {
        name: data.name,
        description: data.description,
        status: data.status,
        createdAt: new Date(),
        owner: {
          connect: { id: userId },
        },
        participants: {
          create: [
            {
              userId: userId,
              role: 'OWNER',
            },
          ],
        },
      },
      include: {
        participants: true,
      },
    });
  }
```

*Система ролей та доступу*

Система передбачає чіткий розподіл прав через користувацькі ролі в межах проєктів:

- OWNER – повний контроль над проєктом;

- MANAGER – управління учасниками, завданнями;

- MEMBER – перегляд і виконання завдань;

- VIEWER – лише перегляд та мінімальні дописи(коментарі та чат);

- GUEST – обмежений доступ за посиланням.

Для реалізації контролю доступу використовується кастомний декоратор @Roles() і guard RolesGuard, який перевіряє, чи має користувач необхідну роль для виконання дії. Реалізація сервісу керування користувачами наведено в лістингу 3.8.

Лістинг 3.8 – Сервіс керування користувачами

```
import { Injectable } from '@nestjs/common'
import { User } from '@prisma/client'
import { CreateUserDto } from 'src/dtos'
import { PrismaService } from '../prisma.service'

@Injectable()
export class UsersService {
  constructor(private prisma: PrismaService){}
  async createUser(dto: CreateUserDto){
    const user = await this.prisma.user.create({
      data: {
        email: dto.email,
        passwordHash: dto.password
      }
    })
    return user;
  }
  async updateUser(id: string, data: Partial<User>): Promise<User> {
    return this.prisma.user.update({
      where: { id },
      data,
    });
  }

  async getUserByField(field: { id?: string; email?: string; username?: string, resetPasswordToken?: string, emailVerificationToken?: string}): Promise<User | null> {
    return this.prisma.user.findFirst({
      where: field,
    });
  }
}
```

### 3.3 Опис клієнтської логіки

Клієнтська частина вебсистеми керування проєктами реалізована з використанням Next.js, який підтримує як серверний, так і клієнтський рендеринг.

Структура клієнту організована навколо логічного розділення на сторінки (pages) та компоненти (components). Такий підхід дозволяє легко масштабувати застосунок, повторно використовувати елементи UI, а також чітко розділити відповідальність між відображенням, логікою та даними.

### 3.3.1 Сторінки та компоненти

У системі реалізовано набір основних сторінок, кожна з яких відповідає за окремий сценарій користувача:

- головна ("/") – вітальна сторінка;
- реєстрація ("/register") – форма для створення нового облікового запису;
- вхід ("/login") – форма автентифікації користувача;
- панель керування ("/dashboard") – головна сторінка після входу – список проєктів користувача;
- проєкти ("/projects") – сторінка зі списком усіх доступних користувачу проєктів;
- проєкт ("/projects/[id]") – сторінка з детальною інформацією про проєкт, його завдання та учасників;
- чати ("/chats") – список усіх чатів користувача, з коротким переглядом останніх повідомлень;
- чат ("/chats/[id]") – сторінка з обраним чатом, де можна переглянути, відправити, відредагувати повідомлення;
- статистика ("/stats") – сторінка зі зведеною статистикою по проєктах користувача.

Для сторінок використовується SSR/SSG залежно від потреб. Сторінки, які містять приватну інформацію, захищені за допомогою middleware.

Компоненти поділено на:

- UI-компоненти (shared) – кнопки, поля вводу, модальні вікна;
- функціональні компоненти (feature-based) – списки завдань, таби проєктів, дошка завдань;
- контейнерні компоненти – обгортають UI і містять логіку завантаження даних, керування станом.

Приклади компонентів:

- TaskCard – відображає коротку інформацію про завдання;
- ProjectList – виводить список проєктів користувача з фільтрацією;
- TaskBoard – дошка з підтримкою drag-and-drop для етапів завдань;
- UserAvatar – компонент для відображення профілю користувача;
- InviteModal – модальне вікно для запрошення учасників у проєкт;
- CommentSection– секція коментарів до завдання.

Переваги підходу:

- повторне використання: компоненти будуються як незалежні модулі, які легко інтегрувати в інші частини застосунку.
- легке тестування: UI-компоненти не мають залежності від API, що спрощує модульне тестування.
- типізація: усі компоненти типізовані з використанням TypeScript, що дозволяє уникати помилок під час компіляції та забезпечує надійність коду.

Таким чином, клієнтська частина системи реалізована з урахуванням масштабованості, зручності користування та розширюваності. Компоненти легко підтримуються й розширюються, а структура сторінок забезпечує інтуїтивно зрозумілу навігацію та хорошу користувацьку взаємодію.

### 3.3.2 Робота з API на клієнті

Клієнт вебсистеми керування проєктами активно взаємодіє з серверною частиною через REST API. Основна мета цієї взаємодії – забезпечити передачу, обробку та оновлення даних між клієнтським інтерфейсом і сервером у режимі реального часу. Завдяки ефективному використанню API користувач отримує актуальну інформацію без необхідності оновлювати сторінку, що суттєво покращує UX.

Для роботи з API використовується бібліотека Axios або fetch API, однак основну логіку побудовано з використанням RTK Query, яка забезпечує:

- автоматичне кешування запитів;
- повторну відправку запитів у разі помилок;
- інвалідацію кешу після мутацій;

- відображення статусу завантаження (loading, error, success).

Основні типи запитів:

- GET – отримання деталей проєкту, завдань, коментарів, тощо;
- POST/PUT – створення/редагування користувача, запрошень, тощо;
- DELETE – видалення проєктів, завдань, коментарів, вихід, тощо.

RTK Query автоматично обробляє помилки, але також реалізовані кастомні механізми для:

- інформування користувача про помилку (toast або modal);
- переадресації на сторінку логіну у випадку 401 Unauthorized;
- логування помилок у сервісі моніторингу (якщо реалізовано).

Завдяки гнучкості RTK Query, можна встановлювати залежності між запитами. Наприклад, при зміні завдання, оновлюються відповідні списки проєкту або статусів.

Клієнтська частина системи реалізована як повноцінний SPA, де більшість логіки взаємодії з даними побудована навколо REST API та бібліотеки RTK Query. Такий підхід дозволяє досягти високої продуктивності, гнучкості у зміні даних та чудового користувацького досвіду.

## 3.4 Інтеграції та сторонні сервіси

Для забезпечення повноцінного функціонування вебсистеми було впроваджено низку інтеграцій зі сторонніми сервісами. Це дозволило розширити функціональні можливості, забезпечити масштабованість, покращити збереження даних і зручність взаємодії з користувачем. Нижче наведено ключові інтеграції, які були реалізовані у рамках розробки

### 3.4.1 Робота з AWS S3 (завантаження/перегляд файлів)

У рамках проєкту вебсистеми керування проєктами реалізована інтеграція з Amazon Web Services S3 (Simple Storage Service) для забезпечення надійного та

масштабованого зберігання файлів, прикріплених до завдань. Цей підхід дозволяє відокремити зберігання великих об’ємів даних від основного застосунку, що позитивно впливає на продуктивність і стабільність системи.

У системі реалізовано багаторівневу архітектуру взаємодії з S3:

- клієнтська частина (Next.js) ініціює завантаження файлів через форму завантаження;

- запит надсилається до бекенду (NestJS), де відбувається генерація підписаного URL для безпечного завантаження;

- фронтенд завантажує файл напряму до S3, використовуючи отриманий URL;

- після успішного завантаження бекенд зберігає метадані файлу в базі даних (наприклад, URL, ім’я, тип, розмір, зв'язок із завданням).

Такий підхід мінімізує навантаження на сервер і дозволяє безпечно працювати з файлами без зберігання їх на власному сервері. Для підключення до S3 використовуються офіційні бібліотеки AWS SDK (фрагменти коду для генерації та збереження файлів показано в лістингах 3.9 та 3.10).

З метою безпеки передбачено:

- обмеження MIME-типів завантажуваних файлів;

- ліміти на розмір файлів;

- перевірка користувача перед генерацією підписаних URL.

Лістинг 3.9 – Фрагмент генерації унікального айді

```
import { Injectable } from '@nestjs/common'
import { S3 } from 'aws-sdk'
@Injectable()
export class AwsS3Service {
  private s3 = new S3({
    region: process.env.AWS_REGION,
    credentials: {
      accessKeyId: process.env.AWS_ACCESS_KEY_ID,
      secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY
    }
  });
  async getPresignedUrl(fileName: string, fileType: string) {
    const key = `attachments/${Date.now()}-${fileName}`;
```

```
    const url = this.s3.getSignedUrl('putObject', {
      Bucket: process.env.AWS_BUCKET_NAME,
      Key: key,
      ContentType: fileType,
      Expires: 600,
    });
    return {
      url,
      key,
      publicUrl:
`https://${process.env.AWS_BUCKET_NAME}.s3.${process.env.AWS_REGION}
.amazonaws.com/${key}`
    };
  }
}
```

Лістинг 3.10 – Запис способу доступу до файлу записаного в S3

```
  async create(dto: { fileName: string; url: string; taskId: string
}) {
    return this.prisma.attachment.create({
      data: {
        fileName: dto.fileName,
        url: dto.url,
        taskId: dto.taskId
      }
    });
  }
```

Форма завантаження реалізована як частина форми створення або редагування завдання. Після вибору файлу на клієнті виконується запит до ендпоінту серверу (/upload/signed-url), який повертає короткочасний URL. Завантаження файлу на S3 відбувається через стандартний PUT запит, безпосередньо з браузера, без потреби пересилання через сервер.

Файли, що прикріплені до завдань, виводяться в інтерфейсі завдання у вигляді списку або галереї (для зображень). Кожен елемент має посилання для перегляду або завантаження. Доступ до файлів здійснюється за допомогою signed URLs, що генеруються при кожному запиті й мають обмежений термін дії.

Також реалізована можливість видалення файлів – при цьому викликається API, яке надсилає команду до AWS S3 на видалення об’єкта та очищає відповідний запис у базі даних.

Інтеграція з AWS S3 має низку переваг:

- масштабованість – система готова до роботи з великим обсягом файлів;

- відмовостійкість – дані зберігаються у кількох зонах доступності;

- оптимізація продуктивності – клієнти напряму взаємодіють із S3;

- безпека – дані не доступні публічно, лише через контрольовані підписані URL.

Інтеграція з AWS S3 забезпечує ефективне керування файлами у системі, дозволяє уникнути перевантаження серверу та зберігає високу безпеку користувацьких даних. Завдяки використанню підписаних URL зчитування та завантаження файлів стало простим, масштабованим і надійним рішенням.

### 3.4.2 Відправка повідомлень через Gmail API

У рамках реалізації вебсистеми керування проєктами було впроваджено механізм надсилання електронних повідомлень користувачам, зокрема для таких випадків:

- підтвердження реєстрації або запрошення до проєкту;

- сповіщення про нові завдання або коментарі;

- нагадування про дедлайни чи оновлення статусу завдань.

Для цього інтегровано Gmail API, який дозволяє безпосередньо надсилати листи через обліковий запис Google, гарантуючи високу доставлюваність та безпечний обмін повідомленнями.

Інтеграція з Gmail API реалізована на стороні бекенду (NestJS). Загальна логіка виглядає так:

- система авторизується через Google OAuth 2.0, отримуючи доступ до облікового запису Gmail;

- після виконання певної дії (реєстрація, запрошення, зміна статусу тощо), бекенд формує тіло листа (HTML/текст);

- через Gmail API відправляється лист від імені системи;

- у разі потреби в логах фіксується факт відправки та час.

Для отримання доступу до Gmail API було створено проєкт у Google Cloud Console, де:

- налаштовано OAuth 2.0 Client ID;

- вказано допустимі URI для редиректів (за потреби);

- активовано Gmail API;

- збережено client_id, client_secret, refresh_token для серверного використання.

Оскільки Gmail API вимагає авторизації, використовується "Service Account" з делегуванням прав або OAuth токени з оновленням через refresh_token, код сервісу відправки повідомлень наведено в лістингу 3.11.

Лістинг 3.11 – Фрагмент коду поштового сервісу

```
import { Injectable } from '@nestjs/common'
import * as nodemailer from 'nodemailer'
@Injectable()
export class MailService {
  private transporter = nodemailer.createTransport({
    service: 'gmail',
    auth: {
      user: process.env.GMAIL_USER,
      pass: process.env.GMAIL_PASS,
    },
  });
  async sendEmailConfirmation(to: string, token: string) {
    const    confirmUrl    =    `${process.env.CLIENT_URL}/confirm-
email?token=${token}`;
    const template = this.getEmailVerificationTemplate(confirmUrl);
    await this.transporter.sendMail({
      to,
      subject: template.subject,
      html: template.html,
    });
  }
  async sendPasswordReset(to: string, token: string) {
    const     resetUrl     =     `${process.env.CLIENT_URL}/reset-
password?token=${token}`;
    const template = this.getPasswordResetTemplate(resetUrl);
    await this.transporter.sendMail({
      to,
      subject: template.subject,
      html: template.html,
    });
  }
}
```

У системі створено кілька базових шаблонів(лістинг 3.12):

- верифікація акаунту та скидання паролю;

- запрошення до проєкту – повідомляє користувача про додання до нового проєкту;

- нове завдання – інформує виконавця про призначення нового завдання;

- коментар до завдання – сповіщає учасників про новий коментар;

- наближення дедлайну – нагадує про термін виконання.

Лістинг 3.12 – Шаблони для повідомлень

```
// Template: Password Reset
getPasswordResetTemplate(resetUrl: string) {
  return {
    subject: 'Reset your password',
    html: `<p>You requested a password reset.</p>
            <p><a href="${resetUrl}">Click here to reset your password</a>.</p>`
  };
}
  // Template: Email Verification
getEmailVerificationTemplate(confirmUrl: string) {
  return {
    subject: 'Email Verification',
    html: `<p>Thank you for registering!</p>
            <p>Please <a href="${confirmUrl}">confirm your email address</a>.</p>`
  };
}
// Template: Project Invitation
getProjectInvitationTemplate(projectName: string, inviter: string) {
  return {
    subject: `You've been added to the project "${projectName}"`,
    html: `<p>${inviter} has invited you to a new project: <b>${projectName}</b>.</p>
            <p>Check your project list for more details.</p>`
  };
}
// Template: New Task
getNewTaskTemplate(taskTitle: string, projectName: string, assigner: string) {
  return {
    subject: `New Task: "${taskTitle}"`,
    html: `<p>You have been assigned a new task "<b>${taskTitle}</b>" in the project "${projectName}".</p>
            <p>Assigned by: ${assigner}.</p>`
  };
}
// Template: Task Comment
getTaskCommentTemplate(taskTitle: string, commenter: string, comment: string) {
  return {
```

```
    subject: `New comment on task "${taskTitle}"`,
    html:  `<p>User  ${commenter}  left  a  comment  on  the  task
"<b>${taskTitle}</b>":</p>
            <blockquote>${comment}</blockquote>`
  };
}
// Template: Deadline Reminder
getDeadlineReminderTemplate(taskTitle: string, deadline: string) {
  return {
    subject: `Deadline approaching for task "${taskTitle}"`,
    html: `<p>This is a reminder that the deadline for the task
"<b>${taskTitle}</b>" is set for ${deadline}.</p>
            <p>Don't forget to complete the task on time!</p>`
  };
```

Переваги використання Gmail API:

- надійність – листи надсилаються через надійний сервіс Google;

- контроль – можливість відслідковувати та логувати помилки;

- безпека – автентифікація через OAuth забезпечує високий рівень безпеки;

- масштабованість – API дозволяє надсилати велику кількість листів за добу (з обмеженням квоти).

Таким чином, Gmail API забезпечує ефективну, безпечну і сучасну систему електронних сповіщень, яка значно підвищує зручність користування вебсистемою, а також покращує залученість і комунікацію між учасниками проєктів.

### 3.4.3 Відео та аудіо дзвінки через Twilio

Для забезпечення зручної та ефективної комунікації між учасниками проєктів у вебсистемі було реалізовано функціональність відео та аудіо зв’язку. У якості платформи для реалізації цих можливостей було обрано Twilio – надійний хмарний сервіс, який забезпечує функціонал реального часу для відео та аудіо зв’язку, обміну повідомленнями тощо.

Twilio має низку переваг, які обумовили його використання у цьому проєкті:

- надійність та масштабованість – сервіс підтримує мільйони одночасних підключень і має SLA високого рівня;

- підтримка WebRTC – використовується відкритий протокол реального часу, який підтримується більшістю сучасних браузерів;

- гнучке API – дозволяє точно налаштувати процес створення кімнат, підключення користувачів, обмеження доступу;

- безпечність – дані передаються у зашифрованому вигляді;

- швидка інтеграція – існують SDK для JavaScript, React, Android, iOS.

Реалізація відеозв'язку базується на кімнатах (rooms), які створюються у Twilio. Основні етапи взаємодії:

- створення кімнати – ініціатор дзвінка (наприклад, менеджер) надсилає запит на сервер, який створює нову кімнату у Twilio через REST API;

- генерація токенів доступу – для кожного учасника сервер генерує унікальний токен доступу (JWT), який дозволяє приєднатися до кімнати;

- підключення клієнтів – користувачі на фронтенді підключаються до кімнати за допомогою SDK Twilio Video (на React);

- взаємодія у реальному часі – здійснюється передача відео, аудіо та даних (наприклад, імен учасників, статусу підключення);

- завершення сеансу – після завершення дзвінка кімната автоматично або вручну закривається.

На серверній стороні реалізовано сервіс Twilio Service(), який:

- автентифікується за допомогою accountSid та authToken;

- має методи для створення кімнат, генерації access token;

- обмежує доступ користувачів до кімнат через ідентифікацію;

- логує факти створення дзвінків для можливості аудиту.

Лістинг 3.13 – Сервіс створення конференцій

```
import { Injectable, Logger } from '@nestjs/common'
import { Twilio } from 'twilio'
import AccessToken from 'twilio/lib/jwt/AccessToken'
import { v4 as uuidv4 } from 'uuid'
const { VideoGrant } = AccessToken;

@Injectable()
export class TwilioService {
  private readonly logger = new Logger(TwilioService.name);
  private readonly twilioClient: Twilio;
  constructor(
    private        readonly        accountSid:        string        =
```

```
process.env.TWILIO_ACCOUNT_SID!,
    private        readonly        authToken:        string        =
process.env.TWILIO_AUTH_TOKEN!,
    private        readonly        apiKeySid:        string        =
process.env.TWILIO_API_KEY_SID!,
    private        readonly        apiKeySecret:        string        =
process.env.TWILIO_API_KEY_SECRET!,
  ) {
    this.twilioClient = new Twilio(this.accountSid, this.authToken);
  }

  async createRoom(roomName?: string) {
    const uniqueRoomName = roomName || `room-${uuidv4()}`;
    const room = await this.twilioClient.video.rooms.create({
      uniqueName: uniqueRoomName,
      type: 'group',
    });
    this.logger.log(`Room            created:            ${room.sid}
(${room.uniqueName})`);
    return room;
  }

  generateAccessToken(identity: string, roomName: string) {
    const token = new AccessToken(
      this.accountSid,
      this.apiKeySid,
      this.apiKeySecret,
      { identity }
    );
    const videoGrant = new VideoGrant({ room: roomName });
    token.addGrant(videoGrant);
    return token.toJwt();
  }
}
```

Для створення токенів використовується бібліотека twilio (npm), де формується токен за допомогою access-token. VideoGrant, який прив’язується до певної кімнати та користувача.

На клієнті використовуються компоненти Twilio Video SDK, які дозволяють:

- запитати доступ до камери й мікрофона;

- підключитися до кімнати за токеном;

- виводити відеопотоки всіх учасників на екран;

- керувати мікрофоном/відео (вмикати/вимикати);

- реагувати на події типу “учасник підключився/вийшов”.

Інтерфейс реалізовано у вигляді модального вікна або окремої сторінки. При

запрошенні до дзвінка користувач отримує сповіщення (email чи push) з можливістю приєднатися.

Всі дзвінки захищені через JWT-токени, які мають обмежений термін дії та доступ лише до конкретної кімнати. Це унеможливлює несанкціонований доступ до дзвінків. Кімнати створюються з параметрами uniqueName, maxParticipants, recordParticipantsOnConnect тощо. У разі потреби адміністратор може обмежити можливість створення дзвінків або переглянути журнал дзвінків.

Завдяки використанню Twilio у системі реалізована сучасна, стабільна та безпечна система відео і аудіо зв’язку. Це дозволяє користувачам оперативно обговорювати завдання, ділові зустрічі, що значно підвищує ефективність роботи в межах команди.

На основі проєкту було реалізовано повнофункціональний прототип системи за допомогою сучасного стеку технологій: NestJS, Prisma, MySQL, React і Next.js. Застосовано JWT-автентифікацію, реалізовано багаторівневу взаємодію з хмарним сховищем AWS S3, передбачено рольову модель користувачів. Кожен модуль реалізовано з урахуванням принципів чистої архітектури та безпеки.

# 4 ВИПРОБУВАННЯ РОБОТИ ВЕБСИСТЕМИ

## 4.1 Авторизація/реєстрація

Одним із ключових етапів взаємодії користувача з вебсистемою керування проєктами є процес автентифікації – тобто входу до особистого кабінету. Система реалізує два способи автентифікації: класичний за допомогою електронної пошти та пароля, а також за допомогою стороннього постачальника через OAuth 2.0 (Google та інші). Цей підхід забезпечує швидкість входу та зручність, особливо для корпоративних користувачів, які звикли до інтеграції з інфраструктурою.

На сторінці реєстрації користувач створює обліковий запис за допомогою email та пароля. Після успішної реєстрації – автоматичне перенаправлення на

форму входу або вхід у систему з одразу виданим токеном.

На рівні серверної частини автентифікація реалізована за допомогою JWT. OAuth-автентифікація (Google) реалізована через бібліотеку, що обробляє отримання authorization code, обмін на токени Google і витяг профілю користувача, який додається або оновлюється в базі даних (рисунок 4.1).

Система реалізує наступні заходи безпеки:

- збереження паролів в хешованому вигляді (bcrypt);
- обмеження на кількість спроб входу (анти-брутфорс логіка);
- перевірка токена на кожному запиті до захищених ресурсів;
- автоматичне оновлення access token при використанні валідного refresh token.

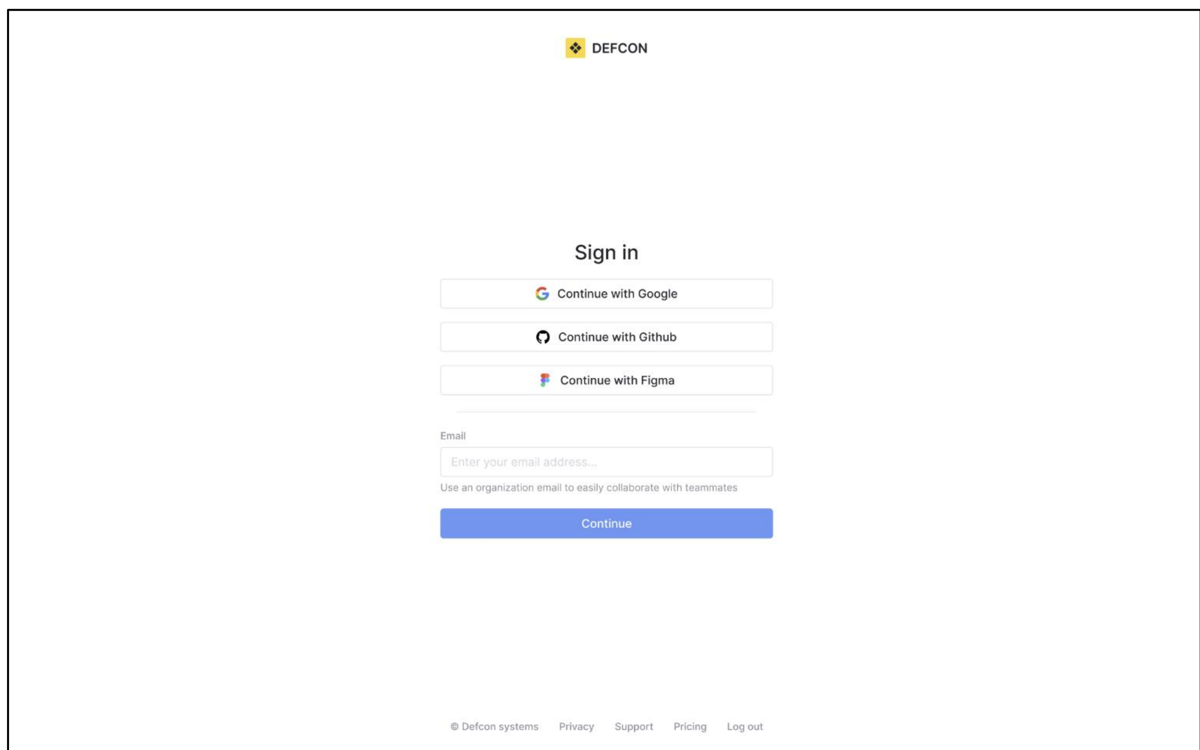



Рисунок 4.1 – Сторінка авторизації

При першій авторизації потрібно ввести додаткові данні задля реєстрації в системі, для цього користувача перекидає на другий екран, вигляд якого вказано на рисунку 4.2

Рисунок 4.2 – Сторінка авторизації

Також при першій реєстрації у діалоговому вікні запитується чи є людина власником команди/компанії та пропонується зареєструвати її задля керування та створення власних проєктів(рисунок 4.3).

Рисунок 4.3 – Сторінка авторизації

Модуль автентифікації реалізовано з урахуванням сучасних вимог до безпеки

та зручності. Поєднання класичного логіну з підтримкою OAuth дозволяє охопити широкий спектр користувачів, а використання JWT забезпечує ефективну та масштабовану модель контролю доступу в межах SPA-додатку. Цей функціонал є базовим для всіх наступних можливостей системи й забезпечує стабільний початок користувацького досвіду.

## 4.2 Сторінка проєктів

Після авторизації користувач потрапляє на основну робочу сторінку – Сторінку проєктів. Цей інтерфейс є центральною панеллю управління, яка дозволяє користувачеві переглядати всі доступні проєкти, створювати нові, а також швидко переходити до перегляду завдань, редагування параметрів проєкту або запрошення учасників. Кожен проєкт відображається у вигляді карточки з наступними елементами:

- назва проєкту;
- короткий опис або теги;
- аватар проєкту (за замовчуванням – іконка з ініціалами);
- кількість завдань або учасників;
- кнопки "Переглянути", "Редагувати", "Запросити учасників";
- індикатор статусу (активний, архівований).

А також у вигляді списку на бічній панелі. Для створення проєкту потрібно заповнити форму наведену на рисунку 4.4.

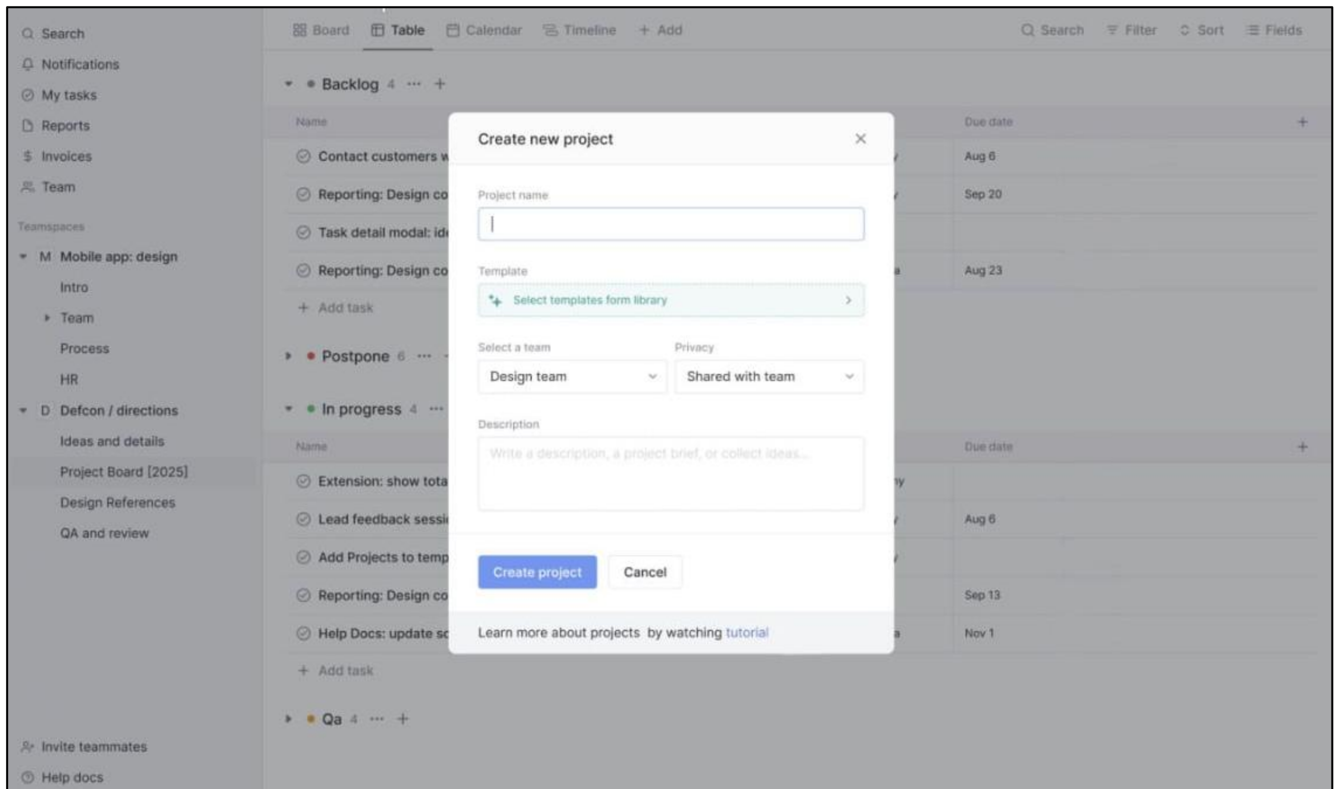



Рисунок 4.4 – Форма створення проєкту

Цей підхід дозволяє швидко орієнтуватися в актуальних проєктах і здійснювати з ними базові дії без додаткових переходів. Залежно від ролі користувача, інтерфейс надає доступ до різних дій:

- адміністратор або менеджер проєкту може створювати нові проєкти, редагувати наявні, призначати ролі, видаляти або архівувати проєкти;
- учасник проєкту має доступ до перегляду, створення та зміни завдань, коментування тощо;
- гість має обмежений за часом та/або функціоналом доступ;
- переглядач має доступ лише до читання вмісту.

Пошук проєктів – реалізовано за допомогою рядка пошуку, який фільтрує список у режимі реального часу. Фільтрація – окремі дозволяють звузити список проєктів.

Завдяки простому, інтуїтивному дизайну користувачі з будь-яким рівнем технічної підготовки можуть легко орієнтуватися в системі. Чіткий поділ на секції, використання кольорових акцентів для статусів і інтерактивні кнопки створюють

приємний досвід.

Сторінка проєктів є точкою входу до всієї подальшої роботи в системі. Вона об’єднує функціональність створення, управління, фільтрації й навігації в одному вікні. Такий підхід дозволяє користувачеві швидко знайти потрібний проєкт, переглянути основну інформацію або почати безпосередню роботу із завданнями.

## 4.3 Список завдань

Сторінка перегляду та керування завданнями є однією з ключових у системі, оскільки основна робота користувачів полягає саме у створенні, призначенні, виконанні та моніторингу завдань у межах проєктів. Інтерфейс реалізований у зручному та інтуїтивному вигляді, щоб забезпечити ефективну командну взаємодію. Інтерфейс дозволяє перемикатися між різними форматами:

- класичний список завдань – у вигляді таблиці або списку з полями: назва, опис, дедлайн, статус, виконавець, пріоритет.

- канбан-дошка – популярний візуальний формат, де завдання переміщуються між колонками відповідно до етапів виконання.

- календарний перегляд – для планування на основі строків виконання (дедлайнів).

Користувачі з відповідними правами можуть виконувати такі дії:

- створення нового завдання – за допомогою модального вікна або окремої сторінки; заповнюються поля назви, опису, термінів, відповідальних осіб, тегів.

- редагування – можливість змінити будь-яке поле, включаючи статус, пріоритет, прикріплені файли.

- видалення – із підтвердженням через модальне вікно.

- призначення користувачів – через випадаючий список або drag&drop у колонках канбану.

- коментування завдання – текстова взаємодія між учасниками, зберігається

історія обговорень.

- сортування завдань за різними критеріями: термінами, пріоритетом, статусом.

- пошук по ключових словах.

- індикатори виконання – візуально показують прогрес завдання або проєкту.

- теги й кольорові мітки – для гнучкого групування завдань.

Приклад з календарним переглядом наведено на рисунку 4.5.

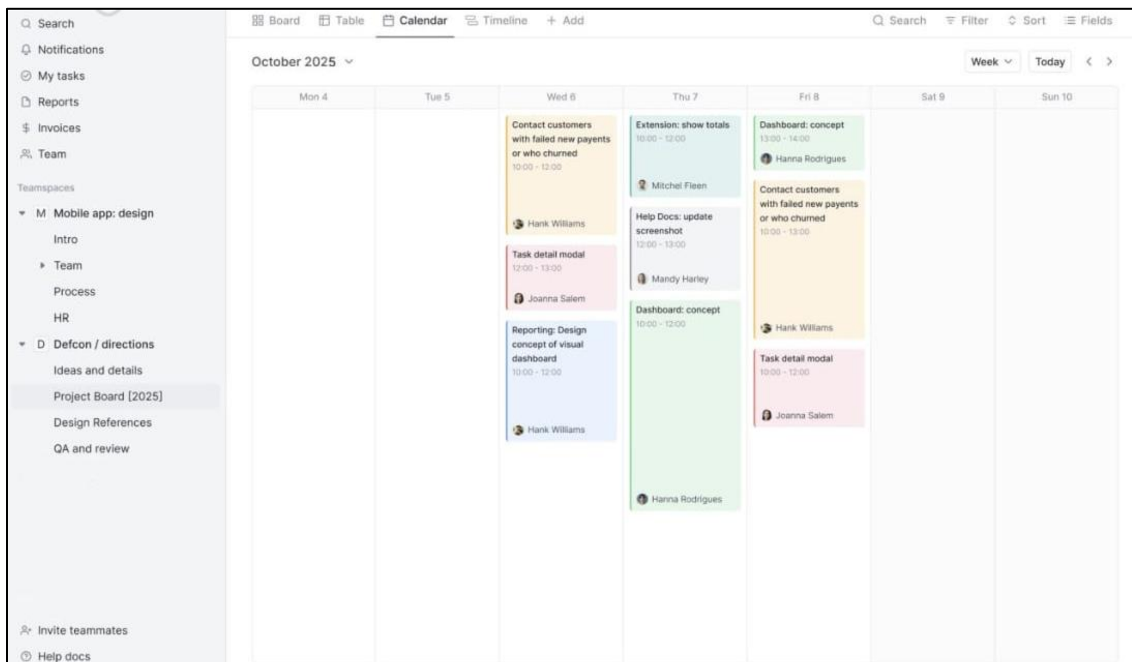



Рисунок 4.5 – Календарний перегляд задач

Сторінка керування завданнями створена таким чином, щоб забезпечити зручний доступ до всіх інструментів командної взаємодії. Гнучкий формат перегляду, розширена система фільтрації, можливість взаємодії в реальному часі через коментарі та зміни статусу – усе це робить управління завданнями швидким, прозорим і зручним.

## 4.4 Коментування та перегляд історії коментарів

У сучасних системах керування проєктами особливу роль відіграє функціональність коментування та перегляду історії змін. Це дозволяє учасникам команди вести комунікацію безпосередньо в межах завдань, залишати важливі уточнення, вести обговорення, фіксувати рішення, а також відстежувати зміни, що відбувалися з кожним завданням. У реалізованій вебсистемі ці можливості були впроваджені як частина модуля взаємодії користувачів із завданнями та проєктами.

Кожне завдання в системі має вбудовану панель коментарів(рисунок 4.5), доступну в нижній частині сторінки перегляду завдання або в окремій вкладці. Основні можливості модуля:

Додавання текстового коментаря – користувач вводить повідомлення у відповідне текстове поле та надсилає його кнопкою або за допомогою комбінації клавіш. Відображення коментарів у хронологічному порядку – нові коментарі автоматично з’являються внизу, з позначенням часу та автора.

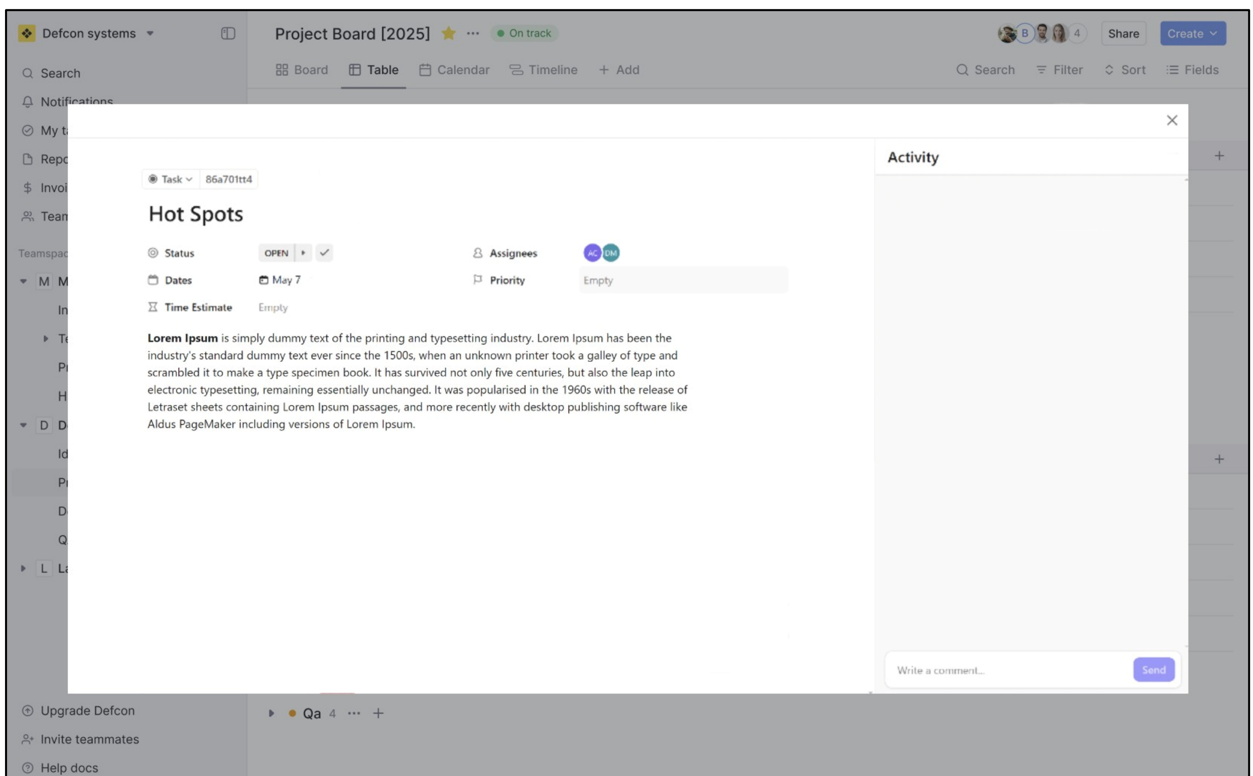



Рисунок 4.5 – Сторінка задачі з полем коментарів

Позначення користувачів через @ – підтримується можливість згадування інших учасників проєкту, що генерує повідомлення на email або push-сповіщення.

Редагування/видалення власних коментарів – доступно впродовж певного часу або без обмеження (залежно від ролі користувача).

Розширення коментарів мультимедіа – можна прикріпити файл, посилання, або додати зображення для більшої наочності.

Формат коментарів максимально спрощено, що дозволяє швидко реагувати на зміни та комунікувати з командою без потреби в сторонніх месенджерах чи пошті.

Механізм коментування реалізовано як інструмент колективної пам'яті та прозорості роботи в проєкті. Він значно підвищує ефективність взаємодії, дозволяє не втрачати контекст завдань та фіксувати всі важливі дії для подальшого аналізу чи перевірки. У комплексі з іншими інструментами системи, ці можливості формують надійний робочий простір для командної співпраці.

### 4.5 Перегляд, додавання та редагування файлів

У системах керування проєктами часто виникає потреба обмінюватися файлами: технічними завданнями, макетами, документами, медіафайлами тощо. Для цього у реалізованій вебсистемі впроваджено функціональність завантаження, перегляду та управління файлами, пов'язаними із завданнями або проєктами.

Файли можна додавати як у контексті конкретного завдання, так і на рівні цілого проєкту або прикріпляти у чаті в конжному інтерфейсі є стандартна іконка канцелярської скріпки, яка відкриває вікно обирання файлу(рисунки 4.7 та 4.8). Інтерфейс реалізовано у вигляді окремої вкладки або секції в картці завдання, яка містить:

- кнопку «Додати файл», що відкриває вікно вибору локального файлу;
- список завантажених файлів з інформацією про назву, тип, дату додавання,

розмір та автора;

- опції для перегляду, завантаження або видалення файлу (залежно від ролі користувача);

- можливість попереднього перегляду зображень або PDF-документів без потреби завантаження.

Рисунок 4.7 – Поле коментарю з можливістю відправити файл

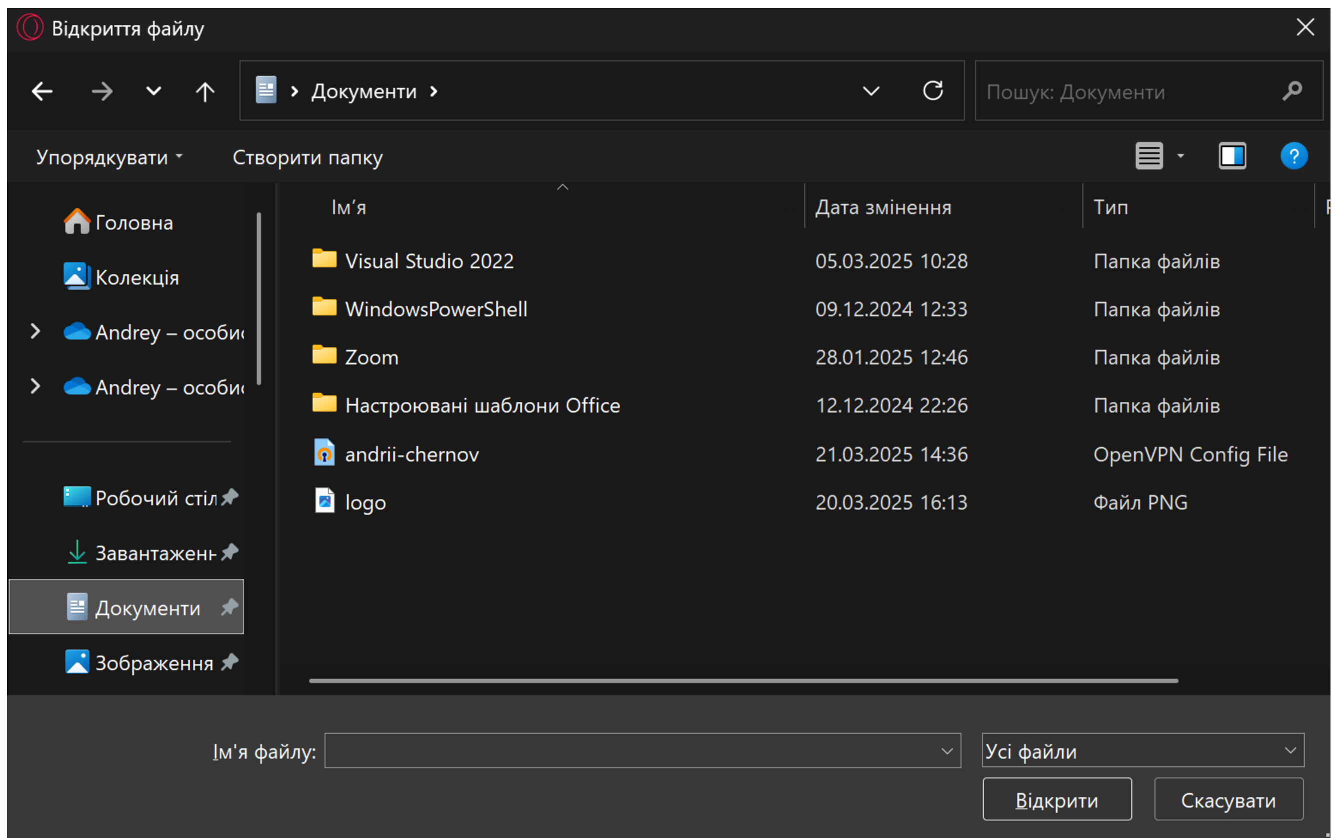



Рисунок 4.8 – Меню обирання файлу

Функціональність завантаження та перегляду файлів є невіддільною частиною вебсистеми, що значно спрощує роботу з проєктною інформацією.

Інтеграція з Amazon S3 дозволила забезпечити надійне та масштабоване зберігання даних, а зручний інтерфейс – зробив взаємодію з файлами інтуїтивною та ефективною. Така система забезпечує прозорість і контроль документообігу в команді.

### 4.6 Чати та комунікація між учасниками

Ефективна комунікація між учасниками проєкту є ключовим фактором успішного виконання завдань. У реалізованій вебсистемі керування проєктами інтегровано зручний інтерфейс чату, що дозволяє оперативно обмінюватися повідомленнями в межах проєкту або конкретного завдання(рисунок 4.9). Завдяки цьому зменшується потреба у зовнішніх месенджерах і забезпечується централізація всієї проєктної інформації.

Основні можливості включають:

- поле введення повідомлення з підтримкою форматування тексту та вставки емодзі;

- вивід історії повідомлень у вигляді діалогового списку з аватарками, іменами авторів, мітками часу та індикацією прочитаного;

- реакції на повідомлення (наприклад, лайк або емодзі), що допомагають швидко дати зворотний зв’язок без написання відповіді;

- можливість згадувати користувачів через @, що сприяє кращій адресності повідомлень;

- система сповіщень про нові повідомлення як у вебінтерфейсі, так і через email або push-повідомлення.

- для кожного завдання чат зберігає повну історію обговорення, що дозволяє новим учасникам швидко ознайомитись із контекстом. Повідомлення не видаляються без адміністративних прав, що гарантує прозорість комунікації.

Технічно, функціональність чату реалізовано з використанням WebSocket

(через бібліотеку socket.io на клієнті та сервері), що забезпечує обмін повідомленнями в реальному часі. Дані зберігаються у базі MySQL, і кожне повідомлення пов’язане з конкретним завданням або проєктом через відповідні зовнішні ключі.

Система пройшла комплексне випробування з використанням ручних сценаріїв. Перевірено коректність основних функцій: створення проєктів, управління завданнями, завантаження файлів, організація зустрічей тощо. Проведено перевірку навантаження та відповідності вимогам безпеки. Виявлені помилки були усунуті, що забезпечило стабільну й надійну роботу системи.

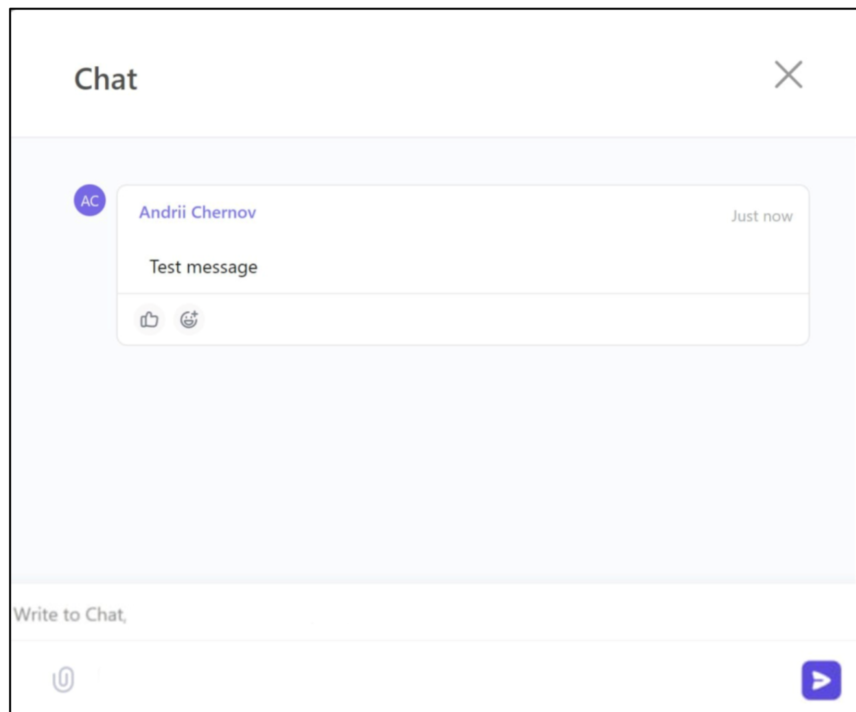



Рисунок 4.9 – Вікно чату

Для додаткової гнучкості передбачено налаштування: користувач може увімкнути або вимкнути звукові сповіщення, налаштувати частоту email-сповіщень або навіть тимчасово вимкнути чат для певних задач, якщо потрібна тиша.

Комунікаційний модуль став важливим елементом взаємодії в команді, дозволяючи обговорювати завдання безпосередньо в контексті їх виконання. Такий підхід мінімізує втрату інформації, зменшує кількість перемикань між додатками та сприяє підвищенню продуктивності команди.

# ВИСНОВКИ

У межах дипломної роботи було реалізовано повнофункціональну вебсистему керування проєктами, яка дозволяє ефективно організовувати спільну роботу над завданнями, контролювати процеси розробки, комунікувати між учасниками команди та інтегрувати сторонні сервіси для покращення користувацького досвіду. А також здобуто цінний практичний досвід розробки повноцінної вебсистеми з нуля, застосовано сучасні технології та принципи програмування, інтегровано зовнішні сервіси. Це не лише дозволило створити корисний програмний продукт, а й поглибити знання в галузі інженерії програмного забезпечення, веброзробки та проєктного менеджменту.

Розроблено архітектуру вебсистеми, яка враховує багаторівневий підхід до обробки даних та взаємодії з користувачем. Спроєктовано структуру бази даних, реалізовано діаграми класів, компонентів і послідовності, що описують основну логіку взаємодії системи. Було враховано принципи модульності, повторного використання коду та розширюваності, що дозволяє легко адаптувати систему до нових вимог.

На основі проєкту було реалізовано повнофункціональний прототип системи за допомогою сучасного стеку технологій: NestJS, Prisma, MySQL, React і Next.js. Застосовано JWT-автентифікацію, реалізовано багаторівневу взаємодію з хмарним сховищем AWS S3, передбачено рольову модель користувачів. Кожен модуль реалізовано з урахуванням принципів чистої архітектури та безпеки.

Подальші напрямки розвитку:

- створення мобільної версії системи на Flutter;
- розбиття серверного коду на мікро-сервіси для подальшої дуплікації та підвищення надійності;
- розширення API для інтеграції з іншими сервісами;
- оптимізація продуктивності під високе навантаження.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ


1. NestJS – Офіційна документація. NestJS. URL: https://docs.nestjs.com (дата звернення: 24.05.2025).

2. Prisma – Офіційна документація. Prisma. URL: https://www.prisma.io/docs (дата звернення: 25.05.2025).

3. MySQL – Офіційний сайт. MySQL. URL: https://www.mysql.com (дата звернення: 28.05.2025).

4. Next.js – Документація. Next.js. URL: https://nextjs.org/docs (дата звернення: 25.05.2025).

5. React – Документація. React. URL: https://react.dev/learn (дата звернення: 26.05.2025).

6. Tailwind CSS – Документація. Tailwind CSS. URL: https://tailwindcss.com/docs (дата звернення: 27.05.2025).

7. TypeScript – Офіційна документація. TypeScript. URL: https://www.typescriptlang.org/docs (дата звернення: 24.05.2025).

8. JWT – JSON Web Tokens. URL: https://jwt.io (дата звернення: 25.05.2025).

9. Google Mail API – Документація. Google Developers. URL: https://developers.google.com/gmail/api (дата звернення: 28.05.2025).

10. Amazon S3 – Документація. AWS Documentation. URL: https://docs.aws.amazon.com/s3 (дата звернення: 27.05.2025).

11. Twilio Video – Офіційна документація. Twilio. URL: https://www.twilio.com/docs/video (дата звернення: 24.05.2025).

12. OAuth 2.0 – Опис протоколу. OAuth. URL: https://oauth.net/2 (дата звернення: 26.05.2025).

13. Postman – Офіційний сайт. Postman. URL: https://www.postman.com (дата звернення: 27.05.2025).

14. GitHub – Платформа для розробки. GitHub. URL: https://github.com (дата звернення: 28.05.2025).

15. OWASP Top Ten – Рекомендації з безпеки вебзастосунків. OWASP. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 25.05.2025).

16. ESLint – Інструмент для аналізу коду JavaScript/TypeScript. ESLint. URL: https://eslint.org (дата звернення: 26.05.2025).

17. Prettier – Форматувальник коду. Prettier. URL: https://prettier.io (дата звернення: 24.05.2025).

18. Zod – TypeScript-first схема валідації. Zod. URL: https://zod.dev (дата звернення: 27.05.2025).

19. React Hook Form – Бібліотека для роботи з формами у React. React Hook Form. URL: https://react-hook-form.com (дата звернення: 28.05.2025).

20. Nodemailer – Модуль для надсилання email у Node.js. Nodemailer. URL: https://nodemailer.com (дата звернення: 26.05.2025).

# ДОДАТОК А
# ЛІСТИНГИ ПРОГРАМ

Лістинг А.1 – Файл user.service.ts

```
import { Injectable } from '@nestjs/common'
import { User } from '@prisma/client'
import { CreateUserDto } from 'src/dtos'
import { PrismaService } from '../prisma.service'

@Injectable()
export class UsersService {
  constructor(private prisma: PrismaService){}

  async createUser(dto: CreateUserDto){
    const user = await this.prisma.user.create({
      data: {
        email: dto.email,
        passwordHash: dto.password
      }
    })
    return user;
  }

  async updateUser(id: string, data: Partial<User>): Promise<User> {
    return this.prisma.user.update({
      where: { id },
      data,
    });
  }

  async getUserByField(field: { id?: string; email?: string;
username?: string, resetPasswordToken?: string,
emailVerificationToken?: string}): Promise<User | null> {
    return this.prisma.user.findFirst({
      where: field,
    });
  }
}
```

Лістинг А.2 – Файл auth.service.ts

```
import { ForbiddenException, Injectable } from '@nestjs/common'
import { ConfigService } from '@nestjs/config'
import { JwtService } from '@nestjs/jwt'
import * as bcrypt from 'bcrypt'
import { AuthDto, CreateUserDto } from 'src/dtos'
import { JwtPayload, Tokens } from 'src/types'
import { UsersService } from 'src/users/users.service'
```

```
import { MailService } from './mail.service'
@Injectable()
export class AuthService {
  constructor(
    private jwtService: JwtService,
    private config: ConfigService,
    private userService: UsersService,
    private mailService: MailService,
  ) {}

async signup(dto: CreateUserDto): Promise<Tokens> {
  const emailExists = await this.userService.getUserByField({ email:
dto.email });
  if (emailExists) throw new ForbiddenException('Email already in
use.');

  dto.password = await this.hashData(dto.password);
  const verificationToken = crypto.randomUUID();
  const newUser = await this.userService.createUser({
    ...dto,
    emailVerificationToken: verificationToken,
  });

  await this.mailService.sendEmailConfirmation(newUser.email,
verificationToken);

  const { passwordHash, ...jwtPayload } = newUser;
  const tokens = await this.getTokens(jwtPayload);
  return tokens;
}

async confirmEmail(token: string) {
  const user = await this.userService.getUserByField({
emailVerificationToken: token });
  if (!user) throw new ForbiddenException('Invalid or expired token');

  await this.userService.updateUser(user.id, {
    emailVerified: true,
    emailVerificationToken: null,
  });
}

async forgotPassword(email: string) {
  const user = await this.userService.getUserByField({ email });
  if (!user) return;

  const token = crypto.randomUUID();
  await this.userService.updateUser(user.id, { resetPasswordToken:
token });
  await this.mailService.sendPasswordReset(email, token);
}

async resetPassword(token: string, newPassword: string) {
```

```
    const    user    =    await    this.userService.getUserByField({
  resetPasswordToken: token });
    if (!user) throw new ForbiddenException('Invalid token');

    const passwordHash = await this.hashData(newPassword);
    await this.userService.updateUser(user.id, {
      passwordHash,
      resetPasswordToken: null,
    });
  }

    async signin(dto: AuthDto): Promise<Tokens> {
      const  user  =  await  this.userService.getUserByField({  email:
  dto.email });
      const   passwordMatches   =   await   bcrypt.compare(dto.password,
  user.passwordHash);

      if  (!passwordMatches)  throw  new  ForbiddenException('Incorrect
  password.');

      const { passwordHash, ...jwtPayload } = user;
      const tokens = await this.getTokens(jwtPayload);
      return tokens;
    }

    async oauthSignin(email: string): Promise<Tokens> {
      let user = await this.userService.getUserByField({ email });

      if (!user) {
        user = await this.userService.createUser({
          email,
          password: null,
        });
      }

      const { passwordHash, ...jwtPayload } = user;
      const tokens = await this.getTokens(jwtPayload);
      return tokens;
    }


    async refresh(id: string): Promise<Tokens>{
      const user = await this.userService.getUserByField({id});
      if (!user || !id) throw new ForbiddenException('User not found.');

      const { passwordHash, ...jwtPayload } = user;
      return this.getTokens(jwtPayload);
    }

    hashData(data: string) {
      return bcrypt.hash(data, 10)
    }
```

```
  async getTokens(user: JwtPayload): Promise<Tokens> {
    const [at, rt] = await Promise.all([
      this.jwtService.signAsync(user, {
        secret: this.config.get<string>('AT_SECRET'),
        expiresIn: '30m',
      }),
      this.jwtService.signAsync(user, {
        secret: this.config.get<string>('RT_SECRET'),
        expiresIn: '7d',
      }),
    ]);

    return {
      access_token: at,
      refresh_token: rt,
    };
  }
}
```

Лістинг А.3 – Файл mail.service.ts

```
import { Injectable } from '@nestjs/common'
import * as nodemailer from 'nodemailer'

@Injectable()
export class MailService {
  private transporter = nodemailer.createTransport({
    service: 'gmail',
    auth: {
      user: process.env.GMAIL_USER,
      pass: process.env.GMAIL_PASS,
    },
  });

  async sendEmailConfirmation(to: string, token: string) {
    const     confirmUrl     =     `${process.env.CLIENT_URL}/confirm-
email?token=${token}`;
    const template = this.getEmailVerificationTemplate(confirmUrl);
    await this.transporter.sendMail({
      to,
      subject: template.subject,
      html: template.html,
    });
  }

  async sendPasswordReset(to: string, token: string) {
    const      resetUrl      =      `${process.env.CLIENT_URL}/reset-
password?token=${token}`;
    const template = this.getPasswordResetTemplate(resetUrl);
    await this.transporter.sendMail({
      to,
      subject: template.subject,
      html: template.html,
```

```
    });
  }

  // Template: Password Reset
  getPasswordResetTemplate(resetUrl: string) {
    return {
      subject: 'Reset your password',
      html: `<p>You requested a password reset.</p>
             <p><a href="${resetUrl}">Click here to reset your
password</a>.</p>`
    };
  }

    // Template: Email Verification
  getEmailVerificationTemplate(confirmUrl: string) {
    return {
      subject: 'Email Verification',
      html: `<p>Thank you for registering!</p>
             <p>Please <a href="${confirmUrl}">confirm your email
address</a>.</p>`
    };
  }

  // Template: Project Invitation
  getProjectInvitationTemplate(projectName: string, inviter: string)
{
    return {
      subject: `You've been added to the project "${projectName}"`,
      html: `<p>${inviter} has invited you to a new project:
<b>${projectName}</b>.</p>
             <p>Check your project list for more details.</p>`
    };
  }

  // Template: New Task
  getNewTaskTemplate(taskTitle: string, projectName: string,
assigner: string) {
    return {
      subject: `New Task: "${taskTitle}"`,
      html: `<p>You have been assigned a new task
"<b>${taskTitle}</b>" in the project "${projectName}".</p>
             <p>Assigned by: ${assigner}.</p>`
    };
  }

  // Template: Task Comment
  getTaskCommentTemplate(taskTitle: string, commenter: string,
comment: string) {
    return {
      subject: `New comment on task "${taskTitle}"`,
      html: `<p>User ${commenter} left a comment on the task
"<b>${taskTitle}</b>":</p>
             <blockquote>${comment}</blockquote>`
```

```
    };
  }

  // Template: Deadline Reminder
  getDeadlineReminderTemplate(taskTitle: string, deadline: string) {
    return {
      subject: `Deadline approaching for task "${taskTitle}"`,
      html: `<p>This is a reminder that the deadline for the task
"<b>${taskTitle}</b>" is set for ${deadline}.</p>
             <p>Don't forget to complete the task on time!</p>`
    };
  }
}
```

Лістинг А.4 – Файл at.strategy.ts

```
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { JwtPayload } from 'src/types';

@Injectable()
export class AtStrategy extends PassportStrategy(Strategy, 'jwt') {
  constructor(config: ConfigService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: config.get<string>('AT_SECRET'),
    });
  }

  validate(payload: JwtPayload) {
    return payload;
  }
}
```

Лістинг А.5 – Файл rt.strategy.ts

```
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { Request } from 'express';
import { ForbiddenException, Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { JwtPayload, JwtPayloadWithRt } from 'src/types';

@Injectable()
export class RtStrategy extends PassportStrategy(Strategy, 'jwt-
refresh') {
  constructor(config: ConfigService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: config.get<string>('RT_SECRET'),
```

```
      passReqToCallback: true,
    });
  }
  validate(req: Request, payload: JwtPayload): JwtPayloadWithRt {
    const refreshToken = req
      ?.get('authorization')
      ?.replace('Bearer', '')
      .trim();

    if (!refreshToken) throw new ForbiddenException('Refresh token malformed');

    return {
      ...payload,
      refreshToken,
    };
  }
}
```

Лістинг А.6 – Файл project.service.ts

```
import { Injectable } from '@nestjs/common'
import { ProjectRole } from '@prisma/client'
import { CreateProjectDto, UpdateProjectDto } from 'src/dtos'
import { PrismaService } from '../prisma.service'

@Injectable()
export class ProjectsService {
  constructor(private prisma: PrismaService) {}

  async getUserProjects(userId: string) {
    return this.prisma.project.findMany({
      where: {
        participants: {
          some: {
            userId: userId,
          },
        },
      },
      include: {
        participants: {
          include: {
            user: {
              select: {
                id: true,
                email: true,
                role: true,
              },
            },
          },
        },
      },
    });
```

```
  }

  async createProject(data: CreateProjectDto, userId: string) {
    return this.prisma.project.create({
      data: {
        name: data.name,
        description: data.description,
        status: data.status,
        createdAt: new Date(),
        owner: {
          connect: { id: userId },
        },
        participants: {
          create: [
            {
              userId: userId,
              role: 'OWNER',
            },
          ],
        },
      },
      include: {
        participants: true,
      },
    });
  }

  async  addParticipantToProject(projectId:  string,  userId:  string,
role: ProjectRole = 'MEMBER') {
  const exists = await this.prisma.projectParticipant.findUnique({
    where: {
      userId_projectId: {
        userId,
        projectId,
      },
    },
  });

  if (exists) {
    throw new Error('Користувач уже є учасником цього проєкту');
  }

  return this.prisma.projectParticipant.create({
    data: {
      userId,
      projectId,
      role,
    },
  });
}


  async updateProject(id: string, data: UpdateProjectDto) {
```

```
    return this.prisma.project.update({
      where: { id },
      data,
    });
  }

  async deleteProject(id: string) {
    return this.prisma.project.update({
      where: { id },
      data:{
        status: "ARCHIVED"
      },
    });
  }
}
```

Лістинг А.7 – Файл task.service.ts

```
import { Injectable } from '@nestjs/common'
import { CreateTaskDto, UpdateTaskDto } from 'src/dtos'
import { PrismaService } from '../prisma.service'

@Injectable()
export class TasksService {
  constructor(private prisma: PrismaService) {}

  async getProjectTasks(projectId: string) {
    return this.prisma.task.findMany({
      where: {
        projectId: projectId,
      },
    });
  }

  async createTask(data: CreateTaskDto) {
    return this.prisma.task.create({ data });
  }

  async updateTask(id: string, data: UpdateTaskDto) {
    return this.prisma.task.update({
      where: { id },
      data,
    });
  }

  async deleteTask(id: string) {
    return this.prisma.task.update({
      where: { id },
      data:{
        status: "ARCHIVED"
      },
    });
  }
```

```
}
```

Лістинг А.8 – Файл comments.service.ts

```
import { Injectable } from '@nestjs/common';
import { CreateCommentDto, UpdateCommentDto } from 'src/dtos';
import { PrismaService } from '../prisma.service';


@Injectable()
export class CommentsService {
  constructor(private prisma: PrismaService) {}

  async getTaskComments(taskId: string) {
    return this.prisma.comment.findMany({
      where: {
        taskId: taskId,
      },
    });
  }

  async createComment(data: CreateCommentDto) {
    const {taskId, authorId, ...commentData} = data; // Assuming
authorId is not needed in the comment creation
    return this.prisma.comment.create({
      data: {
        ...commentData,
        author: { connect: { id: authorId } },
        task: { connect: { id: taskId } },
      },
    });
  }

  async updateComment(id: string, data: UpdateCommentDto) {
    return this.prisma.comment.update({
      where: { id },
      data,
    });
  }

  async deleteComment(id: string) {
    return this.prisma.comment.delete({ where: { id } });
  }
}
```

Лістинг А.1 – Файл attachments.service.ts

```
import { Injectable } from '@nestjs/common'
import { PrismaService } from 'src/prisma.service'

@Injectable()
export class AttachmentsService {
  constructor(
    private prisma: PrismaService
```

```
  ) {}
  async create(dto: { fileName: string; url: string; taskId: string
}) {
    return this.prisma.attachment.create({
      data: {
        fileName: dto.fileName,
        url: dto.url,
        taskId: dto.taskId
      }
    });
  }

  async findByTask(taskId: string) {
    return this.prisma.attachment.findMany({
      where: { taskId },
      orderBy: { uploadedAt: 'desc' }
    });
  }

  async delete(id: string) {
    return this.prisma.attachment.delete({
      where: { id }
    });
  }
}
```

Лістинг А.1 – Файл aws-s3.service.ts

```
import { Injectable } from '@nestjs/common'
import { S3 } from 'aws-sdk'

@Injectable()
export class AwsS3Service {
  private s3 = new S3({
    region: process.env.AWS_REGION,
    credentials: {
      accessKeyId: process.env.AWS_ACCESS_KEY_ID,
      secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY
    }
  });

  async getPresignedUrl(fileName: string, fileType: string) {
    const key = `attachments/${Date.now()}-${fileName}`;

    const url = this.s3.getSignedUrl('putObject', {
      Bucket: process.env.AWS_BUCKET_NAME,
      Key: key,
      ContentType: fileType,
      Expires: 600,
    });

    return {
      url,
```

```
      key,
      publicUrl:
`https://${process.env.AWS_BUCKET_NAME}.s3.${process.env.AWS_REGION}
.amazonaws.com/${key}`
    };
  }
}
```

Лістинг А.11 – Файл chat.service.ts

```
import { Injectable } from '@nestjs/common'
import { CreateChatDto, UpdateChatDto } from 'src/dtos/chat/chat.dto'
import { PrismaService } from 'src/prisma.service'

@Injectable()
export class ChatService {
  constructor(private prisma: PrismaService) {}

  async create(createChatDto: CreateChatDto) {
    return this.prisma.chat.create({
      data: {
        title: createChatDto.title,
        description: createChatDto.description,
                    project: { connect: { id: createChatDto.projectId
} }
      }
    });
  }

  async findAll() {
    return this.prisma.chat.findMany();
  }

  async findOne(id: string) {
    return this.prisma.chat.findUnique({ where: { id } });
  }

  async update(id: string, updateChatDto: UpdateChatDto) {
    return this.prisma.chat.update({
      where: { id },
      data: updateChatDto,
    });
  }

  async remove(id: string) {
    return this.prisma.chat.delete({ where: { id } });
  }
}
```

Лістинг А.12 – Файл message.service.ts

```
import { Injectable } from '@nestjs/common'
import     {     CreateMessageDto,     UpdateMessageDto     }     from
```

```
'src/dtos/chat/message.dto'
import { PrismaService } from 'src/prisma.service'

@Injectable()
export class MessageService {
  constructor(private prisma: PrismaService) {}

  async create(dto: CreateMessageDto) {
    return this.prisma.message.create({
      data: {
        text: dto.text,
        chatId: dto.chatId,
        authorId: dto.authorId,
      },
      include: {
        author: true,
      },
    });
  }

  async findAllByChat(chatId: string) {
    return this.prisma.message.findMany({
      where: { chatId },
      orderBy: { createdAt: 'asc' },
      include: {
        author: true,
      },
    });
  }

  async update(id: string, dto: UpdateMessageDto) {
    return this.prisma.message.update({
      where: { id },
      data: dto,
    });
  }

  async remove(id: string) {
    return this.prisma.message.delete({ where: { id } });
  }
}
```

Лістинг А.13 – Файл meeting.service.ts

```
import { Injectable, Logger } from '@nestjs/common'
import { Twilio } from 'twilio'
import AccessToken from 'twilio/lib/jwt/AccessToken'
import { v4 as uuidv4 } from 'uuid'

const { VideoGrant } = AccessToken;

@Injectable()
```

```
export class TwilioService {
  private readonly logger = new Logger(TwilioService.name);
  private readonly twilioClient: Twilio;

  constructor(
    private        readonly        accountSid:        string        =
process.env.TWILIO_ACCOUNT_SID!,
    private        readonly        authToken:        string        =
process.env.TWILIO_AUTH_TOKEN!,
    private        readonly        apiKeySid:        string        =
process.env.TWILIO_API_KEY_SID!,
    private        readonly        apiKeySecret:        string        =
process.env.TWILIO_API_KEY_SECRET!,
  ) {
    this.twilioClient = new Twilio(this.accountSid, this.authToken);
  }

  async createRoom(roomName?: string) {
    const uniqueRoomName = roomName || `room-${uuidv4()}`;
    const room = await this.twilioClient.video.rooms.create({
      uniqueName: uniqueRoomName,
      type: 'group',
    });
    this.logger.log(`Room            created:            ${room.sid}
(${room.uniqueName})`);
    return room;
  }

  generateAccessToken(identity: string, roomName: string) {
    const token = new AccessToken(
      this.accountSid,
      this.apiKeySid,
      this.apiKeySecret,
      { identity }
    );
    const videoGrant = new VideoGrant({ room: roomName });
    token.addGrant(videoGrant);
    return token.toJwt();
  }
}
```

# РОЗРОБКА ВЕБСИСТЕМИ КЕРУВАННЯ ПРОЄКТАМИ

Національний університет “Запорізька політехніка”
Кафедра комп’ютерних систем та мереж
Виконавець: Студент групи КНТ-512сп Чернов Андрій Олексійович
Керівник: Скрупський Степан Юрійович
Рік захисту: 2025

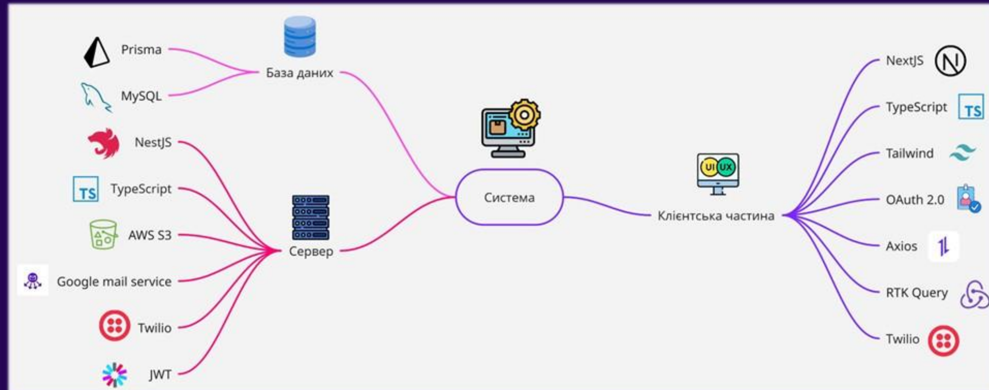
# Мета та Завдання

Мета: Розробити сучасну вебсистему для управління командною проєктною діяльністю з можливістю комунікації та обміну файлами

Завдання:

- створити зручний інтерфейс;
- забезпечити авторизацію та рольову модель користувачів;
- реалізувати модулі керування проєктами та завданнями;
- забезпечити завантаження та перегляд файлів;
- реалізувати відео/аудіозустрічі через Twilio;
- інтегрувати email-розсилку через Gmail API.

# Загальна архітектура системи




Клієнт: Next.js + TypeScript, Tailwind CSS, Twilio

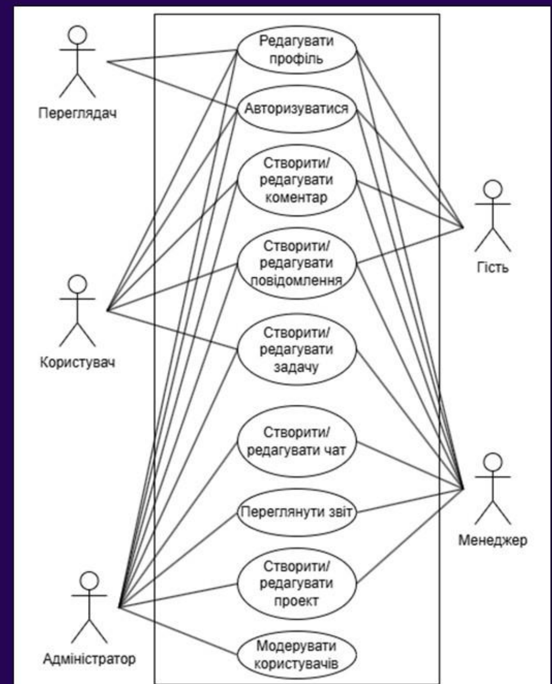
Сервер: NestJS + TypeScript, NextAuth(Oauth 2.0), Axios, RTK Query, REST API, JWT

База даних: MySQL через Prisma ORM

Сервіси: AWS S3, Gmail API, Twilio

# Використання

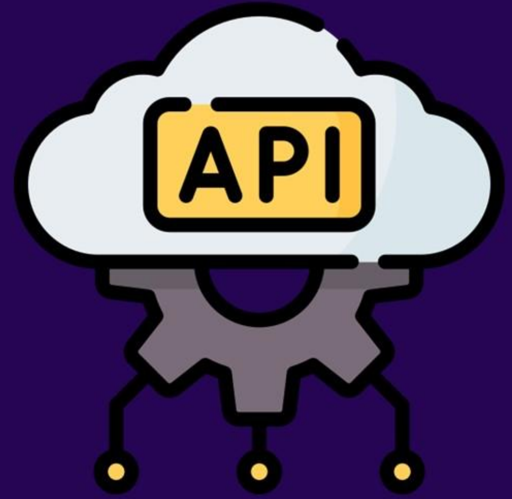
Доступ до функціоналу обмежений ролями. Основні ролі, а саме: переглядач, гість, користувач, менеджер, адміністратор, а також основний функціонал, який їм доступний представлений на діаграмі варіантів використання.



# Клієнт серверна взаємодія

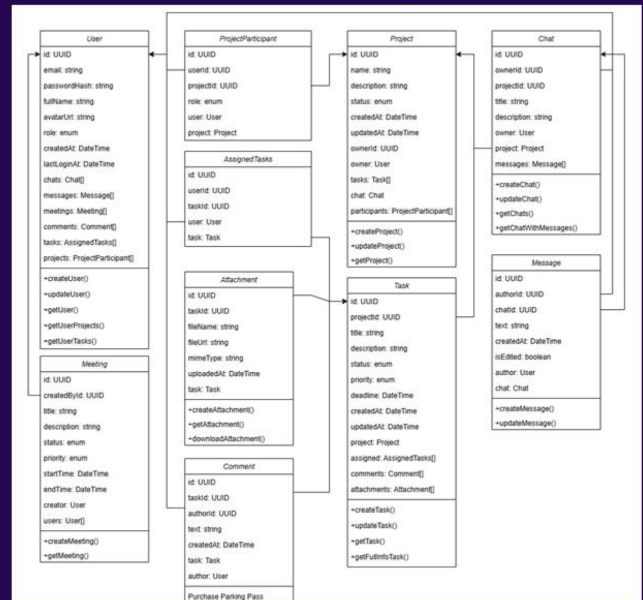
Взаємодія через REST API:

- клієнт надсилає HTTP-запити (GET, POST, PATCH, DELETE) до серверу;
- сервер обробляє запити через контролери NestJS;
- дані передаються у вигляді JSON;
- сесії не зберігаються — використовується JWT для авторизації;
- запити до API перевіряються guard'ами й middleware;
- результати повертаються клієнту як відповіді API з кодами статусу.




# Основні класи та їх архітектура

- User – зберігає користувача та їх ролі;
- Project – інформація про проект та всі відносини до нього;
- Task – інформація про задачу та до якого проекту вона відноситься;
- Comment – інформація про коментар та автора;
- Chat – інформація про чат;
- Message – інформація про повідомлення та автора;
- Attachment – зберігає ім'я файлу та URL для доступу до нього з S3 Bucket;
- Meeting – з інформацією про дзвінок та його учасників.



# Системні вимоги

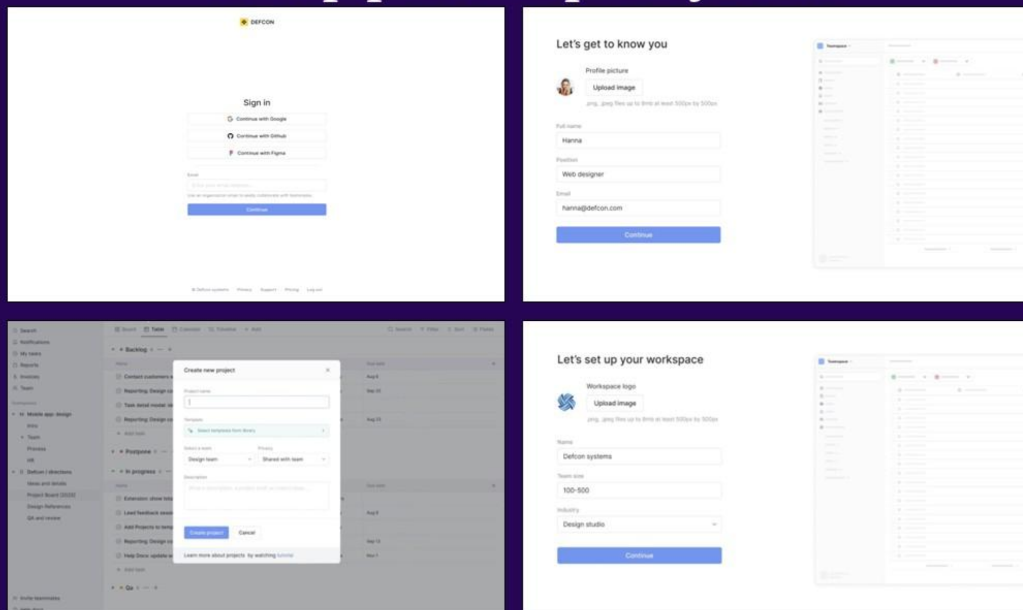
Апаратні вимоги

| Параметр | Мінімальні вимоги | Рекомендовані вимоги |
| --- | --- | --- |
| Процесор | Двоядерний 1.8 ГГц | Чотириядерний 2.4 ГГц і вище |
| Оперативна пам'ять | 2 ГБ | 4–8 ГБ |
| Розширення екрану | 1280×720 | 1920×1080 |
| Відеокамера/Мікрофон | Вебкамера і мікрофон (опціонально) | HD-камера і шумопоглинаючий мікрофон |
| Інтернет | 5 Мбіт/с стабільне з'єднання | 20+ Мбіт/с для відеозустрічей |

Програмні вимоги

| Параметр | Мінімальні вимоги | Рекомендовані вимоги |
| --- | --- | --- |
| Операційна система | Windows 8.1 / macOS 10.13 / Linux | Windows 10+, macOS 12+, Ubuntu 22.04 |
| Браузер | Google Chrome 90+, Firefox 88+, Edge | Остання стабільна версія Chrome або Brave |
| Підтримка JavaScript | Увімкнено | Увімкнено |
| Підтримка WebRTC (дзвінки) | Так | Так |

# Інтерфейс користувача



# Результати розробки та перспективи розвитку

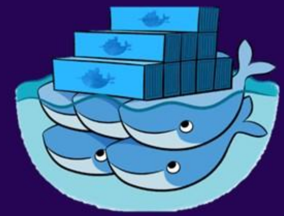
Розроблено:

- повнофункціональну систему управління проєктами;
- підтримка командної роботи з ролями;
- інтеграції з поштою, файлами та дзвінками;
- безпечну авторизацію;

Підтверджено працездатність шляхом тестування.

Перспективи:

- створення мобільної версії системи на Flutter;
- розбиття серверного коду на мікро-сервіси для подальшої дуплікації та підвищення надійності;
- розширення API для інтеграції з іншими сервісами;
- оптимізація продуктивності під високе навантаження.



# Висновки

У межах дипломної роботи було реалізовано повнофункціональний прототип системи керування проєктами.

Розробка виконувалась за допомогою сучасного стеку технологій: NestJS, Prisma, MySQL, React і Next.js.

Застосовано JWT-автентифікацію, реалізовано багаторівневу взаємодію з хмарним сховищем AWS S3, передбачено рольову модель користувачів.

Кожен модуль реалізовано з урахуванням принципів чистої архітектури та безпеки.

Розроблено архітектуру вебсистеми, яка враховує багаторівневий підхід до обробки даних та взаємодії з користувачем.

Спроєктовано структуру бази даних та реалізовано діаграми класів, компонентів і послідовності, що описують основну логіку взаємодії системи.

Дякую за увагу!