

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»**

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО

В.о. завідувача кафедри

_____ Олександр КОВАЛЬ

« ____ » _____ 2026 р.

Дипломна робота

**на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інженерія програмного забезпечення
інтелектуальних кібер-фізичних систем в енергетиці»
спеціальності 121 Інженерія програмного забезпечення
на тему: «Програмний інструмент для відстеження прогресу проєкту з
візуалізацією ключових метрик»**

Виконав:

студент IV курсу, групи ТВ-23

Тополіук Дмитро Сергійович

(прізвище, ім'я, по батькові)

_____ (підпис)

Керівник:

Доцент, к.т.н., Шпурик Вадим Вадимович

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

_____ (підпис)

Рецензент:

_____ (посада, науковий ступінь, вчене звання, прізвище та ініціали)

_____ (підпис)

Засвідчую, що у цій дипломній роботі
немає запозичень із праць інших авторів
без відповідних посилань.

Студент

_____ (підпис)

Київ – 2026

**Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»**

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення
інтелектуальних кібер-фізичних систем в енергетиці»

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

_____ Олександр КОВАЛЬ

«___» _____ 2026р.

ЗАВДАННЯ

на дипломну роботу студенту

_____ Тополюку Дмитру Сергійовичу

(прізвище, ім'я, по батькові)

1. Тема роботи: Програмний інструмент для відстеження прогресу проєкту з візуалізацією ключових метрик

керівник роботи Шпурик Вадим Вадимович, к.т.н.

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

затверджені наказом по університету від «28» травня _____ 2026р. №1992 _____

2. Строк подання студентом роботи «___» _____ 2026р.

3. Вихідні дані до роботи: вебзастосунок, Python, Django, Django REST Framework, PostgreSQL, Angular, TypeScript, PyCharm, Docker, Git, GitHub.

4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно розробити): проаналізувати предметну область та існуючі системи управління проєктами, визначити вимоги до системи, обґрунтувати засоби розробки, спроектувати архітектуру застосунку, розробити серверну та клієнтську частини, реалізувати базу даних, роботу з проєктами, задачами, підзадачами та метриками.

5. Перелік ілюстративного матеріалу: use-case діаграма, концептуальна модель, архітектура системи, інтерфейс користувача, робочі простори, дошка задач, модальне вікно задачі, сторінки метрик проєкту та користувачів.

6. Дата видачі завдання «31» жовтня 2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Строки виконання етапів роботи	Примітка
1	Отримання завдання	31.10.2025	
2	Дослідження предметної області	01.11.2025 – 31.11.2025	
3	Дослідження існуючих рішень	01.12.2025 – 31.12.2025	
4	Постановка вимог до проєктування системи	01.01.2026 – 31.01.2026	
5	Розробка програмного продукту	01.02.2026 – 03.05.2026	
6	Тестування	04.05.2026 – 10.05.2026	
7	Захист програмного продукту	11.05.2026 – 15.05.2026	
8	Оформлення дипломної роботи	18.05.2026 – 31.05.2026	
9	Передзахист	01.06.2026 – 05.06.2026	
10	Захист	15.06.2026 – 19.06.2026	

Студент

(підпис)

Дмитро Тополук

(ім'я, прізвище)

Керівник роботи

(підпис)

Вадим Шпурик

(ім'я, прізвище)

РЕФЕРАТ

Структура та обсяг дипломної роботи. Дипломна робота містить 77 сторінок, 21 рисуноків, 2 додатки та 20 посилань.

Метою роботи є розробка програмного інструменту для відстеження прогресу проєкту з візуалізацією ключових метрик.

У роботі розроблено вебзастосунок, призначений для організації командної роботи над проєктами. Система дозволяє створювати робочі простори, формувати проєкти, додавати учасників, визначати менеджера проєкту, працювати із задачами та підзадачами, призначати виконавців, встановлювати дедлайни, пріоритети, теги та story points. Окрему увагу приділено формуванню аналітичних показників, які дають змогу оцінювати стан виконання проєкту, навантаження учасників, наявність прострочених задач і загальний прогрес роботи.

Практичне значення одержаних результатів полягає у створенні програмного інструменту, який може бути використаний невеликими командами, навчальними групами або внутрішніми проєктними командами для організації роботи та контролю виконання задач. Розроблена система поєднує простоту дошки задач із можливістю перегляду ключових метрик прогресу, що дає змогу швидше оцінювати стан проєкту та виявляти проблемні місця в роботі команди.

Ключові слова: вебзастосунок, управління проєктами, відстеження прогресу, задачі, підзадачі, метрики, Angular, TypeScript, Python, Django, Django REST Framework, PostgreSQL, Docker.

ABSTRACT

Structure and scope of the diploma thesis. The diploma thesis consists of 77 pages, 21 figures, 2 appendices, and 20 references.

The purpose of the work is to develop a software tool to monitor the progress of the project with visualization of key metrics.

The robot has a separate web site for organizing team work on projects. The system allows you to create workspaces, formulate projects, add participants, assign a project manager, work with tasks and subtasks, assign employees, set deadlines, priorities, tags and story points. We pay special attention to the formation of analytical indicators that allow us to evaluate the progress of the project, the participation of participants, the presence of overdue tasks and the progress of the work.

The practical value of the results obtained lies in the creation of a software tool that can be used by small teams, study groups, or internal project teams to organize work and control task execution. The developed system combines the simplicity of a task board with the ability to view key progress metrics, which allows users to assess the project status more quickly and identify problem areas in the team's work.

Keywords: web application, project management, progress tracking, tasks, subtasks, metrics, Angular, TypeScript, Python, Django, Django REST Framework, PostgreSQL, Docker.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ	8
ВСТУП.....	9
1 ПОСТАНОВКА ЗАДАЧІ СТВОРЕННЯ ПРОГРАМНОГО ІНСТРУМЕНТУ ДЛЯ ВІДСТЕЖЕННЯ ПРОГРЕСУ ПРОЄКТУ	11
1.1 Мета.....	12
1.2 Функціональні вимоги системи	13
1.3 Нефункціональні вимоги системи	15
Висновки до розділу 1	16
2 ОГЛЯД ІСНУЮЧИХ СИСТЕМ УПРАВЛІННЯ ПРОЄКТАМИ.....	17
2.1 Jira.....	17
2.2 Asana.....	18
2.3 Trello.....	20
Висновки до розділу 2.....	21
3 ЗАСОБИ РОЗРОБКИ ПРОГРАМНОЇ СИСТЕМИ.....	23
3.1 Середовище розробки PyCharm.....	23
3.2 Система керування базами даних PostgreSQL	25
3.3 Мова Python та фреймворк Django	26
3.4 Мова JavaScript та фреймворк Angular.....	27
3.5 Засоби контейнеризації та система контролю версій	29
Висновки до розділу 3	31
4 ПРОГРАМНА РЕАЛІЗАЦІЯ	32
4.1 Діаграми системи	32
4.1.1 Use-case діаграма	33
4.1.2 Концептуальна модель	35
4.2 Проєктування архітектури системи.....	37
4.3 Реалізація серверної частини	39
4.3.1 Структура backend-частини застосунку	40
4.3.2 Реалізація моделей даних і взаємодії з базою даних	42
4.3.3 Реалізація REST API та обробки запитів.....	44
4.4 Реалізація клієнтської частини	45
4.4.1 Angular-компоненти та сервіси клієнтського застосунку.....	46

4.4.2 Реалізація інтерфейсу робочих просторів і проєктів	47
Висновки до розділу 4.....	49
5 РОБОТА КОРИСТУВАЧА З ПРОГРАМНИМ ІНСТРУМЕНТОМ	50
5.1 Реєстрація та автентифікація користувача.....	50
5.2 Створення та вибір робочого простору.....	51
5.3 Створення та редагування проєкту.....	53
5.4 Робота з дошкою задач.....	55
5.5 Перегляд метрик проєкту та користувачів.....	57
Висновки до розділу 5.....	59
ВИСНОВКИ	60
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	62
ДОДАТОК А Лістинг розробленої системи	64
ДОДАТОК В Презентація	73

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API	-	Інтерфейс програмування застосунків
UI	-	Користувацький інтерфейс
СКБД	-	Система керування базами даних
SQL	-	Мова структурованих запитів для роботи з базами даних
IDE	-	Інтегроване середовище розробки
CRUD	-	Операції створення, перегляду, оновлення та видалення даних
Workspace	-	Робочий простір
JSON	-	Текстовий формат обміну структурованими даними
Git	-	Система контролю версій для відстеження змін у коді

ВСТУП

Командна робота над проєктом рідко обмежується простим переліком задач. Навіть у невеликому проєкті поступово з'являються виконавці, дедлайни, пріоритети, підзадачі, різні рівні відповідальності та потреба бачити не окрему задачу, а загальний стан роботи. Якщо така інформація зберігається в різних місцях або оновлюється вручну, керівнику складно швидко зрозуміти, що вже виконано, які задачі затримуються, хто перевантажений і на якому етапі фактично перебуває проєкт.

Саме тому актуальним є створення програмного інструменту, який поєднує роботу із задачами та відображення основних показників прогресу. На практиці для управління проєктами часто використовують Jira [1], Asana [2] та Trello [3]. Ці системи дають змогу створювати задачі, працювати з дошками, розподіляти відповідальність між учасниками та контролювати виконання роботи [1–3]. Водночас для невеликих команд, навчальних проєктів або внутрішніх робочих груп такі рішення не завжди є зручними: частина функцій може бути зайвою, а швидкий перегляд саме ключових метрик прогресу потребує додаткового налаштування.

Метою дипломної роботи є розробка програмного інструменту для відстеження прогресу проєкту з візуалізацією ключових метрик. Такий інструмент має забезпечувати роботу з робочими просторами, проєктами, задачами та підзадачами, підтримувати призначення виконавців, встановлення строків виконання, визначення ролей учасників і перегляд аналітичної інформації щодо стану проєкту.

Для досягнення поставленої мети необхідно проаналізувати предметну область управління проєктами та наявні програмні рішення, визначити функціональні й нефункціональні вимоги до системи, обґрунтувати вибір засобів розробки, спроектувати архітектуру вебзастосунку й структуру даних, реалізувати клієнтську та серверну частини, а також перевірити роботу системи на типових сценаріях використання.

Об'єктом дослідження є процес управління задачами та відстеження прогресу командного проєкту.

Предметом дослідження є програмні засоби збору, обробки та візуалізації даних про задачі, виконавців, строки виконання, стан робіт і ключові метрики проєкту.

Практичне значення роботи полягає в розробці вебзастосунку, який може використовуватися невеликими командами, навчальними групами або внутрішніми проєктними командами для організації роботи й контролю виконання задач. Система дає можливість створювати робочі простори та проєкти, додавати учасників, працювати із задачами й підзадачами, призначати виконавців, встановлювати пріоритети, дедлайни та story points, а також переглядати метрики прогресу.

Програмний інструмент реалізовано як вебзастосунок із клієнт-серверною архітектурою. Клієнтська частина створюється з використанням Angular [4] і TypeScript [11], серверна частина — за допомогою Python [8], Django [9] та Django REST Framework [5], а для збереження даних використовується PostgreSQL [6]. Для контейнеризації та запуску компонентів системи застосовується Docker [12].

1 ПОСТАНОВКА ЗАДАЧІ СТВОРЕННЯ ПРОГРАМНОГО ІНСТРУМЕНТУ ДЛЯ ВІДСТЕЖЕННЯ ПРОГРЕСУ ПРОЄКТУ

Під час роботи над командним проєктом важливо не тільки фіксувати окремі задачі, а й розуміти, як просувається робота в цілому. У проєкті можуть одночасно існувати задачі різного пріоритету, підзадачі, дедлайни, виконавці та кілька напрямів роботи. Якщо вся ця інформація зберігається розрізнено або контролюється вручну, швидко оцінити фактичний стан проєкту стає складніше. У такій ситуації керівник або учасник команди може бачити окремі задачі, але не мати цілісного уявлення про прогрес, ризики за строками й завантаженість виконавців.

Для організації такої роботи вже існують різні програмні рішення, серед яких Jira [1], Asana [2] та Trello [3]. Вони дають змогу створювати задачі, працювати з дошками, призначати виконавців, контролювати строки та відстежувати статус виконання [1–3]. Проте для невеликих команд, навчальних проєктів або внутрішніх робочих груп ці системи не завжди є оптимальними. Частина їхнього функціоналу може бути надмірною, а налаштування процесів — складнішим, ніж потрібно для базового контролю задач і ключових показників прогресу.

У межах дипломної роботи поставлено задачу розробити програмний інструмент для відстеження прогресу проєкту з візуалізацією ключових метрик. Такий інструмент має поєднувати кілька основних можливостей: роботу з робочими просторами, створення проєктів, ведення задач і підзадач, призначення відповідальних осіб та перегляд аналітичної інформації. Основна ідея полягає в тому, щоб дані, які користувачі вносять під час звичайної роботи із задачами, надалі використовувалися для автоматичного формування показників стану проєкту.

Розроблювана система має підтримувати створення робочих просторів як окремих середовищ для команди або напрямку роботи. У межах робочого простору користувач повинен мати можливість створювати проєкти, додавати учасників і працювати із задачами. Для задач необхідно передбачити назву, опис, виконавця,

пріоритет, строки виконання, story points, теги та статус. Для складніших задач має бути можливість створювати підзадачі, щоб деталізувати роботу й контролювати виконання окремих етапів.

Важливою частиною поставленої задачі є формування метрик проєкту та користувачів. Система має показувати кількість відкритих і завершених задач, відсоток виконання, прострочені задачі, задачі без виконавця, виконані та залишкові story points, а також розподіл навантаження між учасниками. Завдяки таким показникам користувач може швидше оцінити стан проєкту без потреби вручну переглядати всі задачі на дошці.

Отже, основна задача дипломної роботи полягає у створенні програмного інструменту, який допомагає команді організовувати роботу над проєктом і водночас контролювати його виконання за допомогою зрозумілих метрик. Розроблена система має бути достатньо простою для використання у невеликих командах, але при цьому функціональною для роботи із задачами, підзадачами, строками, виконавцями та загальним станом проєкту.

1.1 Мета

Метою дипломної роботи є розробка програмного інструменту для відстеження прогресу проєкту з візуалізацією ключових метрик. Система має допомагати користувачу не тільки створювати задачі, а й бачити, як просувається робота над проєктом загалом. Для цього в застосунку передбачено роботу з робочими просторами, проєктами, задачами, підзадачами, виконавцями, дедлайнами, пріоритетами та story points.

Розроблюваний інструмент повинен використовувати дані, які з'являються під час звичайної роботи із задачами, для автоматичного формування аналітичних показників. До них належать кількість відкритих і завершених задач, відсоток виконання, прострочені задачі, задачі без виконавця, виконані та залишкові story points, а також розподіл навантаження між учасниками команди. Завдяки цьому

користувач отримує не просто список задач, а більш повне уявлення про стан проєкту.

Таким чином, мета роботи полягає у створенні вебзастосунку, який поєднує дошку задач, організацію командної роботи та перегляд метрик проєкту. Такий підхід дозволяє швидше оцінювати поточний стан роботи, контролювати виконання задач і зменшувати потребу в ручному підрахунку показників.

1.2 Функціональні вимоги системи

Функціональні вимоги визначають, які дії користувач повинен мати змогу виконувати в системі. Оскільки розроблюваний програмний інструмент призначений для командної роботи над проєктами, він має охоплювати основні процеси: створення облікового запису, роботу з робочими просторами, створення проєктів, додавання учасників, ведення задач і перегляд показників прогресу.

Насамперед у системі потрібно реалізувати реєстрацію та автентифікацію користувачів. Після входу користувач має працювати тільки з тими робочими просторами й проєктами, до яких він має доступ. Це необхідно для того, щоб дані різних команд не змішувалися між собою, а кожен користувач бачив саме той робочий контекст, у якому бере участь.

Одним із базових елементів системи є робочий простір. Він використовується як окреме середовище для команди або групи пов'язаних проєктів. Користувач повинен мати можливість створити робочий простір, змінити його назву, опис та іконку, переглянути доступні простори й обрати активний для подальшої роботи. У межах робочого простору також має бути передбачена робота з учасниками та їхніми ролями.

Наступним рівнем організації є проєкт. У вибраному робочому просторі користувач має створювати проєкти, задавати їм назву, опис, менеджера, дату початку та дату завершення. До проєкту повинні додаватися учасники, яким надалі

можна призначати задачі. Такий поділ дає змогу відокремити різні напрями роботи й не об'єднувати задачі кількох проєктів в одному загальному списку.

Основна робота в системі відбувається через дошку задач. Користувач повинен мати можливість створювати групи задач, додавати нові задачі, редагувати їхні параметри та змінювати стан виконання. Для задачі доцільно передбачити назву, опис, виконавця, пріоритет, дату початку, дедлайн, story points, теги та позначку виконання. Якщо задача є більш складною, її потрібно мати змогу деталізувати за допомогою підзадач.

Підзадачі потрібні для того, щоб розбивати об'ємну роботу на менші частини. Для них також мають задаватися виконавець, строки, пріоритет і story points. Це дозволяє користувачу бачити не лише загальну задачу, а й окремі етапи її виконання, що особливо важливо для складних або тривалих завдань.

Окремою функціональною вимогою є автоматичне формування метрик. Система має визначати кількість відкритих і завершених задач, відсоток виконання, прострочені задачі, задачі без виконавців, виконані та залишкові story points. Крім цього, потрібно показувати розподіл навантаження між учасниками команди, щоб керівник проєкту міг швидше помітити нерівномірний розподіл роботи або можливі ризики.

Аналітичні дані мають відображатися не тільки як окремі числові значення, а й у зручному для сприйняття вигляді. Для цього в системі передбачається аналітична панель із метриками проєкту, показниками користувачів, розподілом задач і часовою шкалою. Такий формат дає змогу швидко оцінити стан проєкту без ручного перегляду кожної задачі.

Отже, функціональні вимоги охоплюють основні дії, необхідні для командної роботи: керування користувачами, робочими просторами, проєктами, задачами, підзадачами та метриками. Реалізація цих можливостей дозволяє створити систему, яка не просто зберігає задачі, а допомагає бачити реальний стан виконання проєкту.

1.3 Нефункціональні вимоги системи

Нефункціональні вимоги описують не конкретні дії користувача, а загальні властивості системи: наскільки вона зручна, стабільна, безпечна, швидка та придатна до подальшого розвитку. Для програмного інструменту відстеження прогресу проєкту ці вимоги мають важливе значення, оскільки система працює з даними про задачі, строки, учасників команди та метрики, які повинні залишатися актуальними й доступними під час роботи.

Передусім застосунок має бути зрозумілим для користувача. Основні розділи системи — робочі простори, проєкти, дошка задач, налаштування та метрики — мають бути доступними без тривалого навчання. Користувач повинен швидко виконувати базові дії: створити задачу, змінити виконавця, встановити дедлайн, переглянути прогрес або перейти до потрібного проєкту. Це особливо важливо для невеликих команд, яким потрібен практичний інструмент для щоденної роботи, а не складна система з великою кількістю зайвих налаштувань.

Система також має працювати стабільно. Дані про користувачів, робочі простори, проєкти, задачі, підзадачі та метрики повинні зберігатися коректно, без втрати або дублювання інформації. Якщо користувач створює задачу, змінює її статус, призначає виконавця або оновлює дедлайн, ці зміни мають правильно відображатися як на дошці задач, так і в пов'язаних аналітичних показниках.

Окремою вимогою є безпека даних. Користувач повинен бачити лише ті робочі простори та проєкти, до яких має доступ. Дані різних команд не повинні змішуватися між собою, а можливість редагування має залежати від ролі користувача в робочому просторі або проєкті. Це зменшує ризик випадкового перегляду, зміни або видалення інформації, до якої користувач не має бути допущений.

Важливою є і підтримуваність програмного інструменту. Система повинна мати таку структуру, щоб у майбутньому її можна було розширювати без повної переробки застосунку. Наприклад, можна буде додати нові метрики, коментарі до

задач, історію змін, додаткові ролі користувачів або інші способи візуалізації прогресу. Для цього доцільним є поділ системи на клієнтську частину, серверну частину та базу даних.

Ще одна вимога пов'язана з можливістю розгортання системи в іншому середовищі. Застосунок має бути підготовлений до запуску не лише на комп'ютері розробника, а й на іншому пристрої або сервері. Використання контейнеризації дозволяє запускати клієнтську частину, backend і базу даних як окремі компоненти, що спрощує демонстрацію, перенесення й подальше використання програмного продукту.

Отже, нефункціональні вимоги до системи охоплюють зручність інтерфейсу, стабільність роботи, захист даних, достатню швидкодію, підтримуваність і можливість розгортання. Дотримання цих вимог є важливим для того, щоб програмний інструмент був не лише функціональним, а й придатним для реальної роботи з проєктами.

Висновки до розділу 1

У першому розділі було визначено основну задачу дипломної роботи — створення програмного інструменту для відстеження прогресу проєкту. Показано, що під час командної роботи недостатньо просто фіксувати перелік задач. Для нормального управління проєктом важливо бачити, на якому етапі перебуває робота, хто відповідає за виконання окремих задач, які строки наближаються, де виникають затримки та як розподілене навантаження між учасниками.

Отже, у першому розділі сформовано основу для подальшого проєктування й реалізації програмного інструменту. Було визначено, які проблеми має вирішувати система, які функції повинні бути реалізовані та які вимоги слід враховувати під час розробки.

2 ОГЛЯД ІСНУЮЧИХ СИСТЕМ УПРАВЛІННЯ ПРОЄКТАМИ

Перед розробкою власного програмного інструменту доцільно розглянути вже існуючі системи управління проєктами, які використовуються для планування роботи, створення задач, розподілу відповідальності та контролю строків. У цьому розділі розглянуто Jira [1], Asana [2] та Trello [3], оскільки ці інструменти мають різний підхід до організації командної роботи: від гнучкого керування задачами у технічних командах до простого візуального планування за допомогою дошок і карток. Їх аналіз дозволяє визначити корисні рішення, які можна врахувати під час розробки власної системи, а також виявити обмеження, через які для невеликих команд або навчальних проєктів доцільним є створення більш простого й сфокусованого інструменту для відстеження прогресу.

2.1 Jira

Jira [1] розглядається в цій роботі як приклад складнішої системи управління проєктами, яка часто використовується в командах розробки програмного забезпечення. Для аналізу вона важлива не як прямий аналог розроблюваного інструменту, а як приклад системи з широкими можливостями для організації задач, контролю процесів і побудови звітності.

У Jira основна робота зосереджена навколо проєктів і задач. Задача може мати виконавця, статус, пріоритет, строки виконання, опис, коментарі та інші параметри. Для командної роботи важливими є дошки, на яких задачі розміщуються за етапами виконання. Такий формат дає змогу бачити не тільки окрему задачу, а й загальний рух роботи в межах проєкту. У Jira також підтримуються підходи Scrum і Kanban, що робить систему зручною для команд із гнучкою організацією процесів [1].

Сильною стороною Jira є можливість детального налаштування робочого процесу. Команда може змінювати типи задач, статуси, правила переходу між

етапами, фільтри, ролі та звіти. Для великих проєктів це корисно, оскільки дозволяє підлаштувати систему під конкретний процес розробки. Крім того, Jira має інструменти для оцінювання прогресу: звіти, панелі, agile-метрики та інші аналітичні можливості [1]. На рисунку 2.1 зображено робочий простір Jira.

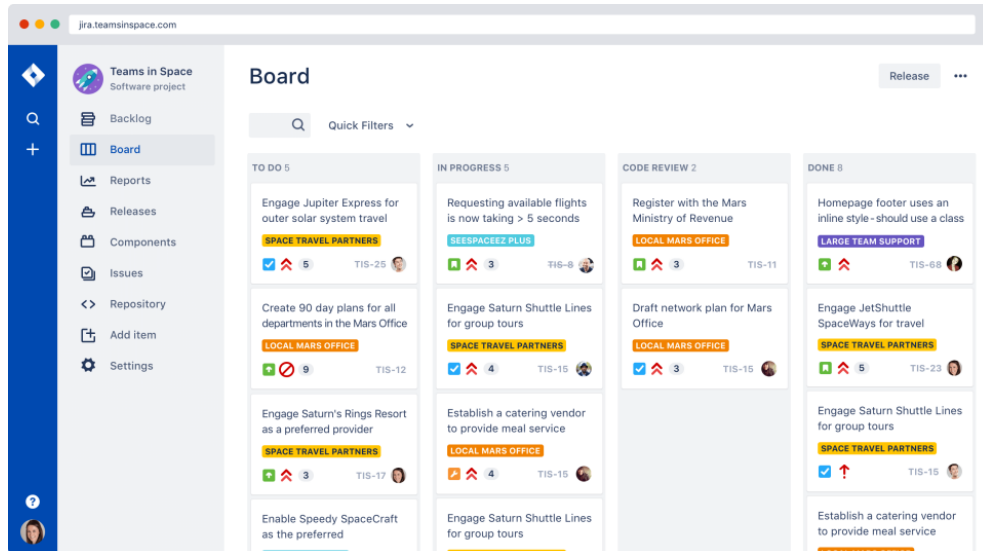


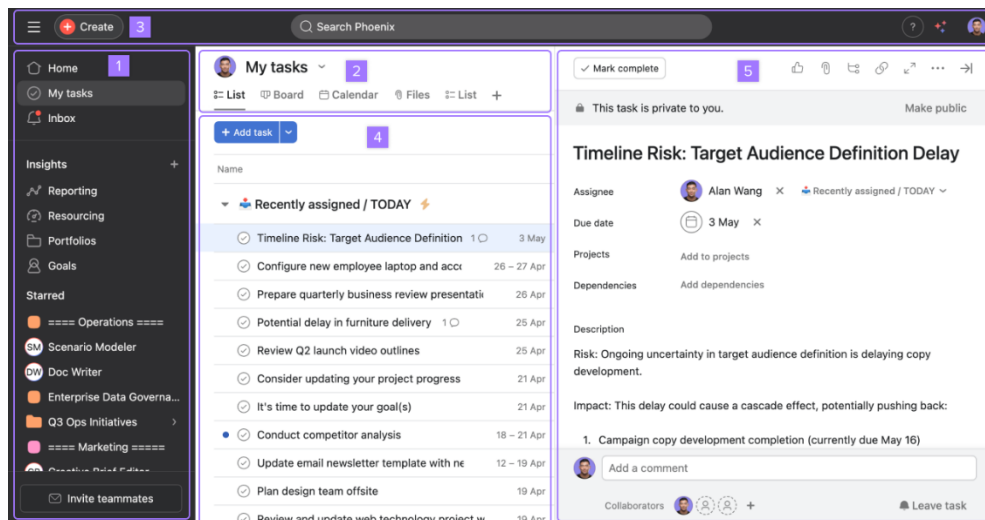
Рисунок 2.1 – Робочий простір Jira

Разом із цим для невеликої команди така кількість налаштувань може бути надмірною. Якщо проєкт має просту структуру, то попереднє налаштування workflow, ролей, прав доступу й звітності може займати більше часу, ніж потрібно для самої організації роботи. У контексті цієї дипломної роботи з Jira доцільно врахувати не всю її складність, а окремі ідеї: дошку задач, статуси, ролі та аналітичне відображення стану проєкту. У власному інструменті ці можливості мають бути реалізовані простіше, без зайвого ускладнення для користувача.

2.2 Asana

Asana [2] у цій роботі розглядається як приклад інструменту, який більше орієнтований не лише на технічні команди, а на ширшу організацію командної роботи. Її можна використовувати для планування навчальних, внутрішніх, управлінських або організаційних проєктів, де важливо розподілити задачі між учасниками, встановити строки та бачити поточний стан виконання.

Основна логіка Asana пов'язана з проєктами та задачами. У межах проєкту команда може створювати окремі задачі, призначати відповідальних, додавати строки виконання, описи, коментарі й підзадачі. Для аналізу цієї системи важливим є те, що один і той самий набір задач можна переглядати по-різному: як список, дошку, календар або часову шкалу. Це дає змогу підлаштовувати спосіб перегляду під характер роботи команди [2]. На рисунку 2.2 зображено робочий простір Asana.



На рисунку 2.2 зображено робочий простір Asana.

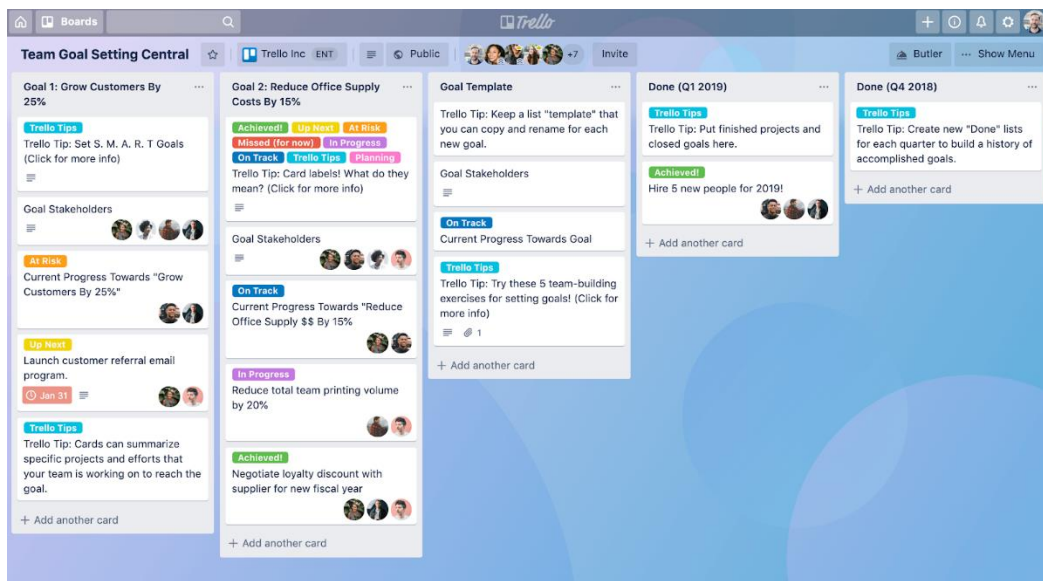
Порівняно з Jira, Asana виглядає простішою для старту. Вона не вимагає настільки глибокого налаштування процесів і більше підходить для команд, яким потрібно швидко організувати роботу, розподілити відповідальність і контролювати строки. Корисною для порівняння є також наявність часової шкали, оскільки вона допомагає бачити задачі не тільки як список, а як послідовність робіт у часі [2].

Водночас для невеликого навчального або внутрішнього проєкту можливості Asana також можуть бути ширшими, ніж потрібно. Якщо команді потрібен компактний інструмент для ведення задач і перегляду базових показників прогресу, то частина функцій автоматизації, звітності або управління навантаженням може залишатися невикористаною. Тому для власної системи доцільно взяти з Asana ідею зручного планування, роботи з відповідальними особами та часовим представленням, але реалізувати це у більш сфокусованому форматі для відстеження прогресу проєкту.

2.3 Trello

Trello [3] є одним із найбільш простих і візуально зрозумілих інструментів для управління задачами. Його основна логіка побудована навколо дошок, списків і карток. Дошка зазвичай відповідає певному проєкту або напрямку роботи, списки відображають етапи виконання, а картки використовуються для окремих задач. Такий підхід робить Trello зручним для невеликих команд, навчальних проєктів, особистого планування або організації нескладних робочих процесів [3].

Головною перевагою Trello є простота входження в роботу. Користувачу не потрібно довго налаштовувати систему або створювати складні робочі процеси. Достатньо створити дошку, додати списки та розмістити в них картки із задачами. У процесі роботи картки можна переміщувати між списками, змінюючи їхній стан. Наприклад, команда може створити списки «Заплановано», «У роботі», «На перевірці» та «Виконано», а потім переносити задачі між ними відповідно до реального стану роботи [3]. На рисунку 2.3 зображено робочий простір Trello.



На рисунку 2.3 зображено робочий простір Trello.

Картки в Trello можуть містити назву, опис, дедлайни, мітки, учасників, коментарі та додаткові матеріали. Завдяки цьому одна картка виступає не просто коротким записом про задачу, а окремим робочим елементом із потрібною для команди інформацією [3]. Це зручно для базового ведення задач, коли важливо

швидко бачити, що саме потрібно зробити, хто залучений до роботи та на якому етапі перебуває задача.

Ще однією сильною стороною Trello є візуальність. Користувач одразу бачить загальну структуру дошки, кількість задач у кожному списку та поточний стан роботи. Такий формат добре підходить для Kanban-подібного підходу, де важливо бачити рух задач між етапами й не витратити багато часу на складне налаштування системи [3].

Водночас простота Trello має і певні обмеження. Базова дошка добре показує поточний стан задач, але не завжди дає достатньо аналітичної інформації про проєкт. Без додаткових налаштувань або ручного аналізу складно швидко оцінити загальний відсоток виконання, навантаження учасників, залишковий обсяг роботи у story points, ризики за строками або динаміку виконання задач.

Отже, Trello є зручним прикладом простої системи візуального управління задачами. Його сильними сторонами є зрозуміла структура дошок, списків і карток, швидкий початок роботи та зручне переміщення задач між етапами. Для розроблюваного програмного інструменту доцільно врахувати саме ці переваги, але доповнити їх можливостями, яких бракує базовому підходу Trello: метриками, story points, аналізом навантаження користувачів, часовою шкалою задач і загальним відстеженням прогресу проєкту.

Висновки до розділу 2

У другому розділі було розглянуто три поширені системи управління проєктами: Jira [1], Asana [2] та Trello [3]. Кожна з них вирішує схожу задачу — допомагає організувати роботу команди, розподіляти задачі між учасниками та контролювати їх виконання. При цьому підхід у цих систем різний: Jira більше підходить для команд розробки та гнучких процесів, Asana орієнтована на планування командної роботи й контроль строків, а Trello робить акцент на простому візуальному представленні задач через дошки, списки та картки.

Огляд цих інструментів дозволив виділити функції, які є корисними для власної системи. До них можна віднести роботу з дошкою задач, поділ роботи за етапами, призначення виконавців, використання дедлайнів і наочне відображення поточного стану проєкту. Саме ці елементи є базовими для того, щоб користувач міг не просто створювати задачі, а й розуміти, як просувається робота загалом.

Разом із цим готові системи не завжди зручні для невеликих команд, навчальних проєктів або внутрішніх робочих груп. Частина їхніх можливостей може бути надмірною для простого відстеження прогресу, а налаштування інколи потребує більше часу, ніж сама організація невеликого проєкту. Тому під час розробки власного інструменту доцільно було зосередитися на більш компактному наборі функцій: робочі простори, проєкти, задачі, підзадачі, виконавці, строки та ключові метрики.

Отже, результати огляду підтвердили доцільність створення власного програмного інструменту. Він має поєднувати просту роботу із задачами, підтримку командного контексту через робочі простори та проєкти, а також візуалізацію показників, які допомагають швидко оцінити стан виконання проєкту.

3 ЗАСОБИ РОЗРОБКИ ПРОГРАМНОЇ СИСТЕМИ

Для створення програмного інструменту відстеження прогресу проєкту необхідно було обрати засоби розробки, які дозволяють реалізувати повноцінний вебзастосунок із клієнт-серверною архітектурою, збереженням даних, роботою з користувачами та подальшим розгортанням. Оскільки система включає робочі простори, проєкти, задачі, підзадачі, ролі учасників і метрики, важливо було забезпечити стабільну взаємодію між інтерфейсом користувача, серверною логікою та базою даних.

У межах роботи для розробки було використано середовище PyCharm [7], яке є зручним для роботи з Python-проєктами та серверною частиною застосунку. Для збереження структурованих даних обрано PostgreSQL [6], оскільки система має багато пов'язаних сутностей: користувачів, робочі простори, проєкти, учасників, задачі та підзадачі. Серверна частина реалізується мовою Python [8] із використанням Django [9] та Django REST Framework [5], що дозволяє створити REST API для обміну даними між backend і frontend. Клієнтська частина розробляється з використанням Angular [4] і TypeScript [11], що підтримують компонентний підхід до побудови інтерфейсу вебзастосунку. Окремо для запуску, перенесення та збереження проєкту використовуються Docker [12], Git [13] та GitHub [14].

3.1 Середовище розробки PyCharm

Для роботи над серверною частиною проєкту було обрано PyCharm [7]. У цій дипломній роботі він використовувався не просто як редактор коду, а як основне середовище для роботи з backend-частиною, файлами налаштувань, терміналом і загальною структурою застосунку.

Оскільки серверна частина реалізована мовою Python [8] із використанням Django [9] та Django REST Framework [5], під час розробки потрібно було постійно працювати з моделями, міграціями, серіалізаторами, маршрутами та API-запитами.

PyCharm був зручним саме тому, що дозволяв швидко переходити між цими файлами й бачити структуру Django-проєкту як єдину систему, а не як окремі непов'язані частини. На рисунку 3.1 зображено середа розробки PyCharm.

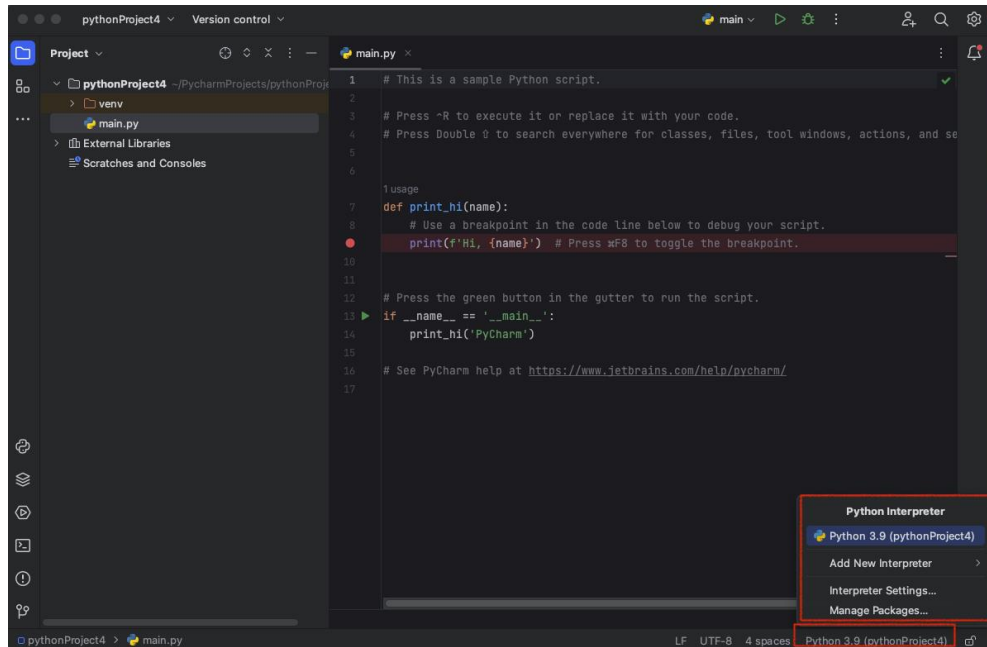


Рисунок 3.1 – Середа розробки PyCharm

Окремо використовувався вбудований термінал PyCharm. Через нього виконувалися команди для роботи з міграціями, запуску backend-частини, перевірки змін і роботи з контейнерами. Це спрощувало процес розробки, тому що більшість дій можна було виконувати в одному середовищі без постійного перемикання між різними програмами.

Також PyCharm був корисним під час роботи з конфігураційними файлами. У проєкті є не тільки Python-код, а й файли, пов'язані з налаштуванням середовища, контейнеризацією, frontend-частиною та базою даних. Можливість переглядати ці файли поруч із backend-кодом допомагала краще контролювати структуру всього застосунку.

Отже, PyCharm [7] використовувався як основне середовище для розробки серверної частини програмного інструменту. Його застосування дало змогу зручно працювати з Django-проєктом, виконувати службові команди через термінал і підтримувати порядок у файлах під час розробки.

3.2 Система керування базами даних PostgreSQL

Для збереження даних у програмній системі було використано PostgreSQL [6]. Це об'єктно-реляційна система керування базами даних з відкритим вихідним кодом, яка підходить для застосунків, де потрібно зберігати структуровані дані та працювати зі зв'язками між різними сутностями [6]. У межах розроблюваного інструменту PostgreSQL використовується як основне сховище інформації про користувачів, робочі простори, проєкти, учасників, задачі, підзадачі, теги, дедлайни та метрики.

Вибір PostgreSQL є доцільним, оскільки система має багато взаємопов'язаних об'єктів. Наприклад, користувач може бути учасником кількох робочих просторів, робочий простір може містити кілька проєктів, проєкт має команду, задачі та підзадачі, а кожна задача може бути пов'язана з виконавцем, тегами, строками виконання та story points. Для такої структури реляційна модель є зручною, тому що дозволяє чітко описати зв'язки між таблицями та забезпечити цілісність даних.

У серверній частині застосунку взаємодія з PostgreSQL [6] відбувається через Django ORM [9]. Це дозволяє описувати структуру даних на рівні моделей Python [8], а потім створювати відповідні таблиці в базі даних за допомогою міграцій. Такий підхід спрощує розробку, оскільки більшість операцій із даними можна виконувати не через прямі SQL-запити, а через об'єкти моделей. Водночас сама база даних зберігає структуровану інформацію у надійному та впорядкованому вигляді.

PostgreSQL [6] також використовується як джерело даних для формування метрик проєкту. Дані про статус задач, виконавців, дедлайни, story points і дату завершення зберігаються в базі та надалі обробляються серверною частиною. На їх основі система може визначати кількість відкритих і завершених задач, прострочені задачі, залишковий обсяг роботи, навантаження користувачів та інші показники, необхідні для аналітичної панелі.

У межах проєкту PostgreSQL [6] запускається як окремий компонент системи. Це дозволяє відокремити базу даних від серверної та клієнтської частин і зробити структуру застосунку більш зрозумілою. Такий підхід також спрощує подальше розгортання системи, оскільки база даних може працювати в окремому контейнері разом з іншими компонентами програмного інструменту [12].

Отже, PostgreSQL [6] було обрано як надійну систему керування базами даних для збереження основної інформації про роботу застосунку. Її використання дозволяє організувати дані системи у структурованому вигляді, підтримувати зв'язки між сутностями та забезпечити основу для розрахунку метрик прогресу проєкту.

3.3 Мова Python та фреймворк Django

Серверна частина програмної системи реалізована мовою Python [8]. Ця мова була обрана через зрозумілий синтаксис, велику кількість готових бібліотек і зручність використання у веброзробці. Для розроблюваного інструменту це важливо, оскільки backend має не лише приймати запити від клієнтської частини, а й працювати з базою даних, перевіряти права користувачів, обробляти задачі та формувати дані для метрик [8].

У межах проєкту Python використовується разом із фреймворком Django [9]. Django є високорівневим вебфреймворком, який дозволяє швидше створювати серверну частину застосунку та має вбудовані засоби для роботи з моделями, маршрутизацією, автентифікацією, адміністративною панеллю й базою даних [9]. Це дало змогу зосередитися не на написанні всіх базових механізмів з нуля, а на реалізації логіки самої системи: робочих просторів, проєктів, задач, підзадач, учасників і ролей.

Для взаємодії з базою даних у Django використовується ORM [9]. Завдяки цьому основні сутності системи описуються у вигляді моделей Python [8], а відповідні таблиці створюються через міграції. Такий підхід є зручним для проєкту,

де існує багато пов'язаних об'єктів: користувачі, робочі простори, проєкти, задачі, підзадачі, теги, виконавці та дедлайни. Замість постійного написання SQL-запитів більшість операцій із даними можна виконувати через моделі, що спрощує розробку та підтримку коду.

Окремо в роботі використовується Django REST Framework [5]. Він потрібен для створення REST API, через яке клієнтська частина обмінюється даними із сервером [5]. Саме через API frontend отримує інформацію про проєкти, задачі, учасників і метрики, а також надсилає запити на створення, редагування або видалення об'єктів. Такий підхід дозволяє чітко розділити серверну та клієнтську частини застосунку.

У межах розроблюваної системи backend відповідає за обробку основної логіки: автентифікацію користувачів, перевірку доступу до робочих просторів і проєктів, роботу із задачами та підзадачами, а також формування даних для аналітичної панелі. На основі інформації з бази даних серверна частина розраховує показники прогресу, кількість відкритих і завершених задач, прострочені задачі, залишкові story points і навантаження користувачів.

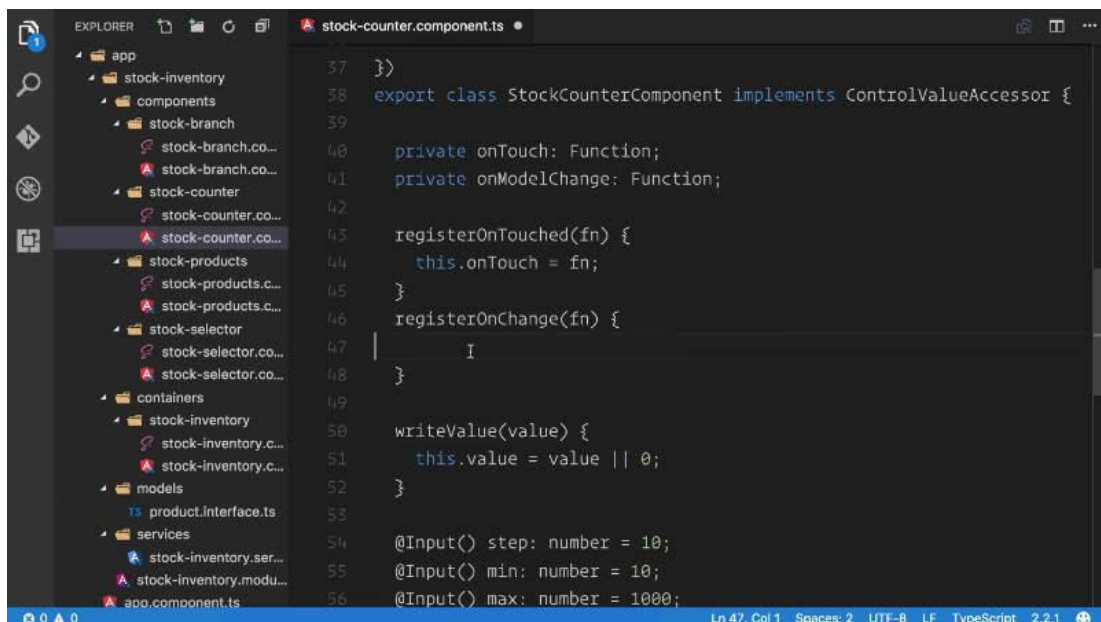
Таким чином, Python [8] і Django [9] були використані як основа серверної частини програмного інструменту. Їх поєднання дозволило реалізувати стабільний backend, організувати роботу з базою даних, створити API для клієнтської частини та забезпечити обробку даних, необхідних для відстеження прогресу проєкту.

3.4 Мова JavaScript та фреймворк Angular

Клієнтська частина програмної системи реалізована з використанням фреймворку Angular [4]. Для вебзастосунку, який має багато сторінок, форм, модальних вікон, таблиць, карток задач і аналітичних блоків, важливо було обрати технологію, що дозволяє зручно розділити інтерфейс на окремі компоненти та підтримувати зрозумілу структуру проєкту. Angular підходить для цього завдання,

оскільки орієнтований на створення динамічних вебінтерфейсів і підтримує компонентний підхід [4, 16].

Основою клієнтської веброзробки є JavaScript [10]. Він використовується для реалізації логіки взаємодії користувача з інтерфейсом, обробки подій, роботи з даними та оновлення сторінки без повного перезавантаження [10]. У межах розроблюваної системи клієнтська частина має реагувати на дії користувача: відкриття задачі, зміну виконавця, встановлення дедлайну, створення підзадачі, перемикання між розділами та перегляд метрик. На рисунку 3.2 зображено приклад коду Angular



На рисунку 3.2 - Приклад коду Angular

У проєкті разом з Angular використовується TypeScript [11]. Він є надбудовою над JavaScript і додає статичну типізацію, що робить код більш передбачуваним і зручним для підтримки [11]. Для цієї системи це важливо, оскільки frontend працює з багатьма типами даних: користувачами, робочими просторами, проєктами, задачами, підзадачами, ролями, тегами та метриками. Завдяки типізації легше описувати структуру таких об'єктів і зменшувати кількість помилок під час розробки.

Angular дозволяє будувати інтерфейс із незалежних компонентів [16]. У межах системи окремими компонентами можуть бути сторінка входу, сторінка

реєстрації, список робочих просторів, сторінка проєкту, дошка задач, картка задачі, модальне вікно задачі, аналітична панель і блоки метрик. Такий поділ робить клієнтську частину більш зрозумілою: кожен компонент відповідає за свою частину інтерфейсу, а не вся логіка зосереджується в одному файлі.

Для обміну даними із серверною частиною використовуються сервіси Angular [17]. Вони надсилають HTTP-запити до REST API та отримують відповідь із backend-частини [15, 20]. Завдяки цьому компоненти можуть відображати актуальні дані про проєкти, задачі, учасників і метрики, а також передавати на сервер зміни, які виконує користувач. Наприклад, після створення нової задачі frontend надсилає запит на сервер, отримує оновлені дані й відображає задачу на дошці.

Отже, Angular [4], JavaScript [10] і TypeScript [11] були використані для реалізації клієнтської частини програмного інструменту. Їх поєднання дозволило створити структурований вебінтерфейс, організувати роботу з компонентами, забезпечити взаємодію з REST API та зробити систему зручною для щоденної роботи користувача з проєктами, задачами й метриками.

3.5 Засоби контейнеризації та система контролю версій

Для запуску програмної системи було використано засоби контейнеризації. Це важливо, оскільки застосунок складається з кількох частин: клієнтської частини, серверної частини та бази даних. Кожен із цих компонентів має власні залежності й умови запуску, тому без єдиного підходу до розгортання можуть виникати проблеми з налаштуванням середовища на іншому комп'ютері або сервері.

Основним інструментом контейнеризації в роботі є Docker [12]. Він дозволяє запускати компоненти застосунку в ізольованих контейнерах, де вже визначено необхідне середовище для їх роботи [12]. У межах розроблюваної системи такий підхід дає змогу окремо запускати backend, frontend і PostgreSQL [6], але при цьому

об'єднувати їх в одну програмну систему. Це спрощує перевірку роботи застосунку та робить його запуск більш передбачуваним.

Для керування кількома контейнерами використовується Docker Compose [12]. За його допомогою можна описати сервіси системи, їхні порти, залежності, змінні середовища та volumes. У результаті всю систему можна запустити однією командою, не налаштовуючи кожен компонент окремо. Для дипломного проєкту це особливо зручно, оскільки застосунок має бути не лише розроблений, а й підготовлений до демонстрації та перенесення в інше середовище.

Окрім контейнеризації, важливу роль у розробці відіграє система контролю версій. Для цього було використано Git [13]. Він дозволяє фіксувати зміни в коді, переглядати історію розробки, повертатися до попередніх версій файлів і безпечніше вносити новий функціонал [13]. У процесі роботи над системою це було важливо, оскільки функціонал поступово розширювався: від базової структури застосунку до робочих просторів, проєктів, задач, підзадач і метрик.

Для збереження репозиторію та організації роботи з кодом використовувався GitHub [14]. У репозиторії розміщується вихідний код frontend- і backend-частин, конфігураційні файли, Docker-файли та інші матеріали, необхідні для запуску проєкту [14]. Наявність віддаленого репозиторію також спрощує перенесення системи на інший пристрій і дозволяє зберігати актуальну версію розробки в одному місці.

Таким чином, Docker [12], Docker Compose [12], Git [13] і GitHub [14] були використані для організації запуску, збереження та супроводу програмної системи. Контейнеризація спростила роботу з кількома компонентами застосунку, а система контролю версій забезпечила послідовне збереження змін і контроль процесу розробки.

Висновки до розділу 3

У третьому розділі було описано засоби, які використовувалися для розробки програмного інструменту відстеження прогресу проєкту. Для роботи над кодом обрано PyCharm [7], оскільки це середовище зручне для Python- і Django-проєктів, роботи з файлами застосунку, терміналом та загальною структурою backend-частини.

Для збереження даних використано PostgreSQL [6]. Такий вибір є доцільним для системи, у якій багато пов'язаних сутностей: користувачі, робочі простори, проєкти, учасники, задачі, підзадачі, теги, строки виконання та метрики. Серверна частина реалізована мовою Python [8] із використанням Django [9] та Django REST Framework [5].

Клієнтську частину розроблено з використанням Angular [4], JavaScript [10] і TypeScript [11]. Завдяки цьому інтерфейс застосунку можна було поділити на окремі компоненти: сторінки робочих просторів і проєктів, дошку задач, модальні вікна, форми та аналітичні блоки.

Також у розділі розглянуто інструменти для запуску, збереження та супроводу проєкту. Docker [12] використовується для контейнеризації компонентів системи, що спрощує запуск frontend, backend і бази даних в узгодженому середовищі. Git [13] і GitHub [14] застосовуються для контролю змін, збереження історії розробки та роботи з репозиторієм проєкту.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ

У цьому розділі описано програмну реалізацію інструменту для відстеження прогресу проєкту. Основну увагу приділено тому, як побудована система, з яких частин вона складається та яким чином взаємодіють між собою клієнтська частина, серверна частина і база даних. Розроблений застосунок реалізовано як вебсистему, у якій користувач працює з робочими просторами, проєктами, задачами, підзадачами, учасниками команди та метриками прогресу.

Програмна реалізація охоплює моделювання основних сценаріїв роботи користувача, структуру бази даних, архітектуру системи, реалізацію backend-частини на Django REST Framework [5], frontend-частини на Angular [4], а також аналітичного модуля. Окремо розглянуто механізм формування метрик проєкту та користувачів, оскільки саме ці показники дозволяють не лише вести задачі, а й оцінювати реальний стан виконання проєкту.

4.1 Діаграми системи

Для опису програмної реалізації системи доцільно спочатку подати її у вигляді діаграм. Це дозволяє перед детальним розглядом коду показати загальну логіку роботи застосунку: хто взаємодіє із системою, які основні дії виконує користувач, які сутності лежать в основі предметної області та як вони пов'язані між собою.

У межах роботи використано дві основні діаграми. Use-case діаграма відображає функціональні можливості системи з погляду користувачів. Вона показує, які дії доступні звичайному користувачу та керівнику проєкту: реєстрація, авторизація, робота з дошкою задач, створення й редагування задач, створення підзадач, перегляд метрик, керування робочим простором, проєктом та учасниками. Така діаграма допомагає зрозуміти основні сценарії використання програмного інструменту без переходу до технічних деталей реалізації.

Концептуальна модель бази даних відображає основні сутності системи та зв'язки між ними на узагальненому рівні. У ній показано користувача, робочий простір, проєкт, задачу, підзадачу, тег і метрики. На відміну від повної логічної схеми бази даних, концептуальна модель не деталізує всі службові таблиці й технічні поля, а демонструє загальну структуру даних, на якій базується робота програмного інструменту.

4.1.1 Use-case діаграма

На рисунку 4.1 наведено use-case діаграму програмного інструменту відстеження прогресу проєкту.

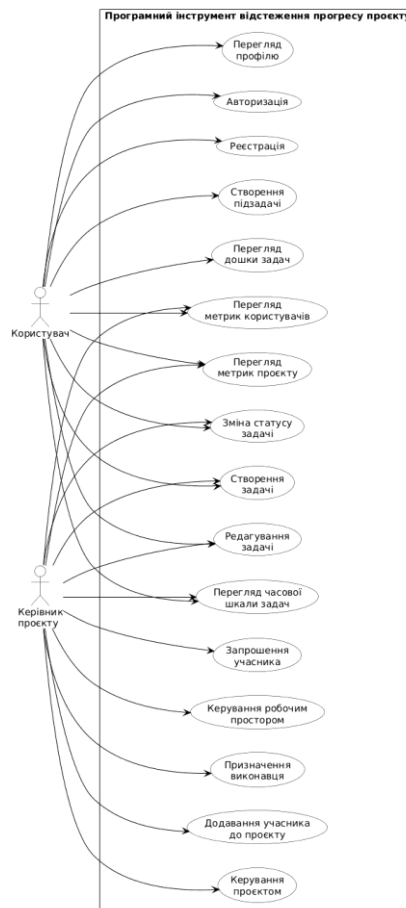


Рисунок 4.1 – Use-case діаграма проєкту

На діаграмі відображено два основні типи взаємодії із системою: роботу звичайного користувача та роботу керівника проєкту. Такий поділ є важливим, оскільки не всі дії в системі мають бути доступними кожному учаснику. Частина

функцій пов'язана з повсякденною роботою із задачами, а частина — з організацією робочого простору, керуванням проєктом і розподілом відповідальності між учасниками.

Користувач у системі виконує базові дії, необхідні для участі в проєкті. До них належать реєстрація, авторизація та перегляд власного профілю. Після входу в систему користувач отримує доступ до дошки задач, де може переглядати поточні задачі, створювати нові, редагувати наявні, змінювати їхній статус і створювати підзадачі. Саме ці сценарії відображають основну роботу учасника команди із завданнями проєкту.

Окремо на діаграмі показано сценарії перегляду аналітичної інформації. Користувач може переглядати метрики проєкту, метрики користувачів і часову шкалу задач. Це дозволяє йому оцінювати не лише окремі задачі, а й загальний стан виконання роботи. Наприклад, через метрики проєкту можна побачити рівень виконання, наявність прострочених задач і залишковий обсяг роботи, а через метрики користувачів — розподіл навантаження між учасниками.

Керівник проєкту має розширений набір дій. Він може керувати робочим простором, запрошувати нових учасників, створювати або редагувати проєкт, додавати учасників до проєкту та призначати виконавців для задач. Ці можливості потрібні для організації командної роботи та підтримки актуальної структури проєкту. Якщо звичайний користувач переважно працює із задачами, то керівник відповідає ще й за налаштування середовища, склад команди та розподіл відповідальності.

Таким чином, use-case діаграма демонструє загальну логіку взаємодії з програмним інструментом. Вона показує, що система охоплює кілька рівнів роботи: обліковий запис користувача, дошку задач, підзадачі, аналітику, керування робочим простором і керування проєктом. Такий набір сценаріїв відповідає основній меті розробки — створенню інструменту, який дозволяє не лише вести задачі, а й контролювати прогрес командного проєкту.

4.1.2 Концептуальна модель

На рисунку 4.2 наведено концептуальну модель бази даних програмного інструменту відстеження прогресу проєкту.

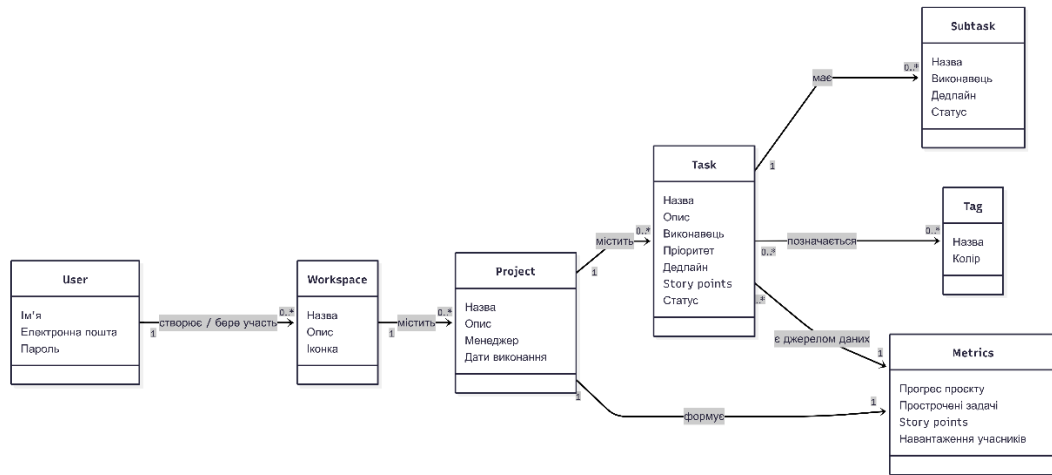


Рисунок 4.2 – Концептуальна модель бази даних

Концептуальна модель відображає основні сутності системи та зв'язки між ними на узагальненому рівні. На цьому етапі не деталізуються всі службові таблиці, технічні поля та допоміжні зв'язки, які використовуються безпосередньо в реалізації. Основна мета моделі — показати, з яких ключових об'єктів складається система і як ці об'єкти пов'язані між собою в межах предметної області.

У моделі виділено сутність User, яка описує користувача системи. Для користувача зберігаються ім'я, електронна пошта та пароль. Саме користувач створює або бере участь у робочих просторах, може бути учасником проєкту, виконувати задачі та переглядати метрики. Зв'язок між User і Workspace показує, що один користувач може бути пов'язаний із кількома робочими просторами, а кожен робочий простір має користувача, який його створив або бере участь у його роботі.

Сутність Workspace використовується для представлення робочого простору. Вона містить назву, опис та іконку. Робочий простір у системі виступає окремим середовищем для команди або групи проєктів. Зв'язок між Workspace і Project показує, що один робочий простір може містити декілька проєктів, але кожен

проект належить до конкретного робочого простору. Такий поділ дозволяє розмежувати дані різних команд або напрямів роботи.

Сутність Project описує окремий проект у межах робочого простору. Для проекту зберігаються назва, опис, менеджер і дати виконання. Проект є центральним об'єктом для подальшої роботи із задачами, оскільки саме до нього належать задачі, підзадачі, теги та метрики. Наявність менеджера дозволяє відокремити звичайного учасника від користувача, який відповідає за організацію роботи в межах конкретного проекту.

Основною робочою сутністю моделі є Task. Вона містить назву, опис, виконавця, пріоритет, дедлайн, story points і статус. Через задачі користувачі фіксують конкретні елементи роботи, змінюють їхній стан, призначають відповідальних осіб і контролюють виконання. Зв'язок між Project і Task показує, що проект може містити багато задач, а кожна задача належить до одного проекту.

Для деталізації складних задач у моделі передбачено сутність Subtask. Вона містить назву, виконавця, дедлайн і статус. Підзадачі дозволяють розбивати велику задачу на менші частини та контролювати виконання кожного етапу окремо. У моделі показано, що одна задача може мати кілька підзадач, але кожна підзадача пов'язана з конкретною задачею.

Сутність Tag використовується для додаткового позначення задач. Вона містить назву та колір. За допомогою тегів користувач може групувати задачі за типом, напрямом, важливістю або іншими ознаками. Зв'язок між Task і Tag показує, що задача може бути позначена тегами, що спрощує візуальне сприйняття дошки задач і пошук потрібних елементів роботи.

Окремо в концептуальній моделі показано сутність Metrics. Вона відображає аналітичний рівень системи: прогрес проекту, прострочені задачі, story points і навантаження учасників. Метрики формуються не вручну, а на основі даних про задачі, їхній статус, виконавців, дедлайни та story points. Саме тому Task виступає джерелом даних для Metrics, а Project пов'язаний із метриками як об'єкт, для якого ці показники розраховуються.

Таким чином, концептуальна модель показує загальну структуру даних програмного інструменту. Вона демонструє, що система побудована навколо робочого простору, проєкту та задачі, а всі інші сутності доповнюють цю логіку: користувачі виконують роботу, підзадачі деталізують задачі, теги допомагають їх класифікувати, а метрики використовують накопичені дані для відстеження прогресу проєкту.

4.2 Проєктування архітектури системи

Програмний інструмент відстеження прогресу проєкту реалізовано як вебзастосунок із клієнт-серверною архітектурою. Такий підхід є зручним для цієї системи, оскільки дозволяє розділити інтерфейс користувача, серверну логіку та збереження даних. Користувач працює із застосунком через клієнтську частину, серверна частина обробляє запити й перевіряє права доступу, а база даних зберігає інформацію про користувачів, робочі простори, проєкти, задачі, підзадачі та метрики.

Клієнтська частина системи розроблена з використанням Angular. Вона відповідає за відображення сторінок застосунку, роботу з формами, навігацію, дошку задач, модальні вікна, сторінки робочих просторів і проєктів, а також за відображення аналітичних даних [4]. Саме через frontend користувач виконує основні дії: створює задачі, редагує їх, змінює статус, переглядає метрики або переходить між розділами системи.

Серверна частина реалізована за допомогою Python, Django та Django REST Framework. Вона виконує роль проміжного рівня між клієнтською частиною та базою даних. Backend приймає HTTP-запити від Angular-застосунку, обробляє їх, звертається до бази даних, виконує перевірку доступу та повертає відповідь у форматі, зручному для відображення на клієнтській частині. Для побудови API використовується Django REST Framework, що дозволяє організувати обмін даними між frontend і backend через REST-запити [5].

База даних побудована на PostgreSQL. Вона використовується для збереження всіх основних даних системи: облікових записів користувачів, робочих просторів, проєктів, задач, підзадач, тегів, строків виконання, story points та інших параметрів, необхідних для роботи застосунку [6]. Оскільки між цими сутностями існує багато зв'язків, використання реляційної бази даних є доцільним для підтримки цілісності інформації та коректного формування метрик.

Загальна схема роботи системи є послідовною. Спочатку користувач виконує певну дію в інтерфейсі, наприклад створює задачу або змінює її статус. Після цього клієнтська частина надсилає запит до серверної частини. Backend перевіряє отримані дані, визначає, чи має користувач право виконати відповідну дію, звертається до бази даних і зберігає або отримує потрібну інформацію. Після завершення обробки сервер повертає відповідь, а клієнтська частина оновлює інтерфейс без необхідності ручного перезавантаження сторінки.

Окрему роль в архітектурі відіграє аналітичний модуль. Він не є повністю ізольованою частиною системи, а формується на основі даних, що вже зберігаються в базі. Для розрахунку метрик використовуються задачі, підзадачі, виконавці, дедлайни, story points і статуси виконання. На основі цих даних серверна частина формує показники прогресу, кількість прострочених задач, навантаження учасників і дані для часової шкали задач. Після цього клієнтська частина відображає ці показники у вигляді аналітичних блоків і візуальних елементів.

Для запуску системи використовується контейнеризоване середовище. Окремими компонентами запускаються frontend, backend і база даних. Такий підхід спрощує розгортання, оскільки кожна частина системи має власне середовище виконання, але всі компоненти працюють разом як єдиний застосунок [12]. Це також полегшує перенесення проєкту на інший комп'ютер або сервер.

Таким чином, архітектура програмного інструменту складається з трьох основних рівнів: клієнтського, серверного та рівня бази даних. Клієнтська частина забезпечує взаємодію користувача із системою, серверна частина реалізує основну бізнес-логіку й API, а база даних відповідає за збереження структурованої

інформації. Такий поділ робить систему зрозумілою, підтримуваною та придатною для подальшого розширення.

4.3 Реалізація серверної частини

Серверна частина програмного інструменту відповідає за основну логіку роботи системи, обробку запитів від клієнтської частини, взаємодію з базою даних, перевірку прав доступу та формування даних для відображення в інтерфейсі. Вона є проміжним рівнем між Angular-застосунком [4] і базою даних PostgreSQL [6]. Коли користувач виконує дію в інтерфейсі, наприклад створює задачу, змінює її статус або переглядає метрики, клієнтська частина надсилає запит до backend-частини, після чого сервер обробляє ці дані та повертає відповідь.

Backend реалізовано мовою Python [8] із використанням фреймворку Django [9] та Django REST Framework [5]. Django використовується для опису моделей даних, роботи з міграціями, налаштування проєкту та взаємодії з базою даних, а Django REST Framework — для створення API, через яке клієнтська частина отримує й надсилає дані [5, 8, 9]. Такий підхід дозволяє розділити відповідальність між frontend і backend: клієнтська частина відповідає за відображення даних і взаємодію з користувачем, а серверна — за їх обробку, перевірку та збереження.

Структура серверної частини побудована за модульним принципом. Окремі частини backend відповідають за користувачів, робочі простори, проєкти, задачі, підзадачі, теги, учасників і метрики. Завдяки такому поділу код легше підтримувати, оскільки логіка кожного функціонального блоку розміщена у відповідному модулі. Наприклад, робота з задачами не змішується з логікою автентифікації або керуванням робочими просторами, що робить структуру проєкту зрозумілішою.

Основними елементами серверної частини є моделі, серіалізатори, представлення та маршрути. Моделі описують структуру даних і зв'язки між сутностями системи. Серіалізатори перетворюють об'єкти Django у формат JSON і

навпаки, щоб дані могли передаватися між сервером і клієнтською частиною. Представлення обробляють HTTP-запити, виконують потрібні операції та повертають відповідь [15]. Маршрутизація визначає, який саме обробник має виконуватися для конкретного API-запиту.

Окрему роль у серверній частині відіграє перевірка доступу. Оскільки система підтримує робочі простори, проекти, учасників і ролі, backend має контролювати, які дії може виконувати конкретний користувач. Наприклад, користувач не повинен мати доступу до проектів іншого робочого простору, якщо він не є його учасником. Так само окремі дії, пов'язані з керуванням учасниками або налаштуваннями проекту, мають бути доступні лише користувачам із відповідними правами.

Серверна частина також відповідає за формування метрик проекту. Для цього backend використовує дані про задачі, підзадачі, виконавців, дедлайни, story points і статус виконання. На основі цих даних розраховуються показники прогресу, кількість відкритих і завершених задач, прострочені задачі, задачі без виконавців, залишкові story points і навантаження користувачів. Після розрахунку ці дані передаються клієнтській частині для відображення в аналітичній панелі.

Таким чином, серверна частина є центральним рівнем програмної системи, який поєднує інтерфейс користувача з базою даних і забезпечує виконання основної логіки застосунку. Вона обробляє запити, зберігає й оновлює дані, контролює доступ користувачів і формує інформацію, необхідну для відстеження прогресу проекту.

4.3.1 Структура backend-частини застосунку

Backend-частина програмного інструменту реалізована як окремий серверний застосунок, що відповідає за обробку запитів, роботу з базою даних, автентифікацію користувачів, перевірку прав доступу та формування даних для

клієнтської частини. Вона не взаємодіє з користувачем напряду через інтерфейс, але забезпечує всю основну логіку, яка стоїть за діями користувача в системі.

Структура backend-частини побудована на основі Django [9] та Django REST Framework [5]. Django використовується як основа серверного застосунку, а Django REST Framework — для створення API, через яке frontend отримує дані та надсилає зміни на сервер [5, 9]. Такий підхід дозволяє розділити систему на два незалежні рівні: клієнтська частина відповідає за відображення інтерфейсу, а серверна — за збереження, обробку та перевірку даних.

У структурі backend-частини можна виділити кілька основних логічних блоків. Перший блок пов'язаний з обліковими записами користувачів: реєстрацією, входом до системи, профілем користувача та автентифікацією. Другий блок відповідає за робочі простори, їх створення, редагування, вибір активного простору та роботу з учасниками. Третій блок реалізує роботу з проєктами: створення проєкту, редагування його параметрів, додавання учасників і визначення менеджера. Окремий блок пов'язаний із задачами, підзадачами, групами задач, тегами, пріоритетами, виконавцями та строками виконання.

У межах кожного такого блоку використовуються стандартні для Django REST Framework елементи: моделі, серіалізатори, представлення та маршрути [5]. Моделі описують структуру даних і зв'язки між сутностями. Серіалізатори перетворюють дані моделей у формат JSON, який може бути переданий клієнтській частині, а також перевіряють дані, що надходять із frontend. Представлення обробляють HTTP-запити, виконують потрібні дії та повертають відповідь [15]. Маршрути визначають, за якою URL-адресою доступний конкретний API-запит.

Важливою особливістю backend-частини є те, що вона не тільки зберігає дані, а й виконує додаткову логіку. Наприклад, під час створення задачі сервер перевіряє, чи має користувач доступ до відповідного проєкту, чи належить виконавець до команди, чи коректно передані пріоритет, дедлайн і story points. Під час зміни статусу задачі backend оновлює її стан, а ці дані надалі можуть впливати на розрахунок метрик проєкту.

Окремо в серверній частині реалізовано логіку формування аналітичних даних. Вона базується на інформації про задачі, підзадачі, виконавців, строки виконання, story points і статуси. Backend отримує ці дані з бази, виконує необхідні обчислення та передає клієнтській частині вже підготовлену інформацію для відображення на аналітичній панелі. Завдяки цьому frontend не виконує складні розрахунки самостійно, а переважно відповідає за візуальне подання отриманих результатів.

Таким чином, структура backend-частини побудована навколо основних сутностей системи: користувачів, робочих просторів, проєктів, задач і метрик. Такий поділ робить серверну частину зрозумілою для підтримки та подальшого розвитку. У майбутньому до неї можна додавати нові модулі, наприклад історію змін задач, коментарі, додаткові ролі або розширені типи аналітики, не змінюючи загальну логіку роботи застосунку.

4.3.2 Реалізація моделей даних і взаємодії з базою даних

У серверній частині застосунку дані описуються через моделі Django [9]. Саме моделі визначають, які об'єкти існують у системі, які поля вони мають і як пов'язані між собою. Для цього проєкту такий підхід є зручним, тому що більшість даних не існують окремо: робочий простір пов'язаний із користувачами, проєкт належить до робочого простору, задача належить до проєкту, а підзадача є частиною більшої задачі.

Django дозволяє працювати з такими даними через ORM [9]. Це означає, що в коді backend-частина взаємодіє не безпосередньо з SQL-запитами, а з Python-об'єктами [8]. Наприклад, задача в коді сприймається як об'єкт із назвою, описом, виконавцем, дедлайном, пріоритетом, story points і статусом виконання. При цьому Django сам перетворює дії з цим об'єктом у відповідні запити до бази даних [9]. Такий підхід спрощує розробку і робить код зрозумілішим, особливо коли потрібно часто отримувати, оновлювати або фільтрувати пов'язані записи.

Для збереження даних використовується PostgreSQL [6]. У базі даних зберігається вся основна інформація, необхідна для роботи системи: користувачі, робочі простори, проєкти, задачі, підзадачі, теги, виконавці, дедлайни та інші параметри [6]. Оскільки структура застосунку має багато зв'язків, реляційна база даних добре підходить для такого проєкту. Вона дозволяє зберігати дані впорядковано й не дублювати одну й ту саму інформацію в різних місцях.

Зв'язки між моделями реалізуються за допомогою стандартних механізмів Django [9]. Наприклад, якщо один проєкт містить багато задач, то задача має посилання на відповідний проєкт. Якщо задача може мати підзадачі, то для цього використовується зв'язок із батьківською задачею. Якщо користувач може брати участь у кількох проєктах, а проєкт має кількох учасників, для цього використовуються проміжні зв'язки. Завдяки цьому система може коректно показувати тільки ті дані, які належать до конкретного робочого простору або проєкту.

Під час роботи користувача з інтерфейсом серверна частина постійно звертається до моделей. Коли створюється нова задача, backend перевіряє отримані дані, пов'язує задачу з потрібним проєктом, групою та виконавцем, після чого зберігає її в базі. Коли користувач змінює статус задачі або оновлює дедлайн, змінюється відповідний запис у базі даних. Якщо задача позначається як виконана, ця інформація надалі використовується не тільки на дошці задач, а й у метриках проєкту.

Таким чином, моделі Django [9], ORM [9] і PostgreSQL [6] утворюють основу роботи backend-частини з даними. Моделі описують структуру об'єктів системи, ORM забезпечує зручну роботу з ними в коді, а PostgreSQL зберігає дані у впорядкованому вигляді. Завдяки цьому система може стабільно працювати з робочими просторами, проєктами, задачами, підзадачами й метриками без ручного написання більшості SQL-запитів.

4.3.3 Реалізація REST API та обробки запитів

Взаємодія між клієнтською та серверною частинами системи реалізована через REST API [5]. Це дозволяє розділити відповідальність між frontend і backend: клієнтська частина відповідає за інтерфейс і дії користувача, а серверна — за обробку даних, перевірку доступу, роботу з базою даних і повернення результату у форматі JSON. Для створення API використано Django REST Framework, який надає зручні засоби для роботи із серіалізаторами, представленнями, маршрутами та правами доступу [5].

У межах розробленої системи API використовується для всіх основних дій користувача. Через відповідні запити клієнтська частина отримує дані про робочі простори, проєкти, задачі, підзадачі, учасників і метрики. Також через API виконуються операції створення, редагування та видалення об'єктів. Наприклад, створення задачі відбувається через POST-запит, отримання списку задач — через GET-запит, оновлення параметрів задачі — через PATCH або PUT, а видалення — через DELETE [15].

Під час обробки запиту backend спочатку визначає, до якого endpoint звернулася клієнтська частина. Після цього перевіряється, чи користувач автентифікований і чи має він доступ до відповідного робочого простору, проєкту або задачі. Якщо дані передані коректно, сервер виконує потрібну дію: отримує записи з бази, створює новий об'єкт, оновлює наявний або формує аналітичні показники. Результат повертається клієнтській частині у форматі JSON, після чого інтерфейс оновлюється.

Важливу роль у цьому процесі виконують серіалізатори Django REST Framework [5]. Вони перетворюють об'єкти моделей Django у формат, придатний для передачі на frontend, а також перевіряють дані, які надходять від користувача. Наприклад, під час створення задачі серіалізатор перевіряє назву, виконавця, пріоритет, дедлайн і story points. Якщо дані некоректні, сервер повертає повідомлення про помилку, і запис не зберігається.

Окремі API-запити використовуються для формування метрик. У цьому випадку сервер не просто повертає записи з бази даних, а обробляє інформацію про задачі, виконавців, дедлайни, story points і статуси виконання. На основі цих даних backend формує показники прогресу, кількість прострочених задач, навантаження учасників і дані для часової шкали. Це дозволяє залишити основну логіку розрахунків на сервері, а клієнтській частині передавати вже підготовлену інформацію для відображення.

Таким чином, REST API [5] є основним механізмом обміну даними між Angular-застосунком [4] і серверною частиною. Його використання забезпечує структуровану взаємодію між інтерфейсом, бізнес-логікою та базою даних, а також дозволяє коректно обробляти дії користувача в межах робочих просторів, проєктів, задач і метрик.

4.4 Реалізація клієнтської частини

Клієнтська частина є тією частиною програмного інструменту, з якою безпосередньо взаємодіє користувач. Вона відповідає за відображення сторінок застосунку, навігацію між розділами, роботу з формами, відкриття модальних вікон, оновлення даних на дошці задач і візуальне подання метрик. Саме через інтерфейс користувач створює робочі простори, налаштовує проєкти, працює із задачами та переглядає стан виконання роботи.

Для реалізації клієнтської частини використано Angular [4], оскільки цей фреймворк дозволяє будувати вебзастосунок із окремих компонентів і підтримувати зрозумілу структуру інтерфейсу [16]. У розробленій системі frontend не виконує основну бізнес-логіку самостійно, а отримує дані від серверної частини через API, відображає їх у зручному вигляді та передає на backend зміни, які вносить користувач [20]. Завдяки цьому клієнтська частина залишається зосередженою на взаємодії з користувачем, а обробка даних і перевірка доступу виконуються на сервері.

4.4.1 Angular-компоненти та сервіси клієнтського застосунку

Клієнтський застосунок побудований за компонентним підходом Angular [4, 16]. У такій структурі інтерфейс поділяється на окремі частини, кожна з яких відповідає за певний фрагмент сторінки або функціональності. Компонент в Angular складається з TypeScript-класу, HTML-шаблону та стилів, що дозволяє розділити логіку, структуру сторінки й зовнішнє оформлення [16]. Для розробленої системи це зручно, оскільки інтерфейс містить різні екрани: авторизацію, профіль користувача, робочі простори, проєкти, дошку задач, модальні вікна та аналітичні блоки.

У межах застосунку окремі компоненти використовуються для сторінок входу й реєстрації, списку робочих просторів, карток проєктів, груп задач, карток задач і модального вікна задачі. Такий поділ робить код більш зрозумілим: наприклад, компонент задачі відповідає за відображення однієї картки, компонент дошки — за групування задач на сторінці, а компонент модального вікна — за детальний перегляд і редагування параметрів задачі.

Для роботи з даними використовуються Angular-сервіси [17]. Вони винесені окремо від компонентів, щоб не змішувати відображення інтерфейсу з обміном даними із сервером. Сервіси в Angular можуть створюватися як окремі класи й підключатися до компонентів через механізм `dependency injection` [17]. У розробленій системі саме сервіси надсилають HTTP-запити до backend-частини, отримують відповіді від REST API та передають дані компонентам [20].

Наприклад, сервіс робочих просторів відповідає за отримання списку доступних `workspace`, створення нового робочого простору та оновлення його параметрів. Сервіс проєктів працює з даними проєктів у межах вибраного робочого простору. Сервіс задач використовується для створення, редагування, переміщення та оновлення задач і підзадач. Завдяки такому поділу компонент не виконує всю логіку самостійно, а лише викликає потрібний метод сервісу й оновлює інтерфейс після отримання відповіді.

Окремим елементом клієнтської частини є маршрутизація [18]. Вона дозволяє переходити між сторінками застосунку без повного перезавантаження сторінки та підтримувати зрозумілу структуру навігації [18]. У системі маршрути використовуються для сторінок авторизації, профілю, робочих просторів, проєктів, дошки задач і метрик. Завдяки цьому користувач може послідовно переходити між основними розділами та зберігати контекст роботи в межах обраного робочого простору або проєкту.

Таким чином, компоненти [16], сервіси [17] та маршрутизація [18] формують основу клієнтського застосунку. Компоненти відповідають за відображення інтерфейсу, сервіси — за отримання й передавання даних, а маршрутизація об'єднує окремі сторінки в цілісний вебзастосунок.

4.4.2 Реалізація інтерфейсу робочих просторів і проєктів

Інтерфейс робочих просторів і проєктів побудований навколо поточного контексту роботи користувача. Робочий простір у системі виступає окремим середовищем, у межах якого завантажуються проєкти, учасники та пов'язані з ними дані. Завдяки цьому клієнтська частина не працює з усіма проєктами одразу, а відображає лише ті елементи, які належать до вибраного workspace.

Для створення та редагування робочих просторів і проєктів у клієнтській частині використовуються форми. Angular підтримує роботу з формами для введення, перевірки та передавання даних користувача, що є важливим для сторінок створення workspace, редагування його параметрів, створення проєкту та оновлення інформації про нього [19]. У таких формах користувач вводить назву, опис, обирає менеджера або задає строки виконання, а клієнтська частина готує ці дані для передавання на сервер.

Обмін даними між інтерфейсом і backend-частиною виконується через Angular-сервіси [17]. Для цього використовується HTTP-клієнт Angular, який дає змогу frontend-застосунку надсилати запити до серверної частини та отримувати

відповіді від API [20]. Наприклад, після вибору робочого простору сервіс надсилає запит для отримання пов'язаних із ним проєктів, а після створення нового проєкту передає на сервер його параметри разом із прив'язкою до активного workspace.

Важливою частиною реалізації є збереження активного робочого простору в інтерфейсі. Після вибору workspace його дані використовуються іншими компонентами застосунку, зокрема сторінкою проєктів. Це дозволяє не дублювати вибір робочого простору на кожній сторінці та підтримувати єдиний контекст роботи. Якщо користувач переходить до проєктів, система вже розуміє, для якого саме workspace потрібно завантажити список проєктів і учасників.

Інтерфейс проєктів працює за тією ж логікою контекстної залежності. Проєкти створюються й редагуються не ізольовано, а в межах конкретного робочого простору. Під час створення нового проєкту frontend передає на backend не тільки назву, опис, менеджера та строки виконання, а й інформацію про workspace, до якого цей проєкт належить. Завдяки цьому на рівні даних зберігається правильний зв'язок між робочим простором і проєктом.

Окремо реалізована логіка роботи з учасниками проєкту. Додавання учасників залежить від поточного робочого простору, оскільки до проєкту можуть бути включені лише ті користувачі, які пов'язані з відповідним workspace. Такий підхід дозволяє уникнути ситуації, коли до проєкту випадково додається користувач з іншого робочого середовища.

Таким чином, інтерфейс робочих просторів і проєктів побудований як послідовна система контекстів. Робочий простір визначає загальне середовище роботи, проєкт уточнює конкретний напрям діяльності, а клієнтська частина забезпечує передавання цього контексту між формами [19], компонентами [16], сервісами [17] та API-запитами [20].

Висновки до розділу 4

У четвертому розділі було описано програмну реалізацію системи відстеження прогресу проєкту. Перед переходом до окремих частин реалізації було подано діаграми системи: use-case діаграму та концептуальну модель. Вони дозволили показати, як користувач і керівник проєкту взаємодіють із застосунком, а також які основні сутності лежать в основі роботи системи.

Архітектуру програмного інструменту побудовано за клієнт-серверним принципом. Такий поділ дав змогу відокремити інтерфейс користувача від серверної логіки та бази даних. Клієнтська частина забезпечує взаємодію з користувачем, серверна частина приймає й обробляє запити, а база даних зберігає інформацію про користувачів, робочі простори, проєкти, задачі, підзадачі та пов'язані з ними параметри.

Під час опису серверної частини було розглянуто її загальну структуру, реалізацію моделей даних, взаємодію з PostgreSQL [6] та обробку запитів через REST API [5]. Backend у системі відповідає не лише за збереження даних, а й за перевірку їх коректності, зв'язок між окремими об'єктами, підготовку відповідей для клієнтської частини та забезпечення стабільної роботи основних сценаріїв.

Клієнтську частину реалізовано на основі Angular-компонентів [4, 16] і сервісів [17]. Такий підхід дозволив розділити інтерфейс на окремі логічні блоки й організувати обмін даними із сервером. Окремо було описано реалізацію інтерфейсу робочих просторів і проєктів, оскільки саме вони задають основний контекст роботи користувача в системі.

Отже, у розділі показано, як загальна ідея програмного інструменту була перенесена в конкретну програмну реалізацію. Система має зрозумілу архітектуру, окрему серверну й клієнтську частини, структуровану роботу з даними та основу для подальшого використання в користувацьких сценаріях.

5 РОБОТА КОРИСТУВАЧА З ПРОГРАМНИМ ІНСТРУМЕНТОМ

Розроблена система призначена для відстеження прогресу проєкту, тому основна увага в інтерфейсі приділяється роботі із задачами, проєктами, робочими просторами та метриками. Робота із застосунком побудована послідовно: користувач проходить реєстрацію або вхід, створює чи вибирає воркспейс, налаштовує проєкт і учасників, працює із задачами та підзадачами, а потім відстежує стан виконання через дошку й метрики проєкту.

5.1 Реєстрація та автентифікація користувача

Після відкриття застосунку користувач потрапляє на сторінку входу. Поле пароля має кнопку для перегляду введених символів. Якщо користувач уже має обліковий запис, він вводить дані та натискає кнопку Sign in. Сторінку входу користувача наведено на рисунку 5.1.

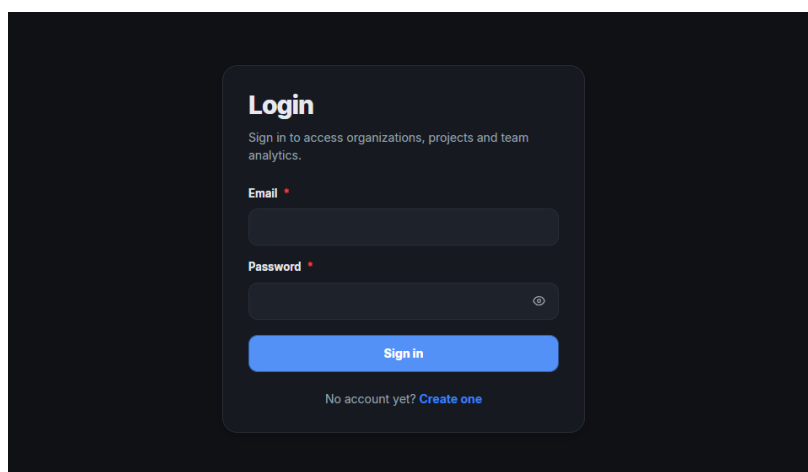


Рисунок 5.1 – Сторінка входу користувача

Якщо користувач ще не зареєстрований, він може перейти до створення акаунта за посиланням Create one. Сторінку реєстрації користувача наведено на рисунку 5.2.

Рисунок 5.2 – Сторінка реєстрації користувача

Після заповнення форми користувач натискає кнопку Create account. Якщо дані введені коректно, створюється новий обліковий запис, після чого користувач може перейти до сторінки входу.

5.2 Створення та вибір робочого простору

Після входу в систему користувач може перейти до розділу Workspaces через бокове меню.

Для створення нового робочого простору користувач відкриває форму, де потрібно вказати назву воркспейсу, а за потреби додати опис і завантажити іконку. Форму створення робочого простору наведено на рисунку 5.3.

Рисунок 5.3 – Форма створення робочого простору

Після створення робочий простір з'являється у списку на сторінці Workspaces. Користувач бачить його назву, опис, іконку та свою роль у цьому воркспейсі. Для додавання нового робочого простору передбачена окрема картка Add workspace. Список доступних робочих просторів наведено на рисунку 5.4.

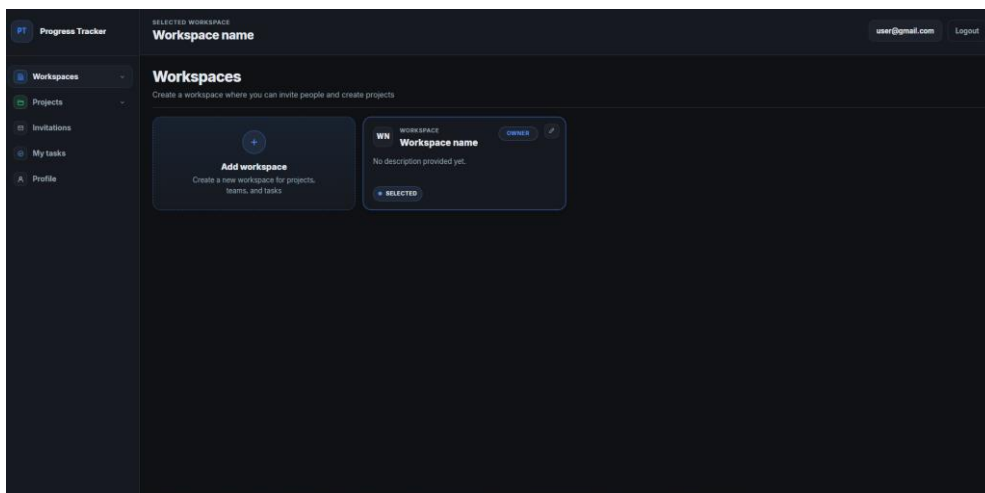


Рисунок 5.4 – Список робочих просторів користувача

Користувач може вибрати потрібний робочий простір зі списку. Після вибору він позначається як Selected, а його назва відображається у верхній частині сторінки в блоці Selected workspace. Вибраний воркспейс використовується як поточний контекст для подальшої роботи з проєктами та іншими розділами застосунку.

Окремо передбачена сторінка налаштувань робочого простору. На ній користувач може переглянути назву, опис, роль поточного користувача та дату створення. Також доступне редагування назви, опису й іконки. Сторінку налаштувань робочого простору наведено на рисунку 5.5.

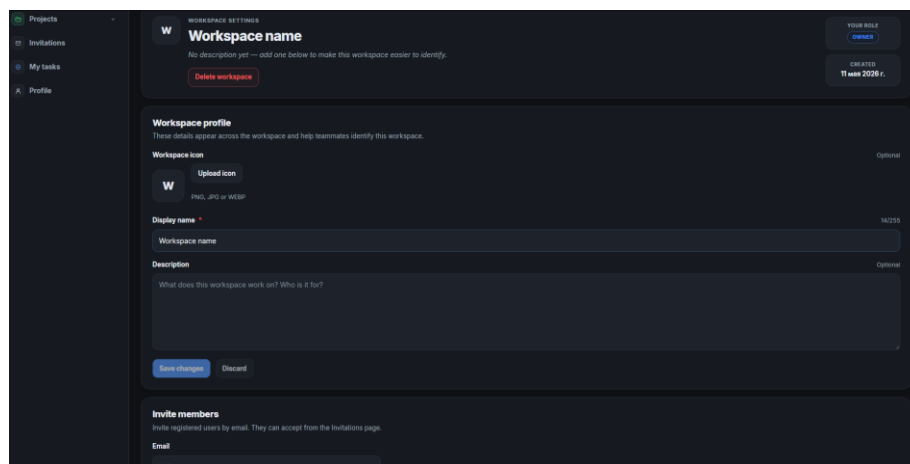


Рисунок 5.5– Сторінка налаштувань робочого простору

Якщо користувач має відповідні права, він може видалити робочий простір або запросити нових учасників за email.

5.3 Створення та редагування проєкту

Після вибору робочого простору користувач може перейти до розділу Projects. На цій сторінці відображаються проєкти поточного воркспейсу. Якщо проєкти ще не створені, система показує повідомлення про їх відсутність і пропонує створити перший проєкт.

Для створення нового проєкту користувач відкриває форму Create project. У ній потрібно вказати назву проєкту, за потреби додати опис, обрати менеджера та зазначити планову тривалість роботи. Назва є обов'язковим полем, а опис, менеджер і дати допомагають краще описати проєкт для команди. Форму створення проєкту наведено на рисунку 5.6.

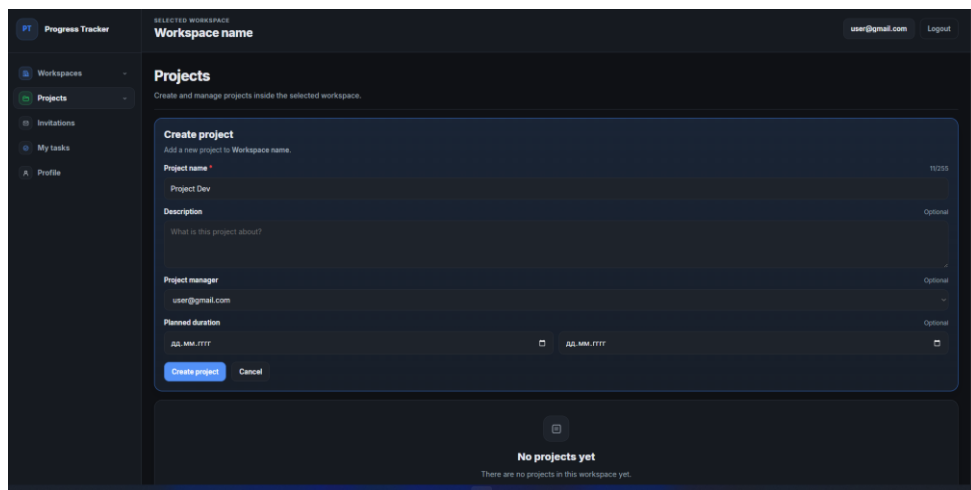


Рисунок 5.6 – Форма створення проєкту

Після створення проєкт з'являється у списку Projects і в боковому меню в межах вибраного воркспейсу. У картці проєкту відображаються його назва, опис, команда, менеджер і планова тривалість. Список проєктів у вибраному робочому просторі наведено на рисунку 5.7.

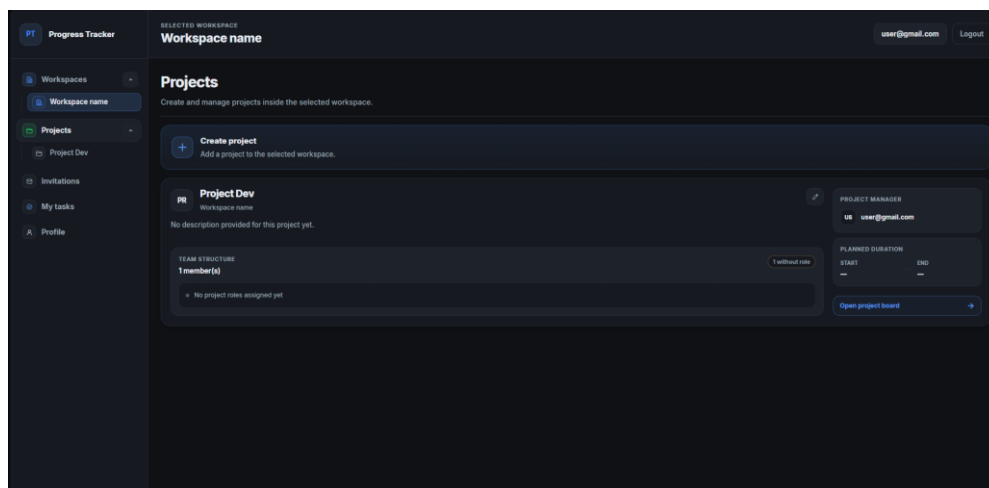


Рисунок 5.7 – Список проєктів робочого простору

Для редагування проєкту користувач відкриває сторінку налаштувань. На ній можна змінити назву, опис, менеджера, дату початку та дату завершення. Також відображається коротка інформація про проєкт. Якщо користувач має відповідні права, він може оновити дані та зберегти зміни. Сторінку редагування проєкту наведено на рисунку 5.8.

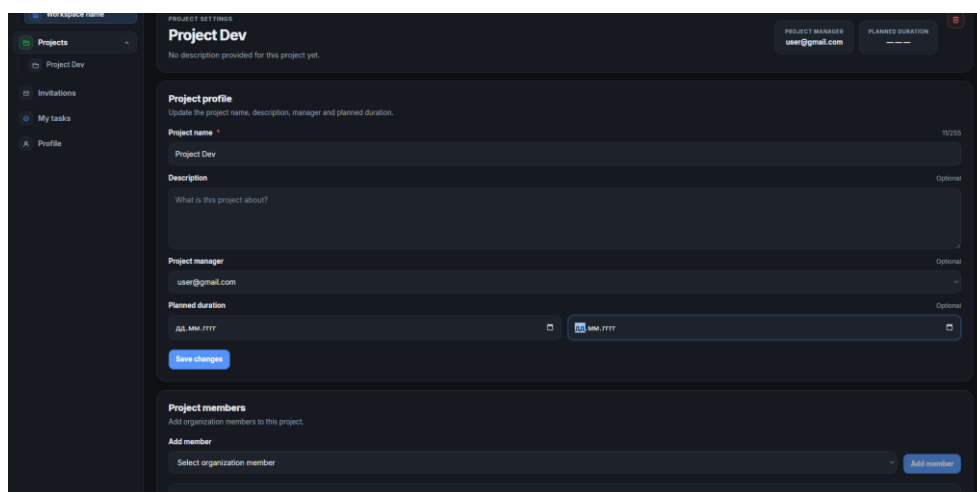


Рисунок 5.8 – Сторінка редагування проєкту

Нижче на сторінці редагування розміщено блок Project members. Він використовується для додавання учасників до конкретного проєкту. Користувач може обрати учасника з робочого простору та додати його до команди, після чого цьому учаснику можна призначати задачі та ролі.

5.4 Робота з дошкою задач

Після відкриття проєкту користувач переходить на сторінку задач. Основною частиною цієї сторінки є дошка, поділена на окремі групи. Кожна група відповідає певному етапу роботи. Дошку задач проєкту наведено на рисунку 5.9.

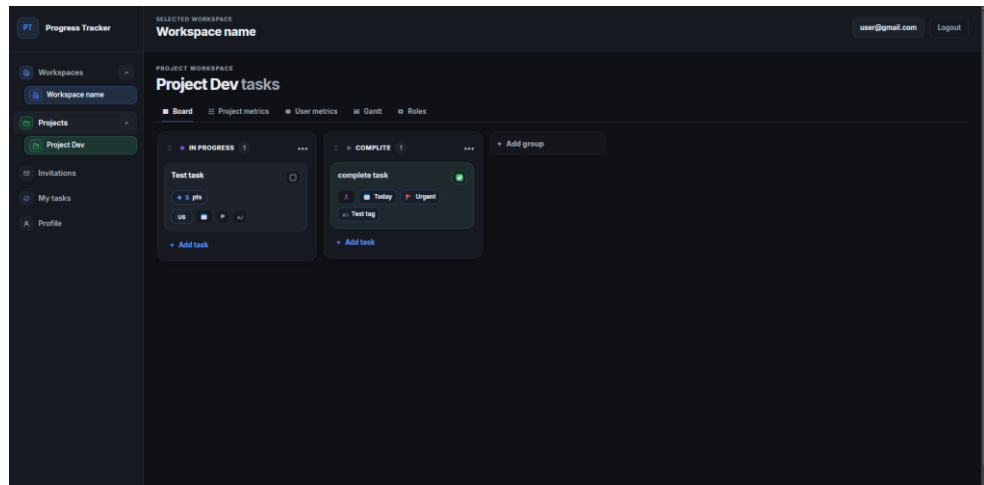


Рисунок 5.9 – Дошка задач проєкту

На дошці користувач може створювати нові групи за допомогою кнопки Add group. Це дозволяє налаштовувати структуру дошки під конкретний проєкт. Також у кожній групі можна додавати задачі за допомогою кнопки Add task.

Під час створення задачі користувач вводить її назву та може одразу додати основні параметри: виконавця, дату, пріоритет, story points і тег. Після збереження задача з'являється у відповідній групі дошки. Приклад створення задачі на дошці наведено на рисунку 5.10.

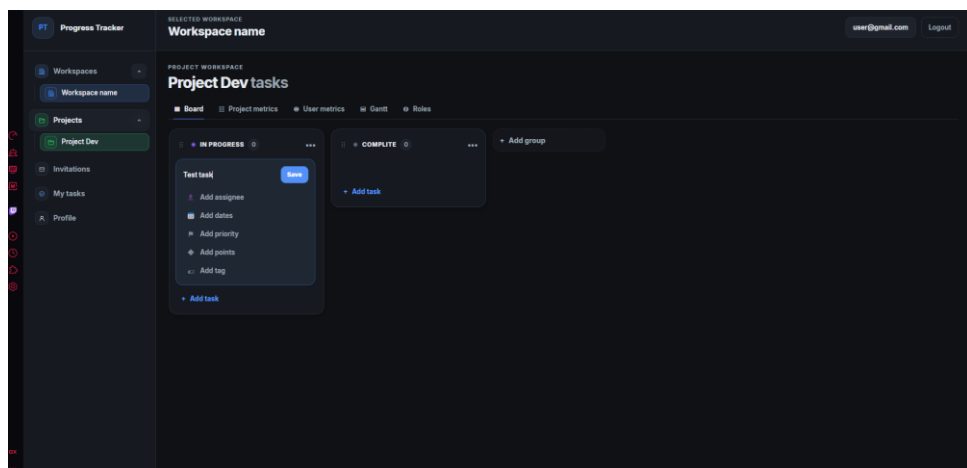


Рисунок 5.10 – Створення задачі на дошці

Після створення задача відображається у вигляді картки. На ній показується назва задачі та основні параметри: story points, виконавець, дата, пріоритет і тег. Якщо задача завершена, вона має відповідну позначку та візуально відрізняється від відкритих задач.

Для детального перегляду користувач може відкрити задачу в модальному вікні. У ньому доступні повна назва, опис, теги, виконавець, пріоритет, дата початку, дедлайн, story points і статус виконання. Також відображається коротке зведення по задачі: поточний статус, кількість підзадач і story points. Детальне вікно задачі наведено на рисунку 5.11.

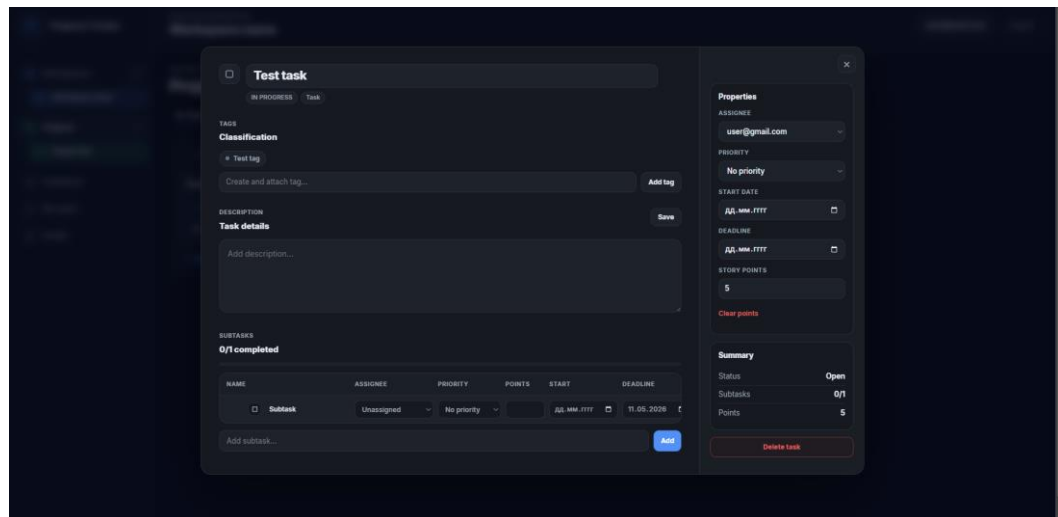


Рисунок 5.11 – Детальне вікно задачі

Окремо в модальному вікні реалізована робота з підзадачами. Користувач може додавати підзадачі, вказувати для них виконавця, пріоритет, story points, дату початку та дедлайн. Це дозволяє розбити складну задачу на менші частини та контролювати її виконання детальніше.

Дані, які користувач вносить під час роботи із задачами, надалі використовуються для формування метрик, Gantt chart і користувацької аналітики. Тому дошка задач є не лише інструментом організації роботи, а й основою для відстеження прогресу проєкту.

5.5 Перегляд метрик проєкту та користувачів

Для аналізу стану проєкту в застосунку передбачено окремі вкладки з метриками. Користувач може перейти до Project metrics, щоб переглянути загальні показники виконання проєкту, або до User metrics, щоб оцінити розподіл роботи між учасниками команди.

На сторінці Project metrics відображаються основні показники проєкту: загальний стан, прогрес за story points, відсоток завершення задач і блок ризиків. Користувач може побачити кількість виконаних і відкритих задач, залишковий обсяг роботи, прострочені задачі, задачі з близьким дедлайном і задачі без виконавця. Сторінку загальних метрик проєкту наведено на рисунку 5.12.

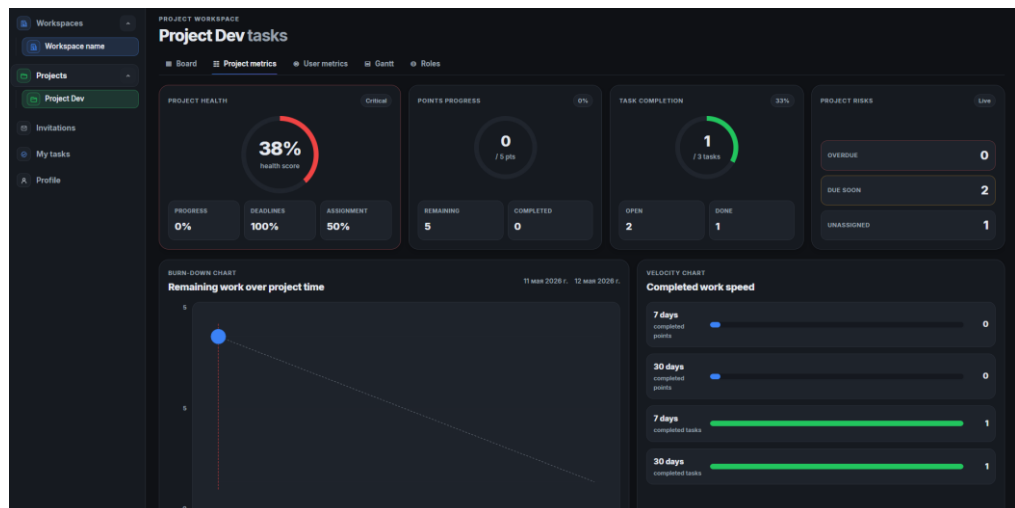


Рисунок 5.12 – Сторінка метрик проєкту

Окремо на цій сторінці показано візуалізацію залишку роботи за story points. Вона використовується як допоміжне представлення для швидкої оцінки обсягу незавершеної роботи, а не як повна історична діаграма змін за кожен день. Також відображається швидкість виконання роботи за останні 7 та 30 днів.

Для оцінки роботи учасників використовується вкладка User metrics. На ній показано кількість учасників проєкту, призначених задач, командний відсоток завершення та задачі без виконавця. Нижче відображається розподіл навантаження: для кожного учасника показано призначені задачі, завершені story points і story

points, які ще залишилися в роботі. Сторінку метрик користувачів наведено на рисунку 5.13.

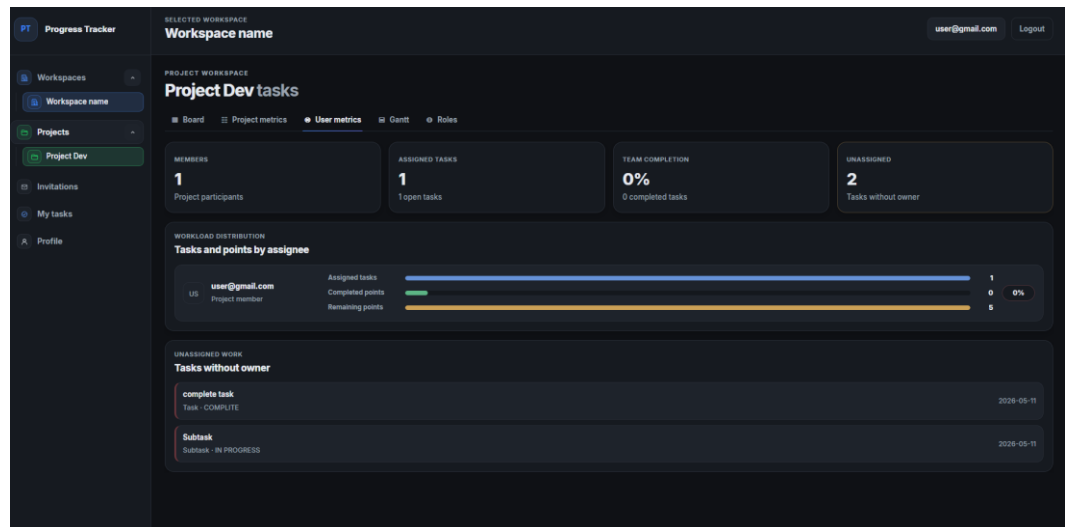


Рисунок 5.13 – Сторінка метрик користувачів

Для перегляду часової структури проєкту використовується вкладка Gantt. У ній задачі відображаються на часовій шкалі відповідно до дати початку та дедлайну. Сторінку часової шкали Gantt наведено на рисунку 5.14.

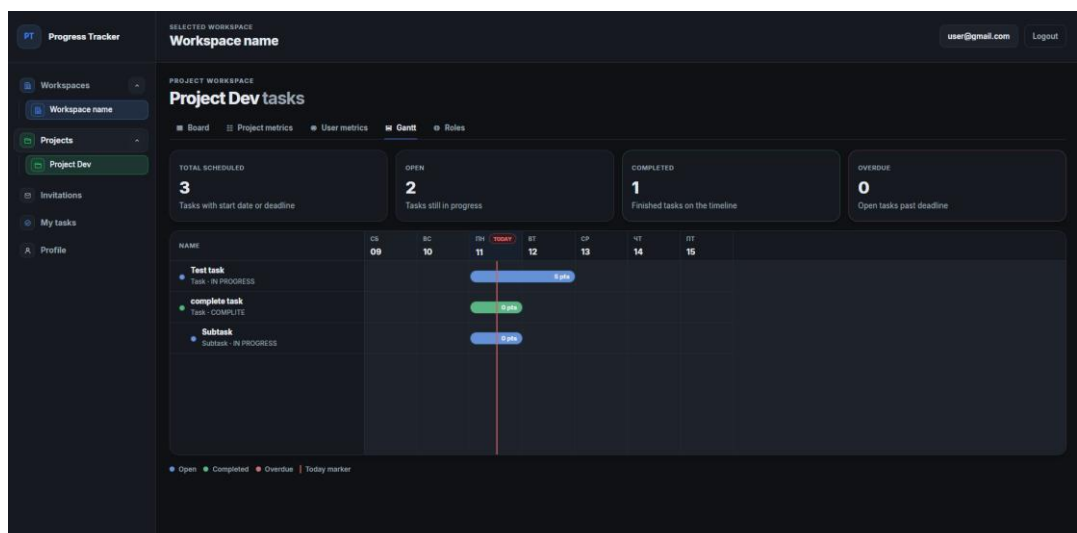


Рисунок 5.14 – Часова шкала Gantt для задач проєкту

Користувач бачить заплановані, відкриті, завершені й прострочені задачі, а також маркер поточного дня.

Висновки до розділу 5

У п'ятому розділі описано роботу користувача з програмним інструментом відстеження прогресу проєкту. Основну увагу приділено практичним сценаріям використання системи: реєстрації та входу, створенню робочого простору, роботі з проєктами, дошкою задач і перегляду метрик.

Робота із застосунком побудована послідовно. Користувач спочатку входить до системи, після чого створює або вибирає робочий простір. Саме робочий простір визначає подальший контекст роботи: у його межах створюються проєкти, додаються учасники та формується структура задач. Такий підхід дозволяє розділяти різні напрями роботи й не змішувати дані кількох команд або проєктів.

Окремо показано роботу з дошкою задач. Користувач може створювати групи, додавати задачі, змінювати їхні параметри, призначати виконавців, встановлювати дедлайни, story points, пріоритети й теги. Для складніших задач передбачені підзадачі, що дають змогу деталізувати роботу та контролювати виконання окремих етапів.

Важливою частиною користувацького сценарію є перегляд аналітичних даних. Метрики проєкту та користувачів допомагають оцінити загальний стан виконання, побачити прострочені задачі, залишковий обсяг роботи, навантаження учасників і часову структуру задач. Завдяки цьому система працює не лише як інструмент для ведення задач, а й як засіб контролю прогресу проєкту.

Отже, розроблений програмний інструмент забезпечує повний шлях роботи користувача: від входу в систему до створення робочого простору, ведення задач і перегляду метрик. Це підтверджує практичну придатність застосунку для організації командної роботи та відстеження стану виконання проєкту.

ВИСНОВКИ

У даній дипломній роботі було розроблено програмний інструмент для відстеження прогресу проєкту з візуалізацією ключових метрик. Основна ідея роботи полягала в тому, щоб створити вебзастосунок, який дає змогу не лише вести задачі, а й швидко оцінювати загальний стан виконання проєкту. У процесі роботи було визначено проблему, сформовано вимоги до системи, розглянуто існуючі інструменти управління проєктами, обрано засоби розробки та реалізовано клієнтську і серверну частини застосунку.

На етапі постановки задачі було встановлено, що для ефективної командної роботи недостатньо мати простий список задач. У проєкті важливо бачити, які задачі вже виконано, які ще залишаються відкритими, хто відповідає за їх виконання, які строки наближаються та як розподілене навантаження між учасниками. Тому розроблювана система була орієнтована на роботу з робочими просторами, проєктами, задачами, підзадачами, виконавцями, дедлайнами, story points і метриками.

Під час аналізу існуючих систем управління проєктами було розглянуто Jira, Asana та Trello. Ці інструменти мають широкі можливості для організації командної роботи, планування задач і контролю строків. Водночас для невеликих команд, навчальних або внутрішніх проєктів вони можуть бути надмірно складними або потребувати додаткового налаштування. Це підтвердило доцільність створення власного програмного інструменту, який поєднує просту роботу із задачами та наочне відображення ключових показників прогресу.

Для реалізації системи було обрано клієнт-серверну архітектуру. Клієнтську частину розроблено з використанням Angular і TypeScript, серверну частину — мовою Python із використанням Django та Django REST Framework, а для збереження даних використано PostgreSQL. Для запуску компонентів системи застосовано Docker, а для контролю версій і збереження коду — Git і GitHub. Такий набір засобів дозволив розділити застосунок на окремі логічні частини: frontend

відповідає за інтерфейс користувача, backend — за обробку запитів і основну логіку, а база даних — за збереження структурованої інформації.

У межах програмної реалізації було спроектовано архітектуру системи, побудовано use-case діаграму та концептуальну модель. Серверна частина забезпечує роботу з моделями даних, міграціями, REST API, обробкою запитів і взаємодією з базою даних. Клієнтська частина побудована на основі Angular-компонентів і сервісів, що дозволило реалізувати інтерфейс робочих просторів, проєктів, задач і метрик у зрозумілій структурі.

Розроблений програмний інструмент забезпечує повний користувацький сценарій роботи. Користувач може пройти реєстрацію та автентифікацію, створити або вибрати робочий простір, створити проєкт, додати учасників, працювати з дошкою задач, створювати задачі й підзадачі, призначати виконавців, встановлювати дедлайни, пріоритети, story points і теги. Окремо реалізовано перегляд метрик проєкту та користувачів, що дозволяє оцінювати загальний стан виконання, навантаження учасників, прострочені задачі, залишковий обсяг роботи та часову структуру задач.

Практичне значення роботи полягає в тому, що створений вебзастосунок може використовуватися невеликими командами, навчальними групами або внутрішніми проєктними командами для організації роботи та контролю виконання задач. Система не перевантажена зайвими функціями, але містить основні можливості, необхідні для ведення задач і відстеження прогресу. У подальшому її можна розширити за рахунок коментарів до задач, історії змін, сповіщень, розширених ролей користувачів, експорту звітів та інтеграцій із зовнішніми сервісами.

Отже, поставлену мету дипломної роботи досягнуто. Було створено програмний інструмент, який поєднує керування задачами, роботу з підзадачами, командну взаємодію та візуалізацію ключових метрик проєкту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Jira. Atlassian. URL: <https://www.atlassian.com/software/jira> (дата звернення: 31.05.2026).
2. Asana's Work Tracking & Project Management Features. Asana. URL: <https://asana.com/features> (дата звернення: 31.05.2026).
3. What is Trello: Learn Features, Uses & More. Trello. URL: <https://trello.com/tour> (дата звернення: 31.05.2026).
4. What is Angular? Angular. URL: <https://angular.dev/overview> (дата звернення: 31.05.2026).
5. Django REST Framework. Django REST Framework. URL: <https://www.django-rest-framework.org/> (дата звернення: 31.05.2026).
6. PostgreSQL: About. PostgreSQL. URL: <https://www.postgresql.org/about/> (дата звернення: 31.05.2026).
7. PyCharm: The Python IDE for Professional Developers. JetBrains. URL: <https://www.jetbrains.com/pycharm/> (дата звернення: 01.06.2026).
8. Python 3.14.5 documentation. Python. URL: <https://docs.python.org/> (дата звернення: 01.06.2026).
9. Django: The web framework for perfectionists with deadlines. Django. URL: <https://www.djangoproject.com/> (дата звернення: 01.06.2026).
10. JavaScript. MDN Web Docs. URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript> (дата звернення: 01.06.2026).
11. TypeScript: JavaScript With Syntax For Types. TypeScript. URL: <https://www.typescriptlang.org/> (дата звернення: 01.06.2026).
12. Docker overview. Docker Docs. URL: <https://docs.docker.com/get-started/docker-overview/> (дата звернення: 01.06.2026).
13. Git documentation. Git. URL: <https://git-scm.com/doc> (дата звернення: 01.06.2026).

14. GitHub Docs. GitHub Docs. URL: <https://docs.github.com/> (дата звернення: 01.06.2026).

15. HTTP request methods. MDN Web Docs. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods> (дата звернення: 01.06.2026).

16. Anatomy of components. Angular. URL: <https://angular.dev/guide/components> (дата звернення: 01.06.2026).

17. Creating and using services. Angular. URL: <https://angular.dev/guide/di/creating-and-using-services> (дата звернення: 01.06.2026).

18. Routing • Overview. Angular. URL: <https://angular.dev/guide/routing> (дата звернення: 01.06.2026).

19. Forms • Overview. Angular. URL: <https://angular.dev/guide/forms> (дата звернення: 01.06.2026).

20. HTTP Client • Overview. Angular. URL: <https://angular.dev/guide/http> (дата звернення: 01.06.2026).

ДОДАТОК А Лістинг розробленої системи

```
from django.db.models import Count, Q
from django.shortcuts import get_object_or_404
from rest_framework import permissions, status
from rest_framework.response import Response
from rest_framework.views import APIView
from datetime import timedelta
from django.utils import timezone
from .metrics import build_project_metrics, build_project_user_metrics
from projects.models import Project
```

```
from .functions import with_task_metrics
from .models import Task, TaskGroup, TaskTag
from .permissions import can_manage_project, has_project_access
from .serializers import (
    TaskCreateSerializer,
    TaskDetailSerializer,
    TaskGroupCreateSerializer,
    TaskGroupSerializer,
    TaskSerializer,
    TaskSubtaskCreateSerializer,
    TaskSubtaskSerializer,
    TaskTagCreateSerializer,
    TaskTagSerializer,
    TaskUpdateSerializer,
)
```

```
class MyTasksView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request):
        tasks = (
            with_task_metrics(
                Task.objects
                .filter(assignee=request.user)
                .select_related("group", "group__project", "assignee")
                .prefetch_related("tags")
            )
        )
```

```

        .order_by("is_completed", "deadline", "position", "id")
    )

    serializer = TaskSerializer(tasks, many=True)
    return Response(serializer.data)

class ProjectMetricsView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request, project_id):
        project = get_object_or_404(Project, id=project_id)

        if not has_project_access(request.user, project):
            return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

        data = build_project_metrics(project)
        return Response(data)

class ProjectUserMetricsView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request, project_id):
        project = get_object_or_404(Project, id=project_id)

        if not has_project_access(request.user, project):
            return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

        data = build_project_user_metrics(project)
        return Response(data)

class TaskGroupsByProjectView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request, project_id):
        project = get_object_or_404(Project, id=project_id)

        if not has_project_access(request.user, project):
            return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

```

```

groups = (
    TaskGroup.objects
    .filter(project=project)
    .annotate(task_count=Count("tasks", filter=Q(tasks__parent_task__isnull=True)))
    .order_by("position", "id")
)

serializer = TaskGroupSerializer(groups, many=True)
return Response(serializer.data)

def post(self, request, project_id):
    project = get_object_or_404(Project, id=project_id)

    serializer = TaskGroupCreateSerializer(
        data=request.data,
        context={
            "request": request,
            "project": project,
        },
    )
    serializer.is_valid(raise_exception=True)
    group = serializer.save()

    response_serializer = TaskGroupSerializer(group)
    return Response(response_serializer.data, status=status.HTTP_201_CREATED)

class TaskGroupDetailView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def patch(self, request, group_id):
        group = get_object_or_404(
            TaskGroup.objects.select_related("project"),
            id=group_id,
        )

        if not can_manage_project(request.user, group.project):
            return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

```

```

        serializer = TaskGroupSerializer(
            group,
            data=request.data,
            partial=True,
        )
        serializer.is_valid(raise_exception=True)
        group = serializer.save()

    return Response(TaskGroupSerializer(group).data)

def delete(self, request, group_id):
    group = get_object_or_404(
        TaskGroup.objects.select_related("project"),
        id=group_id,
    )

    if not can_manage_project(request.user, group.project):
        return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

    if group.tasks.exists():
        return Response(
            {"detail": "Cannot delete group with tasks."},
            status=status.HTTP_400_BAD_REQUEST,
        )

    group.delete()
    return Response(status=status.HTTP_204_NO_CONTENT)

class TaskTagsByProjectView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request, project_id):
        project = get_object_or_404(Project, id=project_id)

        if not has_project_access(request.user, project):
            return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

        tags = TaskTag.objects.filter(project=project).order_by("name", "id")
        serializer = TaskTagSerializer(tags, many=True)

```

```

        return Response(serializer.data)

def post(self, request, project_id):
    project = get_object_or_404(Project, id=project_id)

    serializer = TaskTagCreateSerializer(
        data=request.data,
        context={
            "request": request,
            "project": project,
        },
    )
    serializer.is_valid(raise_exception=True)
    tag = serializer.save()

    response_serializer = TaskTagSerializer(tag)
    return Response(response_serializer.data, status=status.HTTP_201_CREATED)

class TaskTagDetailView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def patch(self, request, tag_id):
        tag = get_object_or_404(
            TaskTag.objects.select_related("project"),
            id=tag_id,
        )

        if not has_project_access(request.user, tag.project):
            return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

        serializer = TaskTagSerializer(
            tag,
            data=request.data,
            partial=True,
        )
        serializer.is_valid(raise_exception=True)
        tag = serializer.save()

        return Response(TaskTagSerializer(tag).data)

```

```

def delete(self, request, tag_id):
    tag = get_object_or_404(
        TaskTag.objects.select_related("project"),
        id=tag_id,
    )

    if not has_project_access(request.user, tag.project):
        return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

    tag.delete()
    return Response(status=status.HTTP_204_NO_CONTENT)


class TasksByProjectView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request, project_id):
        project = get_object_or_404(Project, id=project_id)

        if not has_project_access(request.user, project):
            return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

        tasks = (
            with_task_metrics(
                Task.objects
                .filter(group__project=project, parent_task__isnull=True)
                .select_related("group", "assignee")
                .prefetch_related("tags")
            )
            .order_by("position", "id")
        )

        serializer = TaskSerializer(tasks, many=True)
        return Response(serializer.data)

    def post(self, request, project_id):
        project = get_object_or_404(Project, id=project_id)

        serializer = TaskCreateSerializer(

```

```

        data=request.data,
        context={
            "request": request,
            "project": project,
        },
    )
    serializer.is_valid(raise_exception=True)
    task = serializer.save()

    task = with_task_metrics(
        Task.objects
        .select_related("group", "assignee")
        .prefetch_related("tags")
    ).get(id=task.id)

    response_serializer = TaskSerializer(task)
    return Response(response_serializer.data, status=status.HTTP_201_CREATED)

class TaskDetailView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get_task(self, task_id):
        return get_object_or_404(
            with_task_metrics(
                Task.objects
                .select_related("group", "group__project", "assignee")
                .prefetch_related("tags")
            ),
            id=task_id,
        )

    def get(self, request, task_id):
        task = self.get_task(task_id)
        project = task.group.project

        if not has_project_access(request.user, project):
            return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

        serializer = TaskDetailSerializer(task)

```

```

        return Response(serializer.data)

    def patch(self, request, task_id):
        task = self.get_task(task_id)

        serializer = TaskUpdateSerializer(
            task,
            data=request.data,
            partial=True,
            context={"request": request},
        )
        serializer.is_valid(raise_exception=True)
        task = serializer.save()

        task = self.get_task(task.id)

        response_serializer = TaskDetailSerializer(task)
        return Response(response_serializer.data)

    def delete(self, request, task_id):
        task = get_object_or_404(
            Task.objects.select_related("group", "group__project"),
            id=task_id,
        )

        project = task.group.project

        if not can_manage_project(request.user, project):
            return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

        task.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)

class TaskSubtasksView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get_parent_task(self, task_id):
        return get_object_or_404(
            Task.objects.select_related("group", "group__project"),

```

```

        id=task_id,
    )

def get(self, request, task_id):
    parent_task = self.get_parent_task(task_id)
    project = parent_task.group.project

    if not has_project_access(request.user, project):
        return Response({"detail": "Forbidden."}, status=status.HTTP_403_FORBIDDEN)

    subtasks = (
        parent_task.subtasks
        .select_related("group", "assignee")
        .prefetch_related("tags")
        .order_by("position", "id")
    )

    serializer = TaskSubtaskSerializer(subtasks, many=True)
    return Response(serializer.data)

def post(self, request, task_id):
    parent_task = self.get_parent_task(task_id)

    serializer = TaskSubtaskCreateSerializer(
        data=request.data,
        context={
            "request": request,
            "parent_task": parent_task,
        },
    )
    serializer.is_valid(raise_exception=True)
    subtask = serializer.save()

    subtask = with_task_metrics(
        Task.objects
        .select_related("group", "assignee")
        .prefetch_related("tags")
    ).get(id=subtask.id)
    response_serializer = TaskSubtaskSerializer(subtask)
    return Response(response_serializer.data, status=status.HTTP_201_CREATED)

```

ДОДАТОК В Презентація



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

ПРОГРАМНИЙ ІНСТРУМЕНТ ДЛЯ ВІДСТЕЖЕННЯ ПРОГРЕСУ ПРОЄКТУ З ВІЗУАЛІЗАЦІЄЮ КЛЮЧОВИХ МЕТРИК

виконав: студент групи ТВ-23 Тополук Дмитро Сергійович

керівник: доцент Шпурик Вадим Вадимович, кандидат технічних наук

2026



Актуальність теми

Єдине середовище роботи

Задачі, учасники, строки та стан проєкту мають бути зібрані в одному місці.

Координація команди

Команді важливо бачити, хто за що відповідає і на якому етапі перебуває робота.

Контроль прогресу

Керівнику проєкту потрібно швидко оцінювати виконання без ручного збору інформації.

Прозорість процесу

Метрики допомагають вчасно помічати затримки, навантаження та проблемні місця.



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»



2/9

Постановка задачі

Розробка
вебзастосунку

Дошка задач

Робочі простори

Підзадачі та
параметри задач

Проекти та
учасники

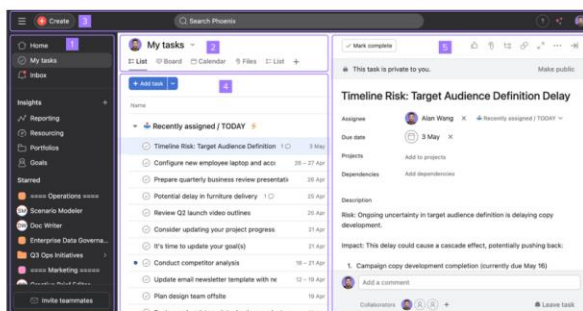
Метрики прогресу



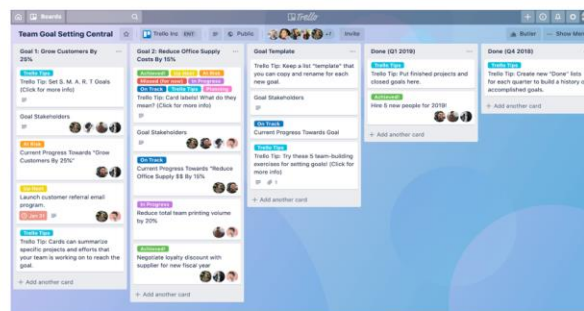
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

3/9

Аналіз предметної області



Asana



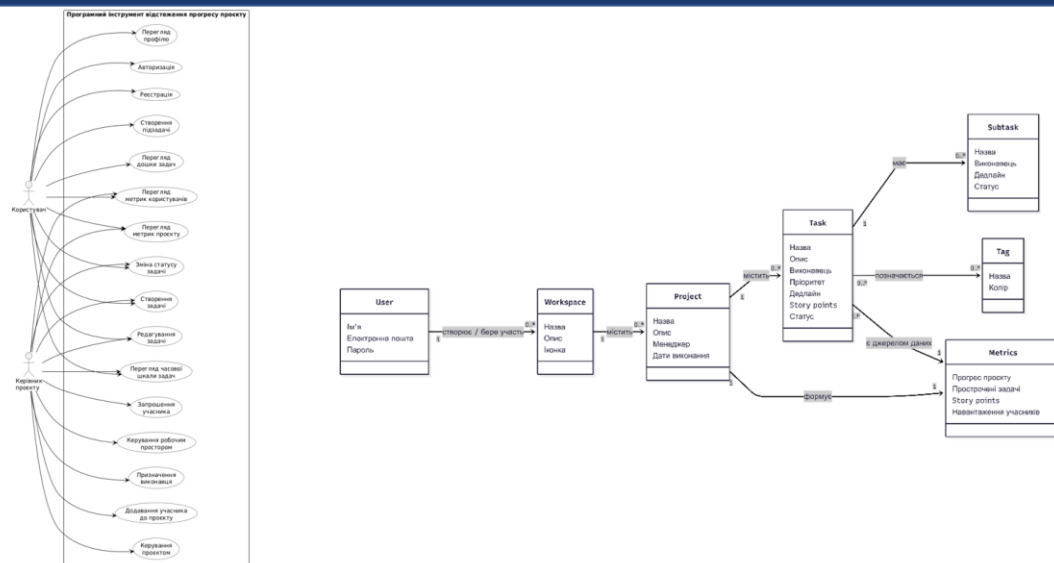
Trello



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

4/9

Проектування системи



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

5/9

Засоби розробки

TypeScript

django

PostgreSQL

python

Angular

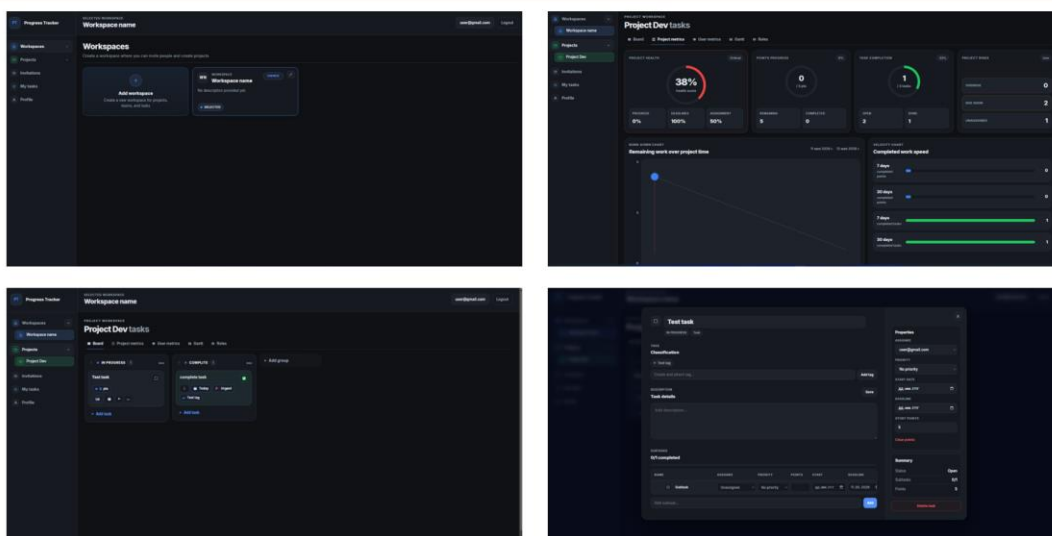
docker



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

6/9

UI



Висновки

	Розроблено програмний інструмент для відстеження прогресу проекту	
	Система не обмежується простою дошкою задач;	
	Реалізовано перегляд загального стану проекту, прогресу та навантаження учасників	
	Інструмент допомагає виявляти проблемні місця в роботі команди	
	Рішення орієнтоване на невеликі команди та навчальні проекти;	
	Подальший розвиток: коментарі, історія змін, сповіщення, експорт звітів та інтеграції.	



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"