НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ
кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

“До захисту допущено”
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА

“___”_____2024 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою
“Цифрові технології в енергетиці”
зі спеціальності *122 “Комп’ютерні науки”*

на тему: Система керування ресурсами та задачами проектів на основі предметно-орієнтованого програмування

Виконав:
студент IV курсу, групи ТР-01

Степаненко Микола Федорович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник:
доцент, к.т.н., Володимир ТИХОХОД _____
(посада, вчене звання ,науковий ступінь, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент: *доцент каф. інженерії програмного забезпечення в енергетиці,*
доцент, к.т.н., Олександр ГАГАРІН _____
(посада, вчене звання, науковий ступінь, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль: *старший викладач Ольга БЕСПАЛА* _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2024

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

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти - перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА

«__»_____2024 р.

ЗАВДАННЯ
на дипломну роботу студенту

СТЕПАНЕНКУ Миколі Федоровичу
(прізвище, ім’я, по батькові)

1. Тема роботи **Система керування ресурсами та задачами проектів на основі предметно-орієнтованого програмування**

керівник роботи ***Тихоход Володимир Олександрович, к.т.н., доцент***
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «___»_____ 202__ р. №_____

2. Термін подання роботи студентом **07.06.2024 року**

3. Вихідні дані до роботи мови програмування C# та TypeScript, фреймворки ASP.NET Core та Angular, СКБД PostgreSQL, Docker, система контролю версій Git, середовища розробки WebStorm та Rider, інструмент для тестування Postman

4. Перелік питань, які потрібно розробити
1) провести аналіз серед наявних аналогів
2) система для керування ресурсами та задачами проектів

3) аргументувати вибрані інструменти, технології та алгоритми для реалізації програмного забезпечення

4) побудувати архітектури програмного забезпечення, баз даних та налаштувати шлюз передачі даних

5) створити прикладний програмний веб-інтерфейс

6) представити процес застосування веб-інтерфейсу

5. Орієнтований перелік ілюстративного матеріалу діаграми, що демонструють роботу алгоритмів веб-інтерфейсу, архітектуру системи, бази даних, взаємодію користувачів із системою; приклади роботи з програмним продуктом.

6. Дата видачі завдання «15» вересня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Затвердження теми роботи		
2.	Вивчення та аналіз задачі	17-20.04.24	
3.	Розробка архітектури та загальної структури системи	21-25.04.24	
4.	Розробка частин окремих підсистем	25.04-02.05.24	
5.	Програмна реалізація системи	03-07.05.24	
6.	Оформлення пояснювальної записки	08-12.05.24	
7.	Захист програмного продукту	13-17.05.24	
8.	Передзахист	03-06.06.24	
9.	Подання готової роботи на кафедру	07.06.2024	
10.	Захист	17-21.06.2024	

Студент _____ **Микола СТЕПАНЕНКО**
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ **Володимир ТИХОХОД**
(підпис) (ім'я, ПРІЗВИЩЕ)

АНОТАЦІЯ

Дипломна робота виконана на 61 сторінках, містить 32 ілюстрацій, 3 таблиці, 1 додаток, 21 джерел в переліку посилань.

Мета роботи — створення платформи для керування задачами та ресурсами проектів на основі предметно-орієнтованого програмування.

Методи та засоби: мови програмування C# та TypeScript, фреймворки ASP.NET Core та Angular, СКБД PostgreSQL, Docker, система керування версіями Git, DDD архітектура, принципи SOLID, KISS, DRY.

Результат — веб-платформа, яка дозволяє користувачам керувати задачами та ресурсами проектів.

Ключові слова: DDD, КЕРУВАННЯ ПРОЕКТАМИ, ООП.

ABSTRACT

The diploma work consists of 61 pages, contains 32 illustrations, 3 tables, 1 appendix, 21 sources in the list of references.

The purpose of the work — to create a platform for managing project tasks and resources based on object-oriented programming.

Methods and tools: C# and TypeScript programming languages, ASP.NET Core and Angular frameworks, PostgreSQL database, Docker, Git version control system, DDD architecture, SOLID, KISS, DRY principles.

The result — a web platform that allows users to manage tasks and project resources.

Keywords: DDD, PROJECT MANAGEMENT, OOP.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

DDD (Domain-Driven Design) — предметно-орієнтоване проектування, підхід до розробки складних програмних систем, який фокусується на основних бізнес-концепціях і логіці предметної області.

ПЗКЗП (Системи керування ресурсами та задачами проектів) — інформаційні системи, що допомагають командам та організаціям будь-якого розміру ефективно керувати своїми проектами, ресурсами та завданнями, забезпечуючи планування, моніторинг та аналіз виконання проектів.

KISS (Keep It Simple, Stupid) — принцип програмування, що закликає до спрощення систем.

DRY (Don't Repeat Yourself) — принцип програмування, що закликає уникати дублювання коду.

SOLID — набір принципів об'єктно-орієнтованого програмування для покращення розробки та підтримки програмного забезпечення.

ВСТУП

Керування ресурсами та задачами проектів є важливою задачею в різних сферах бізнесу, особливо в сфері інформаційних технологій. У сучасному динамічному середовищі, де зміни відбуваються постійно, управління обмеженими ресурсами та задачами з мінливими вимогами є складним викликом. Ефективне управління ресурсами дозволяє оптимізувати витрати, підвищити продуктивність та забезпечити своєчасне виконання проектів, що є критичним для досягнення успіху в конкурентному середовищі.

Підтримка інформаційних систем має важливе значення для побудови ефективної системи керування ресурсами та задачами. Інформаційні системи забезпечують автоматизацію процесів управління, що дозволяє знизити ризики, підвищити точність і оперативність прийняття рішень, а також забезпечити прозорість і контроль виконання задач.

Існує ряд систем, що надають функції управління проектами, серед яких можна виділити такі відомі платформи як Azure DevOps [2], Jira [3] та Trello [4]. Однак, ці системи мають свої недоліки, серед яких: складність у використанні, обмежений функціонал для безкоштовного використання, залежність від хмарних технологій, а також недостатня адаптація до специфічних потреб різних предметних областей.

Актуальною є розробка системи, що надаватиме можливості гнучкого управління ресурсами та задачами проектів з урахуванням специфічних вимог конкретних галузей. Така система повинна забезпечувати високий рівень автоматизації процесів, інтеграцію з іншими інформаційними системами, а також мати можливість адаптації під мінливі вимоги бізнесу.

Оскільки дана проблемна область є досить складною, то доцільно використати шаблони предметно-орієнтованого програмування для реалізації системи. Предметно-орієнтоване програмування дозволяє створювати моделі, що

відображають реальні бізнес-процеси, та забезпечувати тісний зв’язок між програмним забезпеченням і предметною областю.

Метою даної роботи є розробка системи керування ресурсами та задачами проектів на основі предметно-орієнтованого програмування, що дозволить оптимізувати управління проектами, забезпечити високу точність виконання задач та підвищити прозорість процесів управління ресурсами.

Перший розділ даної роботи містить постановку задачі, визначає загальні завдання, які розв’язуються розробленою системою.

У другому розділі розкриваються алгоритми вирішення задачі, які використовувались при реалізації програмного забезпечення.

У третьому розділі обґрунтовано вибір засобів і технологій для створення системи керування задачами та ресурсами проєктів, а також програмну реалізацію, архітектуру та перелік використаних засобів і моделей.

У четвертому розділі описано процес розгортання системи, подано приклади взаємодії з нею, а також проведено аналіз наявних систем, аналогічних до створеної.

1 ЗАДАЧА РОЗРОБКИ СИСТЕМИ КЕРУВАННЯ РЕСУРСАМИ ТА ЗАДАЧАМИ ПРОЕКТІВ НА ОСНОВІ ПРЕДМЕТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

У сучасному динамічному бізнес-середовищі, де конкуренція та вимоги до ефективності постійно зростають, важливим аспектом успіху підприємств і організацій є ефективне управління проектами. Однією з ключових складових цього управління є системи керування ресурсами та завданнями, які забезпечують оптимальне використання доступних ресурсів і контроль за виконанням проектних задач.

Системи керування проектами дозволяють значно підвищити продуктивність команди, зменшити ризики та забезпечити своєчасне виконання проектів. Вони надають можливість координувати роботу різних підрозділів, забезпечувати прозорість і підзвітність процесів, а також оптимізувати використання людських і матеріальних ресурсів.

Предметно-орієнтоване програмування пропонує новітні підходи до розробки таких систем, дозволяючи створювати програмні продукти, які тісно пов'язані з конкретною предметною областю і забезпечують більш інтуїтивне та ефективне управління проектами.

1.1 Задача системи керування ресурсами та задачами проектів

Основні проблеми, з якими стикаються існуючі системи керування проектами, включають недостатню автоматизацію, обмежені можливості інтеграції та низьку прозорість процесів управління. Багато сучасних систем не адаптовані до специфічних потреб різних предметних областей, що ускладнює їх використання і знижує ефективність управління проектами. Це призводить до

збільшення витрат часу і ресурсів, а також знижує продуктивність команди і якість виконання проектів.

Завданням даної роботи є розробка системи керування ресурсами та завданнями проектів, яка буде базуватися на предметно-орієнтованому програмуванні. Ця система повинна забезпечити структурований підхід до планування, виконання та контролю проектів, дозволяючи командам працювати більш організовано та ефективно. Основні завдання системи включають:

— Планування ресурсів: Система повинна допомагати визначити, які ресурси (людські, матеріальні, фінансові тощо) необхідні для успішного виконання проекту. Вона має дозволяти ефективно розподіляти ресурси та встановлювати пріоритети, забезпечуючи оптимальне використання ресурсів у процесі реалізації проекту.

— Управління задачами: Система повинна дозволяти створювати, відстежувати та керувати задачами, необхідними для виконання проекту. Це включає призначення відповідальних осіб, встановлення термінів виконання та моніторинг прогресу робіт.

— Моніторинг та аналіз: Система має надавати засоби для візуалізації та аналізу поточного стану проектів. Вона повинна забезпечувати можливість перегляду поточної статистики та інших інструментів для оцінки ефективності виконання проектів і виявлення можливих проблем або ризиків.

— Інтеграція з іншими системами: Система повинна мати можливість інтеграції з іншими інформаційними системами та платформами, що використовуються в організації, для забезпечення безперервного обміну даними.

— Гнучкість та адаптивність: Система повинна забезпечувати можливість налаштування під специфічні вимоги різних проектів і галузей, що дозволяє використовувати її у широкому спектрі бізнес-середовищ.

Розробка та впровадження такої системи дозволить бізнесам і організаціям значно підвищити ефективність управління проектами, оптимізувати використання ресурсів і забезпечити високу якість виконання завдань. Це є критичним фактором успіху в сучасному конкурентному середовищі.

1.2 Огляд методів предметно-орієнтованого програмування для управління проектами

Предметно-орієнтоване програмування є підходом до розробки програмного забезпечення, який передбачає створення систем, тісно пов'язаних із конкретною предметною областю. Це дозволяє значно підвищити ефективність і зручність використання програмних систем для управління проектами. У цьому підрозділі буде розглянуто основні методи та підходи предметно-орієнтованого програмування, які можуть бути застосовані для розробки систем керування ресурсами та задачами проектів.

Один з основних методів предметно-орієнтованого програмування полягає у створенні моделі предметної області, яка відображає основні сутності, їх властивості та взаємозв'язки, характерні для конкретної галузі. Це дозволяє забезпечити більш точне відображення реальних процесів у програмному забезпеченні. Модель предметної області служить основою для розробки функціональності системи і може включати діаграми класів, об'єктні моделі та інші формалізовані описи.

Предметно-орієнтоване програмування передбачає використання спеціалізованих мов специфікації, які дозволяють описувати вимоги та поведінку системи у термінах, зрозумілих для експертів предметної області. Це можуть бути формальні або напівформальні мови, які забезпечують чіткість і недвозначність опису функціональності системи. Прикладом таких мов є UML (Unified Modeling Language) та DSL (Domain-Specific Languages).

Одним з важливих методів предметно-орієнтованого програмування є автоматична генерація коду з моделей предметної області. Це дозволяє зменшити кількість помилок, що виникають під час ручного кодування, та прискорити процес розробки. Генерація коду може здійснюватися на основі шаблонів або за допомогою спеціалізованих інструментів, які перетворюють моделі у програмний код, готовий до виконання.

Предметно-орієнтоване програмування також включає методи інтеграції розроблюваних систем з іншими інформаційними системами, які використовуються в організації. Це може бути досягнуто шляхом розробки інтерфейсів для обміну даними, використання API або інтеграційних платформ. Інтеграція дозволяє забезпечити безперервний обмін інформацією та підвищити загальну ефективність роботи системи.

Для спрощення розробки системи на основі предметно-орієнтованого програмування широко використовуються спеціалізовані фреймворки та бібліотеки, які забезпечують готові компоненти та інструменти для побудови систем управління проектами. Це дозволяє зосередитися на специфічних вимогах предметної області, зменшуючи витрати часу на розробку базової функціональності.

1.3 Задачі для серверної частини системи

Серверна частина системи керування ресурсами та задачами проектів відіграє ключову роль у забезпеченні функціональності та продуктивності системи. При визначенні задач серверної частини, потрібно враховувати, що вона має бути реалізована на основі предметно-орієнтованого програмування [1]. Таким чином, основні задачі для серверної частини включають:

— Забезпечення бази даних: Розробка та підтримка бази даних для зберігання інформації про проекти, завдання, користувачів та інші важливі дані. Це включає в себе проектування оптимальної структури бази даних, реалізацію механізмів забезпечення цілісності даних та оптимізацію швидкодії доступу до даних. Особливу увагу слід приділити забезпеченню безпеки даних, включаючи механізми резервного копіювання та відновлення, а також захисту від несанкціонованого доступу.

— Моделювання домену: Проектування та реалізація моделей домену, що відображають ключові концепції та бізнес-правила домену. Це включає в себе

ідентифікацію сутностей, значимих для бізнес-логіки системи, та визначення їх взаємодії. Важливим аспектом є забезпечення відповідності моделей реальним бізнес-процесам, що дозволить зменшити розрив між технічною реалізацією та практичними потребами користувачів.

— Управління доменними об'єктами: Визначення інтерфейсів та методів для маніпулювання доменними об'єктами. Це включає в себе реалізацію команд, які виконують дії над об'єктами домену. Особливу увагу слід приділити гнучкості інтерфейсів, щоб забезпечити можливість легкого розширення функціональності системи без суттєвих змін у вже існуючому коді.

— Забезпечення цілісності даних: Реалізація механізмів забезпечення цілісності даних у відповідності до бізнес-правил. Це може включати в себе перевірку даних перед збереженням, валідацію вхідних даних та інші заходи. Важливо забезпечити надійну обробку виключень та механізми відновлення даних у випадку виникнення помилок.

— Управління контекстами: Розділення системи на контексти, кожен з яких відповідає певній області домену та має свої власні моделі, сервіси та правила. Це дозволяє зменшити складність системи та підтримувати її масштабованість. Важливо забезпечити чітке розмежування відповідальності між контекстами та уникнути надмірних залежностей між ними.

— Інтеграція з іншими шарами: Забезпечення взаємодії серверної частини з іншими шарами системи (наприклад, інтерфейсом користувача, шарами доступу до даних) з урахуванням принципів предметно-орієнтованого програмування та збереження чистоти архітектури. Особливу увагу слід приділити забезпеченню зручності API, щоб інші частини системи могли легко взаємодіяти з серверною частиною.

— Забезпечення масштабованості та надійності: Розробка та впровадження механізмів, які забезпечують масштабованість системи для обробки великого обсягу даних та користувачів, а також забезпечення надійності та доступності сервісів навіть у випадку виникнення непередбачених ситуацій. Важливо

враховувати можливість горизонтального масштабування системи, що дозволить легко додавати нові сервери у разі збільшення навантаження.

— Інтеграція з іншими системами: Розробка та налагодження механізмів інтеграції з іншими програмними системами, такими як інструменти розробки, системи контролю версій, електронні поштові сервіси тощо, для забезпечення зручності та ефективності використання системи керування ресурсами та задачами проектів. Важливо забезпечити можливість безшовної інтеграції з існуючими системами та технологіями, що використовуються в організації, для забезпечення цілісності процесів.

— Моніторинг та логування: Впровадження механізмів моніторингу та логування для відстеження стану системи, виявлення та усунення проблем на ранніх стадіях. Це включає налаштування системи збору та аналізу логів, а також інструментів для моніторингу продуктивності та використання ресурсів. Важливо забезпечити можливість автоматичного оповіщення адміністраторів у разі виникнення критичних ситуацій.

— Забезпечення безпеки: Реалізація заходів для забезпечення безпеки системи, включаючи захист від зовнішніх атак, управління доступом користувачів та захист конфіденційної інформації. Важливо враховувати сучасні стандарти безпеки та регулярно оновлювати систему для захисту від нових загроз.

Розробка та впровадження ефективної серверної частини системи керування ресурсами та задачами проектів є ключовим аспектом забезпечення її надійності, продуктивності та безпеки. Дотримання принципів предметно-орієнтованого програмування дозволяє створювати гнучкі та масштабовані системи, які відповідають сучасним вимогам бізнесу.

1.4 Наявні системи для керування ресурсами та задачами проектів

У сучасному світі існує безліч систем керування ресурсами та задачами проектів (ПЗКЗП), кожна з яких пропонує власний набір функцій та можливостей.

Це різноманіття обумовлене зростанням складності проектів, різноманітністю потреб користувачів та швидким розвитком технологій. Різноманіття ПЗКЗП на ринку свідчить про великий попит на ці інструменти, що допомагають командам та організаціям будь-якого розміру успішно керувати своїми проектами. Розглянемо найбільш популярні готові програмні рішення більш детально.

Однією з таких систем є Azure DevOps [2] (рисунок 1.1). Це інтегрована платформа для керування життєвим циклом розробки програмного забезпечення. Вона надає інструменти для спільної роботи, автоматизації, відстеження задач, керування ресурсами та багато іншого. Основні можливості Azure DevOps включають спільну роботу команди, планування та відстеження використання ресурсів, налаштування автоматичних процесів, відстеження прогресу виконання задач та інтеграцію з іншими інструментами. Однак, до мінусів цього рішення можна віднести складність у використанні, обмежений функціонал для безкоштовного використання, залежність від хмарних технологій та надлишковість можливостей.

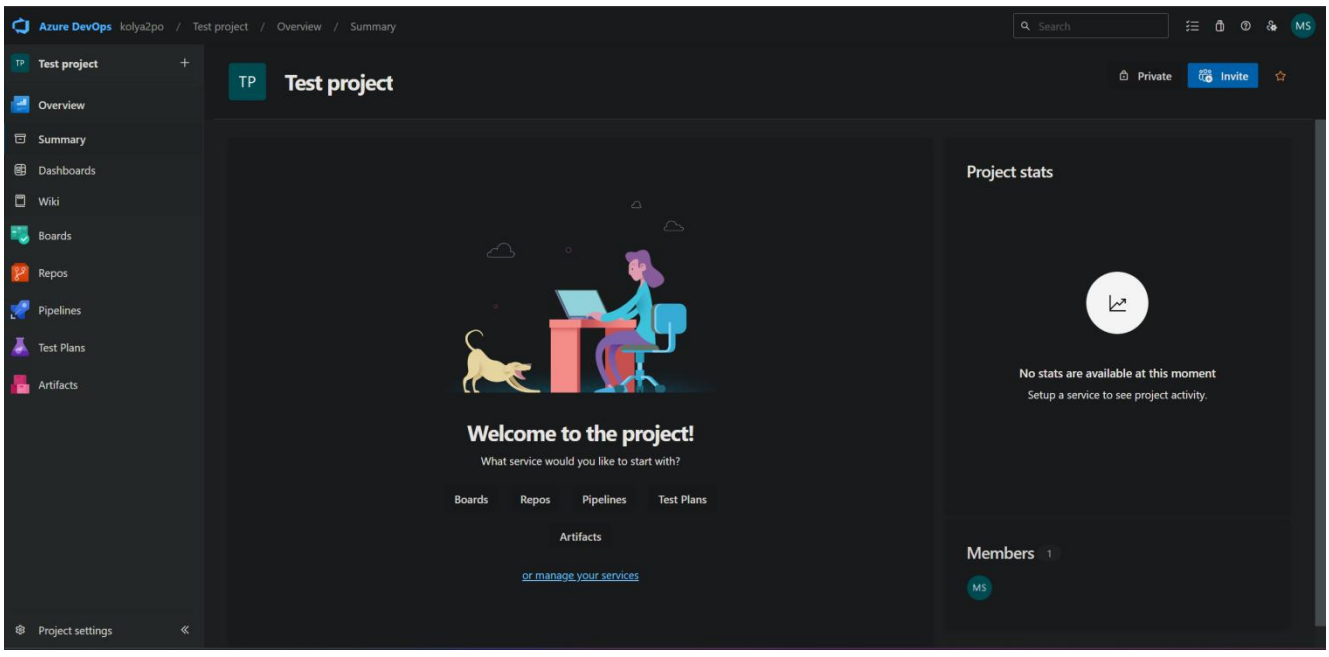


Рисунок 1.1 — Інтерфейс Azure DevOps

Ще однією популярною системою є Jira [3] (рисунок 1.2). Jira використовується великою кількістю організацій по всьому світу та надає

широкий спектр функцій для керування задачами, ресурсами, трекінгом часу, плануванням та іншими аспектами проектного менеджменту. Вона дозволяє створювати, призначати та відстежувати задачі на всіх етапах проекту, створювати детальні плани проектів, відстежувати використання ресурсів та співпрацювати в команді. До недоліків Jira можна віднести складність налаштування, високу вартість для великих команд та потребу в тренуванні користувачів.

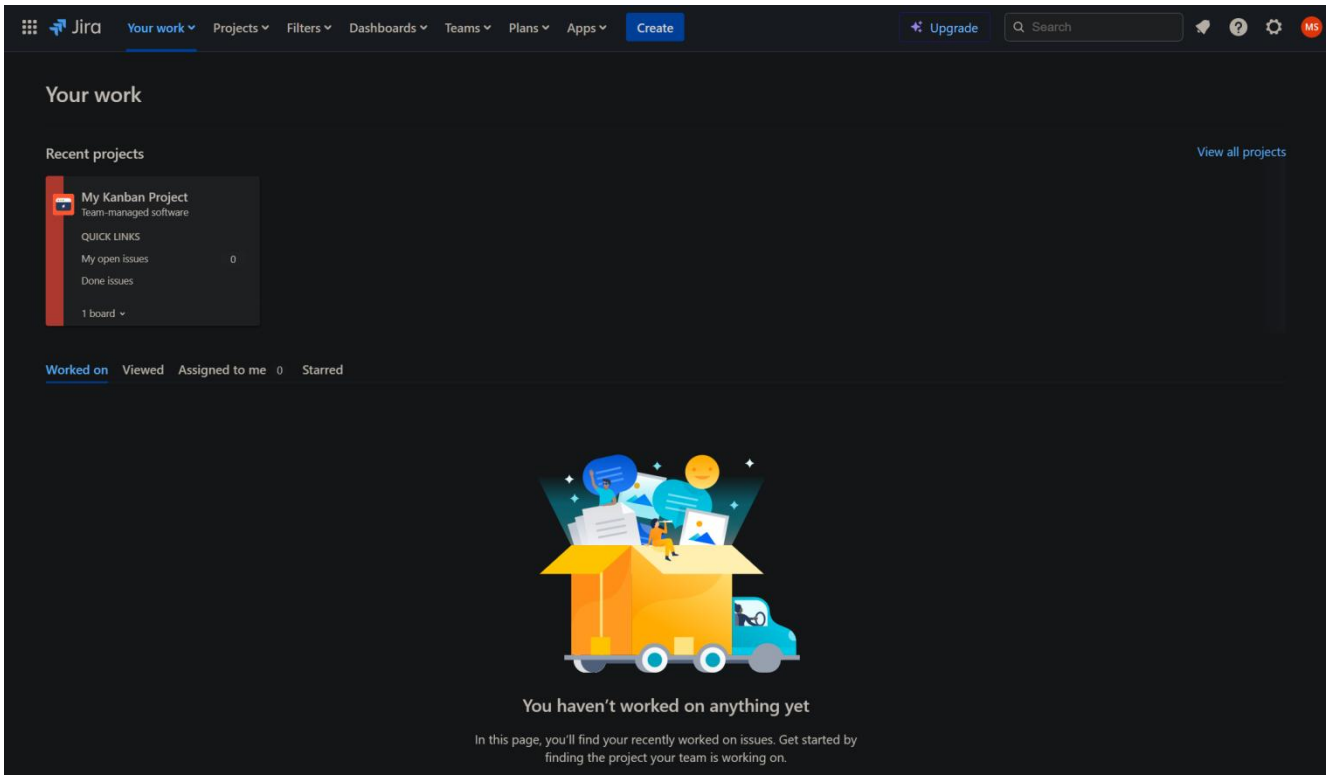


Рисунок 1.2 — Інтерфейс Jira

Trello [4] (рисунок 1.3) є ще одним простим і гнучким інструментом для керування проектами, який використовує методологію Kanban для організації роботи. Він дозволяє користувачам створювати дошки, списки та картки для організації та пріоритизації своєї роботи. Основні можливості Trello включають організацію роботи, співпрацю в команді, інтеграцію з іншими інструментами та мобільність. Однак, Trello має свої недоліки, такі як обмеження у функціональності безкоштовної версії, відсутність деяких розширених функцій

керування проектами та потребу в додаткових інструментах для керування більш складними проектами.

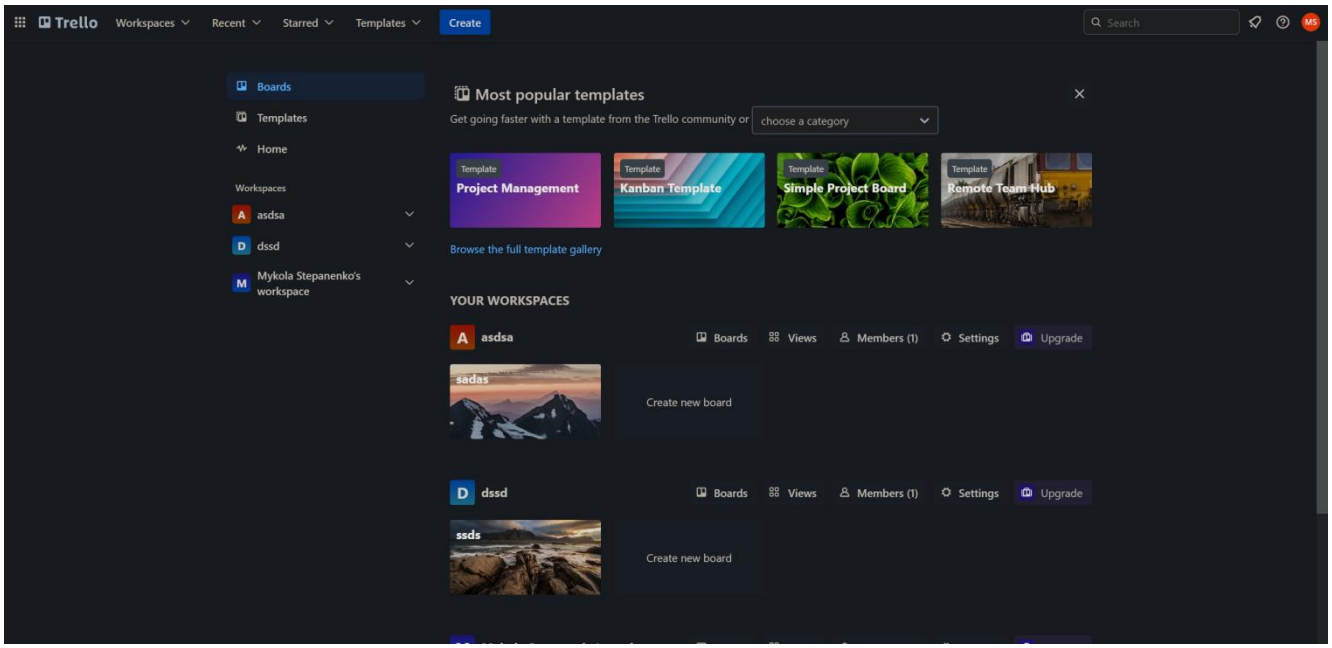


Рисунок 1.3 — Інтерфейс Trello

Для наочності було складено таблицю порівняння основних характеристик трьох найбільш популярних систем керування ресурсами та задачами проектів: Azure DevOps, Jira та Trello.

Таблиця 1 — Порівняння Azure DevOps, Jira та Trello

Параметр	**Azure DevOps**	**Jira**	**Trello**
Основні можливості	Інструменти для спільної роботи, автоматизація, відстеження задач, керування ресурсами	Керування задачами, ресурсами, трекінг часу, планування	Організація роботи, співпраця в команді, мобільність, інтеграція з іншими інструментами
Спільна робота	Так	Так	Так
Автоматизація	Так	Так	Ні
Інтеграція з іншими інструментами	Так	Так	Так

Кінець таблиці 1

Вартість	Обмежений функціонал безкоштовно, платна версія дорога	Висока вартість для великих команд	Безкоштовна версія з обмеженим функціоналом, платна версія доступна
Складність використання	Висока	Висока	Низька
Гнучкість налаштувань	Висока	Висока	Середня
Залежність від хмарних технологій	Так	Так	Ні
Потреба в тренуванні користувачів	Так	Так	Ні

Проаналізувавши вище наведені системи, можна сказати, що вони здійснили значний прогрес у спрощенні та автоматизації процесів керування проектами. Проте, необхідно визнати, що деякі з цих систем мають обмеження та недоліки, які можна виправити. Такий аналіз допомагає не лише уникнути повторних помилок, але й зрозуміти, які аспекти можна покращити та удосконалити в новій системі.

2 РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

Під час розробки програмного продукту було використано підхід, заснований на принципах предметно-орієнтованого програмування. Серверну частину реалізовано за допомогою технологій ASP.NET Core [6], EF Core [8] та мови програмування C# [9]. Для створення веб-додатку були використані такі технології, як Angular [10], TypeScript [11], HTML [12] та SCSS. В процесі створення системи було використано технологію Git [13], яка дозволила слідкувати за еволюцією та змінами коду між версіями додатка. Завдяки цій технології, програмний код доступний на віддаленому сховищі GitHub [20]. У якості середовищ розробки використано JetBrains Rider [17] та JetBrains WebStorm [18].

2.1 Предметно-орієнтоване програмування

Предметно-орієнтоване програмування (Domain-Driven Design, DDD) було обрано для розробки системи керування ресурсами та задачами проектів завдяки його здатності точно відображати специфіку конкретної предметної області. Основна перевага DDD полягає у створенні моделей, що відображають реальні бізнес-процеси, забезпечуючи тісний зв'язок між програмним забезпеченням і бізнес-вимогами. Це дозволяє швидко реагувати на зміни та забезпечувати високу гнучкість системи.

DDD дозволяє створювати структури, стійкі до змін, що знижує витрати на підтримку та розвиток системи в довгостроковій перспективі. Використання цього підходу допомагає ефективніше розподіляти ресурси, знижуючи ризики та підвищуючи продуктивність проекту.

Основний аспект DDD – створення предметних моделей, що відображають сутності та їх взаємозв'язки, що підвищує ефективність управління проектами. Наприклад, у нашій системі були змодельовані сутності “Проект” та “Задача” для забезпечення відповідності реальним бізнес-процесам.

Предметно-орієнтоване програмування є підходом до розробки програмного забезпечення, який передбачає створення систем, тісно пов'язаних із конкретною предметною областю. Це дозволяє значно підвищити ефективність і зручність використання програмних систем [1].

Порівняно з підходом [5], орієнтованим на дані, DDD зменшує складність підтримки проектів з часом, що зображено на рисунку 2.1.

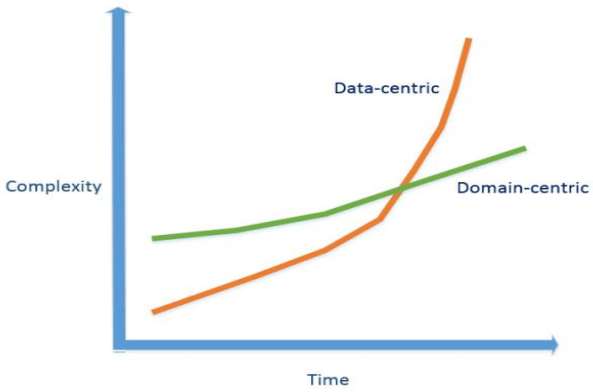


Рисунок 2.1 — Порівняння Domain-centric та Data-centric підходів до проектування інформаційних систем

2.2 Середовище розробки Rider

JetBrains Rider було обрано як основне середовище розробки (IDE) для створення серверного застосунку системи керування ресурсами та задачами проектів з кількох вагомих причин. Перш за все, Rider є потужним інструментом, що поєднує в собі можливості IntelliJ IDEA та ReSharper, що забезпечує високопродуктивні можливості для розробки на мові C#. Це дозволяє значно підвищити продуктивність розробки завдяки інтелектуальним функціям автозавершення коду, автоматичного виправлення помилок і рефакторингу. Крім

того, Rider підтримує всі сучасні технології, що використовуються в проекті, включаючи .NET Core, ASP.NET Core та Angular, що робить його ідеальним вибором для крос-платформної розробки.

JetBrains Rider — це крос-платформне середовище розробки (IDE), яке забезпечує високопродуктивні можливості для розробки на мові C#. Rider використовує ReSharper, що дозволяє проводити аналіз коду на льоту, виконувати автоматичне виправлення помилок і рефакторинг. Rider підтримує .NET Framework, нові крос-платформні версії .NET Core та Mono для розробки серверних додатків, ігор, Unity та Xamarin. Він має вбудований дебагер для допомоги в налагодженні коду, а також вбудовані інструменти для профілювання та перевірки якості коду [18].

2.3 Середовище розробки WebStorm

JetBrains WebStorm було обрано як основне середовище розробки (IDE) для фронтенд-частини нашої системи керування ресурсами та задачами проектів з кількох важливих причин. По-перше, WebStorm спеціалізується на розробці JavaScript і його сучасних діалектів, таких як TypeScript, що є основною мовою для Angular, фреймворку, обраного для розробки клієнтської частини нашої системи. По-друге, WebStorm пропонує багатий набір інструментів для розробки веб-додатків, включаючи інтелектуальне завершення коду, автоматичне виправлення помилок, рефакторинг і інтеграцію з системами контролю версій. Використання WebStorm дозволило швидко і ефективно розробити веб-застосунок.

JetBrains WebStorm — це потужне середовище розробки (IDE), яке спеціалізується на підтримці JavaScript та його сучасних діалектів. WebStorm надає високоякісні інструменти для розробки, включаючи інтелектуальне завершення коду, швидкий перегляд веб-сторінок, автоматичне виправлення помилок, рефакторинг та інтеграцію з системами контролю версій [19].

2.4 Фреймворк для розробки серверної частини ASP.NET Core

Фреймворк ASP.NET Core був обраний для розробки серверної частини нашої системи керування ресурсами та задачами проектів з кількох вагомих причин. По-перше, ASP.NET Core є високопродуктивним, модернізованим і крос-платформним фреймворком, створеним компанією Microsoft. Він підтримує розробку різноманітних веб-додатків, веб-API та мікросервісів, що робить його ідеальним вибором для створення масштабованих і надійних систем.

ASP.NET Core — це високопродуктивний, модернізований, відкритий та крос-платформний фреймворк для розробки веб-додатків, створений Microsoft. Він підтримує розробку веб-додатків, веб-API, мікросервісів, а також розробку на мові C#. ASP.NET Core має модульну архітектуру, що дозволяє вибирати лише ті компоненти, які потрібні для конкретного додатку [6].

ASP.NET Core, у порівнянні з іншими популярними фреймворками, дозволяє обробляти більшу кількість запитів за секунду [7]. Візуально це зображено на рисунку 2.2.

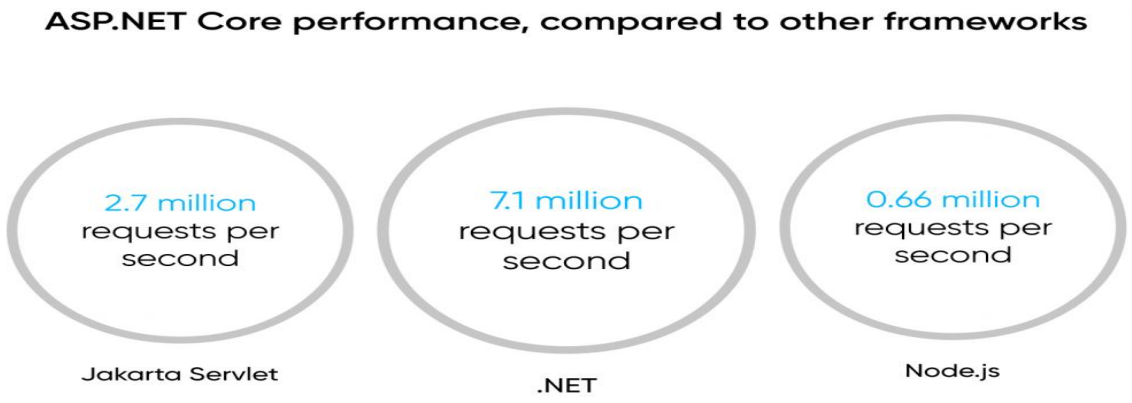


Рисунок 2.2 — Порівняння ASP.NET Core з іншими фреймворками по кількості запитів на секунду

На рисунку можна побачити, що ASP.NET Core пропонує набагато більшу продуктивність, у порівнянні з іншими фреймворками.

2.5 Фреймворк веб-розробки Angular

Angular було обрано для розробки фронтенд-частини нашої системи завдяки його модульності та широкому набору функцій. Він забезпечує високу продуктивність і зручність розробки складних веб-додатків, що робить його ідеальним для створення інтерактивних односторінкових додатків (SPA). Крім того, Angular підтримує TypeScript, що дозволяє використовувати статичну типізацію для зниження кількості помилок та покращення якості коду.

Angular — це потужний фреймворк для розробки веб-додатків, створений командою Google. Він використовує TypeScript як основну мову програмування та надає розробникам високопродуктивні інструменти для створення складних односторінкових додатків (SPA). Angular має вбудовану підтримку двостороннього зв’язування даних, що дозволяє автоматично синхронізувати модель даних та представлення [8].

2.6 Мова програмування C#

Мову програмування C# було обрано для розробки нашої системи завдяки її можливостям, потужним інструментам і глибокій інтеграції з платформою .NET. Це дозволяє створювати надійні, безпечні та високопродуктивні додатки. C# підтримує об'єктно-орієнтоване програмування, що забезпечує гнучкість і модульність коду, а також автоматичне керування пам’яттю, що знижує ризик виникнення помилок, пов'язаних з пам'яттю.

C# — це сучасна об’єктно-орієнтована мова програмування, розроблена компанією Microsoft. Вона є частиною платформи .NET і широко використовується для розробки різноманітних типів програмного забезпечення, включаючи веб-додатки, настільні додатки, мобільні додатки, ігри та багато іншого. C# відрізняється сильною типізацією, автоматичним керуванням пам’яттю, об’єктно-орієнтованими та подієвими моделями програмування [7].

2.7 Мова програмування TypeScript

TypeScript було обрано для розробки клієнтського застосунку системи через його здатності забезпечувати статичну типізацію та покращену структуру коду. Це дозволяє виявляти помилки на етапі компіляції, що підвищує надійність і якість коду. Крім того, TypeScript підтримує сучасні можливості JavaScript, роблячи його ідеальним для створення великих і складних веб-додатків [8].

2.8 Мова гіпертекстової розмітки HTML і мова стилів SCSS

HTML (HyperText Markup Language) — це основна мова розмітки для створення веб-сторінок. HTML використовується для опису структури веб-сторінки за допомогою розмітки, яка складається з ряду коротких кодів, відомих як теги [10].

SCSS (Sassy CSS) — це препроцесор CSS, який додає багато корисних функцій до звичайного CSS, таких як змінні, вкладеність, спадкування та інше. HTML і SCSS працюють разом для створення веб-сторінок, забезпечуючи їх структуру та стилізацію. Завдяки їх взаємодії, можна створювати привабливі та функціональні веб-сторінки.

2.9 Система керування версіями Git та платформа для спільної розробки GitHub

Використання Git та GitHub під час розробки інформаційної системи дозволило ефективно керувати версіями коду, забезпечити прозорість процесу розробки та спростити співпрацю між членами команди. Завдяки цим інструментам було досягнуто високої якості коду та організовано зручний процес розробки.

Система керування версіями Git та платформа для спільної розробки GitHub відіграють ключову роль у сучасній розробці програмного забезпечення, забезпечуючи ефективний контроль за змінами коду та спрощуючи співпрацю між розробниками.

Git — це розподілена система керування версіями, яка дозволяє розробникам відстежувати зміни у вихідному коді проекту та координувати роботу між декількома розробниками. Основними перевагами Git є розподілена архітектура, яка забезпечує кожному розробнику повну копію репозиторію з усією історією змін, що гарантує високу стійкість системи до відмов. Гнучкі робочі процеси Git дозволяють створювати окремі гілки для роботи над новими функціями та об'єднувати їх в основну гілку після завершення роботи. Крім того, операції у Git, такі як створення гілок, коміти і злиття, виконуються дуже швидко, що робить його зручним для великих проектів з великою кількістю змін [13].

GitHub — це веб-платформа для хостингу репозиторіїв Git, яка забезпечує інструменти для спільної розробки та управління проектами. GitHub дозволяє зберігати репозиторії у хмарі, забезпечуючи доступ до них з будь-якого місця, що робить співпрацю між розробниками більш зручною та ефективною. Платформа забезпечує повну підтримку Git, що дозволяє відстежувати зміни, працювати з гілками та об'єднувати зміни в основну гілку проекту. GitHub надає інструменти для спільної роботи, такі як pull requests, code reviews та issues, які дозволяють розробникам обговорювати зміни, перевіряти код та відстежувати помилки. Крім того, GitHub підтримує інтеграцію з багатьма іншими інструментами розробки, такими як CI/CD системи (наприклад, Jenkins, Travis CI), системи управління проектами (наприклад, Jira, Trello) та інші сервіси. Безпека та контроль доступу GitHub забезпечуються гнучкими налаштуваннями доступу до репозиторіїв, що дозволяє контролювати, хто може переглядати та змінювати код [21].

2.10 Використання Docker для розгортання системи

У розробленій системі Docker був використаний для контейнеризації всіх компонентів системи, що забезпечило портативність, швидке розгортання та масштабування додатків, а також ізоляцію та безпеку кожного компонента. Це дозволило створити гнучку та надійну систему, здатну ефективно працювати в різних середовищах.

Docker є провідною платформою для контейнеризації, що дозволяє розробникам упаковувати додатки з усіма необхідними залежностями в стандартизовані одиниці, відомі як контейнери. Використання Docker під час розробки нашої інформаційної системи надало низку важливих переваг [20].

Однією з головних переваг Docker є портативність: контейнери Docker працюють однаково на будь-якій системі, що підтримує Docker, забезпечуючи однакове середовище розробки, тестування та продуктивної експлуатації. Крім того, Docker дозволяє швидко розгортати додатки та легко масштабувати їх, запускаючи кілька контейнерів з одним і тим же додатком. Контейнери працюють в ізольованих середовищах, що підвищує безпеку додатків, оскільки проблеми в одному контейнері не впливають на інші. Також Docker забезпечує ефективне використання системних ресурсів, дозволяючи запускати кілька контейнерів на одній машині без потреби у віртуалізації апаратного забезпечення.

Основними компонентами Docker є Docker Images, Docker Containers, Docker Compose та Docker Swarm. Docker Images — це образи, що містять усе необхідне для запуску додатка, включаючи код, бібліотеки та конфігурації. Docker Containers — це запущені екземпляри образів Docker, які можна запускати, зупиняти, переміщувати або видаляти. Docker Compose є інструментом для визначення та запуску багатоконтейнерних додатків за допомогою YAML-файлів, а Docker Swarm виступає оркестратором для управління кластерами Docker, забезпечуючи високу доступність і масштабованість додатків.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

Головною ціллю при створенні цієї системи було застосування принципів предметно-орієнтованого програмування. Ці принципи дозволяють створювати код, який є зрозумілим та легким для підтримки. У цьому розділі ми детально розглянемо процес програмної реалізації системи, від архітектури та технологій, що були використані під час розробки, до взаємодії користувачів з системою. Важливим аспектом є розуміння того, як різні компоненти системи співпрацюють, щоб забезпечити ефективність та надійність платформи.

3.1 Опис можливих дій користувачів та їх взаємодії з системою

Система керування ресурсами та задачами проектів підтримує всі ролі, що зазвичай передбачені аналогічними системами. Це дозволяє забезпечити гнучкість та адаптивність у використанні системи, надаючи різним категоріям користувачів відповідні права та можливості. Ролі включають:

— Гість

— Власник продукту

— Тестувальник

— Розробник

— Дизайнер

— Тім лід

— DevOps інженер

— Власник компанії

Усі ролі, окрім гостя та власника компанії, мають спільний функціонал, що дозволяє їм виконувати базові операції в системі. Це включає можливість переглядати проекти, в яких вони беруть участь, та переглядати статистику цих

проектів. Користувачі можуть переглядати дошку з картками, маніпулювати ними, створювати коментарі до карток, вносити витрачений час, переглядати та редагувати документи. Цей функціонал забезпечує ефективну співпрацю між членами команди та підтримку робочого процесу.

Власник компанії — це роль, що має найбільше привілей у системі. Вона включає в себе всі вищенаведені можливості, а також додаткові функції, що забезпечують управління проектами та користувачами. Власник компанії може створювати та маніпулювати проектами, запрошувати нових користувачів до проектів, додавати існуючих користувачів на проекти, маніпулювати списками карток. Ці можливості дозволяють власнику компанії ефективно управляти ресурсами та забезпечувати виконання проектних завдань.

Гість має обмежений доступ до системи. Його можливості обмежуються аутентифікацією та авторизацією для доступу до системи. Після успішної авторизації гість може отримати додаткові права доступу залежно від своєї ролі в системі.

На діаграмі прецедентів (рисунок 3.1) зображено ролі акторів: гість, власник продукту, тестувальник, розробник, дизайнер, тім лід, DevOps інженер та власник компанії та дії, що вони можуть виконувати у системі.

Гість у системі має обмежений доступ, його можливості зводяться лише до аутентифікації та авторизації для отримання доступу до системи. Інші ролі, такі як власник продукту, тестувальник, розробник, дизайнер, тім лід, DevOps інженер, мають спільний функціонал, що дозволяє їм переглядати проекти, у яких вони беруть участь, переглядати статистику цих проектів, маніпулювати картками на дошці, створювати коментарі до карток, вносити витрачений час, а також переглядати та редагувати документи.

Власник компанії є користувачем із найбільшими привілеями. Окрім всіх можливостей, доступних іншим ролям, він може створювати та керувати проектами, запрошувати нових користувачів до проектів, додавати існуючих користувачів до проектів, а також маніпулювати списками карток. Така

розширена функціональність дозволяє власнику компанії ефективно управляти ресурсами та забезпечувати виконання проектних завдань.

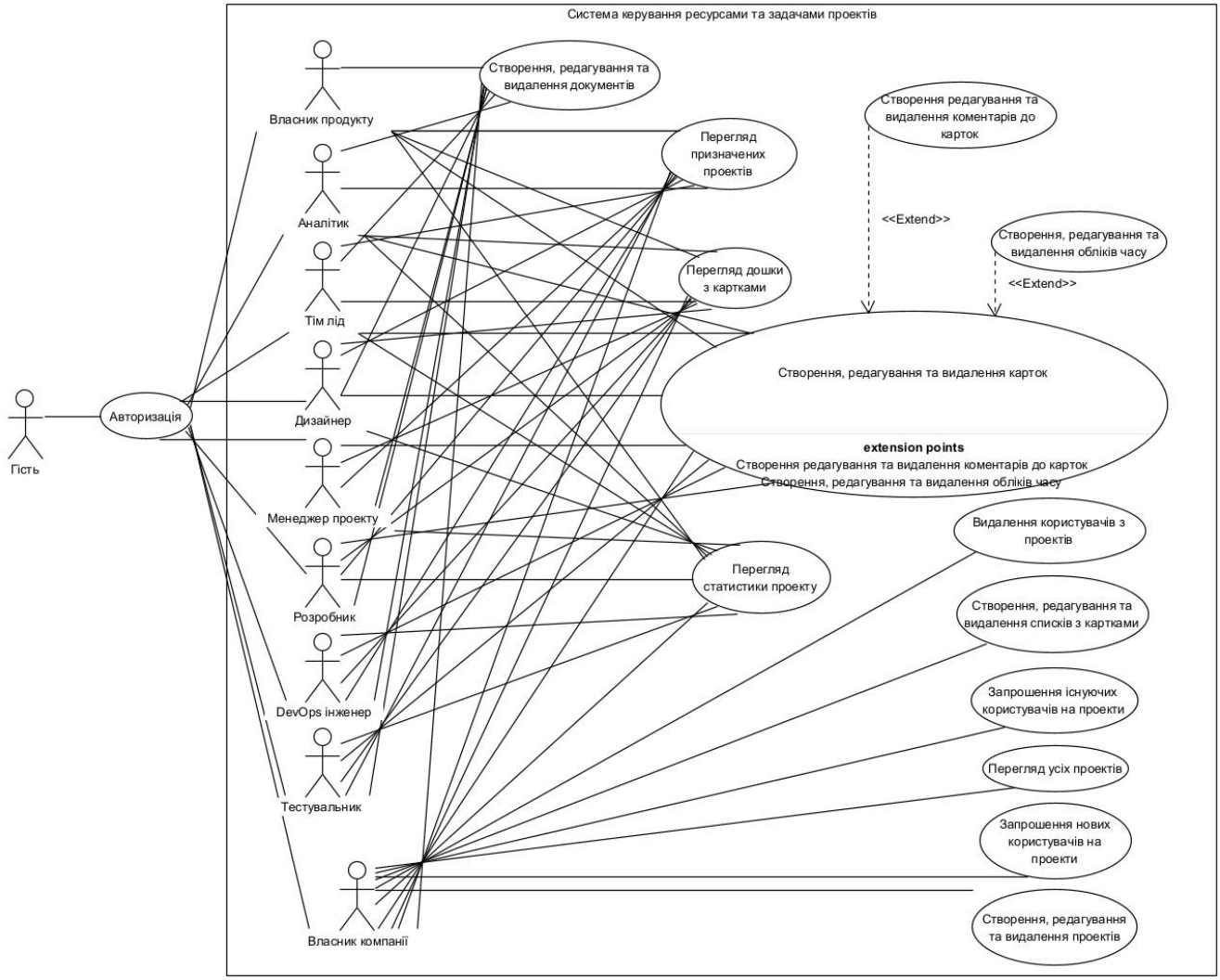


Рисунок 3.1 — Діаграма прецедентів

Ця структура ролей і взаємодій забезпечує ефективну організацію роботи в системі, надаючи кожному користувачу необхідні інструменти для виконання своїх завдань. Діаграма прецедентів наочно демонструє, як різні ролі взаємодіють з системою та одна з одною, а також які дії вони можуть виконувати, що допомагає виявити та реалізувати всі необхідні функції для підтримки робочого процесу.

3.2 Архітектура інформаційно системи

Серверна частина складається з чотирьох проектів: Domain, Application, Infrastructure та Presentation.

Проект Domain містить основні бізнес-моделі та правила. Він відповідає за визначення основних сутностей системи та їх взаємодій. Цей шар повністю незалежний від інших шарів, що дозволяє змінювати бізнес-логіку без впливу на інші частини системи.

Проект Application містить команди, валідації та інтерфейси для виконання бізнес-логіки. Він взаємодіє з Domain для виконання операцій над бізнес-об'єктами та з інфраструктурним шаром для доступу до зовнішніх ресурсів (баз даних, файлових систем тощо). Application є центральною частиною, яка координує роботу всієї системи.

Проект Infrastructure містить реалізації інтерфейсів, визначених у шарі Application, для доступу до зовнішніх ресурсів. Це включає реалізації для роботи з базою даних, веб-сервісом для відправки імейлів та іншими зовнішніми залежностями. Infrastructure забезпечує абстракцію зовнішніх ресурсів, що спрощує їх заміну або оновлення.

Проект Presentation відповідає за взаємодію з користувачами. Він містить контролери, API-ендпоїнти та інші компоненти, які приймають запити від клієнтів, передають їх до Application для обробки та повертають результати. Presentation забезпечує інтерфейс між користувачами та бізнес-логікою системи.

Архітектура серверного застосунку зображена на рисунку 3.2.

Рисунок 3.2 — Діаграма архітектури серверного застосунку

Клієнтська частина системи побудована на основі Angular, що забезпечило модульність та зручність при розробці. Angular дозволяє створювати динамічні односторінкові додатки (SPA), використовуючи компоненти, сервіси та модулі. Таким чином, клієнтський додаток складається з набору компонентів та сервісів, що мають чітко визначене призначення та взаємодіють між собою.

Для того, щоб показати структуру компонентів, була розроблена діаграма компонентів (рисунок 3.3), що зображує компоненти, що стосуються функціоналу для роботи з проектами.

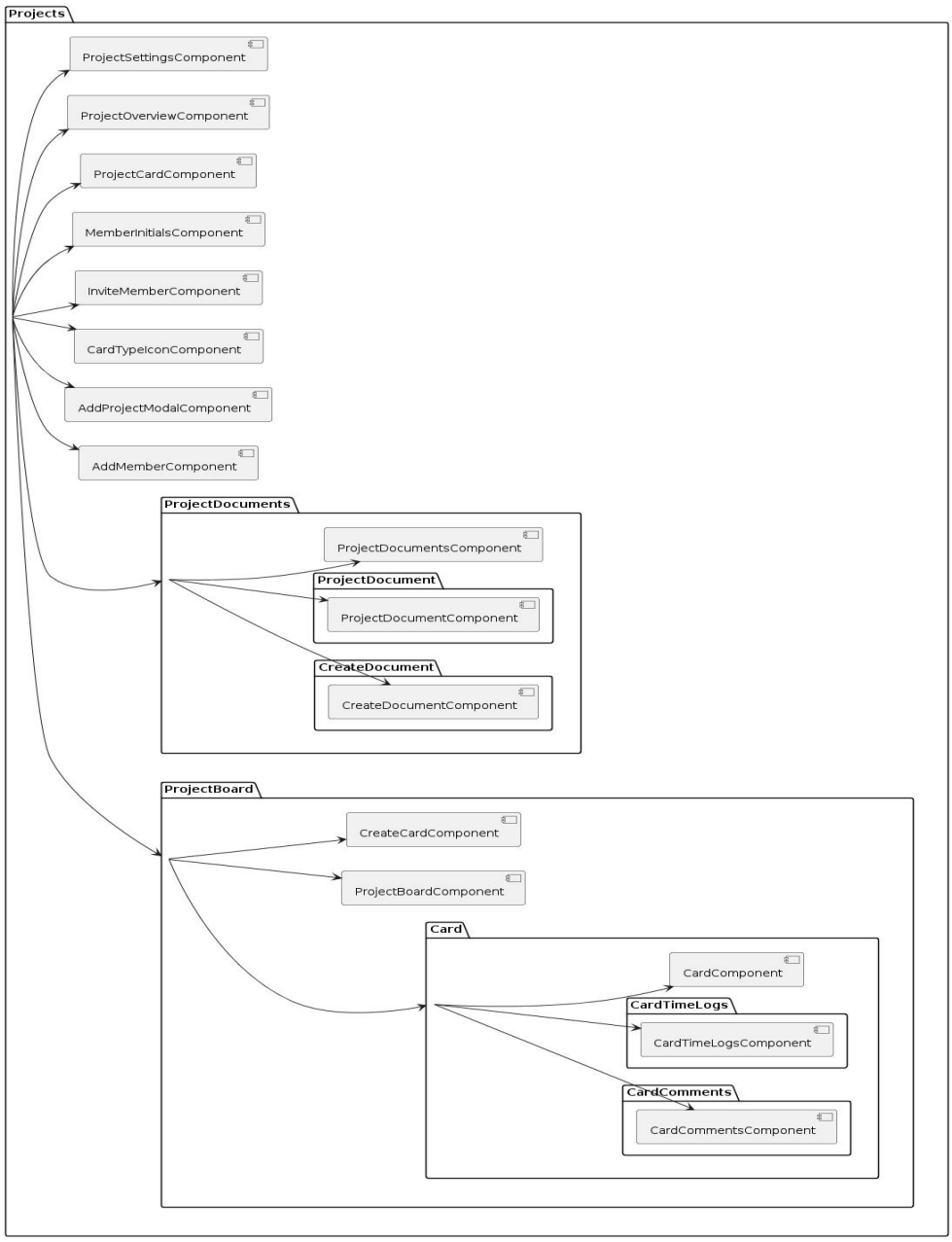


Рисунок 3.3 — Діаграма компонентів

Загальна архітектура інформаційної системи представлена у вигляді діаграми на рисунку 3.4.

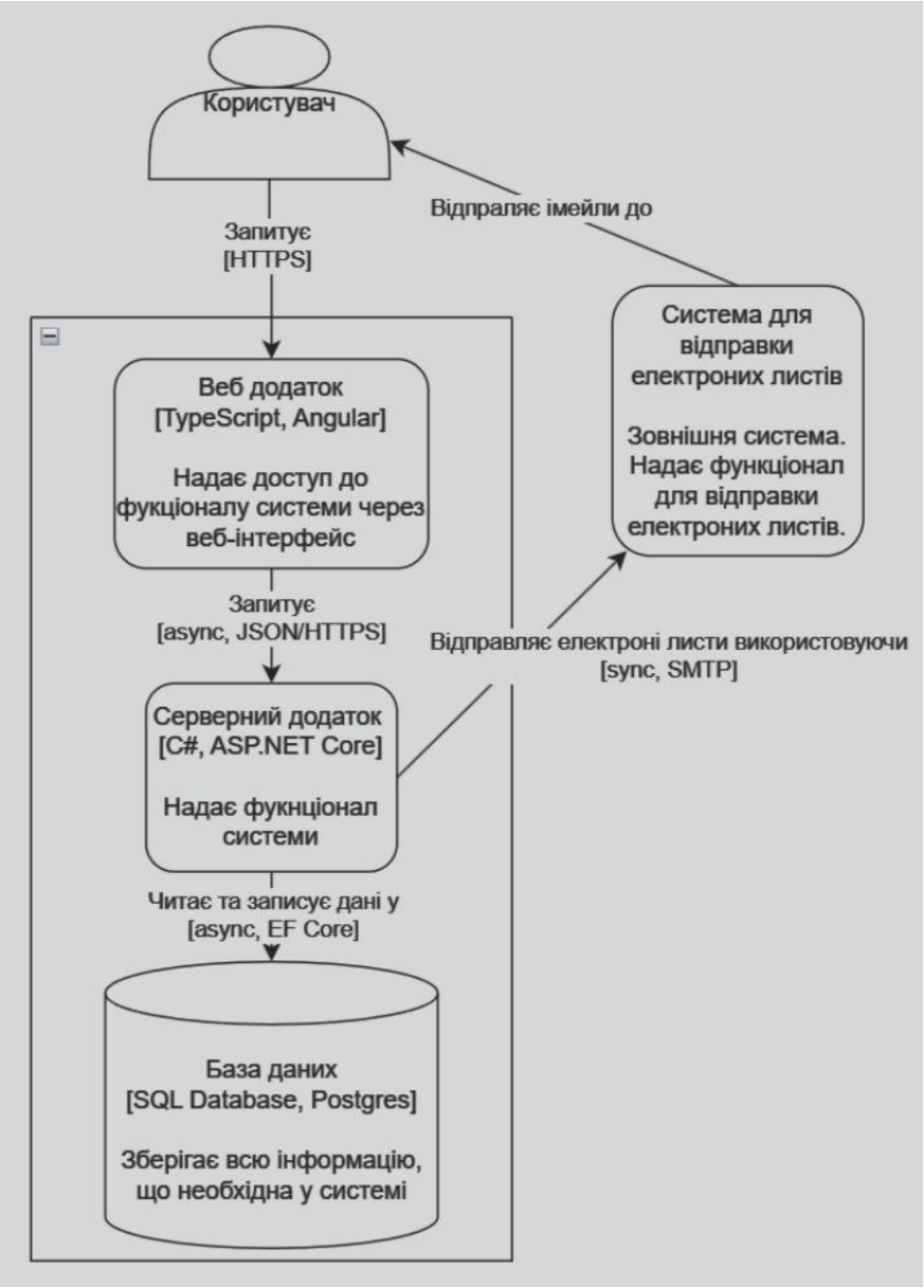


Рисунок 3.4 — Діаграма загальної архітектури інформаційної системи

На діаграмі зображено потік даних та взаємодію між основними компонентами системи. Користувач надсилає запити через веб-додаток, який надає доступ до функціональності системи. Веб-додаток надсилає асинхронні запити до серверного додатку, який обробляє ці запити та взаємодіє з базою даних

для отримання або збереження необхідної інформації. У разі потреби серверний додаток може взаємодіяти із зовнішньою системою для відправки електроних листів користувачам.

3.3 Структура серверного програмного забезпечення

У цьому розділі представлено загальну структуру серверного програмного забезпечення, яка включає в себе основні компоненти та їх взаємозв'язки. Діаграма компонентів основних рівнів серверного застосунку (рисунок 3.5) демонструє логічний поділ системи на окремі рівні, кожен з яких має свою відповідальність та функціональність.

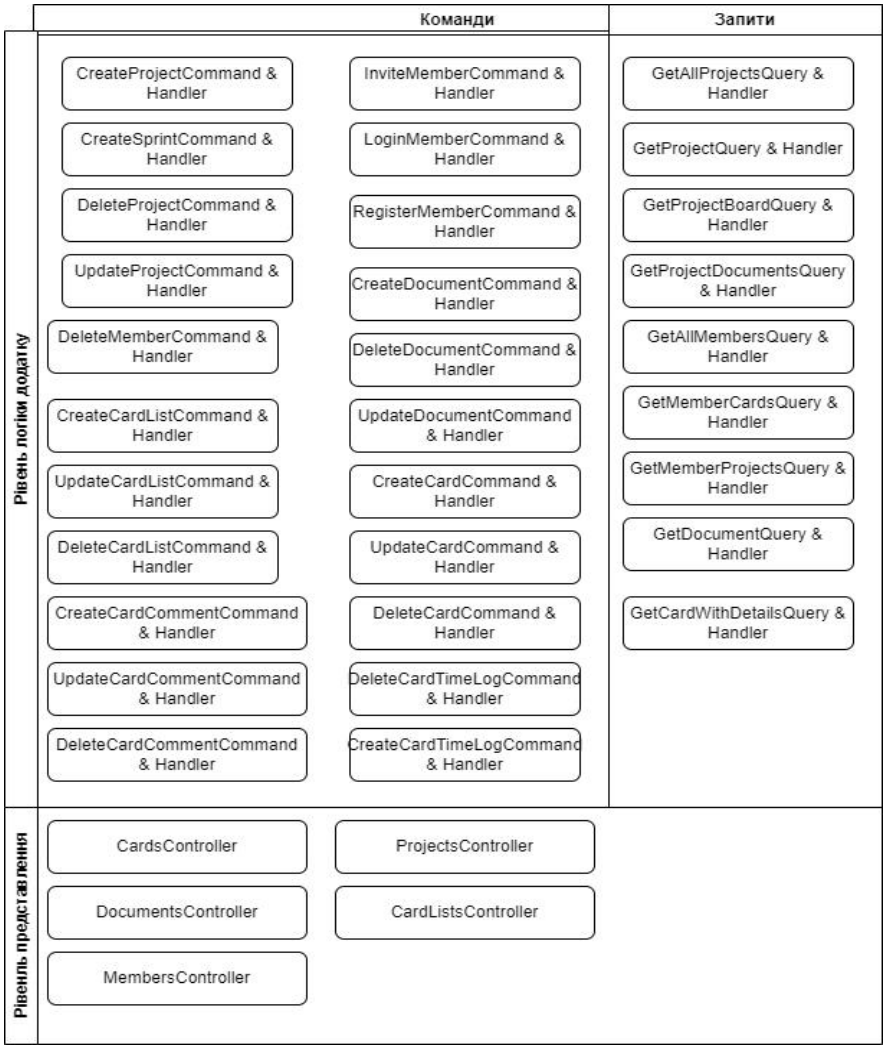


Рисунок 3.5 — Рівні серверного застосунку

Рівень логіки додатку складається з двох основних категорій: команди (Commands) і запити (Queries). Команди відповідають за виконання дій, які змінюють стан системи, тоді як запити призначені для отримання даних без зміни стану.

Цей рівень складається з двох основних категорій: команди (Commands) і запити (Queries). Команди відповідають за виконання дій, які змінюють стан системи, тоді як запити призначені для отримання даних без зміни стану. Команди включають операції створення, оновлення та видалення різних елементів системи, таких як проекти, спринти, картки, коментарі, учасники та документи. Запити, в свою чергу, забезпечують доступ до інформації, включаючи отримання всіх проектів, деталей конкретного проекту, документів проекту, учасників, карток учасників та деталей картки.

На рівні представлення знаходяться контролери, які обробляють HTTP-запити і керують взаємодією з рівнем логіки додатку. Контролери включають обробку операцій з картками, документами, учасниками, проектами та списками карток. Кожен контролер відповідає за певну частину системи, забезпечуючи інтерфейс для взаємодії з відповідними командами та запитами на рівні логіки додатку.

Для розкриття деталей реалізації серверного додатку побудовано діаграму класів (рисунок 3.6), яка демонструє основні класи, їх атрибути та взаємозв'язки між ними. Діаграма класів відображає структуру даних і логіку взаємодії між об'єктами системи, що допомагає краще зрозуміти внутрішню організацію коду та його архітектуру.

На діаграмі класів зображено основні сутності, агрегати, об'єкти значень (value objects) та перерахування (enums) системи керування ресурсами та задачами проектів, а також їх взаємозв'язки.

Агрегати – це основні об'єкти, які групують логічно пов'язані елементи разом для спрощення управління даними та підтримки цілісності. В діаграмі представлені два агрегати: Project та Card. Project містить ідентифікатор проекту, назву, опис та ідентифікатор поточного спринту. Проект може мати кілька

спринтів та документів. Card містить ідентифікатор картки, назву, опис, позицію, ідентифікатори батьківської картки, списку карток, спринту, проекту, учасника, рівень пріоритету, тип картки, дату початку та завершення, очікувану тривалість виконання. Картка може мати коментарі та журнали часу. CardList містить ідентифікатор, ім'я, номер позиції та ідентифікатор проекту. Картка прив'язана до певного списку карток.

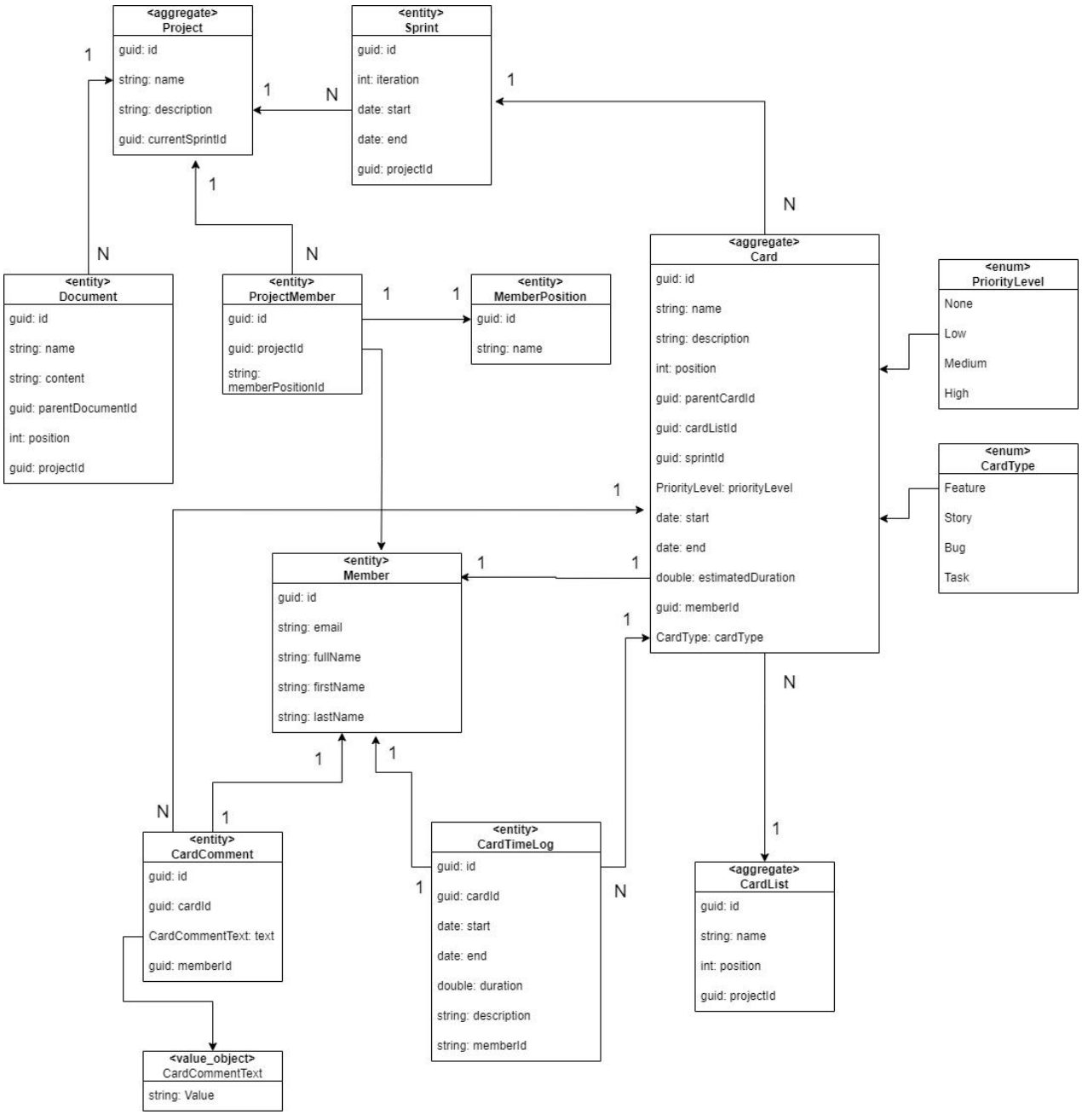


Рисунок 3.6 — Діаграма класів

Сутності – це об'єкти, які мають унікальний ідентифікатор. В діаграмі представлені сутності Sprint, Document, ProjectMember, MemberPosition, Member, CardComment та CardTimeLog. Document представляє документи, що належать до проекту, з такими атрибутами, як ідентифікатор, ім'я, зміст, ідентифікатор батьківського документа та ідентифікатор проекту. ProjectMember відображає інформацію про учасників проекту, включаючи їх ідентифікатор, ідентифікатор проекту та ідентифікатор позиції учасника. MemberPosition зберігає дані про позиції учасників, включаючи ідентифікатор та ім'я позиції. Member містить інформацію про учасників системи, зокрема їх ідентифікатор, адресу електронної пошти, повне ім'я, ім'я та прізвище. CardComment представляє коментарі до карток та включає ідентифікатор, ідентифікатор картки, текст коментаря та ідентифікатор учасника, який залишив коментар. CardTimeLog містить інформацію про витрачений час на виконання завдань і включає ідентифікатор, ідентифікатор картки, дату початку та завершення, проміжок часу (у годинах), опис та ідентифікатор учасника.

Об'єкт значень (value object) в діаграмі представлений класом CardCommentText, який містить єдине поле Value для зберігання тексту коментаря.

Перерахування (enums) включають PriorityLevel та CardType. PriorityLevel визначає рівні пріоритету (None, Low, Medium, High), а CardType – типи карток (Feature, Story, Bug, Task).

Використання агрегатів (aggregates), сутностей (entities) та об’єктів значень (value objects) показує, що при розробці були дотримані шаблони предметно-орієнтованого програмування.

3.4 Опис програмної реалізації серверного застосунку

Серверна частина була розроблена використовуючи останню, на момент розробки, версію платформи .NET з використанням технології ASP.NET Core та

мови програмування C#. Завдяки використанню технології ASP.NET Core, розроблена система отримала високу продуктивність, що дозволяє обробляти більше запитів від користувачів з меншими затримками, забезпечуючи кращу масштабованість та надійність.

В якості архітектури серверної частини була обрана Clean Architecture [14]. Дана архітектура дозволила реалізувати принципи предметно-орієнтовано програмування, за рахунок наявності доменних об’єктів, що чітко описують сутності, що наявні додатку. Також дана архітектура дозволяє якісно описати логіку додатку, що не містить прямих залежностей на зовнішні системи (база даних, SMTP сервер).

Для написання бізнес-логіки додатку була використана бібліотека MediatR [15], яка дозволила реалізувати патерн CQRS (Command and Query Responsibility Segregation) [16]. Цей патерн чітко розмежовує операції, що змінюють стан системи (команди), від операцій, що виконують читання даних (запити). Використання CQRS дозволяє оптимізувати продуктивність та масштабованість системи, а також спрощує підтримку та розвиток бізнес-логіки. Бібліотека MediatR також дозволила втілити централізоване логування подій, обробку помилок, вимірювання метрик продуктивності та відправку подій до клієнтського додатку.

Для валідації вхідних даних була використана бібліотека FluentValidation [17], яка дозволяє прозоро описати вимоги до даних, що потрапляють до системи. FluentValidation забезпечує гнучкість та зручність при створенні правил валідації, що спрощує підтримку та розширення логіки валідації.

Для збереження даних користувачів було обрано реляційну базу даних PostgreSQL [19]. PostgreSQL відома своєю надійністю, високою продуктивністю та підтримкою складних запитів. Вона також забезпечує транзакційну цілісність даних і підтримує розширені функції, такі як індекси, тригери та збережені процедури. PostgreSQL забезпечує можливість масштабування бази даних та підтримку великих обсягів даних, що робить її ідеальним вибором для розробки систем з високими вимогами до продуктивності та надійності.

Для простоти роботи з базою даних була використана бібліотека EF Core, що являє собою ORM (Object-Relational Mapping) технологію, яка дозволяє працювати з таблицями на рівні об’єктів у коді, абстрагуючись від конкретної бази даних та позбавляючи від необхідності писати власноруч SQL запити. EF Core забезпечує автоматичне створення та оновлення схем бази даних, що спрощує розробку та підтримку додатку.

3.5 Опис кінцевих точок серверного застосунку

Серверна частина використовує архітектуру REST (Representational State Transfer), що дозволяє створювати розподілені системи з чіткою та логічною структурою API. Кожен ресурс системи (користувачі, проекти, картки, документи) представлений у вигляді унікального URL, а основні операції (створення, читання, оновлення, видалення) реалізовані за допомогою стандартних HTTP-методів (POST, GET, PUT, DELETE) [18].

Для розкриття того, як можна взаємодіяти з API, опишемо кінцеві точки, що містяться у контролері проєктів. На таблиці 2 представлено HTTP методи, маршрути, описи кінцевих точок та HTTP статус коди, що реалізовані у контролері проектів.

Таблиця 2 — Опис кінцевих точок для ProjectsController

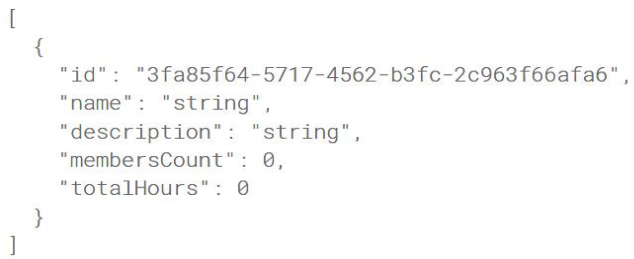
Метод HTTP	**URL**	**Опис**	**HTTP коди статусів**
GET	api/projects	Повертає список всіх проектів	200, 401, 500
POST	api/projects	Створює новий проект	201, 400, 401, 500
PUT	api/projects	Оновлює існуючий проект	204, 400, 401, 404, 500
GET	api/projects/{projectId}	Повертає проект за ідентифікатором	200, 401, 500

Кінець таблиці 2

DELETE	api/projects/{projectId}	Видаляє проект за ідентифікатором	204, 401, 404, 500
GET	api/projects/{projectId}/board	Повертає дошку проекту	200, 401, 404, 500
GET	api/projects/{projectId}/documents	Повертає документи проекту	200, 401, 404, 500
POST	api/projects/{projectId}/members/{memberId}	Додає учасника до проекту	204, 400, 401, 404, 500

Таблиця містить інформацію про операції, які можна виконувати над проектами, такими як отримання списку всіх проектів, створення нового проекту, оновлення існуючого проекту, видалення проекту, отримання детальної інформації про проект, доступ до дошки проекту, а також робота з учасниками та документами проекту.

Кінцева точка GET api/projects повертає список всіх проектів. Відповідь містить масив об'єктів, кожен з яких включає такі поля, як ідентифікатор проекту, назву, опис та інші метадані. Приклад JSON, що повертається показано на рисунку 3.7.



```
[
  {
    "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
    "name": "string",
    "description": "string",
    "membersCount": 0,
    "totalHours": 0
  }
]
```


Рисунок 3.7 — Приклад JSON результату для кінцевої точки GET api/projects

Кінцева точка POST api/projects призначений для створення нового проекту. Запит приймає об'єкт проекту у форматі JSON, що включає поля, такі як назва проекту та опис. Приклад JSON, що вона приймає, показано на рисунку 3.8.

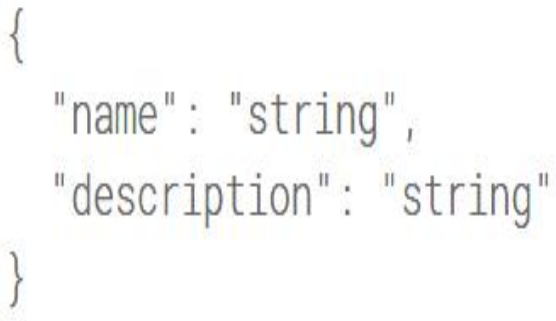


Рисунок 3.8 — Приклад JSON запиту для кінцевої точки POST api/projects

Кінцева точка PUT api/projects використовується для оновлення існуючого проекту. Запит приймає об'єкт проекту з оновленими даними у форматі JSON. Приклад JSON, що вона приймає, показано на рисунку 3.9.

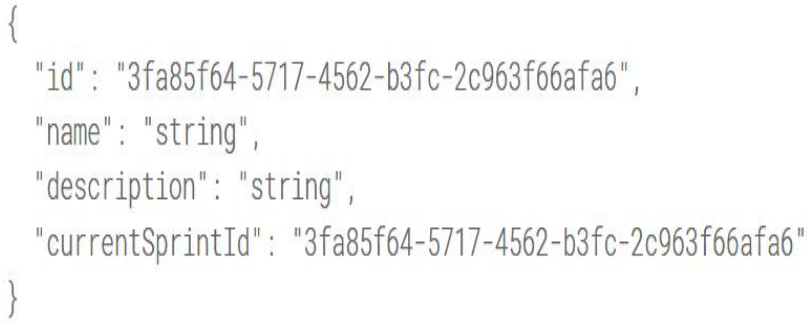


Рисунок 3.9 — Приклад JSON запиту для кінцевої точки PUT api/projects

Кінцева точка GET api/projects/{projectId} повертає проект за його ідентифікатором. Відповідь містить об'єкт проекту з усіма його полями. Приклад JSON, що повертається показано на рисунку 3.10.

Кінцева точка DELETE api/projects/{projectId} видаляє проект за його ідентифікатором.

Ендпоїнт POST api/projects/{projectId}/members/{memberId} додає учасника до проекту. Запит приймає ідентифікатори проекту, учасника та позиції учасники на проєкті.

Кінцева точка DELETE api/projects/{projectId}/members/{memberId} видаляє учасника з проекту. Запит приймає ідентифікатори проекту та учасника.

```
{
  "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "name": "string",
  "description": "string",
  "currentSprintId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "currentSprintStatistics": {
    "name": "string",
    "involvedMembersCount": 0,
    "cardsCount": 0,
    "totalSpentDuration": 0,
    "totalEstimatedDuration": 0,
    "cardListsStatistics": [
      {
        "name": "string",
        "totalEstimatedDuration": 0,
        "totalSpentDuration": 0,
        "cardsCount": 0
      }
    ]
  },
  "members": [
    {
      "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
      "email": "string",
      "userName": "string",
      "firstName": "string",
      "lastName": "string",
      "position": "string",
      "joinedAt": "2024-06-11T00:57:10.493Z"
    }
  ],
  "sprints": [
    {
      "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
      "name": "string"
    }
  ]
}
```


Рисунок 3.10 — Приклад JSON результату для кінцевої точки GET api/projects/{projectId}

Кінцева точка GET api/projects/{projectId}/documents повертає список документів, пов'язаних з проектом. Відповідь містить JSON масив об'єктів документів з відповідними полями. Приклад JSON, що повертається показано на рисунку 3.11.

```
[
  {
    "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
    "name": "string",
    "position": 0,
    "content": "string",
    "childDocuments": [
      "string"
    ],
    "parentDocument": "string"
  }
]
```


Рисунок 3.11 — Приклад JSON результату для кінцевої точки GET api/projects/{projectId}/documents

Кінцева точка GET api/projects/{projectId}/board повертає дошку проекту, що містить списки із завданнями, список ітерацій проєкту, список учасників та ідентифікатор поточної ітерації. Приклад JSON, що повертається показано на рисунку 3.12.


```
{
  "sprints": [
    {
      "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
      "name": "string"
    }
  ],
  "cardLists": [
    {
      "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
      "name": "string",
      "position": 0,
      "cards": [
        {
          "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
          "name": "string",
          "position": 0,
          "priorityLevel": "None",
          "startDate": "2024-06-11T01:01:17.294Z",
          "dueDate": "2024-06-11T01:01:17.294Z",
          "memberId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
          "subCards": [
            "string"
          ],
          "cardType": "Feature",
          "cardListId": "3fa85f64-5717-4562-b3fc-2c963f66afa6"
        }
      ],
      "projectId": "3fa85f64-5717-4562-b3fc-2c963f66afa6"
    }
  ],
  "members": [
    {
      "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
      "email": "string",
      "userName": "string",
      "fullName": "string",
      "projects": [
        {
          "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
          "name": "string",
          "description": "string",
          "memberPosition": "string"
        }
      ]
    }
  ],
  "currentSprintId": "3fa85f64-5717-4562-b3fc-2c963f66afa6"
}
```


Рисунок 3.12 — Приклад JSON результату для кінцевої точки GET api/projects/{projectId}/board

3.6 Опис програмної реалізації клієнтського застосунку

Клієнтський застосунок був розроблений з використанням фреймворку Angular та мови програмування TypeScript. Використання Angular дозволило створити компоненти клієнтського додатку, що виконують чітко відведені функції у системі, що забезпечує простоту підтримки та легкість розуміння. Angular забезпечує модульну структуру додатку, що дозволяє легко додавати нові функції та підтримувати існуючі. Компоненти Angular включають шаблони для відображення даних, сервіси для обробки бізнес-логіки та взаємодії з сервером.

У таблиці 3 представлені компоненти та їх відповідні маршрути.

Таблиця 3 — Компоненти та їх маршрути

Компонент	URL
RegistrationComponent	/register
MainLayoutComponent	/
OverviewComponent	/
ProjectOverviewComponent	/project-overview/:project_id
ProjectDocumentsComponent	/project-documents/:project_id
ProjectDocumentComponent	/project-documents/:project_id/:document_id
ProjectBoardComponent	/project-board/:project_id
ProjectSettingsComponent	/project-settings/:project_id

Маршрутизація у Angular відбудується без оновлення сторінки, на якій знаходиться користувач. Це дозволяє забезпечити позитивний досвід користування розробленим застосунком.

3.7 Схема зберігання даних в системі

Схема зберігання даних в системі є важливим аспектом проектування інформаційних систем, оскільки від її ефективності залежить швидкість доступу

до даних, їхня цілісність та безпека. У даному розділі розглянемо логічну та фізичну схеми зберігання даних, що використовуються в системі керування ресурсами та задачами проектів.

Логічна схема сутностей, зображена на діаграмі (рисунок 3.13), відображає основні сутності системи керування ресурсами та задачами проектів, їх атрибути та взаємозв'язки між ними. Ця діаграма надає структуроване уявлення про дані, які використовуються у системі, та їх взаємодію.

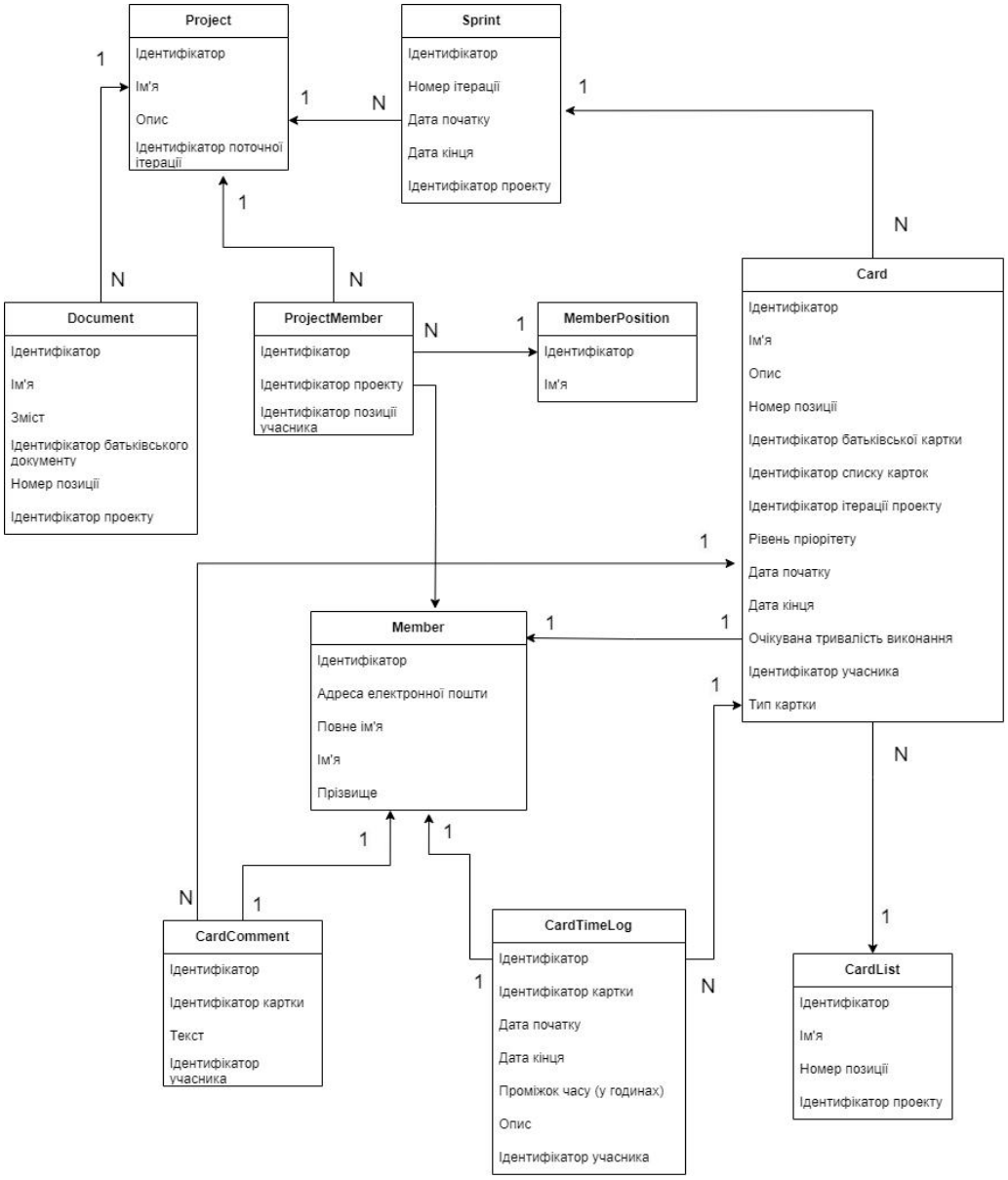


Рисунок 3.13 — Логічна схема сутностей та зв’язків між ними

Сутність Project відповідає за зберігання інформації про проект, включаючи його унікальний ідентифікатор, ім'я, опис та ідентифікатор поточної ітерації. Проект може містити кілька спринтів, кожен з яких також має свій ідентифікатор, номер ітерації, дату початку та завершення, а також ідентифікатор проекту.

Сутність Sprint описує ітерації проекту та пов'язана з проектом через ідентифікатор проекту. Сутність Document представляє документи, що належать до проекту, з такими атрибутами, як ідентифікатор, ім'я, зміст, ідентифікатор батьківського документа та ідентифікатор проекту.

Сутність ProjectMember відображає інформацію про учасників проекту, включаючи їх ідентифікатор, ідентифікатор проекту та ідентифікатор позиції учасника. Сутність MemberPosition зберігає дані про позиції учасників, включаючи ідентифікатор та ім'я позиції.

Сутність Member містить інформацію про учасників системи, зокрема їх ідентифікатор, адресу електронної пошти, повне ім'я, ім'я та прізвище. Учасники можуть створювати та коментувати картки, які представлені сутністю Card. Картка містить ідентифікатор, ім'я, опис, номер позиції, ідентифікатор батьківської картки, ідентифікатор списку карток, ідентифікатор ітерації проекту, рівень пріоритету, дату початку та завершення, очікувану тривалість виконання, ідентифікатор учасника та тип картки.

Сутність CardList відповідає за списки карток у проекті і містить такі атрибути, як ідентифікатор, ім'я та номер позиції, а також ідентифікатор проекту.

Сутність CardComment представляє коментарі до карток та включає ідентифікатор, ідентифікатор картки, текст коментаря та ідентифікатор учасника, який залишив коментар.

Сутність CardTimeLog містить інформацію про витрачений час на виконання завдань і включає ідентифікатор, ідентифікатор картки, дату початку та завершення, проміжок часу (у годинах), опис та ідентифікатор учасника.

Ця структура забезпечує чітке розуміння взаємозв'язків між сутностями у системі та дозволяє ефективно управляти проектами, ресурсами та задачами, забезпечуючи при цьому високу організацію та гнучкість даних.

Фізична схема зберігання даних представляє структуру бази даних на рівні таблиць, їх атрибутів та взаємозв'язків між ними. На другій діаграмі (рисунок 3.6) показано фізичну структуру бази даних, включаючи таблиці, їхні поля та зв'язки.

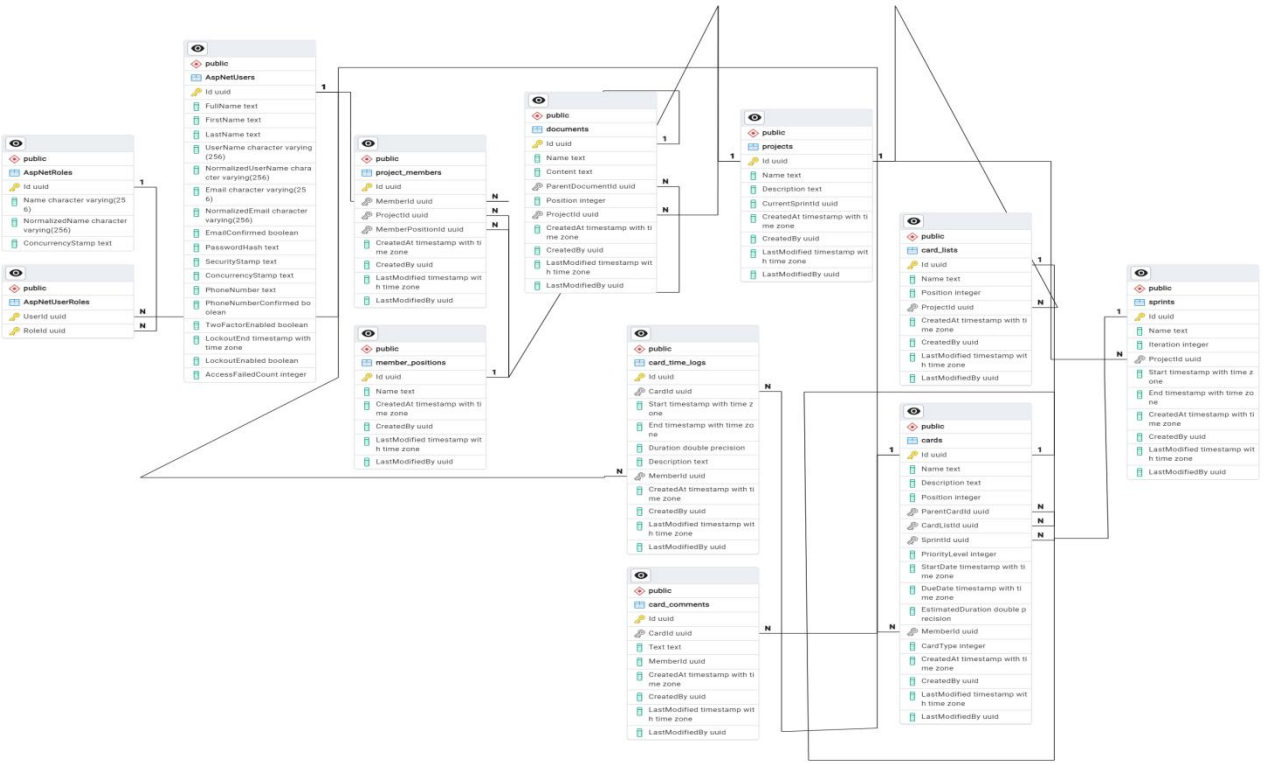


Рисунок 3.14 — Фізична схема сутностей та зв’язків між ними

Таблиця projects зберігає інформацію про проекти, включаючи id, name, description, currentSprintId. Таблиця sprints містить дані про спринти проектів, такі як id, iteration, start, end, projectId. Таблиця project_members зберігає інформацію про учасників проектів, включаючи memberId, projectId, memberPositionId. Таблиця member_positions містить дані про позиції учасників проектів, такі як id, name. Таблиця members зберігає інформацію про учасників проектів, включаючи id, email, fullName, firstName, lastName. Таблиця cards містить дані про картки задач, включаючи id, name, description, position, parentCardId, cardListId, sprintId, priorityLevel, startDate, endDate, estimatedDuration, memberId, cardType. Таблиця card_lists зберігає інформацію про списки карток, включаючи id, name, position, projectId. Таблиця documents містить дані про документи проектів, такі як id, name,

content, parentDocumentId, position, projectId. Таблиця card_comments зберігає коментарі до карток задач, включаючи id, cardId, text, memberId. Таблиця card_time_logs містить дані про витрачений час на виконання задач, включаючи id, cardId, start, end, duration, description, memberId.

Фізична схема відображає логічну модель даних у вигляді таблиць бази даних, які забезпечують ефективне зберігання та доступ до інформації. Кожна таблиця відповідає певному класу логічної схеми, а поля таблиць відповідають атрибутам класів.

Разом ці схеми забезпечують цілісність та узгодженість даних у системі, дозволяючи ефективно керувати ресурсами та задачами проектів. Логічна схема допомагає зрозуміти структуру даних на рівні об'єктів програмування, тоді як фізична схема показує, як ці дані зберігаються у базі даних.

3.8 Опис процесу створення картки

Для розкриття деталей реалізації серверного додатку розглянемо як приклад процес створення картки. Для цього побудовано діаграми активності та послідовності.

Діаграма активності (рисунок 3.15) показує основні кроки, які виконує користувач при створенні нової картки. Спочатку користувач вводить дані у форму створення картки, натискає кнопку для збереження, після чого відбувається валідація даних. Якщо дані не є валідними, виводиться повідомлення з помилкою. У разі успішної валідації, запит відправляється на сервер, де дані перевіряються відповідно до бізнес-правил. Якщо дані не відповідають бізнес-правилам, сервер повертає помилку, інакше картка створюється, зберігається у базі даних та надсилається подія про створення картки. Завершення процесу включає оновлення списку карток та закриття форми створення картки.

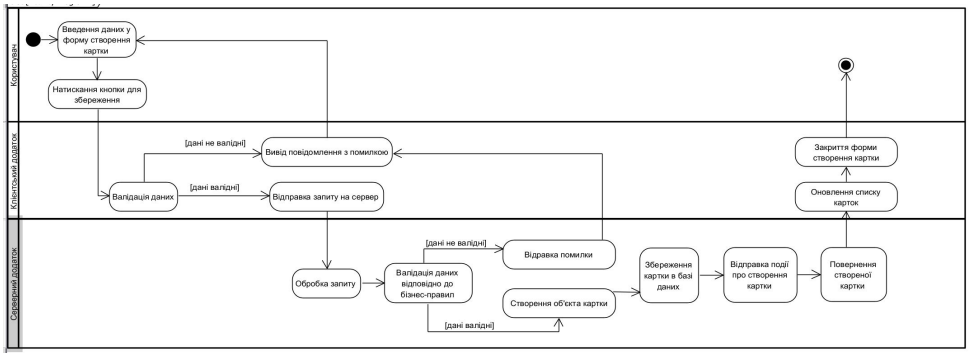


Рисунок 3.15 — Діаграма активності для процесу створення картки

Діаграма послідовності (рисунок 3.16) детально відображає взаємодію між різними компонентами системи під час створення нової картки. Процес починається з запиту користувача на створення картки, після чого здійснюється перевірка валідності форми на клієнтській частині. У разі валідних даних запит на створення картки надсилається на сервер, де відбувається додаткова валідація даних за допомогою CreateCardCommandValidator. Після успішної валідації та виконання команди створення картки, новий об’єкт картки створюється в рамках двох агрегатів: Project та CardList. Далі картка зберігається у базі даних та надсилається подія про створення картки. Нарешті, клієнтська частина отримує створену картку і оновлює інтерфейс користувача.

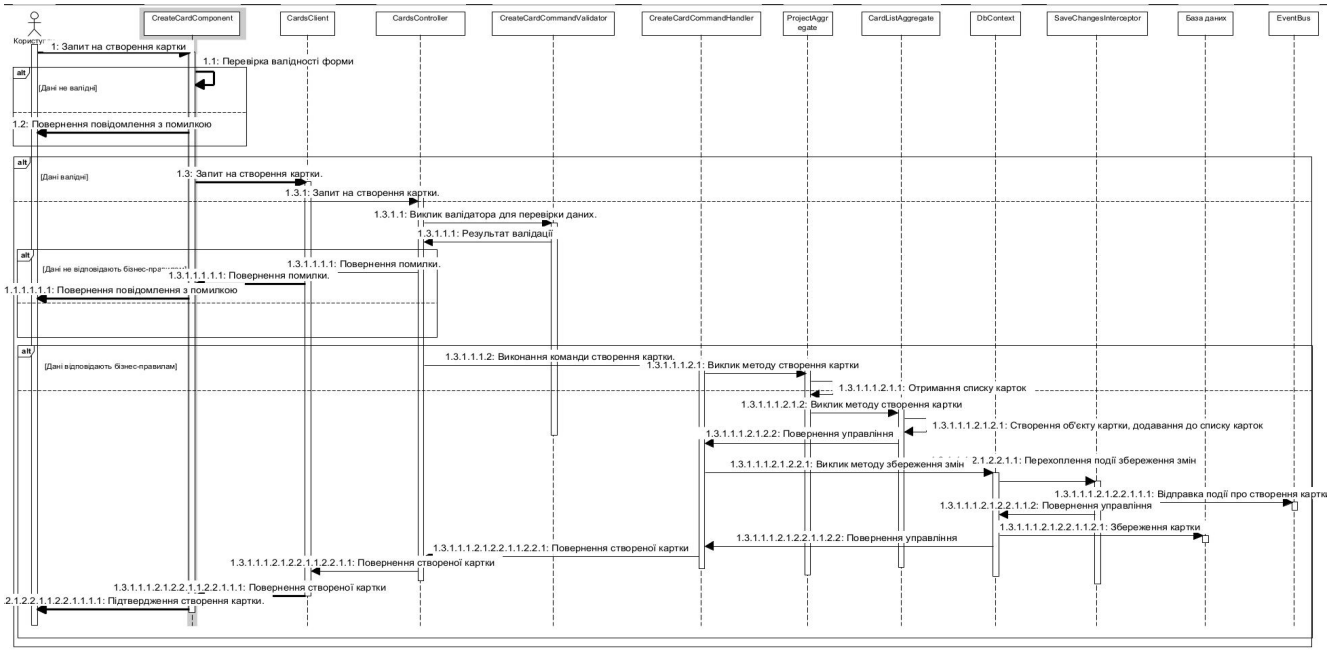


Рисунок 3.16 — Діаграма послідовності для процесу створення картки

Ця діаграма повністю в деталях показує весь процес збереження картки, включаючи перевірку валідності, валідацію на сервері, виконання команди, роботу з агрегатами, збереження в базі даних та надсилання доменної події про створення картки. Це дозволяє показати, що розроблена система слідує шаблонам предметно-орієнтованого програмування, використовуючи агрегати та доменні події. Процес взаємодії компонентів чітко відповідає концепціям DDD (Domain-Driven Design)

Розглянутий процес створення картки є лише одним із багатьох функціональних можливостей системи. Система забезпечує гнучкість та масштабованість, що дозволяє легко адаптувати її до потреб різних проектів і команд.

4 РОБОТА КОРИСТУВАЧА З ІНФОРМАЦІЙНОЮ СИСТЕМОЮ

У цьому розділі ми детально розглянемо системні вимоги та процес встановлення програмної системи, а також опишемо сценарії використання для користувачів. Розуміння системних вимог та правильне встановлення є важливими етапами перед початком роботи з програмним забезпеченням. Додатково, описані сценарії допоможуть користувачам отримати чітке уявлення про функціонал та можливості системи. Для забезпечення стабільної роботи програмної системи необхідно дотримуватися рекомендацій щодо її використання та встановлення. Важливо також враховувати потужність комп'ютера, на якому запускається веб-додаток системи, та відповідати вимогам щодо версії операційної системи.

4.1 Системні вимоги та інсталяція інформаційної системи

Для встановлення платформи на пристрої необхідно дотримуватись наступних технічних умов:

— Рекомендується використовувати багатоядерний процесор для забезпечення оптимальної продуктивності системи.

— Мінімальні вимоги включають 4 ГБ оперативної пам'яті, проте для комфортної роботи з програмним забезпеченням рекомендується 8 ГБ або більше.

— Перед встановленням необхідно переконатися, що на пристрої є достатньо місця для інсталяції програмного забезпечення та зберігання даних.

Розроблена програма підтримує роботу у середовищі Docker Desktop, що забезпечує простоту її встановлення та використання.

Для інсталяції системи необхідно виконати наступні кроки:

1. Клонувати код проекту з платформи GitHub за наступним посиланням: https://github.com/kolya2po/TeamPlan.
2. У кореневій папці проекту виконати команду “docker-compose up”. Ця команда запустить усі необхідні контейнери, включаючи:
   — Контейнер бази даних
   — Контейнер клієнтського додатку
   — Контейнер серверного додатку
   — Контейнер, що виконує роль SMTP сервера

Дотримання цих інструкцій забезпечить правильну інсталяцію та запуск програмної системи на вашому пристрої.

4.2 Сценарій роботи користувача з інформаційною системою

Інтерфейс є ключовим елементом використання будь-якої інформаційної системи, оскільки він створює зручний та зрозумілий механізм взаємодії з користувачем. Якщо інтерфейс розроблено правильно, користувачі можуть легко знаходити потрібну інформацію, виконувати необхідні дії та робити це без непотрібних складнощів. Інтерфейс системи спрямований на забезпечення простоти та зручності використання. Він має інтуїтивно зрозумілу структуру, логічно розташовані елементи управління та чітко відображає всю необхідну інформацію. Дизайн інтерфейсу відповідає сучасним трендам та вимогам до ергономіки, що дозволяє користувачам комфортно працювати з системою протягом тривалого часу.

Перед початком роботи з системою, користувачу необхідно пройти процес аутентифікації, використовуючи свою електронну пошту та пароль (рисунок 4.1). Цей процес забезпечує безпеку даних користувачів та дозволяє системі ідентифікувати користувача, надаючи доступ до персоналізованого контенту.

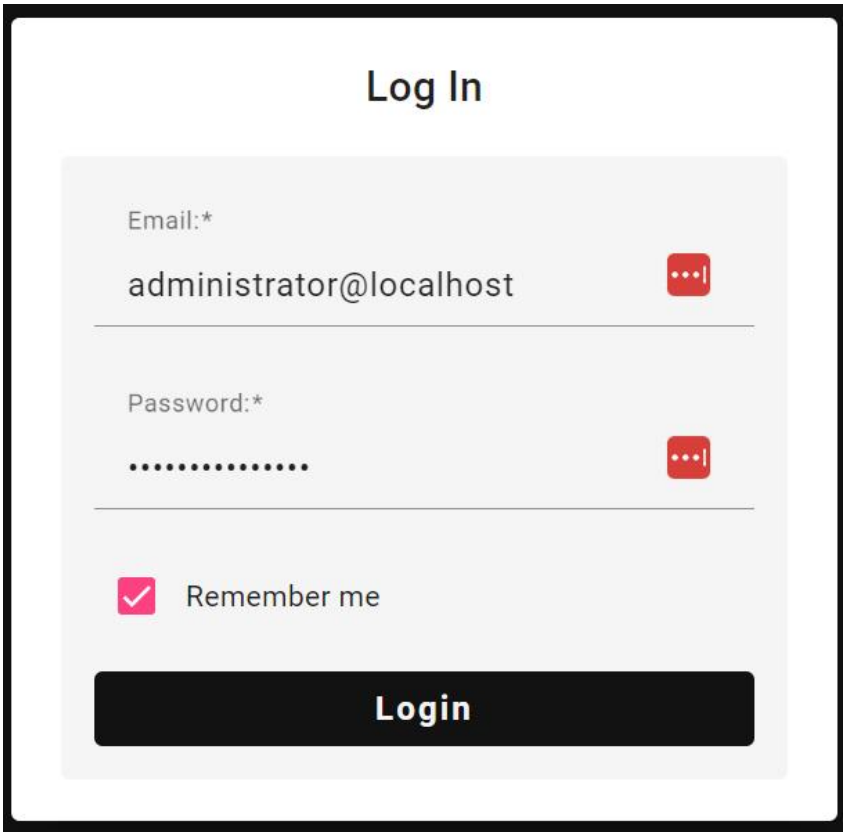


Рисунок 4.1 — Авторизація користувача в системі

Після успішної авторизації користувач бачить список проектів, у яких він бере участь (рисунок 4.2). Кожен проект представлений у вигляді картки, що містить ім’я проекту, його опис та роль користувача на цьому проекті. Це дозволяє користувачам швидко знаходити потрібні проекти та переходити до їх перегляду.

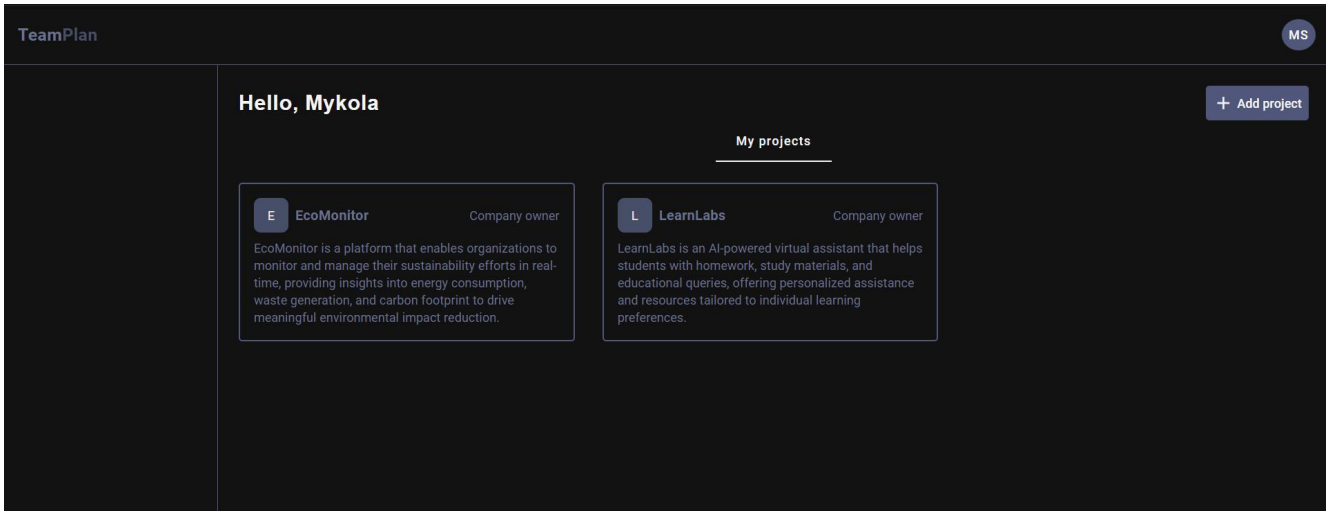


Рисунок 4.2 — Перегляд проектів користувача

Після вибору потрібного проекту користувач потрапляє на сторінку з загальним оглядом проекту (рисунок 4.3). На цій сторінці він може переглядати опис проекту, поточну статистику та список учасників проекту. Ця інформація допомагає користувачам отримати загальне уявлення про стан проекту та його учасників.

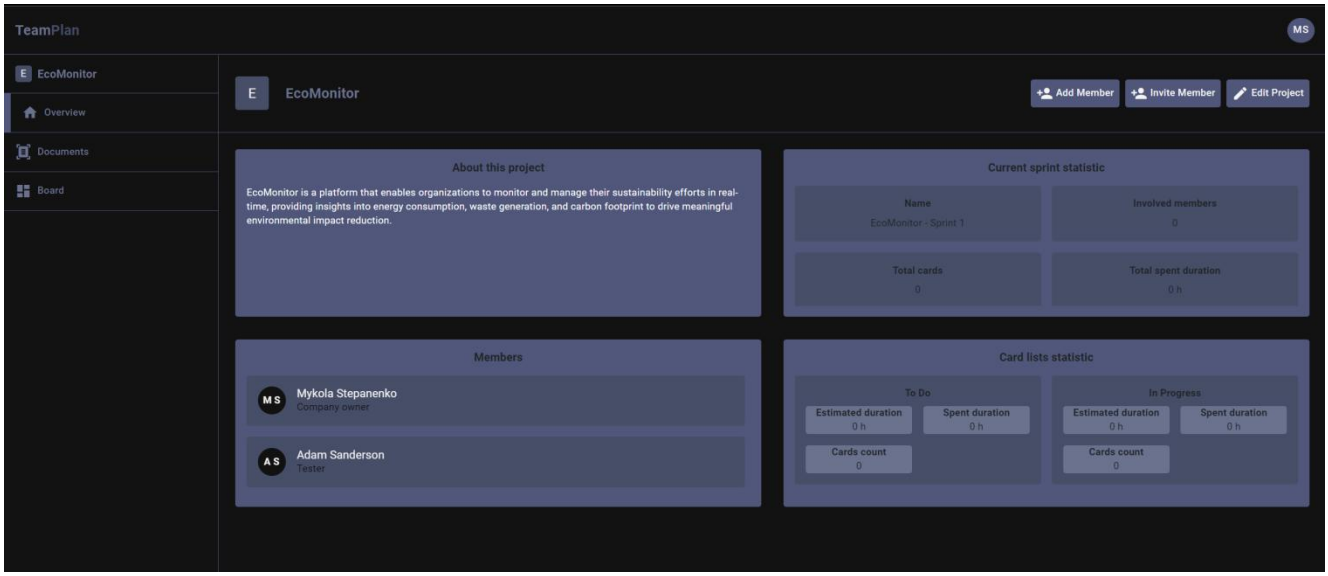


Рисунок 4.3 — Загальний огляд проекту

Користувач може перейти на сторінку з документами (рисунок 4.4), де він може переглянути список усіх доступних документів, обрати документ для детального перегляду та редагувати його. Ця функція дозволяє користувачам ефективно керувати проектною документацією.

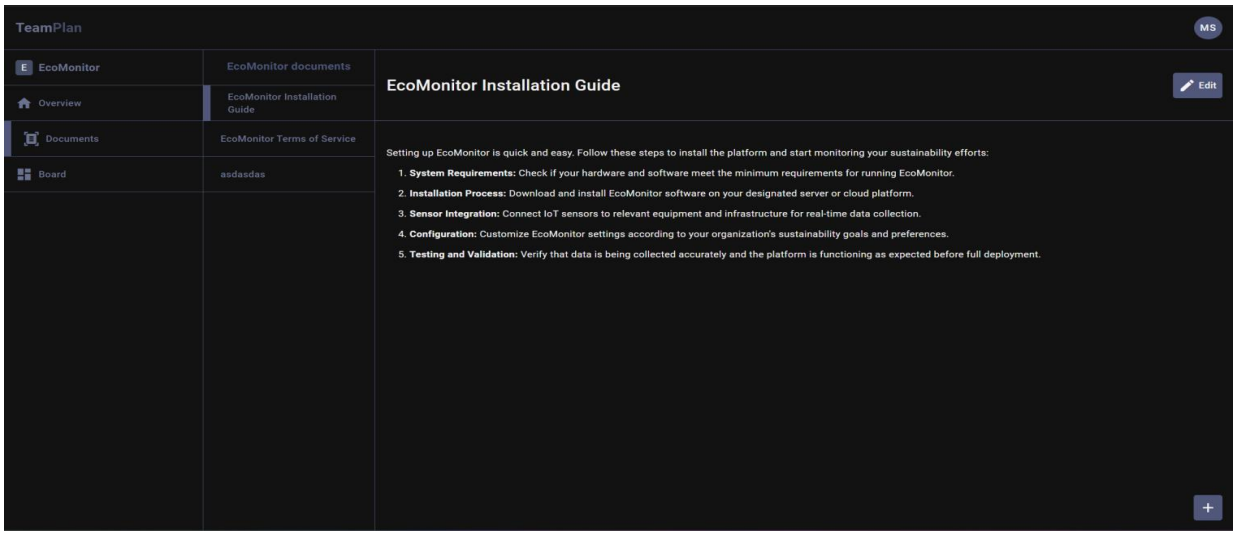


Рисунок 4.4 — Перегляд документів

Для створення нового документа користувач натискає відповідну кнопку у правому нижньому кутку екрану, після чого відкривається нова сторінка для створення документа (рисунок 4.5). На цій сторінці користувач вказує ім'я документа та заповнює його вміст.

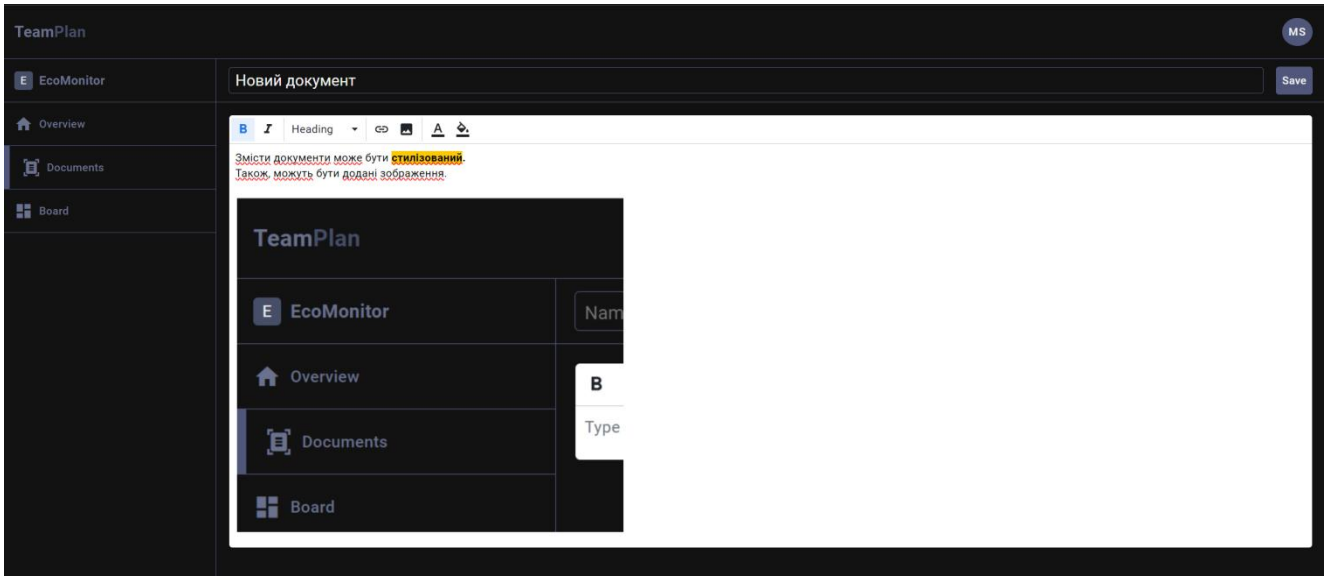


Рисунок 4.5 — Створення нового документу

Для роботи з задачами проекту користувач переходить на відповідну сторінку, де представлена дошка з картками, розташованими по колонках (рисунок 4.6). Користувач може фільтрувати картки за іменами, учасниками, що призначені на них, та за поточною ітерацією проекту. Ця функція допомагає користувачам легко організовувати та відстежувати завдання.

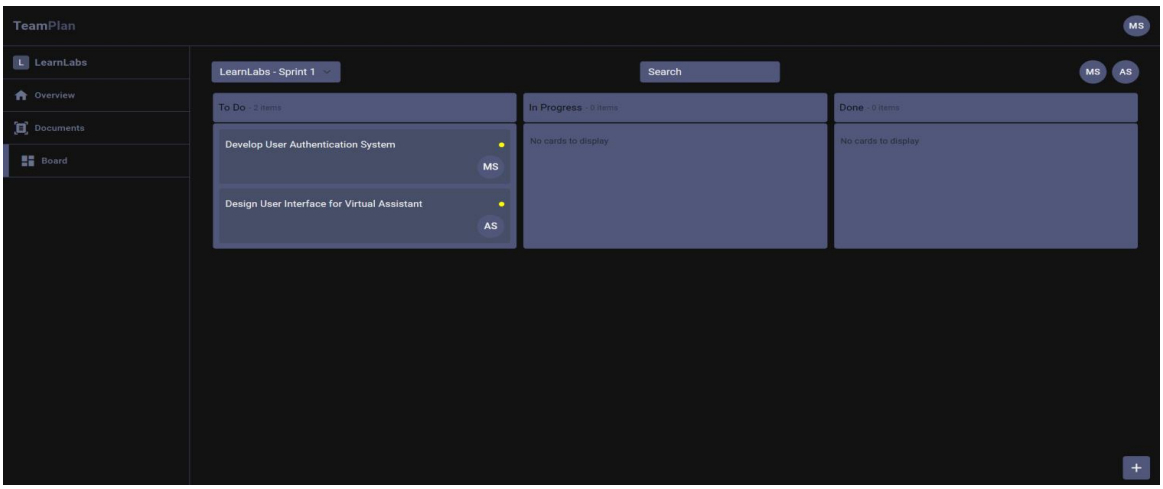


Рисунок 4.6 — Перегляд дошки з картками

Двічі натиснувши на картку, користувач відкриває модальне вікно з детальною інформацією про картку (рисунок 4.7). У цьому вікні можна побачити детальний опис задачі, тип картки, користувача, призначеного для її виконання, дату початку та завершення виконання задачі, а також оцінений час на виконання задачі.

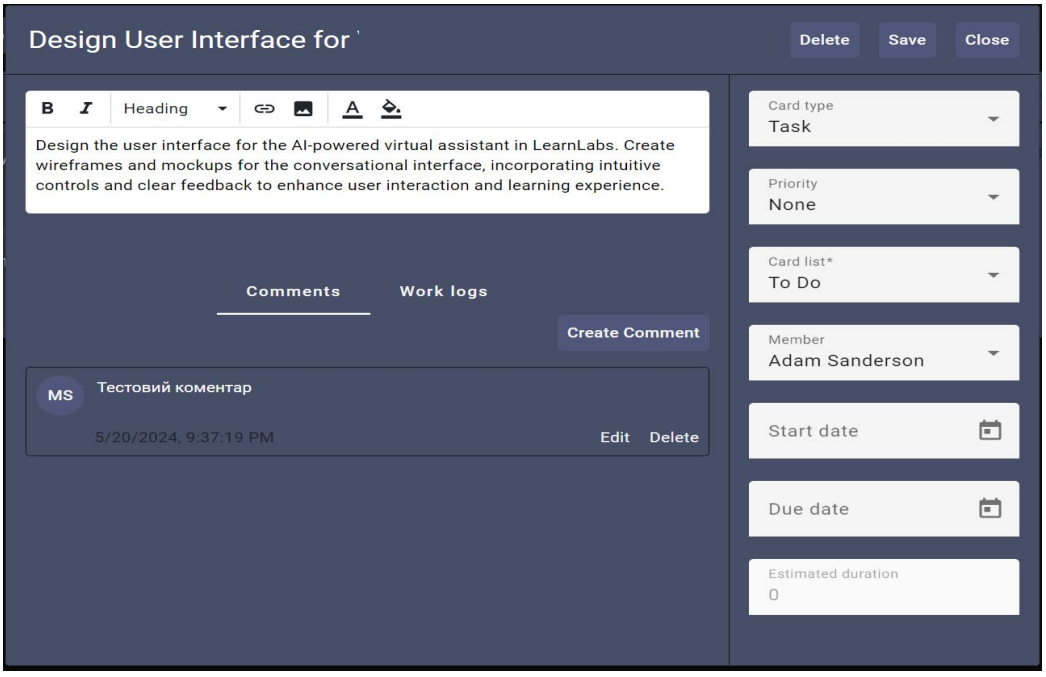


Рисунок 4.7 — Перегляд та редагування картки

У модальному вікні картки користувач також може переглядати, створювати та видаляти коментарі, а також вносити інформацію про витрачений на виконання задачі час. Користувач має змогу змінити тип картки, її пріоритет, список карток, до якого вона належить, учасника, що неї призначений та вказати початку та кінцеві дати. Після внесення змін користувач може натиснути кнопку збереження для оновлення картки або видалити картку.

Для створення нової картки користувач натискає відповідну кнопку у правому нижньому кутку дошки. Це відкриває модальне вікно для створення картки, яке має таку саму структуру, що й вікно перегляду та редагування картки (рисунок 4.8).

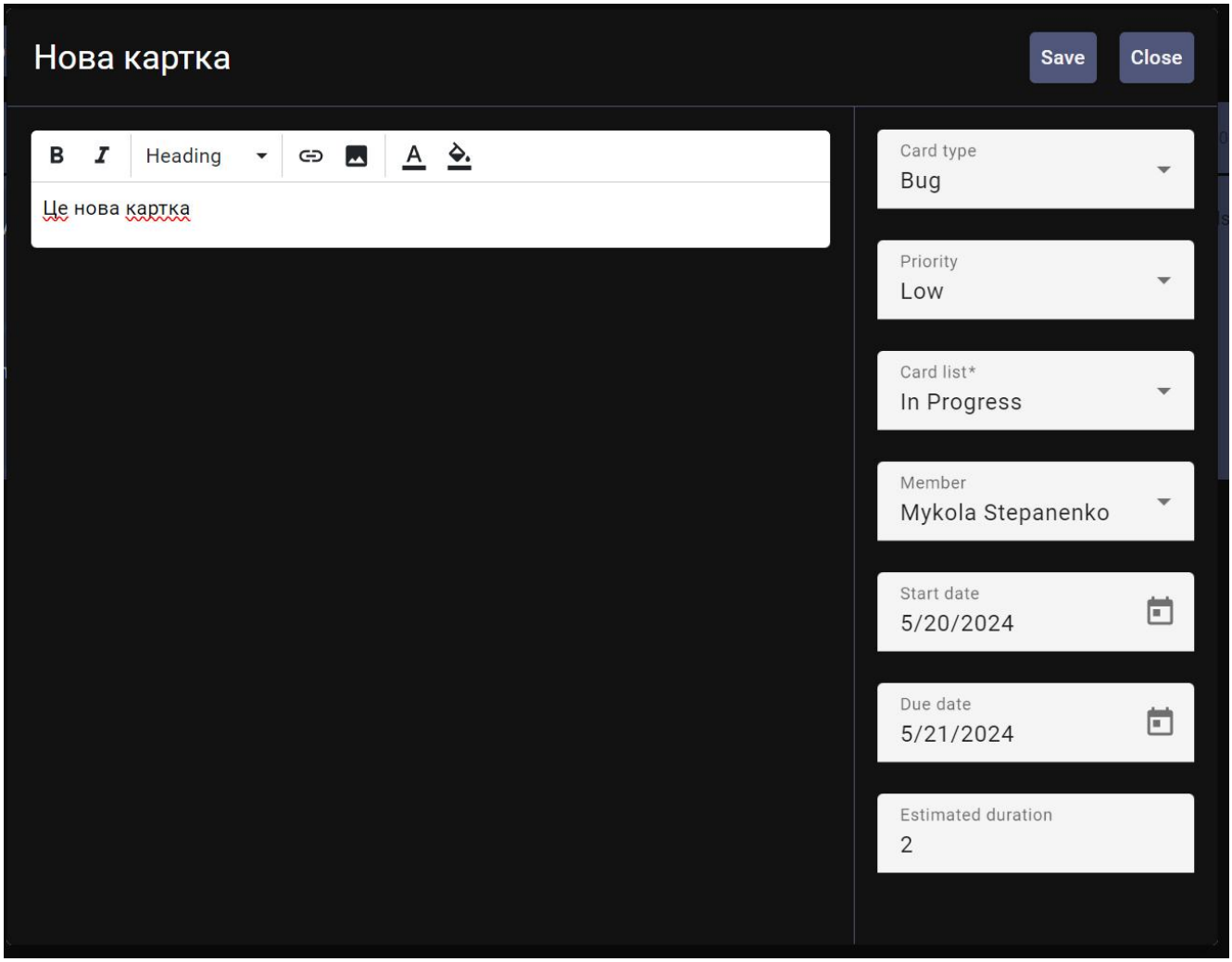


Рисунок 4.8 — Створення нової картки

Функціонал розробленої системи покриває потреби більшості користувачів подібних програмних рішень, надаючи зрозумілий та простий інтерфейс.

4.3 Сценарій роботи власника компанії з інформаційною системою

Якщо користувач є власником компанії, йому доступні додаткові функції для управління проектом та учасниками. Зокрема, він може додавати нових учасників до проекту. Є два варіанти додавання нових учасників: додавання існуючого користувача системи та додавання нового користувача. Для додавання існуючого користувача власник компанії використовує форму, зображену на

рисунку 4.9. Тут власник обирає користувача зі списку та вказує його роль у проекті.

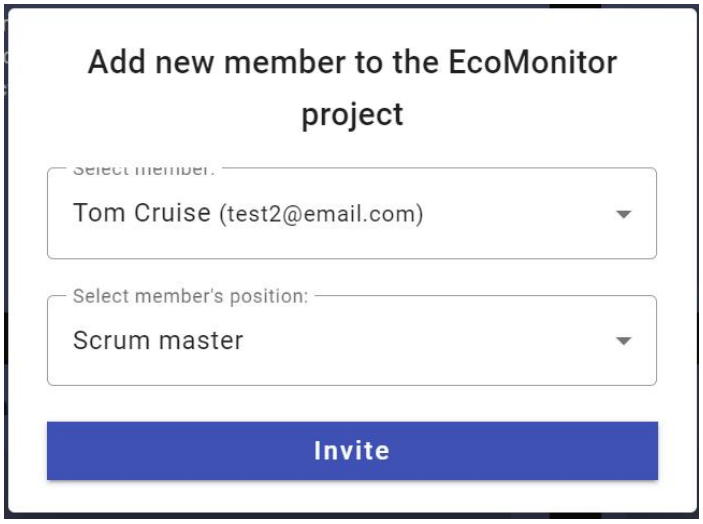


Рисунок 4.9— Додавання існуючого користувача системи на проект

На наведеній вище формі, власник компанії обирає користувача зі списку та вказує його посаду на цьому проекті. Для додавання нового користувача, який ще не зареєстрований у системі, власник компанії використовує іншу форму (рисунок 4.10). У цій формі необхідно вказати адресу електронної пошти користувача, його ім’я, прізвище та посаду. Після заповнення форми користувач отримає запрошення для реєстрації у системі.

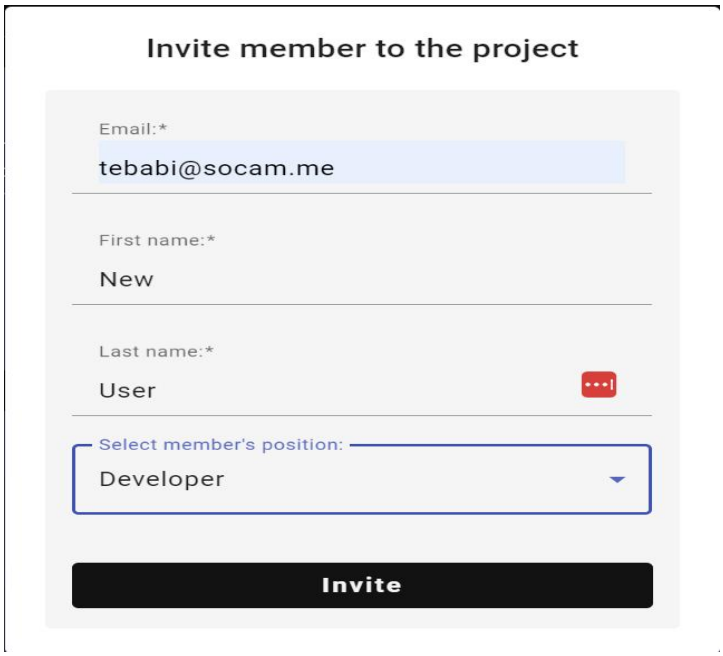


Рисунок 4.10 — Додавання нового користувача на проект

При заповнені даної форми, необхідно вказати адресу поштової скриньки користувача, на яку надійде запрошення для реєстрації, його ім’я, прізвище на посаду.

Сторінка редагування проекту (рисунок 4.11) дозволяє власнику компанії змінювати ім’я проекту та його опис, видаляти проект, видаляти учасників з проекту, а також створювати, редагувати або видаляти списки карток. Це надає власнику компанії повний контроль над управлінням проектом.

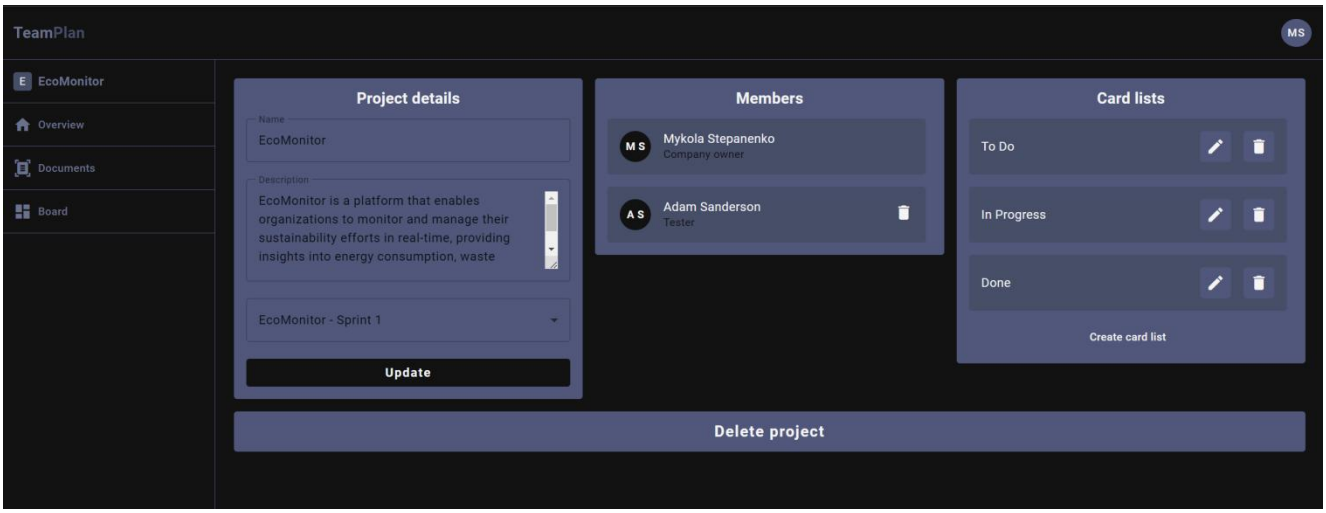


Рисунок 4.11 — Редагування проекту

Цей розширений функціонал для власника компанії забезпечує високу гнучкість і повний контроль над проектами, що дозволяє ефективно керувати ресурсами, задачами і командою. Таким чином, власник компанії може оперативно реагувати на зміни в проекті, вносити необхідні корективи та забезпечувати високий рівень організації і продуктивності команди.

ВИСНОВКИ

1. Аналіз існуючих систем: У процесі роботи було детально проаналізовано такі відомі платформи, як Azure DevOps, Jira та Trello. Виявлено їхні переваги, такі як потужні функціональні можливості та широка популярність, а також недоліки, включаючи складність у використанні, обмежений функціонал для безкоштовного використання, залежність від хмарних технологій та недостатню адаптацію до специфічних потреб різних предметних областей.

2. Розробка нової системи: Основною метою роботи було створення нової інформаційної системи, яка керується принципами предметно-орієнтованого програмування. Це дозволяє створювати моделі, що відображають реальні бізнес-процеси, забезпечуючи тісний зв’язок між програмним забезпеченням і предметною областю. Використання цього підходу підвищує гнучкість системи, спрощує її розширення і підтримку в майбутньому, забезпечує модульність та повторне використання коду.

3. Функціональні можливості системи: Розроблена система охоплює управління проектами, завданнями та документами. Вона надає зручний та інтуїтивно зрозумілий інтерфейс користувачам, підтримує різні ролі користувачів, що дозволяє налаштувати доступ до інформації відповідно до ролей і обов’язків. Це забезпечує підвищення продуктивності та ефективності роботи команди.

4. Веб-версія системи: Додатково була створена веб-версія системи, яка забезпечує доступність та зручність використання з будь-якого пристрою, підключеного до Інтернету. Це робить систему більш гнучкою та доступною для широкого кола користувачів, підвищуючи її практичність та зручність.

5. Надійність та масштабованість: Створена інформаційна система є надійною та масштабованою, що дозволяє використовувати її як частину або основу для більш комплексних рішень у сфері управління проектами. Вона може

бути адаптована під специфічні потреби організацій та сприяти підвищенню ефективності їхньої діяльності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. 1st Edition. 2003. 560 p.

2. Документація платформи Azure DevOps. URL: https://azure.microsoft.com/en—us/products/devops

3. Документація платформи Jira. URL: https://atlassian.com/software/jira

4. Документація платформи Trello . URL: https://trello.com/

5. Порівняння предметно-орієнтованого програмування з іншими підходами до розробки інформаційних систем. URL: https://enterprisecraftsmanship.com/posts/domain-centric-vs-data-centric-approaches/

6. Документація фреймворку ASP.NET Core. URL: https://learn.microsoft.com/en—us/aspnet/core/?view=aspnetcore—8.0

7. Порівняння ASP.NET Core з іншими фрейморками. URL: https://www.abtosoftware.com/blog/why-use-asp-net-mvc-framework-for-web-application-development

8. Документація бібліотеки EF Core. URL: https://learn.microsoft.com/en—us/ef/core/

9. Документація мови програмування C#. URL: https://learn.microsoft.com/en-us/dotnet/csharp/

10. Документація фреймворку Angular. URL: https://angular.io/docs

11. Документація мови програмування TypeScript. URL: https://www.typescriptlang.org/docs/

12. Документація мови гіпертекстової розмітки HTML: URL: https://developer.mozilla.org/en-US/docs/Web/HTML

13. Документація системи керування версіями Git. URL: https://git—scm.com/

14. Мартін Р. Чиста архітектура. Мистецтво розробки програмного забезпечення. Харків: Вид-во "Фабула", 2019. 368 с.

15. Документація бібліотеки MediatR. URL: https://github.com/jbogard/MediatR

16. Betts D. Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure. Microsoft patterns & practices. 2015. 376 p.

17. Документація бібліотеки FluentValidation. URL: https://docs.fluentvalidation.net/en/latest/

18. Biehl M. API Architecture – The Big Picture for Building APIs. API University Series. 2015. 190 p.

19. Документація бази даних PostgreSQL. URL: https://www.postgresql.org/docs/

20. Документація додатку Docker Desktop. URL: https://docs.docker.com/

21. Веб-сервіс GitHub. URL: https://github.com/

ДОДАТОК А

ТЕКСТ ПРОГРАМНОГО МОДУЛЯ

«СИСТЕМА КЕРУВАННЯ РЕСУРСАМИ ТА ЗАДАЧАМИ НА ОСНОВІ ПРЕДМЕТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ»

УКР.НТУУ”КПІ ім. Ігоря Сікорського”_ІАТЕ_ЦТЕ_ТР-01_24Б

Аркушів 19

Київ – 2024

Текст програми

```
public abstract class Entity(Guid id) : IEquatable<Entity>
{
	public Guid Id { get; } = id;

	private readonly List<BaseEvent> _domainEvents = new();

	[NotMapped]
	public IReadOnlyCollection<BaseEvent> DomainEvents =>
_domainEvents.AsReadOnly();

	public void AddDomainEvent(BaseEvent domainEvent)
	{
		_domainEvents.Add(domainEvent);
	}

	public void RemoveDomainEvent(BaseEvent domainEvent)
	{
		_domainEvents.Remove(domainEvent);
	}

	public void ClearDomainEvents()
	{
		_domainEvents.Clear();
	}

	public override bool Equals(object? obj)
	{
		if (ReferenceEquals(null, obj))
		{
			return false;
		}

		if (ReferenceEquals(this, obj))
		{
			return true;
		}

		if (obj.GetType() != this.GetType())
		{
			return false;
		}
```

```
            return Equals((Entity)obj);
        }

        public bool Equals(Entity? other)
        {
            if (ReferenceEquals(null, other))
            {
                return false;
            }

            if (ReferenceEquals(this, other))
            {
                return true;
            }

            return _domainEvents.Equals(other._domainEvents) && Id == other.Id;
        }

        public override int GetHashCode()
        {
            return HashCode.Combine(_domainEvents, Id);
        }

        public static bool operator ==(Entity? left, Entity? right)
        {
            return Equals(left, right);
        }

        public static bool operator !=(Entity? left, Entity? right)
        {
            return !Equals(left, right);
        }
    }

    public abstract class AuditableEntity(Guid id) : Entity(id)
    {
        public DateTime CreatedAt { get; set; }

        public Guid? CreatedBy { get; set; }

        public DateTime? LastModified { get; set; }

        public Guid? LastModifiedBy { get; set; }
    }
```

```
public class Result
{
    public ResultType ResultType { get; init; }

    public string[] Errors { get; init; }

    protected Result(
        ResultType resultType,
        IEnumerable<string> errors)
    {
        ResultType = resultType;
        Errors = errors.ToArray();
    }

    public static Result Success(ResultType resultType)
    {
        return new Result(resultType, Array.Empty<string>());
    }

    public static Result<T> Success<T>(ResultType resultType, T value)
    {
        return new Result<T>(resultType, Array.Empty<string>(), value);
    }

    public static Result Failure(ResultType resultType, params string[] errors)
    {
        return new Result(resultType, errors);
    }

    public static Result<T> Failure<T>(ResultType resultType, params string[] errors)
    {
        return new Result<T>(resultType, errors, default);
    }
}

public class Result<T> : Result
{
    public T? Value { get; init; }

    protected internal Result(
        ResultType resultType,
        IEnumerable<string> errors,
        T? value) : base(resultType, errors)
    {
        Value = value;
```

```
    }
}

public sealed class Project : AuditableEntity
{
    public required string Name { get; set; }

    public string? Description { get; set; }

    public Guid CurrentSprintId { get; set; }

    public IEnumerable<Sprint> Sprints { get; init; } = new List<Sprint>();

    public IEnumerable<ProjectMember> ProjectMembers { get; init; } = new
List<ProjectMember>();

    public IEnumerable<Document> Documents { get; init; } = new
List<Document>();

    public IEnumerable<CardList> CardLists { get; init; } = new List<CardList>();

    private Project(Guid id) : base(id)
    {
    }

    public static Project Create(
        string name,
        string? description = null)
    {
        var project = new Project(Guid.NewGuid())
        {
            Name = name,
            Description = description
        };

        return project;
    }
}

public sealed class Sprint : AuditableEntity
{
    public required string Name { get; init; }

    public required int Iteration { get; init; }
```

```
    public required Guid ProjectId { get; init; }

    public Project Project { get; init; } = null!;

    public DateTime? Start { get; init; }

    public DateTime? End { get; init; }

    public IEnumerable<Card> Cards { get; init; } = new List<Card>();

    private Sprint(Guid id) : base(id)
    {
    }

    public static Sprint Create(
        int iteration,
        Project project,
        DateTime? start = null,
        DateTime? end = null)
    {
        var projectSprint = new Sprint(Guid.NewGuid())
        {
            Name = $"{project.Name} — Sprint {iteration}",
            Iteration = iteration,
            ProjectId = project.Id,
            Start = start,
            End = end
        };

        return projectSprint;
    }
}

public sealed class ProjectMember : AuditableEntity
{
    public required Guid MemberId { get; init; }

    public required Guid ProjectId { get; init; }

    public Project Project { get; init; } = null!;

    public required Guid MemberPositionId { get; init; }

    public MemberPosition MemberPosition { get; init; } = null!;
```

```
    private ProjectMember(Guid id) : base(id)
    {
    }

    public static ProjectMember Create(
        Guid memberId,
        Guid projectId,
        Guid memberPositionId)
    {
        var projectMember = new ProjectMember(Guid.NewGuid())
        {
            MemberId = memberId,
            ProjectId = projectId,
            MemberPositionId = memberPositionId
        };

        return projectMember;
    }
}

public sealed class CardList : AuditableEntity
{
    public required string Name { get; set; }

    public required int Position { get; set; }

    public required Guid ProjectId { get; init; }

    public IEnumerable<Card> Cards { get; init; } = new List<Card>();

    private CardList(Guid id) : base(id)
    {
    }

    public static CardList Create(
        string name,
        int position,
        Guid projectId)
    {
        var cardList = new CardList(Guid.NewGuid())
        {
            Name = name,
            Position = position,
            ProjectId = projectId
        };
```

```
            return cardList;
        }
}

public sealed class Card : AuditableEntity
{
    public required string Name { get; set; }

    public string? Description { get; set; }

    public int Position { get; set; }

    public Guid? ParentCardId { get; set; }

    public Card? ParentCard { get; init; }

    public required Guid CardListId { get; set; }

    public CardList CardList { get; init; } = null!;

    public required Guid SprintId { get; init; }

    public Sprint Sprint { get; init; } = null!;

    public PriorityLevel PriorityLevel { get; set; } = PriorityLevel.None;

    public DateTime? StartDate { get; set; }

    public DateTime? DueDate { get; set; }

    public double? EstimatedDuration { get; init; }

    public Guid? MemberId { get; set; }

    public IEnumerable<Card> SubCards { get; init; } = new List<Card>();

    public CardType CardType { get; set; }

    public IEnumerable<CardComment> Comments { get; init; } = new
List<CardComment>();

    public IEnumerable<CardTimeLog> TimeLogs { get; init; } = new
List<CardTimeLog>();
```

```
    private Card(Guid id) : base(id)
    {
    }

    public static Card Create(
        string name,
        string? description,
        int position,
        Guid cardListId,
        Guid sprintId,
        PriorityLevel priorityLevel,
        CardType cardType,
        Guid? memberId = null,
        DateTime? startDate = null,
        DateTime? dueDate = null,
        Guid? parentCardId = null,
        double? estimatedDuration = null)
    {
        var card = new Card(Guid.NewGuid())
        {
            Name = name,
            Description = description,
            Position = position,
            ParentCardId = parentCardId,
            CardListId = cardListId,
            SprintId = sprintId,
            PriorityLevel = priorityLevel,
            StartDate = startDate,
            DueDate = dueDate,
            MemberId = memberId,
            CardType = cardType,
            EstimatedDuration = estimatedDuration
        };

        return card;
    }
}

public sealed class CardComment : AuditableEntity
{
    public required Guid CardId { get; init; }

    public Card Card { get; init; } = null!;

    public required CardCommentText Text { get; set; }
```

```
    public required Guid MemberId { get; init; }

    private CardComment(Guid id) : base(id)
    {
    }

    public static CardComment Create(
        string text,
        Guid cardId,
        Guid authorId)
    {
        var cardComment = new CardComment(Guid.NewGuid())
        {
            Text = CardCommentText.Create(text),
            CardId = cardId,
            MemberId = authorId
        };

        return cardComment;
    }
}

public sealed class CardTimeLog : AuditableEntity
{
    public required Guid CardId { get; init; }

    public Card Card { get; init; } = null!;

    public required DateTime Start { get; set; }

    public required DateTime End { get; set; }

    public double Duration { get; init; }

    public string? Description { get; set; }

    public required Guid MemberId { get; init; }

    private CardTimeLog(Guid id) : base(id)
    {
    }

    public static CardTimeLog Create(
        Guid cardId,
```

```
            DateTime start,
            DateTime end,
            Guid memberId,
            string? description = null)
        {
            var cardTimeLog = new CardTimeLog(Guid.NewGuid())
            {
                CardId = cardId,
                Start = start,
                End = end,
                Description = description,
                MemberId = memberId,
                Duration = (end — start).TotalHours
            };

            return cardTimeLog;
        }
}


public class CreateProjectCommand : IRequest<Result<ProjectDto>>
{
    public required string Name { get; init; }

    public string? Description { get; init; }
}

internal sealed class CreateProjectCommandHandler(
    IApplicationDbContext applicationDbContext,
    ICurrentUserService currentUserService)
    : IRequestHandler<CreateProjectCommand, Result<ProjectDto>>
{
    public async Task<Result<ProjectDto>> Handle(CreateProjectCommand request,
CancellationToken cancellationToken)
    {
        var project = Project.Create(request.Name, request.Description);

        applicationDbContext.Projects.Add(project);

        project.AddDomainEvent(new ProjectCreatedEvent(project));

        var userId = currentUserService.UserId;

        var companyOwnerPosition = await applicationDbContext
            .MemberPositions
```

```
                .SingleAsync(mp => mp.Name == MemberPositions.CompanyOwner,
cancellationToken);

            var projectMember = ProjectMember.Create(userId!.Value, project.Id,
companyOwnerPosition.Id);

            await applicationDbContext.ProjectMembers.AddAsync(projectMember,
cancellationToken);

            var sprint = Sprint.Create(iteration: 1, project);

            await applicationDbContext.Sprints.AddAsync(sprint, cancellationToken);

            project.CurrentSprintId = sprint.Id;

            var cardLists = new List<CardList>
            {
                CardList.Create("To Do", 0, project.Id),
                CardList.Create("In Progress", 1, project.Id),
                CardList.Create("Done", 2, project.Id)
            };

            await applicationDbContext.CardLists.AddRangeAsync(cardLists,
cancellationToken);

            try
            {
                await applicationDbContext.SaveChangesAsync(cancellationToken);
            }
            catch (Exception ex)
            {
                return Result.Failure<ProjectDto>(ResultType.InternalServerError,
ex.Message);
            }

            return Result.Success(ResultType.Created, new ProjectDto(project));
        }
}

internal class CreateProjectCommandValidator :
AbstractValidator<CreateProjectCommand>
{
    private readonly IApplicationDbContext _applicationDbContext;

    public CreateProjectCommandValidator(IApplicationDbContext
```

```
applicationDbContext)
    {
        _applicationDbContext = applicationDbContext;

        RuleFor(x => x.Name)
            .NotEmpty().WithMessage("Name is required.")
            .MaximumLength(200).WithMessage("Name must not exceed 200
characters.");

        RuleFor(x => x.Name)
            .MustAsync(HaveUniqueNameAsync)
            .WithMessage(command => $"The specified name already exists:
{command.Name}.");
    }

    private Task<bool> HaveUniqueNameAsync(
        string name,
        CancellationToken cancellationToken)
    {
        return _applicationDbContext
            .Projects
            .AllAsync(b => b.Name != name, cancellationToken);
    }
}

public class CreateCardCommand : IRequest<Result<CardDto>>
{
    public required string Name { get; init; }

    public string? Description { get; init; }

    public required Guid CardListId { get; init; }

    public required Guid SprintId { get; init; }

    public Guid? ParentCardId { get; init; }

    public PriorityLevel PriorityLevel { get; init; }

    public DateTime? StartDate { get; init; }

    public DateTime? DueDate { get; init; }

    public Guid? MemberId { get; init; }
```

```
    public required CardType CardType { get; init; }

    public double? EstimatedDuration { get; init; }
}

public class CreateCardCommandHandler(IApplicationDbContext
applicationDbContext)
    : IRequestHandler<CreateCardCommand, Result<CardDto>>
{
    public async Task<Result<CardDto>> Handle(
        CreateCardCommand request,
        CancellationToken cancellationToken)
    {
        var cardsCount = await applicationDbContext.Cards
            .Where(c => c.CardListId == request.CardListId && c.SprintId ==
request.SprintId)
            .CountAsync(cancellationToken);

        var card = Card.Create(
            request.Name,
            request.Description,
            cardsCount + 1,
            request.CardListId,
            request.SprintId,
            request.PriorityLevel,
            request.CardType,
            request.MemberId,
            request.StartDate,
            request.DueDate,
            request.ParentCardId,
            request.EstimatedDuration);

        await applicationDbContext.Cards.AddAsync(card, cancellationToken);

        try
        {
            card.AddDomainEvent(new CardCreatedEvent(card));
            await applicationDbContext.SaveChangesAsync(cancellationToken);
        }
        catch (Exception e)
        {
            return Result.Failure<CardDto>(ResultType.InternalServerError,
e.Message);
        }
```

```
            return Result.Success(ResultType.Created, new CardDto(card));
        }
}

internal class CreateCardCommandValidator :
AbstractValidator<CreateCardCommand>
{
    private readonly IApplicationDbContext _applicationDbContext;

    public CreateCardCommandValidator(IApplicationDbContext
applicationDbContext)
    {
        _applicationDbContext = applicationDbContext;

        RuleFor(command => command.Name)
            .NotEmpty()
            .WithMessage("Name is required.");

        RuleFor(command => command.ParentCardId)
            .MustAsync(UseExistingParentCardAsync)
            .When(command => command.ParentCardId != null)
            .WithMessage("The specified parent card does not exist.");

        RuleFor(command => command.ParentCardId)
            .MustAsync(ParentIsNotOfTypeTaskAsync)
            .When(command => command.ParentCardId != null)
            .WithMessage("Task card can't have a child card.");

        RuleFor(command => command.ParentCardId)
            .MustAsync((command, parentCardId, cancellationToken) =>
                ParentCardMustBeHigherInHierarchyAsync(parentCardId,
command.CardType, cancellationToken))
            .When(command => command.ParentCardId != null)
            .WithMessage("Parent card must be higher in hierarchy.");

        RuleFor(command => command.CardType)
            .Must(NotBeBugWithParent)
            .When(command => command.ParentCardId != null)
            .WithMessage("Bug card can't have a parent card.");

        RuleFor(command => command.StartDate)
            .LessThanOrEqualTo(command => command.DueDate)
            .WithMessage("Start date must be less than or equal to due date.");
    }
```

```
    private Task<bool> UseExistingParentCardAsync(
        Guid? parentCardId,
        CancellationToken cancellationToken)
    {
        return _applicationDbContext
            .Cards
            .AnyAsync(c => c.Id == parentCardId, cancellationToken);
    }

    private async Task<bool> ParentIsNotOfTypeTaskAsync(
        Guid? parentCardId,
        CancellationToken cancellationToken)
    {
        var parentCard = await _applicationDbContext
            .Cards
            .FindAsync([parentCardId], cancellationToken);

        return parentCard!.CardType != CardType.Task;
    }

    private async Task<bool> ParentCardMustBeHigherInHierarchyAsync(
        Guid? parentCardId,
        CardType cardType,
        CancellationToken cancellationToken)
    {
        var parentCard = await _applicationDbContext
            .Cards
            .FindAsync(new object?[] { parentCardId }, cancellationToken);

        return (int)cardType > (int)parentCard!.CardType;
    }

    private bool NotBeBugWithParent(CardType cardType)
    {
        return cardType != CardType.Bug;
    }
}

public class CreateCardListCommand : IRequest<Result<CardListDto>>
{
    public required string Name { get; init; }

    public required Guid ProjectId { get; init; }
}
```

```
internal class CreateCardListCommandHandler(
    IApplicationDbContext applicationDbContext)
    : IRequestHandler<CreateCardListCommand, Result<CardListDto>>
{
    public async Task<Result<CardListDto>> Handle(
        CreateCardListCommand request,
        CancellationToken cancellationToken)
    {
        var cardListsCount = await applicationDbContext
            .CardLists
            .Where(cardList => cardList.ProjectId == request.ProjectId)
            .CountAsync(cancellationToken);

        var cardList = CardList.Create(
            request.Name,
            cardListsCount + 1,
            request.ProjectId);

        applicationDbContext.CardLists.Add(cardList);

        try
        {
            cardList.AddDomainEvent(new CardListCreatedEvent(cardList));

            await applicationDbContext.SaveChangesAsync(cancellationToken);
        }
        catch (Exception ex)
        {
            return Result.Failure<CardListDto>(ResultType.InternalServerError,
ex.Message);
        }

        return Result.Success(ResultType.Created, new CardListDto(cardList));
    }
}

internal class CreateCardListCommandValidator :
AbstractValidator<CreateCardListCommand>
{
    private readonly IApplicationDbContext _applicationDbContext;

    public CreateCardListCommandValidator(IApplicationDbContext
applicationDbContext)
    {
        _applicationDbContext = applicationDbContext;
```

```
            RuleFor(command => command.Name)
                .NotEmpty()
                .MaximumLength(20)
                .MustAsync(UniqueNameAsync)
                .WithMessage(command => $"The specified name already exists:
{command.Name}.");

            RuleFor(x => x.ProjectId)
                .MustAsync(ProjectExistsAsync)
                .WithMessage("The specified project does not exist.");
        }

        private async Task<bool> UniqueNameAsync(
            CreateCardListCommand command,
            string name,
            CancellationToken cancellationToken)
        {
            var cardListWithTheSameNameExists = await _applicationDbContext
                .CardLists
                .Where(cardList => cardList.ProjectId == command.ProjectId)
                .AnyAsync(cardList => cardList.Name.Equals(name),
cancellationToken);

            return !cardListWithTheSameNameExists;
        }

        private Task<bool> ProjectExistsAsync(
            Guid projectId,
            CancellationToken cancellationToken)
        {
            return _applicationDbContext
                .Projects
                .AnyAsync(project => project.Id == projectId, cancellationToken);
        }
}

public class DeleteProjectCommand : IRequest<Result>
{
        public Guid ProjectId { get; init; }
}

internal sealed class DeleteProjectCommandHandler(IApplicationDbContext
applicationDbContext)
        : IRequestHandler<DeleteProjectCommand, Result>
```

```
{
    public async Task<Result> Handle(
        DeleteProjectCommand request,
        CancellationToken cancellationToken)
    {
        var project = await applicationDbContext
            .Projects
            .FindAsync(
                new object?[] { request.ProjectId },
                cancellationToken);

        if (project == null)
        {
            return Result.Failure(
                ResultType.NotFound,
                $"Project with id {request.ProjectId} wasn't found");
        }

        applicationDbContext.Projects.Remove(project);

        project.AddDomainEvent(new ProjectDeletedEvent(project));

        await applicationDbContext.SaveChangesAsync(cancellationToken);

        return Result.Success(ResultType.NoContent);
    }
}
```