

**МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ
ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ**

**ФАКУЛЬТЕТ ФІНАНСІВ ТА ЦИФРОВИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І
СИСТЕМ**

КВАЛІФІКАЦІЙНА РОБОТА

на тему: “Розробка системи управління завданнями для проєктної команди”

Здобувача вищої освіти першого (бакалаврського) рівня,
IV курсу, групи КБ-21-1, спеціальності 122 «Комп'ютерні науки»,
освітня програма «Інформаційні управляючі системи і технології (за галузями)»



(підпис)

Сидорука Дмитра
Олександровича

(прізвище, ім'я, по батькові)

Науковий керівник



(підпис)

Редич Олександр
Володимирович, к.ф.-м.н.,
доцент

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

Керівник освітньо-
професійної програми



(підпис)

Омельчук Антон
Анатолійович, к.т.н.

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

Допущено до захисту за рішенням кафедри комп'ютерних та інформаційних
технологій і систем протокол від «13» травня 2025 № 18

Завідувач кафедри



(підпис)

Філоненко Михайло
Миколайович, к.ф.-м.н., доцент

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

Ірпінь – 2025

Реєстрація № 44

23.05.2025

(дата)



(підпис відповідальної особи)

Результати перевірки

до захисту

(до захисту, на доопрацювання)

23.05.2025

(дата)



(підпис)

Результат захисту

4 (добре)

(оцінка)

04.06.2025

(дата)



(підпис)

Голова комісії:



(підпис)

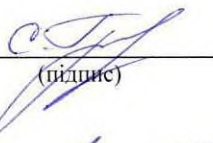
В.М. Франчук

Члени комісії:



(підпис)

М.М. Філоненко



(підпис)

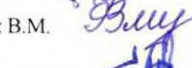
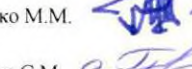
С.М. Грищенко



(підпис)

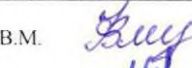
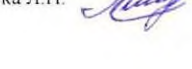
А.А. Омельчук

БЛАНК ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Об'єкти контролю	Діапазон оцінки в балах	Фактична кількість балів	Примітки	Дата, П.І.Б.: керівника, голови комісії членів комісії, секретаря, (підписи)
ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ КОМІСІЄЮ ПО ЗАХИСТУ				
1. Зміст випускної роботи (якість теоретичного матеріалу, його зв'язок з практичним, вміння представляти матеріал в логічній послідовності, наявність висновків та пропозицій)	від 0 до 30	26		Редич О.В.  Франчук В.М. 
2. Дотримання вимог щодо оформлення	від 0 до 10	8		Філоненко М.М.  Грищенко С.М. 
3. Вчасне виконання роботи на всіх етапах у встановлені кафедрою терміни	від 0 до 5	4		Омельчук А.А.  Суховерх Л.П. 
4. Рецензент	від 0 до 5	5		
РАЗОМ БАЛІВ	від 0 до 50	43		

Оцінювання кваліфікаційної роботи науковим керівником – 42 балів.

ПЕРЕВЕДЕННЯ ОЦІНКИ ЗА 100-БАЛЬНОЮ ШКАЛОЮ ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою	Дата	П.І.Б.: голови, членів комісії, секретаря, (підписи)
85	B	4 (добре)	04.06.2025	Франчук В.М.  Філоненко М.М.  Грищенко С.М.  Омельчук А.А.  Суховерх Л.П. 

ПРАВИЛА ПЕРЕВЕДЕННЯ ОЦІНОК ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою
90-100	A	5 (відмінно)
80-89	B	4 (добре)
70-79	C	
60-69	D	
50-59	E	3 (задовільно)
35-49	FX	2 (незадовільно)

Факультет фінансів та цифрових технологій
Кафедра комп'ютерних та інформаційних технологій і систем
Галузь знань 12 «Інформаційні технології»
Спеціальність 122 «Комп'ютерні науки»
Освітня програма «Інформаційні управляючі системи і технології (за галузями)»

Затверджую

Завідувач кафедри  Філоненко М.М.

«15» листопада 2024

**Завдання на кваліфікаційну роботу
здобувачу вищої освіти**

Сидоруку Дмитру Олександровичу

1. Тема кваліфікаційної роботи **«Розробка системи управління завданнями для проєктної команди»**

Затверджена наказом ректора від «13» листопада 2024 № 1717

2. Строк здачі закінченої роботи «5» травня 2025

3. Мета роботи: розробка веб-додатку для управління завданнями проєктної команди.

Об'єкт дослідження: методи та технології розробки веб-додатків для автоматизації управління завданнями у проєктних командах.

Предмет дослідження: інструментарій розробки веб-застосунків з використанням ASP.NET, Core MVC, Entity Framework Core.

Орієнтовна база дослідження: системи управління проєктами Trello, Jira, Asana, ClickUp, наукові праці в галузі управління проєктами.

4. Вхідні дані до роботи: відкриті дані та документація по ASP.NET, Core MVC, систем управління завданнями, таких як Trello, Jira, Asana, ClickUp.

5. Перелік графічного матеріалу: рисунки інтерфейсів додатку, структурні схеми, алгоритми, лістинги програм.

6. Зміст кваліфікаційної роботи (перелік питань за кожним розділом)

РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ СИСТЕМ УПРАВЛІННЯ ЗАВДАННЯМИ.

1.1 Огляд систем управління завданнями

1.2. Порівняльний аналіз функціональних можливостей систем управління завданнями

1.3 Проблеми використання стандартних систем у невеликих командах

1.4 Обґрунтування необхідності розробки власної системи

РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ

2.1. Вибір технологій та інструментів для розробки

2.2 Архітектура системи

2.3 Проєктування бази даних та о сновного функціоналу

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБ-ДОДАТКУ

3.1 Налаштування середовища розробки та налаштування проєкту

3.2 Реалізація серверної частини

3.3 Реалізація клієнтської частини

3.4 Тестування системи

7. Календарний план виконання роботи

№ п/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	
		За планом	Фактично
1	2	3	4
1	Вибір теми, підбір літературних джерел, формування структури кваліфікаційної роботи	до 01.11.2024	« 01 » 11 2024 р.
2	I розділ (розкрити сутність і стан дослідження наукової проблематики роботи, після чого навести її загальну характеристику; огляд літератури за темою і вибір конкретних напрямів досліджень)	30.01.2025	« 30 » 01 2025 р.
3	II розділ (аналіз експериментальної частини про методики досліджень)	28.02.2025	« 28 » 02 2025 р.
4	III розділ (проведені теоретичні і (або) експериментальні дослідження та проєктні рішення)	30.04.2025	« 30 » 04 2025 р.
5	Висновки	01.05.2025	« 01 » 05 2025 р.
6	Доопрацювання роботи з урахуванням зауважень наукового керівника	25.05.2025	« 25 » 05 2025 р.
7	Отримання відгуку керівника	25.05.2025	« 26 » 05 2025 р.
8	Отримання зовнішньої рецензії	28.05.2025	« 28 » 05 2025 р.

8. Дата видачі завдання «15» листопада 2024 р.

9. Керівник кваліфікаційної роботи


(підпис)

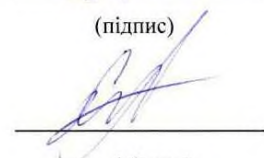
Редич О.В.

10. Керівник освітньо-професійної програми


(підпис)

Омельчук А.

11. Завдання до виконання отримав,
здобувач вищої освіти


(підпис)

Сидорук Д.

АНОТАЦІЯ

“Розробка системи управління завданнями для проєктної команди”

Мета роботи – створення веб-застосунку для управління завданнями в команді з використанням ASP.NET Core MVC та Entity Framework Core.

Розроблена система дозволяє автоматизувати ключові процеси планування, контролю виконання задач і взаємодії між учасниками проєктної команди. У роботі проведено аналіз сучасних СУЗ, обґрунтовано вибір архітектурних рішень, описано етапи розробки, реалізовано функціонал створення та редагування завдань, коментування, призначення ролей і фільтрації. Отримані результати можуть бути використані в ІТ-командах, освітніх або стартап-проєктах. Кваліфікаційна робота містить 66 сторінок, 3 таблиць, 7 рисунків, 1 список використаних джерел з 39 найменувань, 13 додатків.

Ключові слова: управління завданнями, ASP.NET Core, MVC, Entity Framework, автоматизація, веб-додаток.

ANNOTATION

“Development of a task management system for the project team”

The purpose of the work is to create a web application for managing tasks in a team using ASP.NET Core MVC and Entity Framework Core. The developed system allows you to automate the key processes of planning, monitoring the execution of tasks and interaction between the participants of the project team. The work analyzes modern CPS, substantiates the choice of architectural solutions, describes the stages of development, implements the functionality of creating and editing tasks, commenting, assigning roles and filtering. The obtained results can be used in IT teams, educational or startup projects. The qualification paper contains 66 pages, 3 tables, 7 figures, a list of used sources from 39 names, 13 applications.

Keywords: task management, ASP.NET Core, MVC, Entity Framework, automation, web application.

Зміст

ВСТУП.....	3
РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ СИСТЕМ УПРАВЛІННЯ ЗАВДАННЯМИ.....	5
1.1 Огляд систем управління завданнями	5
1.2. Порівняльний аналіз функціональних можливостей систем управління завданнями	8
1.3 Проблеми використання стандартних систем у невеликих командах	20
1.4 Обґрунтування необхідності розробки власної системи	24
РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ.....	27
2.1. Вибір технологій та інструментів для розробки	27
2.2 Архітектура системи	32
2.3 Проєктування бази даних.....	39
2.4 Основний функціонал.....	45
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБ-ДОДАТКУ.....	48
3.1 Налаштування середовища розробки та налаштування проєкту	48
3.2 Реалізація серверної частини	51
3.3 Реалізація клієнтської частини.....	58
3.4 Тестування системи.....	62
ВИСНОВОК	64
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	66
ДОДАТКИ	70

ВСТУП

Актуальність теми. У сучасних умовах цифрової трансформації ефективне управління завданнями у командах розробників програмного забезпечення відіграє ключову роль у забезпеченні продуктивності, організації робочих процесів та досягненні поставлених цілей. Впровадження цифрових рішень дозволяє автоматизувати рутинні процеси, забезпечити прозорість роботи команди та підвищити ефективність виконання проєктів. На даний момент існують десятки різних систем управління завданнями, таких як Trello, Jira, Asana, ClickUp, які пропонують широкий функціонал для організації командної роботи. Проте, більшість таких рішень або мають надлишковий функціонал, або не враховують специфічні вимоги, що виникають під час роботи над ІТ-проєктами.

Тому розробка власної системи управління завданнями з урахуванням сучасних потреб команд розробників, використовуючи технологію .NET, є актуальним завданням. Це дозволить створити гнучкий масштабований та ефективний продукт, який спростить організацію роботи команди, мінімізує зайві дії та забезпечить зручну взаємодію між учасниками проєкту.

Мета дослідження. Метою дослідження є розробка веб-додатку для управління завданнями проєктної команди, який забезпечить автоматизацію процесів планування, контролю виконання та взаємодії між учасниками команди.

Завдання дослідження. Для досягнення поставленої мети необхідно:

- Виконати аналіз існуючих систем управління завданнями, знайти їхні слабкі та сильні сторони.
- Обґрунтувати вибір технологій для розробки веб-додатку.
- Спроектувати архітектуру системи на основі .NET, враховуючи вимоги масштабованості та зручності використання.
- Реалізувати веб-додаток з основним функціоналом:
 - Створення, редагування, призначення завдань;

- контроль статусу виконання;
- система повідомлень та коментарів для покращення командної взаємодії
- Виконати тестування та оцінку ефективності системи.

Об'єктом дослідження є процес управління завданнями у проєктних командах, а також методи та технології розробки веб-додатків для автоматизації вищезазначеного процесу.

Предметом дослідження є інструментарій розробки веб-застосунку для управління завданнями в межах проєктної діяльності, з реалізацією авторизації, ролей, CRUD операцій, фільтрації та коментування, з використанням ASP.NET Core MVC і Entity Framework Core.

Практична значимість розробки. Результатом роботи є мінімально життєздатний веб-застосунок з усіма реалізаціями модулів, який може бути використаний невеликими командами для ефективної організації проєктної роботи. Застосунок підтримує багаторівневий доступ, контроль виконання задач, коментування та тегування, що забезпечує нам покращення командної взаємодії та ведення обліку задач.

Основні положення і результати кваліфікаційної роботи були представлені у тезах доповіді на студентській науково-практичній конференції «ЕКОНОМІКА І БІЗНЕС: НОВІ РЕАЛІЇ, ТЕХНОЛОГІЧНІ ТРЕНДИ, ЦИФРОВІ ІННОВАЦІЇ» 15 травня 2025 р. У роботі було проведено аналіз існуючих рішень, обґрунтування розробки власного рішення, розкрито особливості архітектури веб-застосунку, використані технології та досягнуті результати розробки.

РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ СИСТЕМ УПРАВЛІННЯ ЗАВДАННЯМИ.

1.1 Огляд систем управління завданнями

У сучасних умовах розробки програмного забезпечення та управління проєктами важливою складовою успішно виконаного завдання є ефективне управління робочими процесами. Щоб забезпечити це, були створенні спеціальні програмні рішення – системи управління завданнями. Вони допомагають розподіляти завдання між розробниками і контролювати виконання цих завдань. Зазначені вище системи забезпечують автоматизацію процесів планування, розподілу та моніторингу виконання завдань, що значно впливає на ефективність роботи команди розробників та знижує ризик невчасного виконання проєктів.

При розробці ІТ-продукту виконується багато завдань, які можуть залежати одне від одного або ж просто є взаємопов'язаними і швидкість їх виконання залежить від готовності інших завдань. В такому випадку важливо забезпечити можливість для розробника або будь-якого члена команди, який працює над конкретним завданням, відстежувати прогрес пов'язаних задач. Це дозволяє йому своєчасно отримувати інформацію про виконання критично важливих завдань, які впливають безпосередньо на його роботу, і при необхідності вносити корективи або контактувати з іншими учасниками проєкту і пропонувати свої вирішення. Саме такі можливості і надають сучасні системи управління завданнями. Завдяки створенню та розподілу завдань між учасниками команди визначаються відповідальні особи за кожне конкретне завдання, що сприяє кращій організації роботи та підвищенню відповідальності кожного члена команди. Ці системи надають механізми контролю статусу виконання, що дозволяє відстежувати прогрес виконання завдання. Як правило, завдання можуть мати різні статуси, такі як “В очікуванні”, “В роботі”, “На перевірці” та “Завершено”. Така система маркування є найпоширенішою і вона дає стислу та чітку інформацію про

стан виконання задач, а за необхідності дозволяє оперативно реагувати на зміни [14, 22].

Ще одним важливим механізмом є можливість пріоретизації завдань. Кожне завдання може мати певний рівень важливості та чітко встановлені терміни виконання. Це дозволяє керівникам проєкту або іншим відповідальним особам більш ефективно розподіляти ресурси та уникати перевантаження працівників. Більшість систем управління завданнями надають необхідний для цього функціонал, дозволяючи використовувати автоматичні нагадування та дедлайни, щоб мінімізувати ризики зриву строків.

Спільна робота та комунікація є найважливішим елементом сучасних систем управління завданнями, адже без комунікації буде дуже складно організувати роботу між усіма учасниками команди. Ці системи надають можливість коментування завдань, ведення внутрішніх обговорень, обміну файлами та спільного редагування документів. Завдяки цьому члени команди мають можливість оперативно обмінюватися інформацією, ефективно вирішувати проблеми та підтримувати зв'язок завжди [22].

Додатково сучасні системи підтримують інтеграцію з іншими інструментами, такими як електронна пошта, календарі, месенджери, сервіси для підвищення продуктивності і навіть інтегровані середовища розробки, які надають можливість розробникам вносити корективи у роботу своїх колег та швидко змінювати статуси виконання завдань. Це дозволяє користувачам отримувати сповіщення про зміни у завданнях, планувати зустрічі, а також синхронізовувати робочий процес між різними платформами, що значно підвищує ефективність розробки і продуктивність команди загалом [14].

На ринку існує багато програмних рішень для управління завданнями, які мають свої особливості та переваги. Наприклад, Jira є потужним і найпоширенішим інструментом для управління Agile-проєктами, який широко використовується в IT-сфері. Agile – це метод управління проєктами. Ця система є досить гнучкою і її характерними ознаками є надання кінцевого продукту на кожному етапі роботи та незрозумілий фінал проєкту. Основний

принцип роботи – поділ проєкту на короткі цикли, які ще називають ітераціями, наприкінці кожного з яких замовник отримує певний продукт. Jira дозволяє ефективно працювати з Kanban-дошками і Scrum-дошками, інтегруватися з системами контролю версій, такими як GitHub та Bitbucket, а також використовувати розширені засоби аналітики [4, 27, 38].

Інший популярний інструмент – Trello – вирізняється простотою та гнучкістю. Він базується на Kanban-дошках – це інструмент для спільної роботи над проєктами, коли на одній дошці фіксуються всі завдання проєкту та відповідальні за їх реалізацію. Це дозволяє командам легко організовувати роботи за допомогою карток і списків. Trello особливо корисний для невеликих команд та фрілансерів, яким потрібен простий і зрозумілий інструмент для організації завдань [6].

Для більш комплексного управління завданнями можна використовувати Asana, яка дозволяє організовувати роботу у вигляді списків, таймлайнів або Kanban-дошок. Вона оснащена інструментами для відслідковування продуктивності команди та підтримує інтеграцію із Google Drive, Slack, Zoom та іншими сервісами [3].

Ще однією потужною системою управління проєктами є ClickUp, яка поєднує в собі широкий набір функцій і гнучкість у використанні. Основною перевагою даної системи є велика палітра інструментів для співпраці, наприклад призначення завдань і статусів, сповіщення, призначення статусів не лише завданням, а й коментарям, створення власних статусів тощо. Також ця система підтримує різні методології управління проєктами і дозволяє налаштовувати рівні доступу для різних учасників команди [20].

Останньою системою із найбільш відомих варто розглянути Microsoft Planner. Цю систему обирають компанії, які активно використовують продукти Microsoft, адже ця система інтегрується з Microsoft Teams, Outlook та OneDrive, дозволяючи безперешкодно керувати завданнями без необхідності використання стороннього програмного забезпечення [30, 31].

Це свідчить, що системи управління завданнями є критично важливими для IT-команд, забезпечуючи ефективну організацію командної роботи. Ці системи дозволяють не лише розподіляти завдання, але й вести контроль за їх виконанням, відслідковувати залежності між задачами, налаштовувати комунікацію між членами команди та інтегруватися з іншими інструментами. Вибір конкретної системи залежить від специфіки проєкту, розмірів команди і необхідного функціоналу. Це дозволяє кожному проєкту знайти оптимальне рішення для своїх потреб. В наступному підрозділі буде проведено детальний порівняльний аналіз функціональних можливостей зазначених вище систем управління завданнями.

1.2. Порівняльний аналіз функціональних можливостей систем управління завданнями

Із зростанням популярності систем управління завданнями виникає потреба не лише в загальному огляді таких рішень, але й у ґрунтовному аналізі їхніх функціональних можливостей. Кожен із продуктів, які активно використовуються сучасними командами розробників, надає власний набір інструментів, підходів до організації роботи та ступінь гнучкості у налаштуванні. Вибір тієї чи іншої системи в основному залежить від специфіки проєкту, методології управління, розміру команди та інтеграції або бажання інтегруватися із вже існуючими сервісами, можливість формування звітності, а також масштабованість для подальшого зростання команди або проєкту.

Кожен із популярних інструментів – Jira, Trello, Asana, ClickUp та Microsoft Planner – орієнтований на різні категорії користувачів та типи завдань. Деякі із вищезазначених систем використовують підтримку методологій Agile та Scrum, а інші пропонують саме візуальне планування або базовий трекінг завдань. Враховуючи це, варто провести порівняльний аналіз їх функціональних можливостей, щоб виявити ключові переваги та недоліки, а також зрозуміти, яку систему обрати для конкретних умов використання.

Виконуючи аналіз функціональних можливостей сучасних систем управління завданнями, варто спиратися на набір універсальних критеріїв, що дозволять нам об'єктивно оцінити продукт як підходящий до використання у конкретному середовищі, адже кожна команда має свої індивідуальні потреби, підходи до розробки, специфіку внутрішніх механізмів взаємодії, яка прямо впливає на вимоги до обраного продукту. Саме тому детальний аналіз за чітко визначеними критеріями є гарантією правильного вибору системи, яка дозволить команді досягати поставлених цілей більш ефективно.

Серед необхідних критеріїв варто виділити зручність користування(UI/UX), яка визначає, наскільки швидко нові користувачі адаптуються до системи та зможуть ефективно працювати з нею. Не менш важливою є гнучкість у налаштуванні – можливість адаптувати систему під свої потреби та структуру проєкту. Значну роль відіграє підтримка сучасних методологій управління проєктами, таких як Agile, Scrum або Kanban, що є основою для команд у сфері розробки програмного забезпечення. Окрім цього, варто зазначити наступним критерієм функціонал для спільної роботи, наскільки багато інструментів для цього містить продукт, наприклад коментування завдань, призначення відповідальних за завдання, контроль дедлайнів та використання маркувань або тегів.

Інтеграція з іншими сервісами також відіграє важливу роль у загальній ефективності командної роботи, адже дозволяє створити єдине середовище із ще більш розширеним функціоналом, якщо це потрібно. Такими сервісами можуть бути Google Workspace, Microsoft 365, Slack, Zoom, GitHub та різні інтегровані середовища розробки. Варто зауважити, що можливість створення звітності та аналітики також є важливим критерієм, адже дозволяють візуалізувати прогрес, приймати рішення на основі отриманих даних та виявляти слабкі місця у командній роботі. Також системи управління завданнями можуть працювати на різних платформах, таких як веб, десктоп, мобільні додатки. Цей критерій визначає гнучкість у доступі до інструменту, що є вирішальним для віддалених

команд. Звісно, варто враховувати також ціну – особливо важливою вона є для невеликих команд, стартап проєктів та компаній із малим бюджетом.

Jira.

Jira – це потужний інструмент для управління проєктами та відстеження завдань, який був розроблений компанією Atlassian. Він надає широкий функціонал, який сприяє ефективній організації робочих процесів та підвищенню продуктивності командної розробки.

З основних переваг цього інструменту варто виділити зручність користування(UI/UX): Jira пропонує інтуїтивно зрозумілий інтерфейс із можливістю налаштування під конкретні потреби команди. Користувачі можуть створювати та налаштовувати інформаційні панелі, відображати завдання у вигляді списків, канбан-дошок або таймлайнів. Це забезпечує зручність у відстеженні процесу виконання роботи [Див. рис. 1.1].

Варто також виділити гнучкість у налаштуванні, як було згадано вище: інструмент дозволяє налаштовувати робочі процеси, статуси завдань, поля та дозволи відповідно до конкретних вимог проєкту чи власне команди. Саме це забезпечує гнучкість та можливість адаптації системи до різних методологій управління та специфіки роботи.

Jira була розроблена спеціально для підтримки Agile-методологій, включаючи Scrum і Kanban. Вона надає інструменти для планування спринтів – це короткий фіксований період часу, за який команда повинна виконати певний обсяг роботи, зазвичай це завершити частину продукту, тобто додати новий функціонал. Зазвичай він триває від 1 до 4 тижнів і має чітко визначену мету та список завдань, який формується на початку, тобто на командних зібраннях.

Також Jira надає інструменти для створення беклогів та управління історіями користувачів, що сприяє ефективному впровадженню гнучких підходів у розробці. Варто згадати ще раз беклог – це впорядкований список усіх завдань,

функцій або змін, які необхідно реалізувати в межах проєкту. Беклог є основою для планування спринтів і визначенні пріоритетів у командній роботі [37].

Як і у всіх інших системах управління завданнями, Jira забезпечує можливість командної розробки, яка супроводжується централізованою платформою для співпраці членів команди, дозволяючи коментувати завдання, прикріплювати файли, відстежувати історію змін та отримувати сповіщення про зміни у проєкті. Одним словом, це сприяє прозорості та ефективності комунікації у команді [Див. рис. 1.1], [14].

Даний інструмент підтримує інтеграцію з великою кількістю сторонніх додатків та сервісів завдяки Atlassian Marketplace, включаючи такі популярні інструменти як Slack, Microsoft Teams, GitHub, GitLab та багато інших. Це забезпечує об'єднання різні частини робочих процесів у єдиному середовищі [38].

Jira надає ефективні інструменти для створення звітів та аналітики, що дозволяють відстежувати продуктивність команди, аналізувати прогрес проєкту та виявляти вузькі місця в процесах. Основними інструментами є створення діаграм для аналізу даних та звіти, які можна налаштувати на свій розсуд [Див. рис. 1.1], [27, 38].

Даний інструмент доступний через веб-інтерфейс, а також дозволяє завантажити мобільні додатки для IOS та Android, які нічим не поступаються веб-версії, що забезпечує зручний доступ до системи з будь-якого пристрою [Див. рис. 1.1], [27].

Jira має декілька тарифних планів, серед яких безкоштовний, який дозволяє створити команду до 10 користувачів, проте має обмежений функціонал, стандартний за вартістю \$7.50 за користувача на місяць, що включає розширені функції та підтримку, преміум за вартістю \$17.00 за користувача на місяць, який надає необмежене зберігання даних, документ, який гарантує, що сервіс буде працювати з певною надійністю та швидкістю (Service Level

Agreement), а також Enterprise-план, який надає індивідуальні умови для великих організацій [14].

Отже, Jira є досить потужною та гнучкою системою управління завданнями, що особливо добре підходить для середніх команд розробників, які працюють за методологією Agile. Завдяки вищезазначеному інструментарію Jira забезпечує високий рівень адаптації до специфіки будь-якого проєкту.

Нижче показано інтерфейс даної системи управління завданнями:

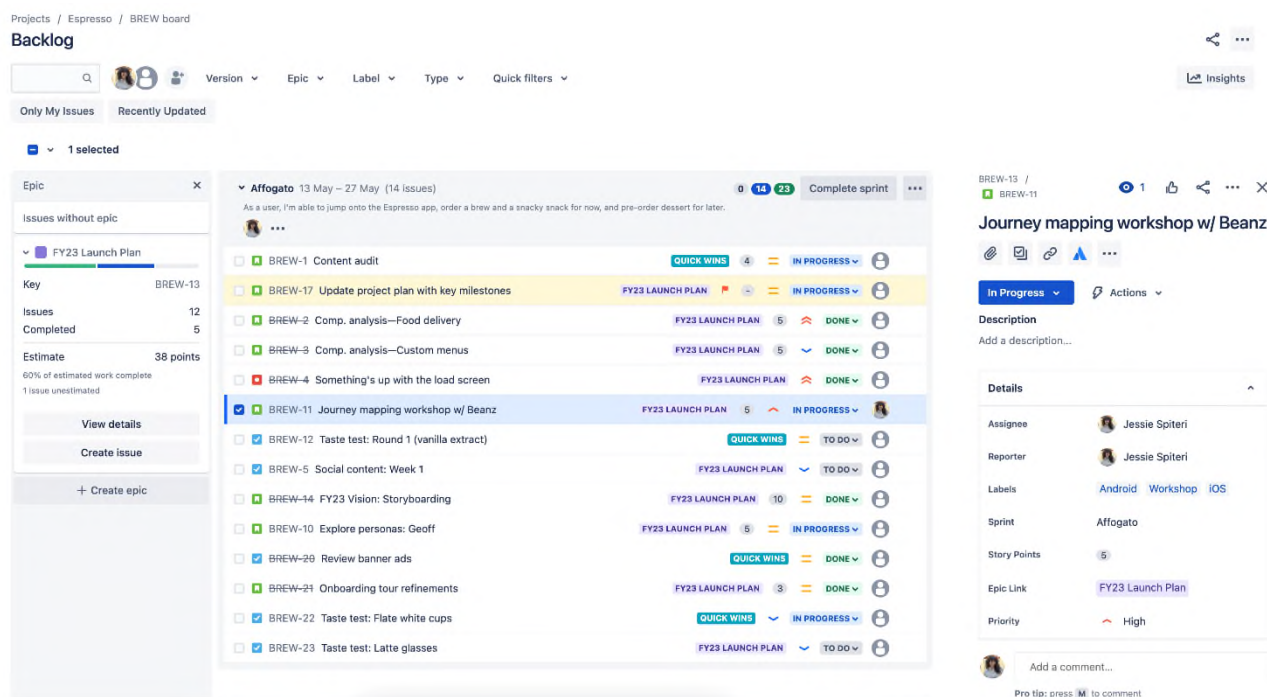


Рис. 1.1 Інтерфейс Jira

Trello.

Однією з найвідоміших та найзручніших систем управління завданнями є Trello – легка у використанні система, яка базується на візуальній моделі канбан-дошок. Цей інструмент спочатку був орієнтований на невеликі команди або індивідуальних користувачів, але з зі зростанням популярності сервіс переріс у повноцінну платформу для організації командної роботи, планування проєктів та відстеження і контролю особистої продуктивності. Його простота поєднується із гнучкістю, що дозволяє адаптувати Trello під різні сценарії використання – від

ведення особистого to-do-списку до управління робочими процесами стартапу чи креативної команди [3].

Завдяки зрозумілому інтерфейсу, великій кількості інтеграцій та швидкій адаптації нових користувачів Trello набув великої популярності серед команд, які шукають мінімалістичне, але ефективне рішення для співпраці. Розробником даної системи є також компанія Atlassian, як і у Jira, проте ці системи зовсім різні, незважаючи на те як здається на перший погляд. Trello націлений на більш широке коло користувачів, включаючи маркетингологів, HR-фахівців, менеджерів та для особистого використання на відміну від Jira, яка орієнтована саме на розробників. У цій частині буде детально розглянуто функціональні можливості Trello, його переваги та недоліки, типові сценарії використання, а також оцінено доцільність його впровадження для різних команд [3, 9, 14].

Система управління завданнями Trello має широке коло користувачів завдяки своїй канбан-моделі, яка дозволяє організовувати робочі процеси у вигляді карток, списків і дошок, що робить її зручною для усіх типів користувачів – від студентів до корпоративних команд.

Як і з Jira, варто виділити зручність користування системою Trello: вона має інтуїтивно зрозумілий інтерфейс, що дозволяє новим користувачам швидко адаптуватися до роботи з платформою. Користувачі мають можливість легко створювати дошки для проєктів, додавати списки для етапів роботи та картки для окремих завдань. Картки можна переміщувати між списками за допомогою простого перетягування, що забезпечує візуально приємне відображення прогресу завдань. До кожної картки можна додати опис, чек-листи, терміни виконання, коментарі та мітки, що сприяє детальному управлінню завданнями [3, 28, 38].

Trello надає можливість налаштування дошок відповідно до власних потреб, потреб команди або проєкту. Користувачі можуть створювати власні шаблони дошок, використовувати мітки для розбиття завдань по категоріях, а

також налаштовувати права доступу. Головною перевагою можна виділити автоматизацію повторюваних дій за допомогою функції Butler, наприклад переміщення карток, встановлення термінів або надсилання сповіщень. Це дозволяє не витрачати часу на повторювані дії і підвищити ефективність роботи команди [28, 38].

Хоча дана система управління завданнями не є спеціалізованим інструментом для Agile або Scrum, її гнучкість дозволяє адаптувати платформу під вищезгадані методології. Користувачі мають можливість створювати дошки для спринтів, використовувати списки для відображення етапів виконання завдання(наприклад To Do, In Progress, Done) та додавати картки для окремих завдань. Також є можливість встановити додаткові розширення, щоб інтегрувати функціональність для відстеження спринтів, створення беклогів або ведення аналізу [28].

В контексті командної роботи теж можна виділити декілька переваг для даної системи. Trello дозволяє учасникам команди працювати над завданнями в реальному часі, призначати завдання конкретним членам команди, залишати коментарі, додавати вкладення на отримувати сповіщення. Це дозволяє забезпечити прозорість робочих процесів та сприяти кращій координації між учасниками проєкту.

Trello підтримує інтеграцію із сервісами, такими як Slack, Google Drive, Dropbox, Microsoft Teams та багатьма іншими, що дозволяє об'єднати всі ці інструменти в одну систему, яка використовується командою. Також за допомогою надбудов можна додати функціональність створення звітів та діаграм, наприклад Screenful та Time in List.

Система є доступною через мобільні додатки або веб-інтерфейс, що забезпечує зручний доступ до проєктів з будь-якого пристрою. Завдяки цьому користувачі завжди залишаються на зв'язку та керують завданнями незалежно від місця перебування.

Щодо вартості, існує 4 тарифні плани, тому користувачі зможуть обрати найбільш оптимальний для себе, серед яких безкоштовний план, що надає основний функціонал для індивідуальних користувачів, стандартний план за вартістю \$5 на місяць при щорічній оплаті, що включає необмежену кількість дошок, розширені чек листи та інші додаткові функції для невеликих команд, преміум-план за вартістю \$10 на місяць, який додає функції перегляду у вигляді календаря, діаграм Ганта, а також розширені можливості автоматизації та план для організацій за вартістю \$17.50 за користувача на місяць, який створений спеціально для компаній, з додатковими вимогами до безпеки та адміністрування.

Попри численні переваги, Trello має низку обмежень, які слід враховувати при виборі інструменту для командної роботи:

- Обмежений функціонал у базовій версії – безкоштовний план має суттєві обмеження щодо кількості надбудов, автоматизацій та переглядів, що може бути критичним для зростаючих команд.
- Відсутність розширених функцій звітності – для отримання можливості проводити аналіз потрібні сторонні розширення або інтеграції, що ускладнює моніторинг продуктивності без додаткових налаштувань [24].
- Менш придатний для великих технічних проєктів – на відміну від Jira, Trello не підтримує складні структури завдань, що обмежує його ефективність у команді розробників [24].
- На відміну від Jira, тут складно створювати гнучкі робочі простори або ролі з глибоким рівнем доступу [24].

Загалом, Trello надзвичайно зручна система управління завданнями, яка чудово підходить для невеликих команд або особистого використання, втім для більш складних проєктів або команд, функціоналу цієї системи може бути недостатньо, адже зазвичай команди працюють за суворими методологіями розробки, а для реалізації цього в Trello може знадобитися час.

Asana

Asana – це хмарна платформа для управління проєктами та завданнями, розроблена для покращення командної взаємодії та підвищення продуктивності. Ця система дозволяє командам швидко та ефективно організовувати робочі процеси, відстежувати прогрес та досягати цілей ефективніше. Систему виділяють як одну із кращих у проєктному менеджменті, адже вона має велику кількість переваг і невелику кількість недоліків [10, 21].

Основними перевагами даної платформи є інтуїтивно зрозумілий інтерфейс, що дозволяє новачкам швидко адаптовуватися та гнучкість у налаштуванні завдяки можливості створення власних робочих процесів, використання шаблонів, налаштувань полів та автоматизації завдань за допомогою правил [10, 21].

Asana підтримує методології управління проєктами, як і більшість інших систем управління завданнями, включаючи Scrum і Kanban, що дозволяє адаптувати платформу під потреби команди або проєкту. Інтеграція з іншими сервісами, такі як Zoom, Gmail, Slack також присутня, завдяки чому можна створити єдину систему для роботи [9, 12, 21].

Ця система також підтримує створення діаграм та аналіз роботи команди, що дозволяє в режимі реального часу оцінювати, наскільки швидко та ефективно виконуються завдання на проєкті. Також основною перевагою є розділення цілей на ключові результати, що полегшує кількісне вимірювання успіху. Ця функція є дуже корисною при впровадженні цілей і ключових результатів або подібних рамок встановлення цілей в організації [12, 21].

Asana підтримує декілька тарифних планів, як і вище описані системи управління завданнями, такі як базовий, преміум, бізнес і план для підприємства за ціною \$0, \$10.99, \$24.99 та індивідуальна ціна відповідно. Для стартапів та маленьких компаній може бути достатньо базового або преміум-плану, але для великих організацій варто обрати план для підприємства, адже він пропонує

більшу безпеку та гнучкість, необмежену кількість інформаційних панелей для звітності, спеціальну цілодобову підтримку і багато інших привілеїв [29].

В порівнянні з іншими системами Asana поступається ним завдяки обмеженням безкоштовного плану, якого досить часто може бути недостатньо для малих організацій або складних проєктів та вартості тарифних планів, які можуть бути дорогими для малих підприємств [38].

Отже, Asana це потужний інструмент для управління проєктами, який підходить для команд середнього розміру із серйозними проєктами. Завдяки великому набору функцій, інтеграціям та гнучкості в налаштуванні, ця система допомагає командам ефективно організовувати роботу на проєкті та швидко досягати поставлених цілей. Однак, перед вибором цієї системи управління проєктами варто враховувати обмеження та ціну тарифних планів, навіть для великих організацій.

ClickUp

ClickUp – це універсальна платформа для управління завданнями та проєктами, яка об'єднує інструменти для планування, співпраці та аналітики в одну систему. Ця платформа дозволяє командам будь-якого розміру ефективно організовувати роботу в команді та підвищувати продуктивність. Основними перевагами даної системи є гнучка ієрархія, а саме можливість створювати по черзі простір, папки, списки, завдання і підзавдання, що дозволяє створити зручну структуру під потреби будь якого проєкту. Ця система підтримує канбан-дошки, календарі, діаграми Ганта, списки та інформаційні панелі для візуалізації процесів. У ClickUp відразу вбудовані чати, документи, цілі, білі дошки та автоматизації, що відразу створює цілу систему. Ця платформа має безкоштовний тарифний план із необмеженою кількістю користувачів та завдань, що є вирішальним фактором при виборі системи управління завданнями для малих команд та стартапів. Проте, існують також недоліки, попри усі переваги, такі як зниження швидкості роботи інтерфейсу при великій кількості даних у великих

проектах, що може створити незручності згодом, коли проєкт буде зростати ще більше і обмеження безкоштовного плану, такі як розширені автоматизації та аналітика, доступні лише в платних тарифах [3, 14, 20].

Загалом, дана система управління завданнями є хорошим вибором для малих команд, хоча кількістю функціоналу вона поступається усім попереднім, які пропонують більшу кількість інтеграцій та функцій для налагодження командної роботи.

Microsoft Planner

Microsoft Planner – це інтуїтивно зрозуміла система для управління завданнями, яка інтегрована у пакет Microsoft 365 і націлена на аудиторію, котра вже працює із екосистемою Microsoft. Дана система дозволяє командам створювати плани, призначати завдання, відстежувати прогрес та співпрацювати у реальному часі. Планер підтримує інтеграцію із абсолютно усіма сервісами Microsoft такими як Teams, Outlook, OneDrive та інші, організовує завдання у вигляді карток на канбан-дошках, дозволяє спільно працювати над завданнями, додавати коментарі, прикріплювати файли та відстежувати прогрес, а також доступний на усіх пристроях, включаючи IOS [30, 31].

На жаль, дана система не підтримує розширене управління ресурсами, бюджетування або детальне планування залежностей, що для великих проєктів може не підходити. Порівнюючи з іншими системами управління завданнями, планер має обмежений функціонал для створення звітів та аналітики, а також має сильну залежність від екосистеми Microsoft, що може бути недоцільним для команд, які не користуються продуктами Microsoft [30].

Усі розглянуті системи мають спільну мету – спростити управління завданнями, покращити командну взаємодію та забезпечити контроль розробки програмного продукту. Вони всі підтримують базовий функціонал, такий як створення задач, візуалізація процесів, коментарі та інтеграції із іншими сервісами.

Водночас, між системами існують відмінності. Наприклад, Jira та Asana орієнтовані на середні або великі команди і серйозні проекти, в той час як ClickUp, Microsoft Planner та Trello орієнтовані на стартапи та малі команди. При виборі однієї з цих систем варто аналізувати свої потреби, бюджет та рівень складності. Жодне рішення не буде ідеальним для кожного, що і створює умови для опису та розробки власної системи, адаптованої під конкретні вимоги. Нижче наведено порівняльну таблицю, яка узагальнює в собі аналіз проведений вище:

Таблиця 1.2

Порівняльна таблиця систем управління завданнями

Критерій	Jira	Trello	Asana	ClickUp	Microsoft Planner
Зручність користування (UI/UX)	Складний інтерфейс для новачків	Дуже простий і візуальний	Інтуїтивно зрозумілий	Гнучкий, але насичений інтерфейс	Мінімалістичний, простий
Гнучкість налаштування	Висока	Обмежена	Середня	Дуже висока	Обмежена
Підтримка методологій	Scrum, Kanban, Agile	Kanban	Scrum, Kanban, Timeline	Scrum, Kanban, Waterfall	Лише Kanban
Командна робота	Розширена: коментарі, статуси, ролі	Коментарі, чек-листи	Розвинена співпраця	Все включено: чати, коментарі,	Базові можливості

Критерій	Jira	Trello	Asana	ClickUp	Microsoft Planner
				документація	
Інтеграції	GitHub, Bitbucket, Slack тощо	Slack, Google Drive, Power-Ups	Slack, Zoom, Google, Microsoft	Більше 1000 інтеграцій	MS Teams, Outlook, OneDrive
Звіти та аналітика	Потужна система звітів	Обмежено	Наявні таймлайни, графіки	Розширена аналітика	Мінімальна
Платформа	Web, мобільний	Web, мобільний	Web, мобільний	Web, мобільний	Web, мобільний
Вартість	Безкоштовно до 10 осіб, далі від \$8.60	Безкоштовно, з доплатами за Power-Ups	Безкоштовно, платні плани від \$10	Безкоштовно, платні — з \$7	Входить у Microsoft 365
Найкраще підходить для:	Середні та великі технічні команди	Невеликі проекти, фрілансери	Універсальне рішення для малих та середніх бізнесів	Невеликі команди, які хочуть «все в одному»	Організації в екосистемі Microsoft

1.3 Проблеми використання стандартних систем у невеликих командах

Хоча сучасні системи управління завданнями пропонують широкий функціонал і підтримують гнучкі методології, вони не завжди є хорошим вибором для невеликих команд. Більшість таких систем розроблялися саме для

середніх та великих організацій, що призводить до низки незручностей при використанні даних систем у менш масштабних проєктах.

Малі команди зазвичай працюють у більш динамічному середовищі, мають обмежені ресурси та потребують швидких рішень для коректної організації своєї роботи. У цьому контексті використання вищеописаних систем із надмірною кількістю налаштувань, складною структурою та платними функціями може лише ускладнити процес управління командою та розробки. У даному підрозділі буде розглянуто основні проблеми, з якими стикаються невеликі команди під час використання стандартних продуктів.

Одна з головних проблем, з якою стикаються малі команди використовуючи системи управління завданнями – це перевантажений інтерфейс та надлишковий функціонал, який ускладнює взаємодію між учасниками та їх адаптацію до нової системи. Багато популярних рішень, таких як Jira, ClickUp чи Asana, мають десятки модулів, типів завдань, статусів і рівнів доступу. У невеликих командах, де швидкість і простота мають вищий пріоритет, така велика кількість функціоналу лише заважає використовувати продукт ефективно.

Наприклад, замість того, щоб швидко створити завдання і призначити його відповідальній особі, користувач змушений витратити час на навігацію по складному меню, заповнювати неважливі, але необхідні поля, вибирати статуси, пріоритети і зв'язки між задачами. Простіше кажучи, вся ця рутинна перетворюється на окрему велику задачу, яка знижує продуктивність учасників команди.

Ще однією типовою проблемою для невеликих команд є складність адаптації до нової системи управління завданнями. Навіть якщо обрана платформа має потужний функціонал і теоретично підходить усім учасникам команди, то на практиці її повноцінне впровадження часто потребує часу, зусиль і навчання персоналу. Особливо критично це тоді, коли в команді немає

технічного спеціаліста, який взяв би на себе роль адміністратора системи або відсутній спеціальний менеджер проекту.

Такі системи, як Jira та ClickUp мають детальну документацію, відеоінструкції та спільноти користувачів. Проте, сам факт необхідності вивчати матеріали може стати бар'єром для багатьох команд, адже у невеликих командах та стартапах час має значення і треба розпочинати роботу якнайшвидше.

Також варто враховувати, що в малих командах часто працюють фахівці різного профілю – маркетологи, дизайнери, аналітики, які можуть не мати попереднього досвіду з подібними інструментами. Якщо інтерфейс незрозумілий, а термінологія перевантажена технічними поняттями, це спричиняє плутанину та знижує мотивацію користуватися системою.

Це може призвести до того, що команда починає комбінувати кілька різних систем управління завданнями або повністю ігнорувати обране програмне забезпечення, що призводить до фрагментації даних, втрати прозорості і зниження продуктивності.

Для невеликих команд з обмеженим бюджетом вартість використання популярних систем управління завданнями часто є критичним фактором, адже більшість продуктів, таких як Jira, Asana та ClickUp, мають безкоштовні тарифи, в яких суттєво знижений функціонал, чого може бути недостатньо. Повноцінне використання системи – з можливістю створювати розширені звіти, автоматизувати процеси, обмежувати доступи та інтегрувати зовнішні сервіси – зазвичай потребує зміни тарифного плану на платний.

У випадку навіть невеликої команди, яка складається з 5-10 осіб це може перетворитися на критичну щомісячну витрату, особливо для стартапів, студентських проєктів або тимчасових ініціатив. Такі витрати не завжди є виправданими, особливо якщо команда не використовує повний набір функцій платформи.

В такому випадку, малі команди змушені користуватися обмеженим функціоналом в безкоштовному плані або взагалі відмовлятися від платформи, що ще раз підтверджує необхідність у доступному й адаптивному рішенні.

Ще одна проблема, з якою стикаються малі команди – це складність або обмеження в налаштуванні системи під свої потреби. У деяких платформах, таких як Microsoft Planner, функціонал є досить фіксованим і не дозволяє гнучко змінювати структуру завдань, статусів або ролей користувачів. Водночас, у більш потужних системах, таких як Jira та ClickUp, кастомізація можлива, але вона потребує додаткових знань, часу і часто – адміністратора, що не завжди доцільно для невеликого колективу. Малі команди, які прагнуть швидкого старту без складних налаштувань, часто змушені або працювати із шаблонами, які їм не підходять, або витратити зусилля на адаптацію системи, замість зосередження на основних завданнях.

Відсутність гнучкості та простоти у налаштуванні – це ще один аргумент на користь створення власного рішення, яке відповідає конкретним вимогам команди без надлишкової складності.

Частина сучасних систем управління завданнями тісно інтегровані з певними екосистемами – наприклад, Microsoft Planner повністю прив'язаний до Microsoft 365, а Jira активно інтегрується з Atlassian Cloud. Для великих компаній це часто є перевагою, однак для малих команд така залежність може обмежувати гнучкість.

У разі якщо команда не використовує весь пакет сервісів, система може виявитися надмірною або навіть недоступною без платної підписки на супутні продукти. Крім того, багато платформ реалізують частину ключового функціоналу через доповнення, які часто теж є платними.

Для невеликих команд це часто створює додаткові бар'єри – як фінансові, так і технічні. Замість універсального інструменту розробники отримують

конструктор, який потрібно зібрати самостійно з модулів, залежних від інших сервісів, що ускладнює використання і підтримку.

І останньою проблемою використання вищеописаних систем є низька ефективність при простих задачах. У малих командах завдання часто мають короткий життєвий цикл: їх потрібно швидко створити, призначити та закрити. Для таких випадків великі системи управління завданнями іноді виявляються “важкими” – час, витрачений на створення задачі, встановлення пріоритетів, додавання міток та зв’язків, перевищує час її фактичного виконання.

У таких випадках замість підвищення ефективності все відбувається навпаки – платформа починає ускладнювати та гальмувати роботу, замість того щоб спростувати її. Це особливо помітно, коли завдань небагато, а структура проєкту – максимально проста. Часто такі команди або мінімізують використання інструментів, або шукають альтернативу у вигляді легких рішень.

Аналіз використання стандартних систем управління завданнями у невеликих командах показав, що попри широкий функціонал, ці інструменти часто є неефективними для малих проєктів. Перевантаженість інтерфейсу, складність адаптації, висока вартість, обмеження налаштування та прив’язаність до екосистем створюють додаткові труднощі, які не відповідають динаміці створення проєктів у невеликих колективах.

Ці фактори свідчать про актуальність створення власної системи управління завданнями, яка буде адаптована до конкретних вимог команди, простою у використанні та функціонально достатньою для ефективного управління процесами без надлишкових ускладнень.

1.4 Обґрунтування необхідності розробки власної системи

На основі проведеного аналізу можна дійти висновку, що існуючі системи управління завданнями хоч і пропонують широкий набір функціоналу та підтримку сучасних підходів до проєктного менеджменту, однак не завжди задовольняють усім потребам невеликих команд розробників. У багатьох

випадках ці системи є занадто складними, важкими в освоєнні, дорогими або недостатньо гнучкими для підлаштування під конкретні умови роботи. Саме завдяки цьому виникає потреба у створенні власного рішення, орієнтованого на простоту, ефективність та конкретний контекст команди. Розробка своєї системи управління завданнями дозволяє уникнути надмірного функціоналу, зосередившись лише на необхідному – створенні завдань, призначенні виконавців, зміні статусів, додаванні коментарів та отриманні сповіщень. Це дозволяє побудувати інтерфейс, орієнтований на кінцевого користувача, без перевантаження зайвими полями, меню та складними налаштуваннями. Крім того, власна система забезпечує контроль над логікою її роботи – з можливістю гнучко налаштовувати статуси задач, шаблони, рівні доступу та структуру даних. Ще однією перевагою є можливість інтеграції лише з тими сервісами, які дійсно використовуються в команді, наприклад, внутрішній чат, поштові сервіси або календарі. І нарешті, власна розробка дозволяє уникнути зайвих витрат на ліцензії, оскільки така система не потребує оплати за підписки чи обмеження за кількістю користувачів.

Крім того, реалізація проєкту на сучасних платформах дозволяє забезпечити високу продуктивність, масштабованість, розширюваність і безпеку додатку, що робить його набагато ціннішим і хорошим рішенням навіть у перспективі зростання команди і функціоналу.

Також важливо, що створення такої системи може враховувати внутрішню специфіку проєктної діяльності: наприклад, спосіб формування задач, особливості командної взаємодії, типові сценарії використання. У стандартних рішеннях подібні нюанси або взагалі не передбачені, або реалізовані через складні механізми налаштування або розширення.

Таким чином, створення власного веб-застосунку для управління завданнями не лише виправдане, але й є логічною відповіддю на ті проблеми, з якими стикаються невеликі команди у своїй щоденній роботі. Такий підхід

дозволяє досягти оптимального балансу між функціональністю, зручністю та ефективністю.

РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ.

Після аналізу існуючих систем управління завданнями, їхніх переваг та недоліків, а також виявлення актуальних проблем у використанні схожого програмного забезпечення у малих командах виникає необхідність створити власну систему, яка буде відповідати конкретним вимогам, буде простою у використанні, масштабованою та достатньо функціональною для ефективної командної роботи.

Проектування системи є важливим етапом, який включає вибір архітектурного підходу, визначення технологій, які будуть використані у створенні додатку, структуру бази даних, основні програмні модулі та різні сценарії взаємодії користувачів із продуктом. Правильний вибір інструментів та архітектурних рішень є запорукою стабільності, надійності та подальшої підтримки веб-додатку.

У даному розділі буде описано процес формування технічного бачення системи, аргументовано вибір технологій, описано загальну архітектуру, структуру бази даних і перелік основних функцій, які реалізуються у рамках даної роботи.

2.1. Вибір технологій та інструментів для розробки

Вибір технологій для розробки веб-додатку є найважливішим етапом проектування, адже саме завдяки технологіям які ми використовуємо визначається не лише зручність розробки, але й стабільність, продуктивність та масштабованість системи. Враховуючи вимоги до системи управління завданнями для малої проєктної команди, основними критеріями при виборі технологій стали зручність розробки, зрозуміла структура, надійна база даних, підтримка авторизації та автентифікації, гнучкий інтерфейс і простота розгортання додатку.

Система буде реалізовувати найпоширенішу клієнт-серверну архітектуру, в якій сервер є обробником запитів, виконувачем бізнес-логіки та взаємодіє з

базою даних, а клієнт відповідає за відображення інформації та взаємодію з користувачем, в нашому випадку клієнтом може бути браузер. З огляду на вищезазначені вимоги було обрано перевірений та зростаючий за популярністю набір технологій на базі платформи .NET, зокрема ASP.NET Core MVC, Entity Framework Core, Microsoft SQL Server, а також для розробки сторони клієнта Razor Pages та Bootstrap.

Ще одним із варіантів розглядалося створення веб-додатку використовуючи мову програмування Java, а також пов'язаний з нею стек технологій під назвою Spring Boot. Spring Boot – це розширення фреймворку Spring для Java, створене для швидкого запуску та розробки додатків. Він автоматизує налаштування програми, пропонує вбудовані сервери, наприклад Tomcat та Jetty, інтеграцію із базами даних, безпекою, REST API та іншими цікавими доповненнями. Його перевагою є те, що він об'єднує в собі багато ключових фреймворків Spring екосистеми, завдяки чому стає можливим створення додатків використовуючи схожі між собою інструменти. Spring Boot об'єднує в собі такі фреймворки, які б знадобилися при створенні системи управління завданнями:

Spring Core – базовий модуль, який є базовим в даній екосистемі. Він включає механізми інверсії керування та ін'єкції залежностей.

Spring MVC – фреймворк, без якого стає неможливим створення веб-застосунку. Він використовує шаблон Model-View-Controller, що дозволяє поділити застосунок на різні шари, кожен з яких відповідає за конкретну задачу. Spring Boot інтегрує його для побудови RESTful API та класичних веб-додатків.

Spring Data JPA – є абстракцією над JPA. JPA – це інструмент, який використовує Java Persistence Query Language, схожу на SQL мову яка дозволяє використовувати об'єктну модель Java, а не SQL таблиці, тобто працювати із таблицями в об'єктно-орієнтованому стилі. Spring Boot підключає Spring Data до будь-якої підтримуваної системи управління базами даних (СУБД).

Spring Security – модуль для забезпечення авторизації та аутентифікації. Даний фреймворк дозволяє швидко додати безпеку до REST API або веб-застосунку – включно із OAuth2, JWT, базовою авторизацією та базовим логіном.

Ці модулі забезпечили б нам створення системи управління завданнями, проте дана екосистема вирізняється складністю у налаштуванні, а це не потрібно для невеликого веб-застосунку, такого як власна система управління завданнями.

З огляду на проблему, вказано вище, було обрано екосистему .NET – сучасну платформу від компанії Microsoft, яка дозволяє створювати веб, настільні, мобільні, хмарні та інші типи додатків. У контексті створення веб-застосунку більш привабливими є засоби .NET, без потреби в сторонніх технологіях. Серед переваг варто виділити єдність середовища розробки, швидкість створення продукту, багату стандартну бібліотеку, офіційну підтримку, простоту розгортання, а також інтеграцію із хмарними сервісами та системами безпеки.

Розробку застосунку можна поділити на використання уніфікованих інструментів, які чудово взаємодіють між собою, зокрема такі:

ASP.NET Core MVC – фреймворк для побудови веб-додатків, який також реалізує шаблон Model-View-Controller. Він також дозволяє розділити логіку, подання та обробку запитів, що особливо важливо при зростанні масштабу проєкту. Також є фреймворк ASP.NET, проте попередній підтримує різні платформи, а не лише Windows, має високу продуктивність та модульність [36].

Entity Framework Core – фреймворк, який забезпечує взаємодію з базами даних у вигляді об'єктної моделі. Подібно до Spring Data JPA, EF Core підтримує запити на основі Language Integrated Query(LINQ), механізми міграцій бази даних, ліниве завантаження(тобто завантаження даних за потреби, а не під час запуску додатку), зміни відстеження тощо. Завдяки підходу Code First база даних створюється на основі класів, які були описані мовою C# [11].

Identity Framework / ASP.NET Identity – механізм аутентифікації та авторизації користувачів, який забезпечує все те ж саме що і Spring Security, але більш простий у налаштуванні [33].

Razor Pages / Razor Views – технологія для створення інтерфейсу користувача безпосередньо на сервері, яка дозволяє змішувати HTML і C# у єдиному шаблоні. Це робить розробку швидкою, зручною та наочною, без потреби у складних front-end фреймворках. За використання Java ми б використовували Thymeleaf – двигун шаблонів, який використовував би HTML у зв'язку із Java, але цього було б недостатньо для створення необхідного інтерфейсу користувача [19].

Також варто зауважити, що ASP.NET Core має вбудовану ін'єкцію залежностей, а також механізми для логування, конфігурації та розгортання додатку. Також платформа надає широкі можливості для тестування, такі як бібліотека xUnit та NUnit, локалізації та є повноцінною основою для професійного розробницького циклу. У контексті даного проєкту, .NET дозволяє не лише реалізувати функціональність системи управління завданнями, а й зробити це у зрозумілому, зручному та підтримуваному вигляді, що особливо важливо для невеликої команди розробників.

Таким чином, вибір цієї платформи є обґрунтованим через її баланс між функціональністю, швидкістю розробки, зрозумілою архітектурою та підтримкою повного циклу життєдіяльності веб-додатку – від створення до розгортання.

Нижче наведено порівняльну таблицю двох платформ для розробки додатків, щоб узагальнити все вищесказане.

Таблиця 2.1

Порівняльна таблиця Spring Boot та .NET

Критерій	.NET (ASP.NET Core)	Spring Boot (Java)
Платформа	Кросплатформенна (.NET Core працює на Windows, Linux, macOS)	Кросплатформенна (завдяки Java Virtual Machine)
Архітектура	MVC, Razor Pages, Blazor, підтримка REST API, SignalR	MVC, REST API, підтримка microservices через Spring Cloud
Налаштування	Просте, мінімум конфігурацій "з коробки"	Гнучке, але потребує більше конфігурації
Основні фреймворки	ASP.NET Core MVC, Entity Framework Core, Identity	Spring MVC, Spring Data JPA, Spring Security
Object Relational Mapping	Entity Framework Core	Spring Data JPA (на основі Hibernate)
Безпека	ASP.NET Identity, підтримка OAuth2, JWT	Spring Security, підтримка OAuth2, JWT, LDAP
Тестування	xUnit, NUnit, MSTest	JUnit, Mockito, Testcontainers
Модульність	ASP.NET Core модульний за замовчуванням	Spring Boot надає auto-configuration, легко розширюється
Продуктивність	Висока (з .NET Core 6+ показники на рівні з Node.js)	Висока, залежно від конфігурації JVM
Застосування в індустрії	Чудово підходить для бізнес-додатків, CRM, ERP	Часто використовується у fintech, e-commerce, cloud-рішеннях

2.2 Архітектура системи

Архітектура є критичним аспектом проєктування будь-якого програмного продукту. Вона визначає структуру застосунку, організацію усіх компонентів, способи взаємодії між ними, а також забезпечує масштабованість, надійність і можливість підтримувати код протягом довгого періоду часу. Особливо це важливо для систем, що керують життєвим циклом завдань у команді – таких як системи управління завданнями [1].

Якщо спробувати дати визначення даному поняттю, то архітектура програмного забезпечення – це набір структур, необхідних для обґрунтування проєктних рішень. В даному підрозділі буде розглянуто декілька найпоширеніших архітектурних стилів, проаналізовано їхні переваги та недоліки, а також обґрунтовано вибір тієї архітектури, яка буде використовуватися у рамках даного дипломного проєкту [1].

У рамках даної роботи архітектура системи має відповідати кільком ключовим вимогам:

- Простота реалізації та підтримки.
- Чітке розділення логіки, даних і представлення.
- Можливість розширення в майбутньому (наприклад, мобільний додаток, API).
- Зручність у тестуванні окремих компонентів.
- Підтримка ролей та рівнів доступу.
- Безпечне оброблення запитів та робота з користувацькими даними.

Перед початком розробки було загострено увагу на таких підходах:

1. Монолітна архітектура – найпростіший з архітектурних стилів, у якому вся логіка програми реалізується в межах лише одного застосунку. Наприклад, уся логіка системи, база даних, обробка запитів та інтерфейс реалізовані в межах

одного модуля. Такий підхід є найпростішим у реалізації, проте з часом він призводить до так званого “спагеті-коду”, а також ускладнює масштабування [8].

Окрему увагу варто виділити терміну “спагеті-код”, який є дуже доречним в контексті даного архітектурного підходу. Даний термін описує погано організований, хаотичний код із заплутаною логікою та щільно зв’язаними елементами. Цей термін повністю описує монолітну архітектуру, адже при використанні цього підходу виникає велика кількість залежностей, які з часом важко контролювати, дублювання логіки, адже все пишеться в одному файлі без розділення на функції чи модулі та безконтрольне використання умовних операторів. Такий код важко тестувати та підтримувати і складно вносити в його зміни без ризику нашкодити вже існуючому функціоналу [39].

Переваги:

- Простота реалізації
- Швидке розгортання
- Менше витрат на комунікацію між модулями

Недоліки:

- Зі зростанням об’єму коду підтримка додатку стає складною
- Неможливість масштабувати окремі частини
- Оновлення потребує перекомпіляції всього застосунку, що є досить довгим процесом

Раніше цим підходом користувалися при створенні старих веб-додатків на мові PHP, WordPress та старих ASP.NET WebForms.

2. Мікросервісна архітектура

Мікросервіси – це стиль програмування, за якого застосунок поділяється на окремі сервіси, кожен із яких виконує одну задачу та спілкується з іншими через API. API (Application Programming Interface) – це інтерфейс для взаємодії між різними програмами або компонентами програмного забезпечення [16]. В

даному випадку, кожен мікросервіс має можливість взаємодіяти з іншим мікросервісом через певний інтерфейс, це може бути, наприклад, HTTP(REST), gRPC або черги повідомлень, але пояснення цього виходить за рамки даної роботи, адже використовують такий підхід в основному великі компанії. Для невеликого проєкту, такого як система управління завданнями даний підхід є занадто складним в реалізації [1, 15].

Переваги:

- Можна масштабувати окремі сервіси
- Можливість використання різних мов або технологій
- Сервіси легко замінюються та повторно використовуються

Недоліки:

- Висока складність налаштування
- Потребує впровадження систем логування, моніторингу та балансування для контролю виконання окремого сервісу
- Вимагає складної інфраструктури DevOps

DevOps – це такий набір практик, що об'єднує розробку і експлуатацію (Development і Operations) програмного забезпечення для досягнення швидкої, стабільної та безперервної доставки продуктів (коду).

Прикладами продуктів, які використовують такий архітектурний підхід є Netflix, Amazon, Uber [15].

3. REST API (Representational State Transfer)

REST – це архітектурний стиль який базується на взаємодії клієнта і сервера через запити HTTP та використовує прості принципи [26, 35]:

- Клієнт робить HTTP-запити до серверного API та отримує дані у форматі JSON або XML [26, 35].

- API в свою чергу реалізує CRUD-операції (Create, Read, Update, Delete) за допомогою методів: GET для отримання даних, POST для надсилання нових даних, PUT для оновлення існуючих даних та DELETE для видалення даних [26, 35].
- Завдяки цим принципам додаток розділяється між клієнтом та сервером

Для даного додатку можна було б реалізувати REST API з наступними URL-адресами:

- GET/POST /tasks – створити або отримати завдання
- PUT /tasks/{id} – оновити існуюче завдання
- DELETE /tasks/{id} – видалити завдання
- GET /users/{id}/tasks – отримати усі завдання користувача

У такому випадку сторона клієнта, наприклад JavaScript React, зверталася б до API та на основі отриманих даних створювала б інтерфейс у браузері, і вся логіка взаємодії була б реалізована окремо [26, 35].

Переваги:

- Повна гнучкість інтерфейсу
- Можливість створення кількох клієнтів, наприклад браузер, мобільний застосунок або ж настільний застосунок
- Висока розділеність – набагато простіше масштабувати

Недоліки:

- Потрібно окремо реалізувати клієнтську сторону(тобто front-end), наприклад на React
- Вимагає авторизації через JSON Web Tokens(JWT), що може бути досить складною задачею для такого додатку
- Загалом ускладнює розробку для невеликого проєкту

Цей підхід виглядав би найбільш привабливим, якби не було ще одного і в даній роботі фінального архітектурного рішення – MVC.

4. Архітектура MVC (Model-View-Controller):

Обрана для даного проєкту архітектура – MVC на платформі ASP.NET Core. Це шаблон, що дозволяє розділити застосунок на три шари:

- Model (Модель) – шар, який відповідає за відображення даних та бізнес-логіку, працює з базою даних [Див. рис. 2.1], [13].
- View (Подання) – шар, який можна описати як HTML-розмітку з Razor-синтаксисом, що відповідає за клієнтський інтерфейс [Див. рис. 2.1], [13].
- Controller (Контролер) – шар, який керує логікою обробки запитів, взаємодіє з моделлю і повертає подання [Див. рис. 2.1], [13].

Даний підхід є фаворитом серед усіх, адже дозволяє нам створювати окремі контролери для кожної окремої моделі. Наприклад, запити від клієнта надходять до контролера TaskController, який взаємодіє із моделлю Task, обробляє логіку та повертає TaskView.cshtml.

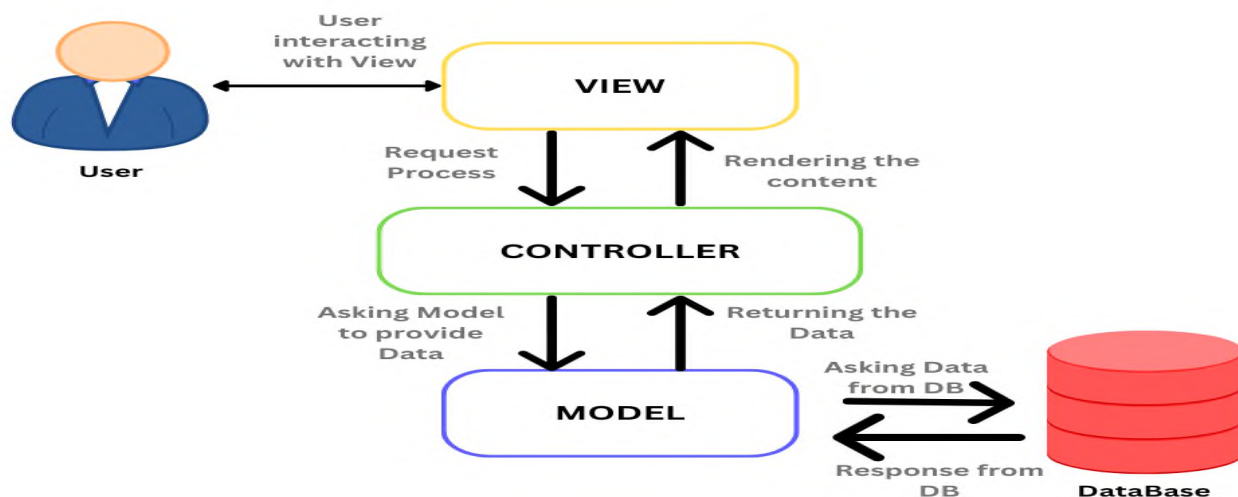


Рис. 2.1 Рисунок-опис архітектурного підходу MVC

При використанні MVC сервер керує маршрутизацією, безпекою та сесіями, всі сторінки формуються на сервері, підтримується ін'єкція залежностей та є можливість використовувати фільтри.

Перевагами цього рішення є простота впровадження, вбудована підтримка авторизації (ASP.NET Identity), зрозумілий шаблон для будь-якого рівня розробника, можливість масштабування через окремі контролери та зниження навантаження на клієнтську частину, адже сторінки передаються із сервера.

На мою думку, єдиним недоліком даного підходу є низька динамічність інтерфейсу, на відміну від REST API з його використанням Single Page Application(SPA), хоча це не є критичним в даному випадку [13].

SPA – це такий спосіб реалізації клієнтської сторони, коли за допомогою фреймворків JavaScript, таких як React, Angular чи Vue одна сторінка постійно оновлюється, залежно від відповіді сервера, яка отримується за допомогою JSON або XML. Тобто, існує завжди лише одна сторінка, яка лише міняє свої частини.

Враховуючи обраний архітектурний підхід, можна розділити архітектуру на певні складові, серед яких ми виділяємо:

1. Презентаційний рівень:

- Razor Pages для виводу HTML-сторінок
- Bootstrap – фреймворк для стилізації сторінок
- JavaScript/jQuery – для динамічних елементів

2. Рівень контролерів

- ASP.NET Core Controllers (HTTP-запити)
- Валідація даних, отриманих із форм
- Перенаправлення на інші сторінки
- Авторизація та аутентифікація

3. Сервісний рівень

- Створення сервісів для обробки запитів, ролей, задач, коментарів
- Ін'єкція залежностей через спеціальний Dependency Injection контейнер

4. Рівень доступу до даних

- Створення контексту бази даних
- Створення репозиторіїв на EF Core (для реалізації CRUD-операцій)

Отже, після обрання архітектурного підходу, поділу додатку на окремі шари та визначення основних задач варто узагальнити все вищеписане та перейти до ще однієї важливої задачі, без якої не обійдеться жоден веб-додаток – проєктування бази даних.

Нижче буде наведено узагальнену таблицю, яка описує усі вищезазначені архітектурні підходи.

Таблиця 2.2

Порівняльна таблиця архітектурних підходів

Архітектура	Основна ідея	Переваги	Недоліки	Доречність використання
Монолітна	Уся логіка в одному застосунку	Простота, швидкість розгортання	Важко масштабувати, складна підтримка при зростанні	Маленькі, однофункціональні проєкти
Мікросервіси	Розподіл на незалежні сервіси, що взаємодіють через API	Масштабованість, автономність модулів	Висока складність, потреба в DevOps	Великі, розподілені системи
REST API + SPA	Сервер надає API, клієнт (JS) робить	Гнучкість, багатоплатформеність, сучасний UI	Потрібна окрема front-end команда,	Веб-сервіси, мобільні клієнти, SPA

Архітектура	Основна ідея	Переваги	Недоліки	Доречність використання
	рендеринг UI		складність аутентифікації	
MVC (обрано)	Розділення на Model, View, Controller. Рендеринг на сервері	Простота, інтегровані механізми безпеки, швидкий старт, чітка структура	Менше динаміки, оновлення сторінки при кожному запиті	Корпоративні системи, внутрішні веб-додатки

2.3 Проєктування бази даних

При створенні будь-якого веб-додатку, що працює із великими обсягами інформації, важливою складовою є правильне проєктування бази даних. Особливо важливим це є для системи управління завданнями, де дані є взаємопов'язаними, динамічними і піддаються постійним оновленням — починаючи від створення нових задач і проєктів до зміни їх статусу виконання, дедлайнів та відповідальних осіб [22].

Чому проєктування є таким важливим?

- Воно дозволяє зберегти цілісність даних. Якщо база не є правильно спроектованою, дані можуть дублюватися, суперечити один одному або взагалі втрачатися при оновленнях. Добре спроектована база даних гарантує, що зв'язки між об'єктами є стабільними, передбачуваними і саме головне, контрольованими.
- Система повинна бути в змозі масштабовуватися. Вона повинна залишатися ефективною навіть за умови збільшення кількості користувачів, проєктів і задач. Проєктування з можливістю масштабування дозволяє уникнути болючих міграцій бази даних або зміни дизайну в майбутньому. Міграції — це незамінний інструмент у розробці

програмного забезпечення, особливо в контексті реляційних баз даних. Це скрипти або файли, які дозволяють керувати змінами в структурі бази даних організовано та контрольовано. Основна мета міграцій — підтримувати узгодженість між кодом програми та схемою бази даних.

- Система повинна завжди залишатися продуктивною. Запити до правильно нормалізованої бази даних виконуються швидше, споживають менше ресурсів і краще індексуються. Це особливо важливо при використанні фільтрації, сортування, пошуку по статусах, дедлайнах або виконавцях завдання. Нормалізація це техніка розробки бази даних, яка зменшує надмірність даних і усуває небажані характеристики, такі як аномалії вставки, оновлення та видалення. Правила нормалізації розділяють великі таблиці на менші та зв'язують їх за допомогою зв'язків.
- Система повинна завжди бути гнучкою. Добре спроектована база дозволяє впроваджувати нову функціональність(теги, коментарі, аналітику, права доступу), не ламаючи наявну структуру.
- Безпека і контроль доступу. Завдяки правильному розподілу ролей, таблицям користувачів і ролей можна побудувати чітку систему авторизації та розмежування доступу до функціоналу.

Основними етапами проєктування бази даних є:

1. Аналіз вимог — визначення функціональності, яку має підтримувати система, тобто які об'єкти будуть існувати, які зв'язки між ними.
2. Виявлення сутностей — визначення головних логічних одиниць, наприклад користувача, завдань, проєктів.
3. Визначення атрибутів — що саме характеризує кожну сутність. В контексті програмування це називають полями об'єктів, тобто ім'я користувача, опис завдання, кінцевий термін виконання і так далі.
4. Моделювання зв'язків — визначення того, як об'єкти між собою пов'язані. Наприклад користувач має декілька проєктів, проєкт має декількох користувачів, завдання має лише один проєкт і так далі. Це визначається

спеціальними зв'язками, наприклад один-до-одного, один-до-багатьох та багато-до-багатьох.

5. Нормалізація – це процес впорядкування структури бази даних з метою усунення надлишкових даних, а також уникнення аномалій вставки, оновлення, видалення.
6. Побудова ER-діаграми – ця діаграма виглядає як візуальне представлення усіх сутностей, їх полів(або ж атрибутів) і зв'язків між ними.
7. Вибір СУБД і реалізація схеми – впровадження структури в конкретній системі керування БД, в нашому випадку MS SQL Server.
8. Оптимізація – додавання індексів, обмежень, зовнішніх ключів та налаштування продуктивності.

Перевагами формалізованого підходу є зменшення ймовірності помилок які важко відслідковувати, забезпечення ефективної взаємодії кількох розробників, легше тестування, оновлення та масштабування, а також уніфіковане зберігання даних, що підвищує цілісність і безпеку.

Після аналізу вимог до системи управління завданнями, визначено основні сутності, які відображають головні об'єкти та процеси в системі. Кожна сутність(програмно це клас, в базі даних це таблиця) має набір атрибутів(полів), що характеризують її властивості.

1. Сутність Користувач(User):

Дана сутність призначена для зберігання інформації про користувачів системи. Вона також буде використовуватися також для авторизації. Основними атрибутами є:

- UserID(Primary Key): Унікальний ідентифікатор користувача. Він задається автоматично.
- FullName: Повне ім'я користувача.
- Email: Електронна пошта користувача.
- PasswordHash: Закодований пароль для автентифікації.

- RoleID(Foreign Key): Посилання на роль користувача у системі.

[Див. рис. 2.2]

2. Сутність Ролі(Roles):

Вона призначена для визначення ролей користувачів. Буде тісно пов'язана із попередньою сутністю, адже обидві відіграють ключову роль в автентифікації та авторизації. Основними атрибутами є:

- RoleID(Primary Key): Унікальний ідентифікатор ролі.
- RoleName: Назва ролі, наприклад адміністратор або користувач.

[Див. рис. 2.2]

3. Сутність Проекти(Projects):

Призначена для зберігання інформації про проекти, над якими працюють команди. Дана сутність Основними атрибутами є:

- ProjectID(Primary Key): Унікальний ідентифікатор проекту.
- Title: Назва проекту.
- Description: Опис проекту.
- CreatedAt: Дата створення проекту.
- CreatedByUser(Foreign Key): Посилання на користувача, який створив даний проект.
- Tasks: Згодом буде додано програмно. Це буде перелік усіх завдань у даному проекті.

[Див. рис. 2.2]

4. Сутність Завдання(Tasks):

Призначена для зберігання інформації про завдання в межах проектів. Основними атрибутами є:

- TaskID (Primary Key): Унікальний ідентифікатор завдання.

- Title: Назва завдання.
- Description: Опис завдання.
- Status: Статус виконання завдання (наприклад, Нове, В процесі, Завершено).
- Priority: Пріоритет завдання.
- DueDate: Кінцева дата виконання.
- ProjectID (Foreign Key): Посилання на проєкт, до якого належить завдання.
- AssignedUserID (Foreign Key): Посилання на користувача, відповідального за виконання завдання.

[Див. рис. 2.2]

5. Сутність Коментарі(Comments):

Призначена для зберігання коментарів до завдань для забезпечення комунікації між учасниками. Основними атрибутами є:

- CommentID (Primary Key): Унікальний ідентифікатор коментаря.
- Content: Текст коментаря.
- CreatedAt: Дата створення коментаря.
- AuthorID (Foreign Key): Посилання на користувача, який залишив коментар.
- TaskID (Foreign Key): Посилання на завдання, до якого належить коментар.

[Див. рис. 2.2]

6. Сутність Теги(Tags):

Призначена для категоризації завдань за допомогою тегів. Основними атрибутами є:

- TagID (Primary Key): Унікальний ідентифікатор тега.
- TagName: Назва тега.

[Див. рис. 2.2]

7. Сутність Завдання-Теги(TaskTags):

Призначена для реалізації зв'язку багато-до-багатьох між завданнями та тегами.

Основні атрибути:

- TaskID (Foreign Key): Посилання на завдання.
- TagID (Foreign Key): Посилання на тег.

[Див. рис. 2.2]

Ці сутності та їх атрибути формують основу для побудови ER-діаграми, яка відображатиме зв'язки між об'єктами системи. Для створення її використовувався інструмент Oracle SQL Data Modeler [Див. рис. 2.2]. Ця програма дозволяє створювати, редагувати та аналізувати логічні, фізичні, реляційні і багатовимірні моделі. Інструмент підтримує створення ER-діаграм і генерацію SQL-скриптів. Для створення діаграми було імпортовано .sql файл зі скриптами створення таблиць, а потім було експортовано результат у вигляді рисунку. [Див. рис. 2.2]

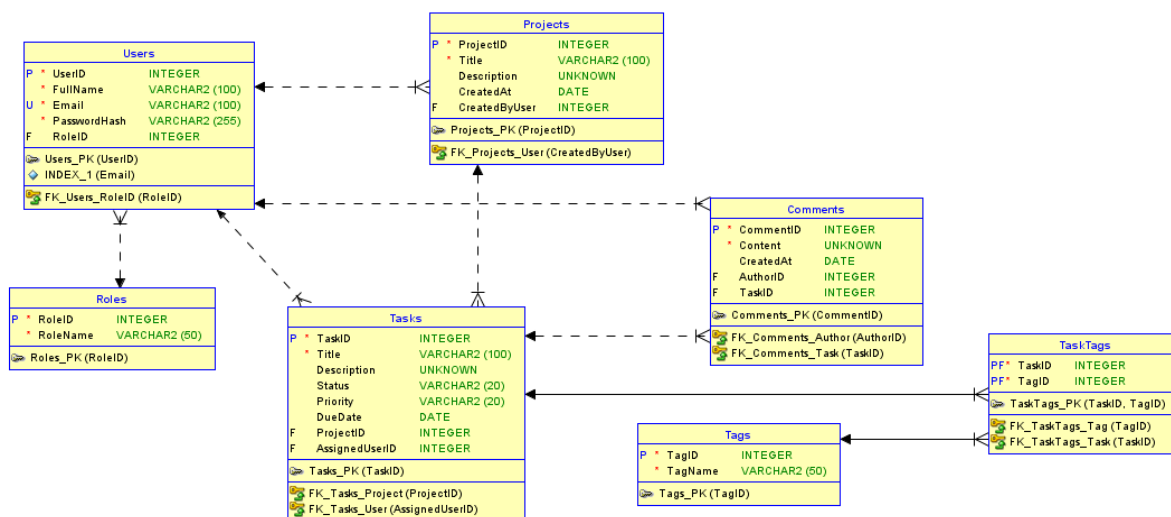


Рис. 2.2 ER-діаграма бази даних

2.4 Основний функціонал

Успішна розробка інформаційної системи управління завданнями потребує чіткого опису основного функціоналу, який безпосередньо реалізує вимоги кінцевих користувачів даного продукту. Основною метою є надання інструментів для ефективної організації роботи проєктних команд – починаючи від фіксації завдань до контролю їхнього виконання й забезпечення комунікації між учасниками.

До ключового функціоналу можна віднести:

1. Створення, редагування і контроль виконання завдань [22].
2. Управління користувачами і ролями [22].
3. Організація зворотного зв'язку через коментарі
4. Гнучке налаштування доступу [22].
5. Відображення актуального статусу роботи над проєктами

У цьому підрозділі буде розглянуто три базові компоненти: створення завдань, розподіл ролей та створення коментарів. Кожен із них є частиною загальної архітектурної системи, яка дозволяє підвищити прозорість процесів, зменшити кількість непорозумінь між учасниками і забезпечити контроль за виконанням поставлених цілей. Це наблизило нас до поняття MVP – Minimal Valuable Product – мінімально життєздатний продукт, а саме тестова версія програми, за допомогою якої можна зрозуміти, чи варто впроваджувати ті чи інші функції, а також зрозуміти, що саме необхідно користувачам [18].

Створення завдань:

Однією з центральних функцій системи є модуль управління завданнями. Завдання – це одиниці роботи із чіткою структурою, які закріплюються за учасниками проєкту та мають відповідні атрибути, що забезпечать контроль виконання.

Функціональними можливостями модуля є:

- Створення нового завдання: користувачі з відповідними правами можуть створити завдання в межах конкретного проєкту, вказавши назву, опис, пріоритет, дату дедлайну і призначити виконавця.
- Редагування завдання: повинна бути можливою змінна будь-якого атрибуту завдання, а також зміна виконавця.
- Фільтрація та сортування: користувачі можуть швидко знайти потрібне завдання за статусом, пріоритетом або виконавцем. Це особливо важливо у великих проєктах з десятками активних завдань.
- Додавання тегів: для швидкої категоризації завдань можливе прикріплення тегів, що допомагає групувати завдання та структурувати робочий процес.
- Завдання мають бути легкі у сприйнятті, обмежені за обсягом та мати чітко визначений результат. У рамках архітектури платформи передбачається використання статусної моделі з фіксованим набором станів, які відповідають етапам виконання завдання.

Розподіл ролей:

Система дозволяє чітко розмежовувати повноваження користувачів шляхом впровадження ролей. Це реалізується за допомогою сутності Roles, пов'язаної з кожним користувачем через зовнішній ключ RoleID.

Основні ролі:

- Адміністратор (Admin): має повний доступ до всіх функцій системи, включно з управлінням користувачами, проєктами, завданнями, коментарями. Саме адміністратор створює нові проєкти, призначає відповідальних осіб, встановлює права доступу.
- Користувач (User): може переглядати і редагувати лише ті завдання, до яких він має доступ (наприклад, де є виконавцем або учасником). Має змогу залишати коментарі, оновлювати статус виконання.

- Менеджер проєкту (в MVP не додається): у майбутньому система може бути розширена для підтримки проміжних ролей з окремими правами, наприклад, для делегування частини адміністративного функціоналу.

З використанням ролей з'являється можливість забезпечити безпеку даних, обмеживши деякі функції для звичайних користувачів, такі як призначення виконавця. Це дозволить нам створити контрольовану систему розмежування повноважень, що є важливим для корпоративного середовища.

Коментарі до завдань:

Ефективна командна робота потребує постійної комунікації. Саме тому в системі реалізовано модуль коментарів (Comments), який забезпечує текстову взаємодію користувачів у межах кожного конкретного завдання.

Ключові функції:

- Залишення коментаря до завдання: кожен користувач може залишити повідомлення під завданням, що зберігається із зазначенням автора та дати створення.
- Читання історії комунікації: усі коментарі зберігаються хронологічно, що дозволяє відслідковувати логіку змін і прийнятих рішень.
- Повідомлення про нові коментарі: у перспективі може бути реалізована система повідомлень, яка повідомляє користувача про нові записи у завданнях, що його стосуються. До MVP не додається.

Описані компоненти становлять базову інфраструктуру даної системи управління завданнями. У наступному розділі буде описано функціонал інших модулів, адже зараз головне створити продукт, який вже можна буде вводити в експлуатацію та відслідковувати результат. Також у наступному розділі буде описано весь процес створення продукту, від налаштування робочого середовища до тестування вже існуючої системи.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБ-ДОДАТКУ.

3.1 Налаштування середовища розробки та налаштування проєкту

Розробка системи здійснювалася з використанням сучасного технологічного стеку Microsoft, який забезпечує високу продуктивність, гнучкість та інтеграцію з хмарними сервісами. Для реалізації серверної частини було використано ASP.NET Core – кросплатформену фреймворк-систему з відкритим вихідним кодом, яка дозволяє створювати продуктивні веб-додатки. Робота з базою даних здійснювалася через ORM Entity Framework Core.

Було використанні наступні інструменти та середовище:

- IDE: Visual Studio 2022 Community Edition
- .NET SDK: .NET 8.0(Long-term support)
- ORM: Entity Framework Core 9
- База даних: Microsoft SQL Server 2022 Developer Edition
- Інтерфейс адміністрування бази даних: SQL Server Management Studio(SSMS) [Рис. 3.2]
- Контроль версій: Git + GitHub
- Керування залежностями: NuGet Package Manager
- Інтерфейс користувача: ASP.NET Core MVC
- Автентифікація: ASP.NET Identity
- Front-End бібліотеки: Bootstrap 5, jQuery
- Операційна система: Windows 11

Підготовка проєкту здійснювалася таким чином:

У Visual Studio було створено новий проєкт ASP.NET Core MVC. [Див. рис. 3.1] Ключовою перевагою створення цього проєкту у Visual Studio є те, що структура проєкту повністю створюється середовищем розробки, а саме:

- **Controllers/** : Дана директорія містить контролери, які відповідають за обробку HTTP-запитів. Контролер зв'язує користувача з логікою додатку. [Див. рис. 3.1]
- **Models/** : Тут розміщені класи-моделі, які представляють сутності з бази даних або бізнес логіку. [Див. рис. 3.1]
- **Views/** : Містить .cshtml файли – представлення, які відповідають методам контролерів.
- **wwwroot/** : Це публічна папка, в яку потрапляє вся статична інформація, а саме CSS-файли JS-файли, зображення тощо.

Також проєкт містить такі файли:

- **appsettings.json**, який є файлом конфігурації, де зберігаються рядки підключення до бази даних, ключі, змінні середовища, параметри логування та будь-які інші налаштування проєкту.
- **Program.cs**, який є точкою входу в програму. Тут ми ініціалізуємо робоче середовище, налаштовуємо залежності, задаємо параметри маршрутизації тощо.
- **TaskManager.csproj**, який є файлом проєкту. Тут визначені деякі залежності, версії SDK(Software Development Kit), цільова платформа та збірка. [Див. рис. 3.1]

Також вручну були додані такі директорії, як **Services** для бізнес-логіки, яка не має стосунку до контролерів чи моделей та **Data** для контексту бази даних, конфігурації міграцій та ініціалізаторів даних. [Див. рис. 3.1]

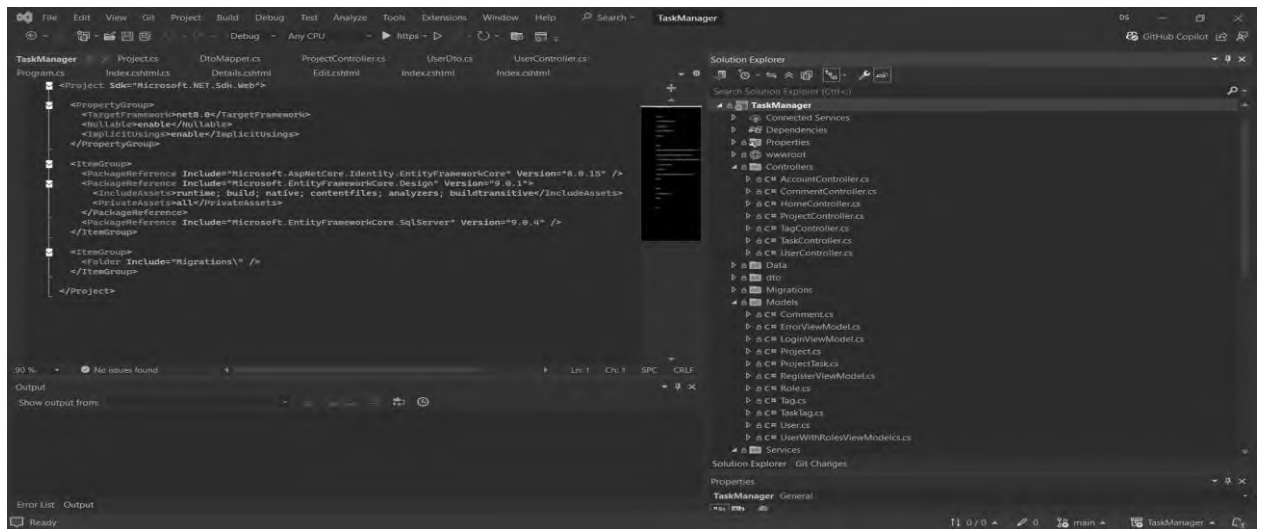


Рис. 3.1 Основна структура проекту

Для роботи з базою даних було додано рядок підключення до бази даних у файл `appsettings.json`:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
      "Server=localhost;Database=TaskManagerDb;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Це дозволить додатку автоматично підключатися до бази даних. Це налаштування не працює без окремого налаштування у файлі `Program.cs`, де ми додаємо залежність:

```
builder.Services.AddDbContext<TaskManagerContext>(options =>
```

```
options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));
```

За наявності такої структури, було створено віддалений репозиторій GitHub, куди ми вносимо наш проєкт для контролю версій. Це забезпечить нам можливість відхилити небажані зміни та повернутися до минулих версій.

3.2 Реалізація серверної частини

Серверна частина веб-додатку реалізована на базі фреймворку ASP.NET Core MVC, який забезпечує чітке розділення відповідальностей між логікою обробки запитів (контролери), відображенням (представлення) та обробкою даних (моделі). В якості системи управління базою даних було використано Entity Framework Core — сучасну ORM-бібліотеку, яка дозволяє взаємодіяти з базою даних на рівні об'єктів без потреби писати SQL-запити вручну.

Було обрано підхід Code First, що передбачає створення структури бази даних на основі C# класів. Такий підхід спрощує розробку, дозволяє швидко змінювати модель даних і автоматично оновлювати схему бази, використовуючи міграції. Це особливо зручно під час активної фази розробки, коли структура бази може змінюватися відповідно до нових функціональних вимог.

Відповідно до функціоналу системи, реалізовано набір моделей, які відображають ключові об'єкти предметної області:

- User — розширена модель користувача, яка наслідує IdentityUser<int>. Використання цілочисельного ключа (int) як первинного ключа дозволяє краще контролювати типізацію на рівні бази даних та узгодженість із іншими сутностями. Крім стандартних полів, таких як логін та пароль, ця модель включає додаткові атрибути: ім'я користувача, створені користувачем проєкти, завдання які виконує користувач та коментарі які залишив користувач.

- Role — модель ролі, яка наслідує IdentityRole<int>. Це дозволяє додавати додаткові поля у разі потреби, наприклад опис ролі або рівень доступу. У системі реалізовано базову підтримку ролей для керування правами доступу.
- Project — описує проєкт, що містить назву, опис, дату створення та зв'язок з користувачами та завданнями. Кожен проєкт є логічною одиницею для групування завдань.
- Task — представляє окреме завдання в межах проєкту. Містить заголовок, опис, статус, пріоритет, дату дедлайну та посилання на виконавця (користувача) і проєкт. Забезпечено можливість відслідковувати виконання, зміну статусу та коментування.
- Comment — модель для коментарів до завдань. Містить текст коментаря, дату створення та автора. Це дозволяє організувати живу дискусію в межах конкретного завдання.
- Tag — модель мітки, яка дозволяє категоризувати завдання. Одна мітка може бути пов'язана з багатьма завданнями, і навпаки — одне завдання може мати кілька міток.
- TaskTag — проміжна модель, яка реалізує зв'язок типу "багато-до-багатьох" між Task і Tag. Такий підхід є типовим для реалізації гнучкої системи тегування. Кожен запис цієї таблиці містить зовнішні ключі до відповідного завдання та мітки.

Усі класи моделей будуть наведені у додатку А.

Для взаємодії з базою даних реалізовано контекстний клас TaskManagerContext, який наслідує IdentityDbContext<User, Role, int>. Це дозволяє об'єднати функціональність ASP.NET Identity з власними моделями додатку та забезпечити повноцінну інтеграцію користувачів і ролей у систему [Додаток Б].

Основні моменти в даній конфігурації:

- Спадкування від `IdentityDbContext` дозволяє зберігати всю інформацію про користувачів і ролі в єдиній базі, поряд з іншими доменними сутностями.
- `Fluent API` використовується для конфігурації усіх зв'язків, зокрема "багато-до-багатьох" (`Task` та `Tag` через `TaskTag`).
- Унікальний ключ у `TaskTag` (комбінований `TaskId` + `TagId`) забезпечує правильну роботу без дублікатів.

Як було сказано вище, для створення зв'язків між моделями було використано спосіб конфігурації моделі `Fluent API`. За допомогою `Fluent API` ми визначаємо правила конфігурації для моделі та зв'язків між таблицями без використання атрибутів.

Цей спосіб конфігурації моделі через `Fluent API` використовує методи для побудови налаштувань. Це дозволяє точніше контролювати структуру бази даних і зберігає більшу гнучкість порівняно з атрибутами в самій моделі.

Міграції створювалися за допомогою стандартного інструменту `Entity Framework Core Migrations`. Даний інструмент дозволяє нам лише за допомогою двох команд у командному рядку запустити процес, при якому на основі моделей створених нами генеруються SQL-скрипти і створюються таблиці у базі даних із заданими нами налаштуваннями. Вищезазначені команди виглядають ось так:

```
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

Команда `dotnet ef migrations add` створює нову міграцію. Вона генерує код, який описує зміни, які необхідно внести в базу даних, щоб вона відповідала новому стану наших моделей. У прикладі вище ми додали міграцію з назвою `InitialCreate`, хоча назва може бути довільною.

Кожна міграція зберігається в окремому класі в папці Migrations в проєкті. Ці класи містять код для створення або зміни таблиць і стовпців, а також визначають зовнішні ключі, індекси, каскадні операції та інші налаштування.

Команда `dotnet ef database update` застосовує міграції до бази даних. Вона аналізує міграції, які ще не були застосовані, і застосовує їх до поточної бази даних. Це дозволяє автоматично створювати таблиці, індекси та інші базові структури, визначені в наших моделях.

При виконанні цієї команди Entity Framework Core перевіряє, яка версія бази даних актуальна, порівнює її з останньою міграцією, і лише тоді виконує необхідні зміни. Таким чином, ця команда дозволяє оновити структуру бази даних до нової версії, не перезавантажуючи чи не втрачаючи дані [7].

Після виконання команд для створення та застосування міграцій у проєкті з'являється папка Migrations, де зберігаються всі міграції для проєкту. Кожна міграція — це окремий клас, що описує певні зміни в базі даних [Додаток В].

Метод `Up` у класі міграції описує зміни, які будуть виконані у базі даних при застосуванні міграції (створення таблиць, додавання стовпців тощо).

Метод `Down` визначає, як повернутися до попереднього стану бази даних, скасувавши зміни, що були внесені міграцією. Цей метод викликається під час відкату міграції.

Оскільки кожна міграція є окремим класом, в базі даних також створюється таблиця `__EFMigrationsHistory` (за замовчуванням), в якій зберігається інформація про всі міграції, які були застосовані до цієї бази даних. [Див. рис. 3.2] Ця таблиця містить такі поля, як `MigrationId` та `ProductVersion`, що дозволяє Entity Framework Core відслідковувати, яка версія бази даних була застосована, і чи потрібно виконувати нові міграції.

Іноді потрібно скасувати зміни в базі даних або повернути її до попереднього стану. Для цього можна використовувати команду `dotnet ef database update <назва міграції>`, вказуючи міграцію, до якої хочеш повернутися.

У командному рядку це вказується наступним чином:

dotnet ef database update 20230506010234_InitialCreate

Як було зазначено вище, ми використовуємо СУБД Microsoft SQL Server та Microsoft SQL Server Management Studio для того щоб графічно працювати із базою даних та перевіряти її структуру. Нижче буде наведено рисунок із представленням інтерфейсу адміністрування SSMS, а також в Object Explorer можна побачити результат виконання міграцій у проєкті:

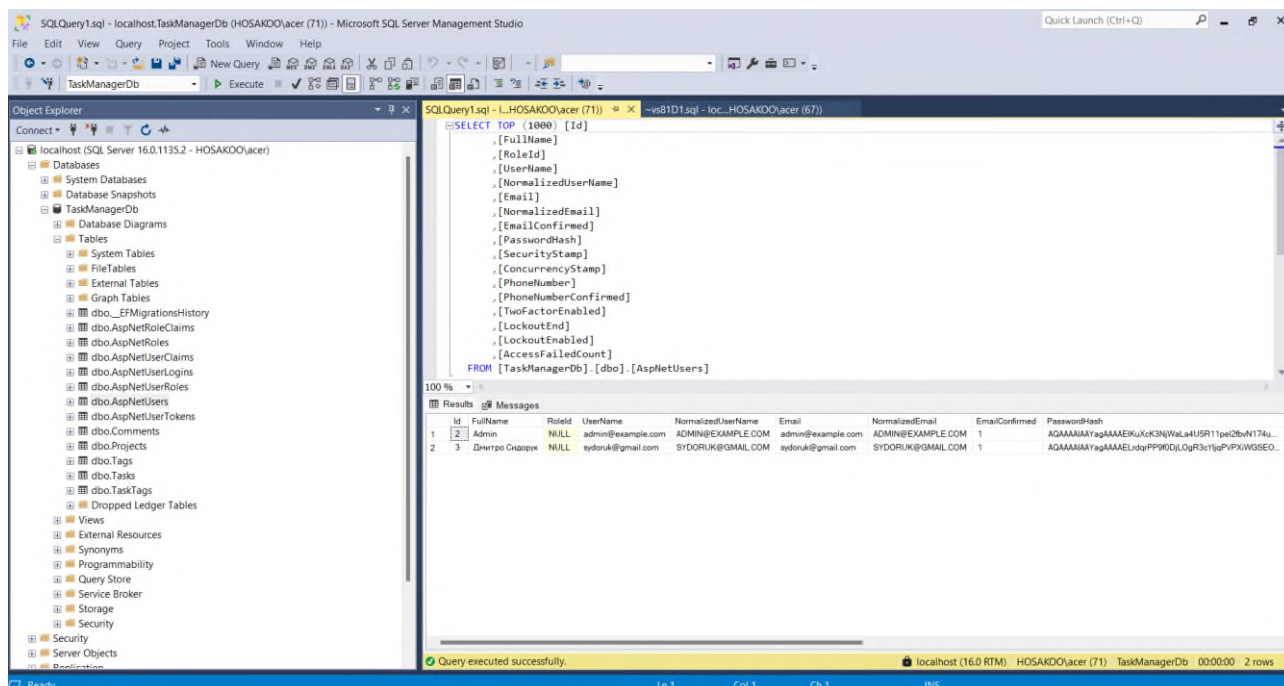


Рис. 3.2 Графічний інтерфейс SQL Server Management Studio та структура бази даних TaskManagerDb

Контролери в ASP.NET Core MVC відіграють ключову роль у взаємодії між користувачем і серверною логікою. У межах нашого застосунку було створено кілька контролерів, відповідальних за різні аспекти управління проєктами, завданнями, тегами та коментарями. Кожен контролер відповідає за

CRUD-операції для відповідної сутності та взаємодіє з сервісним шаром або безпосередньо з контекстом бази даних.

Основні контролери:

- `ProjectsController`: перегляд, створення, редагування та видалення проєктів.
- `TasksController`: управління завданнями в рамках проєктів.
- `TagsController`: створення та асоціювання тегів із завданнями.
- `CommentsController`: додавання та перегляд коментарів до завдань.
- `UsersController` (лише для адміністраторів): керування користувачами та їх ролями.

Контролери реалізують обробку як GET-запитів (для відображення сторінок), так і POST-запитів (для внесення змін). Усі критичні дії захищено авторизацією.

Для перевірки вхідних даних використовувалися атрибути валідації з простору `System.ComponentModel.DataAnnotations`. Наприклад:

- `[Required]` — обов'язкове поле
- `[StringLength]` — обмеження довжини
- `[EmailAddress]`, `[Phone]` — перевірка на правильний формат

Крім того, для складніших сценаріїв реалізовані кастомні атрибути валідації. Наприклад, атрибут, що перевіряє, чи дата дедлайну завдання не є у минулому.

Помилки валідації автоматично відображаються у відповідних формах завдяки `ModelState`. Це забезпечує зручний досвід користувача та надійність даних.

У застосунку реалізована рольова авторизація на основі `ASP.NET Core Identity`. Були створені дві основні ролі:

- Admin — має доступ до всіх функцій, включаючи управління користувачами, проєктами та системними налаштуваннями.
- User — може створювати власні проєкти та завдання, коментувати та взаємодіяти з командою.

Ось конфігурація авторизації:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));
    options.AddPolicy("ProjectMember", policy =>
        policy.RequireClaim("ProjectMember", "true"));
});
```

Завдяки конфігурації авторизації ми маємо можливість обмежувати доступ до контролера UserController за допомогою атрибуту [Authorize(Policy = "AdminOnly")].

У застосунку передбачено багаторівневу обробку помилок:

1. Глобальна обробка — у Program.cs налаштований middleware UseExceptionHandler, який перенаправляє на спеціальну сторінку помилки.
2. Логування — усі винятки журналюються через вбудований ILogger.
3. Обробка ModelState помилок — неправильні запити користувача (наприклад, незаповнені форми) не проходять далі і повертаються з відповідними повідомленнями.
4. Спеціальні сторінки — створено окремі View для помилок 404 (Not Found), 403 (Forbidden) та 500 (Internal Server Error), що підвищує зручність і стабільність інтерфейсу.

Нижче наведено action-метод контролера UserController, який відповідає за зміну даних користувача:

```

[HttpPost]
[Authorize(Roles = "Admin")]
public async Task<IActionResult> Edit(UserWithRolesViewModel model)
{
    var user = await userManager.FindByIdAsync(model.Id.ToString());
    if (user is null) return NotFound();
    user.FullName = model.FullName;
    user.Email = model.Email;
    user.UserName = model.Email;
    user.NormalizedEmail = model.Email.ToUpper();
    user.NormalizedUserName = model.Email.ToUpper();

    var currentRoles = await userManager.GetRolesAsync(user);
    await userManager.RemoveFromRolesAsync(user, currentRoles);
    await userManager.AddToRoleAsync(user, model.RoleName);
    await userManager.UpdateAsync(user);
    return RedirectToAction(nameof(Index));
}

```

3.1 Лістинг коду контролера UserController

Повний лістинг коду даного контролера буде вказано у додатку Г.

3.3 Реалізація клієнтської частини

Клієнтська частина веб-додатку реалізована з використанням Razor Pages в поєднанні зі стандартними технологіями веб-розробки, такими як HTML5, CSS3, JavaScript та бібліотеки Bootstrap 5, яка забезпечує адаптивну, зручну і привабливу верстку.

Основна мета фронтенду — забезпечити користувачеві зручний інтерфейс для взаємодії з функціоналом системи: створення та перегляд проєктів і завдань, фільтрація, авторизація тощо.

Для розмітки інтерфейсу використовується Razor-синтаксис (.cshtml файли), який дозволяє безпосередньо інтегрувати серверні дані у HTML. Завдяки ньому ми маємо можливість створювати динамічні сторінки, які наповнюються залежно від того, які дані передаються нам в модель зі сторони сервера. Також, для валідації форм використовується технологія ASP.NET Core Tag Helpers, що поєднує в собі серверну та клієнтську валідацію, базуючись на jQuery Validate.

Варто розглянути детальніше Razor:

Razor — це синтаксис шаблонів, що використовується у технології ASP.NET Core, який дозволяє поєднувати HTML та C# в одному файлі для динамічного формування веб-сторінок. Razor є ключовим компонентом шаблонного рушія у ASP.NET Core MVC і Razor Pages, забезпечуючи гнучкий та зручний спосіб створення подання (View), що відображає дані з моделі.

При створенні додатку усі необхідні службові класи та моделі створювалися з використанням ASP.NET Core Identity, тому в першу чергу було реалізовано авторизацію та автентифікацію, разом із сторінками реєстрації [Див. рис. 3.3] та входу [Додаток Д].

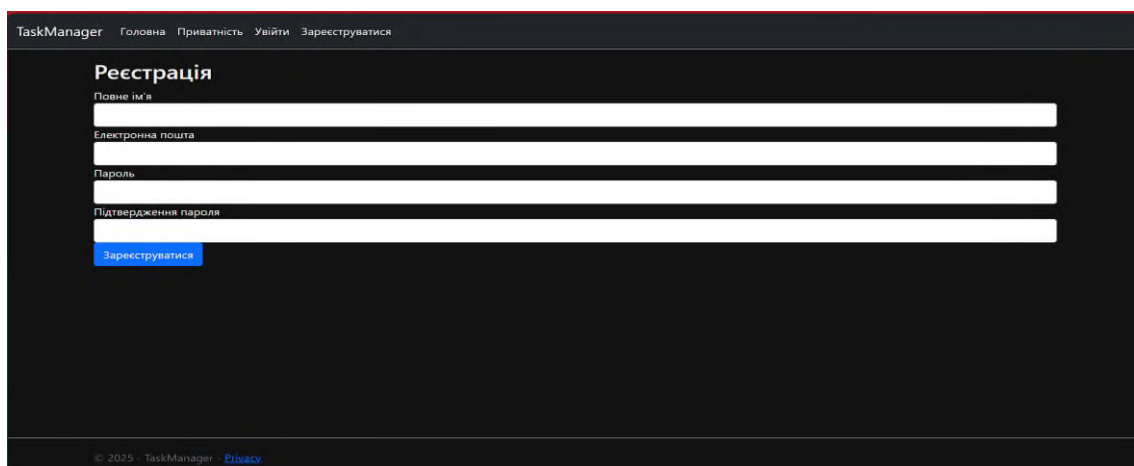


Рис. 3.3 Сторінка реєстрації

При створенні вищезазначених сторінок було використано моделі DTO(Data Transfer Object), щоб відображати моделі було зручніше. Для користувача було створено дві різних моделі, перша – для реєстрації та входу в акаунт, а друга – для передачі користувача як моделі у необхідні View. Завдяки цьому, при створенні клієнтської частини я зосереджувався лише на необхідних властивостях моделей, а на серверній стороні ці моделі швидко перетворювалися у необхідні повноцінні моделі і над ними виконувалися різного типу операції.

Також, було реалізовано заборону доступу до окремих сторінок веб-додатку, таких як наприклад список усіх користувачів або сторінка зміни інформації про проєкт, якщо він не належить користувачу. Це було зроблено завдяки серверній частині, де такі випадки оброблялися і повертали помилку Forbid 403, а також завдяки прихованню посилань на ці сторінки у меню навігації. Якщо користувач зареєстрований, то він може бачити в меню навігації усі посилання, окрім сторінки з користувачами, а якщо користувач є адміністратором, то він може бачити дану сторінку.

Це було реалізовано в окремому файлі _Layout.cshtml, який розміщений у папці Shared, а тому кожен файл Razor має до його доступ [Додаток Е].

Було створено усі необхідні форми для зміни або створення усіх моделей, такі як Project/Create [Додаток Ж], [Див. рис. 3.4], Project/Edit, Task/Create, Task/Edit.

The screenshot shows the 'Create New Project' form in the TaskManager application. The form is centered on a dark background. It has a title 'Create New Project' at the top. Below the title, there are three input fields: 'Title' with a placeholder 'Enter project title', 'Description' with a placeholder 'Describe your project', and 'Creation Date' with a placeholder 'dd.mm.pppp --:--'. At the bottom of the form, there are two buttons: 'Back to List' and 'Create'.

Рис. 3.4 Інтерфейс створення проєкту

Для валідації помилок у формах використовуються невеликі JavaScript-скрипти, а для покращення UX, наприклад реалізації Drag and Drop у проєкті із Kanban-дошкою використовуються скрипти більш просунуті скрипти JS, що дає змогу нам забезпечити більш приємну взаємодію з інтерфейсом [Див. рис. 3.5], [Додаток 3].

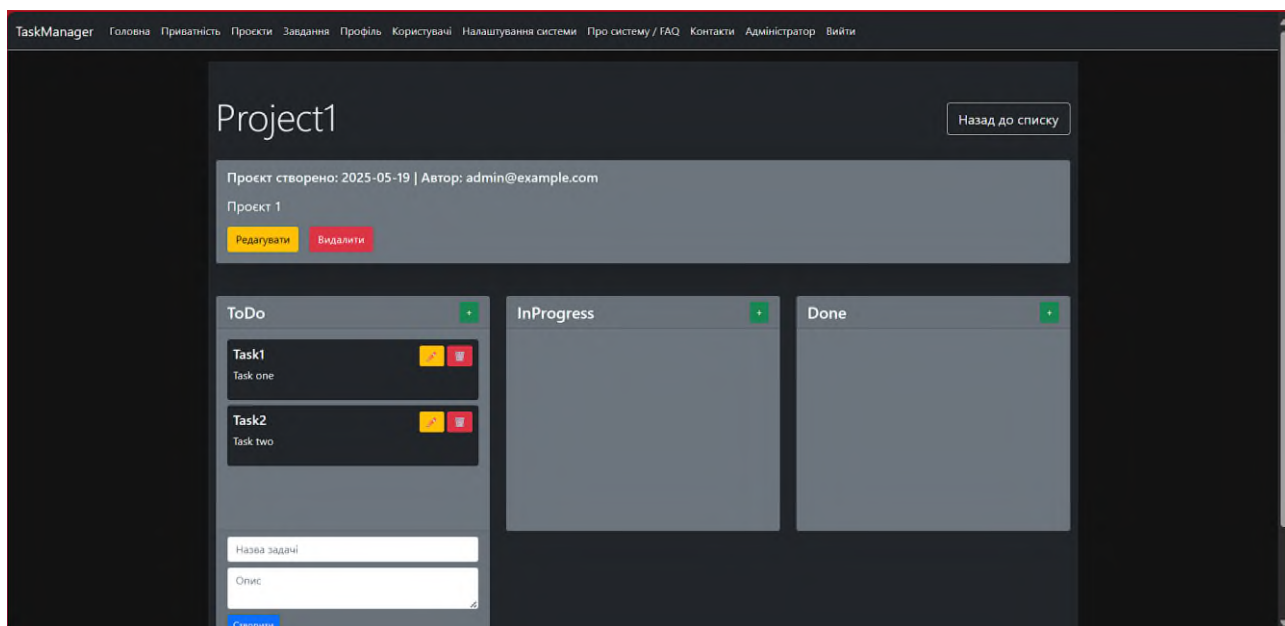


Рис. 3.5 Інтерфейс взаємодії із Kanban-дошкою

Авжеж, використовуються класи Bootstrap для спрощення стилізації сторінок. Дана бібліотека надає готові стилі, компоненти та інструменти для швидкої розробки адаптивного та сучасного інтерфейсу веб-застосунку. В її основі лежать HTML, CSS і JavaScript, які забезпечують створення стильових елементів інтерфейсу, таких як кнопки, таблиці, форми, модальні вікна і так далі.

Ключовою особливістю Bootstrap є адаптивна сітка, яка забезпечує коректне відображення макету сторінки на різних пристроях із різними розмірами екрану, включаючи мобільні пристрої. Стилiзація здійснюється за допомогою CSS-класів(наприклад btn, form-control, card, alert, row і так далі), які застосовуються відразу при їх заданні у HTML-розмітці.

Отже, в результаті було створено адаптивний інтерфейс із потужним механізмом авторизації. Завдяки Razor сторінки забезпечують динамічний рендеринг та зручну роботу з даними на сторінці.

3.4 Тестування системи

Тестування є невід’ємною частиною процесу розробки програмного забезпечення, що дозволяє переконатися в коректності, надійності та відповідності реалізованого функціоналу заданим вимогам. У межах реалізації веб-додатку для керування завданнями було застосовано два основних підходи до тестування: юніт-тестування (unit testing) та інтеграційне тестування (integration testing).

Юніт-тестування спрямоване на перевірку окремих функціональних одиниць системи — методів, сервісів або компонентів. У реалізації даного додатку передбачено покриття юніт-тестами таких частин, як:

- сервіси (Один з них TaskService) [Додаток К];
- Контролери(UserController) [Додаток Л];
- валідаційна логіка;
- логіка фільтрації та сортування задач.

Тестування здійснюється з використанням фреймворку xUnit. Для ізоляції залежностей застосовано бібліотеку Moq, яка дозволяє створювати підроблені реалізації інтерфейсів. Простіше кажучи, за допомогою цієї бібліотеки ми створюємо “фейковий” об’єкт.

Інтеграційне тестування дозволяє перевірити взаємодію між компонентами системи, включаючи взаємодію з базою даних, системою автентифікації та авторизації. Для цього використано тестову базу даних SQLite in-memory, а також стандартні інструменти ASP.NET Core для створення тестового сервера (WebApplicationFactory).

Основна мета — перевірити, що система коректно реагує на HTTP-запити та забезпечує відповідну поведінку з урахуванням ролей користувачів, прав доступу та логіки обробки даних.

Після запуску усіх реалізованих тестів не було виявлено критичних помилок. Усі тести пройдені успішно, що свідчить про стабільну роботу основних компонентів системи. В подальшому планується розширити покриття тестами, особливо в частині авторизації, фільтрації та обробки винятків.

ВИСНОВОК

У результаті виконання кваліфікаційної роботи на тему “Розробка системи управління завданнями для проєктної команди” було досягнуто поставлену мету – створити веб-додаток, який дозволяє контролювати, координувати та організовувати роботу проєктної команди в межах одного проєкту. Було проведено повний цикл дослідження, від аналізу вже існуючих рішень і їх порівняння до реалізації власного рішення з урахуванням потреб малих команд.

Перший розділ був присвячений аналізу предметної області та функціоналу популярних систем управління завданнями, таких як Jira, Trello, Asana, ClickUp та Microsoft Planner. В ході порівняльного аналізу було виявлено такі недоліки, як перевантаженість функціоналом для невеликих команд та обмежений функціонал без можливості адаптації до індивідуальних потреб.

У другому розділі було обґрунтовано вибір технологічного стеку. Серверна частина розроблялася із використанням ASP.NET Core MVC – сучасного веб-фреймворку з підтримкою архітектурного патерну Model-View-Controller. Для взаємодії з базою даних було обрано ORM-бібліотеку Entity Framework Core із підходом Code First, що забезпечило нам зручне управління сутностями та виконання міграцій. Також було розроблено систему авторизації та автентифікації з використанням ASP.NET Identity. Також було розроблено клієнтську частину з використанням Razor Web Pages, JavaScript, CSS, Bootstrap та jQuery, що забезпечило нам гарний вигляд додатку та легку взаємодію із сервером.

У третьому розділі детально розглянуто реалізацію серверної логіки. Було створено сервіси, що інкапсулюють бізнес-логіку додатку, DTO-моделі для обміну даними між шарами, мапінг сутностей, а також контролери, які забезпечують маршрутизацію запитів. Всі компоненти побудовані з дотриманням принципів SOLID і мають чіткий розподіл відповідальностей. Реалізовано базову авторизацію, захист доступу до ресурсів та обробку помилок. Також створено проєкт бази даних, що охоплює основні сценарії командної роботи.

У результаті реалізації системи отримано гнучкий, масштабований веб-додаток, який дозволяє створювати проєкти, призначати завдання, контролювати їх виконання, коментувати, додавати теги, відстежувати виконавців, а також розмежовувати права доступу залежно від ролі користувача. Така система здатна стати ефективним інструментом організації командної роботи в рамках малих і середніх розробницьких груп.

Система, яка була створена, є основою для подальшого вдосконалення та масштабування, зокрема для розробки зручного інтерфейсу, розширення аналітичного функціоналу та впровадження інтеграції з іншими сервісами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Архітектура програмного забезпечення. Wezom. URL: <https://wezom.com.ua/ua/blog/arhitektura-programmnogo-obespecheniya> (Дата звернення: 28.02.2025)
2. Бази даних. Поняття ER-моделі. Поняття сутності (entity). Атрибути. Види атрибутів. BestProg. Програмування: теорія та практика. URL: <https://surl.li/hsglsp> (Дата звернення: 30.01.2025)
3. Коросташовець А. 10 найкращих альтернатив Jira у 2025 році. Worksection. URL: <https://worksection.com/ua/blog/jira-alternatives.html> (Дата звернення: 30.01.2025)
4. Методології управління проектами, або Що таке Waterfall, Agile та Scrum. De Visu. URL: <https://surli.cc/enmqus> (Дата звернення: 30.01.2025)
5. Методологія проектування баз даних. Stud.com.ua. URL: https://stud.com.ua/77215/informatika/metodologiya_proektuvannya_danih (Дата звернення: 28.02.2025)
6. Методологія Kanban: дошки, принципи та сервіси, які вам допоможуть. UASpectr. URL: <https://surl.li/zcghfc> (Дата звернення: 30.01.2025)
7. Міграційне тестування. QATestLab. URL: <https://training.qatestlab.com/blog/technical-articles/migration-testing/> (Дата звернення: 30.04.2025)
8. Монолітна архітектура ПЗ. QaLight. URL: [https://qalight.ua/baza-znaniy/shho-take-monolitna-arhitektura/#:~:text=Монолітна%20архітектура%20\(Monolithic%20Architecture\)%20–%20це%20традиційни.](https://qalight.ua/baza-znaniy/shho-take-monolitna-arhitektura/#:~:text=Монолітна%20архітектура%20(Monolithic%20Architecture)%20–%20це%20традиційни.) (Дата звернення: 28.02.2025)
9. Нахутін М. Asana vs trello: обираємо інструмент для керування проектами. Cloudfresh. URL: <https://surl.li/kzvozo> (Дата звернення: 30.01.2025)
10. Нахутін М. Jira vs asana: обираємо найкращий інструмент для менеджменту. Cloudfresh. URL: <https://surl.li/irdiix> (Дата звернення: 30.01.2025)

11. Оволодіння Entity Framework Core: Глибоке занурення. javascript.org.ua.
URL: <https://javascript.org.ua/ovolodinnya-entity-framework-core-gliboke-zanurennya-2/> (Дата звернення: 30.04.2025)
12. Огляд Asana, ціни, особливості з плюсами та мінусами. Ciroapp. URL:
<https://ciroapp.com/uk/asana-review/> (Дата звернення: 30.01.2025)
13. Підручник з MVC Framework для початківців: що таке, Archітектура та приклад. Guru99. URL: <https://www.guru99.com/uk/mvc-tutorial.html> (Дата звернення: 28.02.2025)
14. Худенко Д. ТОП 7 інструментів управління проектами у 2025 році. Worksection. URL: <https://surl.li/pkbmzl> (Дата звернення: 30.01.2025)
15. Що таке архітектура мікросервісів? Основні концепції та приклади. PayPro Global. URL: <https://payproglobal.com/uk/відповіді/що-таке-мікросервісна-архітектура/> (Дата звернення: 28.02.2025)
16. Що таке API. QaLight. URL: <https://qalight.ua/baza-znaniy/shho-take-api/> (Дата звернення: 28.02.2025)
17. Що таке JSON. QaLight. URL: <https://qalight.ua/baza-znaniy/shho-take-json/> (Дата звернення: 28.02.2025)
18. Що таке MVP: 10 кроків щоб створити MVP. Wezom. URL:
<https://wezom.com.ua/ua/blog/chto-takoe-mvp> (Дата звернення: 28.02.2025)
19. An Introduction To ASP.NET Razor Pages. Learn Razor Pages. URL:
<https://www.learnrazorpages.com/> (Last accessed: 28.02.2025)
20. Aranda A. ClickUp 2025: benefits, features & pricing. Software Advice.
URL: <https://www.softwareadvice.com/project-management/clickup-profile/> (Last accessed: 30.01.2025)
21. Asana pros and cons: top 4 advantages & disadvantages. FreshBooks.
URL: https://www.freshbooks.com/hub/projects-management/asana-pros-and-cons?utm_source=chatgpt.com (Last accessed: 30.01.2025)
22. Chapter 2: designing a task management system – nocobase. NocoBase. URL:
<https://www.nocobase.com/en/tutorials/task-tutorial-system-design> (Last accessed: 30.01.2025)

23. Clifford D. Monday.com vs asana vs trello vs jira. Medium. URL:
<https://medium.com/goodbusinesskit/monday-com-vs-asana-vs-trello-vs-jira-6862e73d9839> (Last accessed: 30.01.2025)
24. Frei K. What is trello? How it works, features, and pros & cons. Ahsuite Blog. URL: <https://blog.ahsuite.com/what-is-trello/> (Last accessed: 28.02.2025)
25. Gasimov O. What is a microservice. GlobalLogic Ukraine. URL:
<https://www.globallogic.com/ua/insights/blogs/microservices-architecture-for-beginners-part-one/> (Last accessed: 28.02.2025)
26. GeeksforGeeks. REST API architectural constraints. GeeksforGeeks. URL:
<https://www.geeksforgeeks.org/rest-api-architectural-constraints/> (Last accessed: 28.02.2025)
27. GeeksforGeeks. What is JIRA Software. GeeksforGeeks. URL:
<https://www.geeksforgeeks.org/what-is-jira-software/> (Last accessed: 30.01.2025)
28. GeeksforGeeks. What is Trello and How To Use It. GeeksforGeeks. URL:
<https://www.geeksforgeeks.org/what-is-trello-and-how-to-use-it/> (Last accessed: 30.01.2025)
29. Matveyeva A. Asana pricing overview: a complete guide to all plans. Chanty. URL: <https://www.chanty.com/blog/asana-pricing/> (Last accessed: 30.01.2025)
30. Microsoft planner review: simple task and work management. UC Today. URL: <https://www.uctoday.com/collaboration/microsoft-planner-review-simple-task-and-work-management/> (Last accessed: 30.01.2025)
31. Microsoft planner service description - service descriptions. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/office365/servicedescriptions/project-online-service-description/microsoft-planner-service-description> (Last accessed: 30.01.2025)
32. MVC: шаблон проектування архітектури додатку. Brander. URL:
<https://brander.ua/technologies/mvc#:~:text=MVC%20-%20найперший%20із%20шаблонів%20проектування%20архіт> (Last accessed: 28.02.2025)

33. Patel R. A beginner's guide to ASP.NET core identity for authentication and authorization (with .NET 8). Medium. URL: <https://medium.com/@ravipatel.it/a-beginners-guide-to-asp-net-core-identity-for-authentication-and-authorization-with-net-8-e6c8deb612f4> (Last accessed: 28.02.2025)
34. Pierce D. Trello's new update helps you manage Slack, email, and everything else. The Verge. URL: https://www.theverge.com/news/618349/trello-app-ai-inbox?utm_source=chatgpt.com (Last accessed: 30.01.2025)
35. REST API: що це, як працює, переваги і приклади. GoIT. URL: <https://goit.global/ua/articles/rest-api-shcho-tse-iak-pratsiuie-perevahy-i-pryklady/> (Дата звернення: 28.02.2025)
36. Tutorial: get started with EF core in an ASP.NET MVC web app. Microsoft Learn. URL: https://learn.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-9.0&utm_source=chatgpt.com (Last accessed: 28.02.2025)
37. What is a backlog. ProductPlan. URL: <https://www.productplan.com/glossary/backlog/> (Last accessed: 30.01.2025)
38. Z R. Maximizing efficiency with agile tools: jira, trello, and asana review. LinkedIn. URL: <https://www.linkedin.com/pulse/maximizing-efficiency-agile-tools-jira-trello-asana-review-zaveri-zfi7f/> (Last accessed: 30.01.2025)
39. Програмування по-італійськи: Звідки з'являється спагеті-код і чи варто з ним боротися. R_D media. URL: <https://robotdreams.cc/uk/blog/551-zvidky-zyavlyayetsya-spahetti-kod> (Дата звернення: 28.02.2025)

ДОДАТКИ

Додаток А.1

Лістинг коду класу User

```
namespace TaskManager.Models
{
    public class User : IdentityUser<int>
    {
        public string FullName { get; set; }
        public int? RoleId { get; set; }
        public Role Role { get; set; }
        public virtual ICollection<ProjectTask> AssignedTasks { get; set; }
        public virtual ICollection<Project> CreatedProjects { get; set; }
        public virtual ICollection<Comment> Comments { get; set; }
    }
}
```

Додаток А.2

Лістинг коду класу Project

```
namespace TaskManager.Models
{
    public class Project
    {
        public int ProjectId { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
    }
}
```

```

    public DateTime CreatedAt { get; set; }

    public int CreatedByUserId { get; set; }
    public User Creator { get; set; }

    public ICollection<ProjectTask> Tasks { get; set; } = new List<ProjectTask>();
}
}

```

Додаток А.3

Лістинг коду класу ProjectTask(Task)

```

namespace TaskManager.Models
{
    public class ProjectTask
    {
        public int TaskId { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public string Status { get; set; }
        public string Priority { get; set; }
        public DateTime? DueDate { get; set; }
        public int ProjectId { get; set; }
        public virtual Project Project { get; set; }
        public int? AssignedUserId { get; set; }
        public virtual User AssignedUser { get; set; }
        public virtual ICollection<Comment> Comments { get; set; }
        public virtual ICollection<TaskTag> TaskTags { get; set; }
    }
}

```

Лістинг коду класу TaskManagerContext

```
namespace TaskManager.Data
{
    public class TaskManagerContext : IdentityDbContext<User, Role, int>
    {
        public TaskManagerContext(DbContextOptions<TaskManagerContext>
options) : base(options) { }

        public DbSet<Project> Projects { get; set; }
        public DbSet<ProjectTask> Tasks { get; set; }
        public DbSet<Comment> Comments { get; set; }
        public DbSet<Tag> Tags { get; set; }
        public DbSet<TaskTag> TaskTags { get; set; }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
            builder.Entity<User>()
                .HasOne(u => u.Role)
                .WithMany()
                .HasForeignKey(u => u.RoleId)
                .OnDelete(DeleteBehavior.Restrict);

            builder.Entity<Project>()
                .HasOne(p => p.Creator)
                .WithMany(u => u.CreatedProjects)
                .HasForeignKey(p => p.CreatedByUserId)
```

```
.OnDelete(DeleteBehavior.Restrict);
```

```
builder.Entity<ProjectTask>()  
    .HasOne(t => t.Project)  
    .WithMany(p => p.Tasks)  
    .HasForeignKey(t => t.ProjectId);
```

```
builder.Entity<ProjectTask>()  
    .HasOne(t => t.AssignedUser)  
    .WithMany(u => u.AssignedTasks)  
    .HasForeignKey(t => t.AssignedUserId)  
    .OnDelete(DeleteBehavior.Restrict);
```

```
builder.Entity<Comment>()  
    .HasOne(c => c.Task)  
    .WithMany(t => t.Comments)  
    .HasForeignKey(c => c.TaskID);
```

```
builder.Entity<Comment>()  
    .HasOne(c => c.Author)  
    .WithMany(u => u.Comments)  
    .HasForeignKey(c => c.AuthorID)  
    .OnDelete(DeleteBehavior.Restrict);
```

```
builder.Entity<TaskTag>()  
    .HasKey(tt => new { tt.TaskId, tt.TagId });  
builder.Entity<TaskTag>()  
    .HasOne(tt => tt.Task)  
    .WithMany(t => t.TaskTags)
```

```
.HasForeignKey(tt => tt.TaskId);
```

```
builder.Entity<TaskTag>()
```

```
.HasOne(tt => tt.Tag)
```

```
.WithMany(t => t.TaskTags)
```

```
.HasForeignKey(tt => tt.TagId);
```

```
builder.Entity<ProjectTask>().HasKey(t => t.TaskId);
```

```
builder.Entity<Project>().HasKey(p => p.ProjectId);
```

```
builder.Entity<Comment>().HasKey(c => c.CommentId);
```

```
builder.Entity<Tag>().HasKey(t => t.TagId);
```

```
builder.Entity<Role>().HasKey(r => r.Id);
```

```
}
```

```
}
```

```
}
```

Лістинг коду класу міграції Initial Create

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: «Users»,
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation(«SqlServer:Identity», «1, 1»),
                UserName = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey(«PK_Users», x => x.Id);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: «Users»);
    }
}
```

Лістинг коду контролера UserController

```
namespace TaskManager.Controllers
{
    [Authorize(Roles = «Admin»)]
    public class UserController : Controller
    {
        private readonly IUserService _service;
        private readonly UserManager<User> userManager;
        private readonly RoleManager<Role> roleManager;

        public UserController(IUserService service, RoleManager<Role> roleManager,
            UserManager<User> userManager)
        {
            _service = service;
            this.roleManager = roleManager;
            this.userManager = userManager;
        }

        public async Task<IActionResult> Index()
        {
            var users = await userManager.Users.ToListAsync();
            var model = new List<UserWithRolesViewModel>();
            foreach (var user in users)
            {
                var roles = await userManager.GetRolesAsync(user);
                model.Add(new UserWithRolesViewModel
                {
                    Id = user.Id,
                    FullName = user.FullName,
                    Email = user.Email,
```

```

        RoleName = roles.FirstOrDefault() ?? «—»
    });}
    return View(model);
}

public async Task<IActionResult> Edit(int id)
{
    var user = await userManager.FindByIdAsync(id.ToString());
    if (user is null) return NotFound();
    var roles = await roleManager.Roles.ToListAsync();
    var currentRoles = await userManager.GetRolesAsync(user);
    return View(new UserWithRolesViewModel
    {
        Id = user.Id,
        FullName = user.FullName,
        Email = user.Email,
        RoleName = currentRoles.FirstOrDefault(),
        AvailableRoles = roles.Select(role => new SelectListItem { Text =
role.Name, Value = role.Name }).ToList(),
    });
}

[HttpPost]
public async Task<IActionResult> Edit(UserWithRolesViewModel model)
{
    var user = await userManager.FindByIdAsync(model.Id.ToString());
    if (user is null) return NotFound();
    user.FullName = model.FullName;
    user.Email = model.Email;
    user.UserName = model.Email;
    user.NormalizedEmail = model.Email.ToUpper();
}

```

```

user.NormalizedUserName = model.Email.ToUpper();

var currentRoles = await userManager.GetRolesAsync(user);
await userManager.RemoveFromRolesAsync(user, currentRoles);
await userManager.AddToRoleAsync(user, model.RoleName);
await userManager.UpdateAsync(user);
return RedirectToAction(nameof(Index));
}

public async Task<IActionResult> Details(int id)
{
    var result = await userManager.FindByIdAsync(id.ToString());
    var role = await userManager.GetRolesAsync(result);
    if (result == null) return NotFound();
    return View(new UserDto
    {
        Id = result.Id,
        FullName = result.FullName,
        Email = result.Email,
        RoleName = role.FirstOrDefault()
    });
}
}

```

Лістинг коду сторінки логіну

```

@model TaskManager.Models.LoginViewModel
@{
    ViewData["Title"] = "Login";
}
<h2>Вхід</h2>

<form asp-action="Login">
    <div class="form-group">
        <label asp-for="Email"></label>
        <input asp-for="Email" class="form-control" />
        <span asp-validation-for="Email" class="text-danger"></span>
    </div>

    <div class="form-group">
        <label asp-for="Password"></label>
        <input asp-for="Password" class="form-control" />
        <span asp-validation-for="Password" class="text-danger"></span>
    </div>

    <div class="form-check">
        <input asp-for="RememberMe" class="form-check-input" />
        <label asp-for="RememberMe" class="form-check-label"></label>
    </div>

    <button type="submit" class="btn btn-primary">Вхід</button>
</form>

```

Лістинг коду сторінки реєстрації

```
@model TaskManager.Models.RegisterViewModel
@{
    ViewData["Title"] = "Реєстрація";
}

<h2>Реєстрація</h2>

<form asp-action="Register">
    <div class="form-group">
        <label asp-for="FullName"></label>
        <input asp-for="FullName" class="form-control" />
        <span asp-validation-for="FullName" class="text-danger"></span>
    </div>

    <div class="form-group">
        <label asp-for="Email"></label>
        <input asp-for="Email" class="form-control" />
        <span asp-validation-for="Email" class="text-danger"></span>
    </div>

    <div class="form-group">
        <label asp-for="Password"></label>
        <input asp-for="Password" class="form-control" />
        <span asp-validation-for="Password" class="text-danger"></span>
    </div>

    <div class="form-group">
```

Продовження додатка Д.2

```
<label asp-for="ConfirmPassword"></label>
<input asp-for="ConfirmPassword" class="form-control" />
<span asp-validation-for="ConfirmPassword" class="text-danger"></span>
</div>

<button type="submit" class="btn btn-primary">Зареєструватися</button>
</form>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

Лістинг коду навігаційного меню

```

<nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white
border-bottom box-shadow mb-3">
    <div class="container-fluid">
        <a class="navbar-brand" asp-area="" asp-controller="Home" asp-
action="Index">TaskManager</a>
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-
bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"
aria-expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
            <ul class="navbar-nav flex-grow-1">
                <li class="nav-item">
                    <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Index">Головна</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Privacy">Приватність</a>
                </li>
                @if (SignInManager.IsSignedIn(User))
                {
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-controller="Project" asp-
action="Index">Проекту</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-controller="Task" asp-
action="Index">Завдання</a>
                    </li>
                }
            </ul>
        </div>
    </div>
</nav>

```

```

</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-controller="Account" asp-
action="Profile">Профіль</a>
</li>
@if (User.IsInRole("Admin"))
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-controller="User" asp-
action="Index">Користувачі</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-controller="Settings" asp-
action="Index">Налаштування системи</a>
    </li>
}

<li class="nav-item">
    <a class="nav-link text-dark" asp-controller="Help" asp-
action="FAQ">Про систему / FAQ</a>
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-controller="Contact" asp-
action="Index">Контакти</a>
</li>
}
@if (User.Identity.IsAuthenticated)
{
    <li class="nav-item">
        <a class="nav-link text-dark">@User.Identity.Name</a>

```

```

        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-controller="Account" asp-
action="Logout">Вийти</a>
        </li>
    }
    else
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-controller="Account" asp-
action="Login">Увійти</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-controller="Account" asp-
action="Register">Зареєструватися</a>
        </li>
    }
</ul>

</div></div></nav>

```

Лістинг коду Project Create.cshtml

```
@using TaskManager.dto
```

```
@model ProjectDto
```

```
@{
```

```
    ViewData["Title"] = "Create Project";
```

```
}
```

```
<div class="container py-5 bg-dark text-light min-vh-100">
```

```
    <div class="row justify-content-center">
```

```
        <div class="col-md-8">
```

```
            <h2 class="mb-4">Create New Project</h2>
```

```
            <form asp-action="Create" method="post" class="bg-secondary p-4 rounded shadow">
```

```
                <div class="mb-3">
```

```
                    <label asp-for="Title" class="form-label">Title</label>
```

```
                    <input asp-for="Title" class="form-control bg-dark text-light border-light" placeholder="Enter project title" />
```

```
                    <span asp-validation-for="Title" class="text-danger"></span>
```

```
                </div>
```

```
                <div class="mb-3">
```

```
                    <label asp-for="Description" class="form-label">Description</label>
```

```
                    <textarea asp-for="Description" class="form-control bg-dark text-light border-light" rows="4" placeholder="Describe your project"></textarea>
```

```
                    <span asp-validation-for="Description" class="text-danger"></span>
```

```
                </div>
```

```

<div class="mb-3">
    <label asp-for="CreatedAt" class="form-label">Creation Date</label>
    <input asp-for="CreatedAt" class="form-control bg-dark text-light
border-light" type="datetime-local" />
    <span asp-validation-for="CreatedAt" class="text-danger"></span>
</div>

<div class="d-flex justify-content-between">
    <a asp-action="Index" class="btn btn-outline-light">Back to List</a>
    <button type="submit" class="btn btn-primary">Create</button>
</div>
</form>
</div>
</div>
</div>

@section Scripts {
    @{
        await Html.RenderPartialAsync("_ValidationScriptsPartial");
    }
}

```

Лістинг коду Project Details.cshtml

```

@model ProjectDto
@using Microsoft.AspNetCore.Identity
@using TaskManager.dto
@inject UserManager<User> UserManager

@{
    ViewData["Title"] = "Project Details";
    var isAdmin = User.IsInRole("Admin");
    var userId = UserManager.GetUserId(User);
    var statuses = new[] { "ToDo", "InProgress", "Done" };
}

@section Scripts {
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/Sortable/1.15.0/Sortable.min.js"></script
>

    <script>
        document.addEventListener("DOMContentLoaded", function () {
            const containers = document.querySelectorAll(".task-column");
            containers.forEach(column => {
                new Sortable(column, {
                    group: "tasks",
                    animation: 150,
                    onEnd: function (evt) {
                        const taskId = evt.item.dataset.taskId;
                        const newStatus = evt.to.dataset.status;

                        fetch(`/Task/UpdateStatus/${taskId}?newStatus=${newStatus}`, {

```

```

        method: "POST"
    }).then(r => {
        if (!r.ok) alert("Помилка при оновленні статусу");
    });
    }
    });
    });
    });
</script>
}

<div class="container py-5 bg-dark text-light min-vh-100">
    <div class="mb-4 d-flex justify-content-between align-items-center">
        <h1 class="display-4">@Model.Title</h1>
        <a asp-action="Index" class="btn btn-outline-light btn-lg">Назад до
        списку</a>
    </div>

    <div class="card bg-secondary text-white shadow-lg mb-5">
        <div class="card-body">
            <h5 class="card-subtitle mb-3 text-light">
                Проект створено: @Model.CreatedAt.ToString("yyyy-MM-dd") | Автор:
                @Model.CreatedByUserName
            </h5>
            <p class="card-text fs-5">@Model.Description</p>

            <div class="d-flex gap-3 mt-3">
                @if (Model.CreatedById.ToString() == userId || isAdmin)
                {

```

```

        <a asp-action="Edit" asp-route-id="@Model.Id" class="btn btn-
warning">Редагувати</a>

        <a asp-action="Delete" asp-route-id="@Model.Id" class="btn btn-
danger">Видалити</a>

    }

</div>

</div>

</div>

<div class="row">

    @foreach (var status in statuses)
    {
        <div class="col-md-4">

            <div class="card bg-secondary text-white mb-4">

                <div class="card-header d-flex justify-content-between align-items-
center">

                    <h4 class="mb-0">@status</h4>

                    <!-- Кнопка для відкриття форми -->

                    <button class="btn btn-sm btn-success"
onclick="toggleForm('@status')">+</button>

                </div>

                <div class="card-body task-column" data-status="@status" style="min-
height: 300px;">

                    @foreach (var task in Model.Tasks.Where(t => t.Status == status))
                    {

                        <div class="card bg-dark text-white mb-2 p-2" data-task-
id="@task.Id">

                            <div class="d-flex justify-content-between">

                                <h5 class="card-title">@task.Title</h5>

                                <div>

```

Продовження додатка 3

```

        <a asp-controller="Task" asp-action="Edit" asp-route-
id="@task.Id" class="btn btn-sm btn-warning">■</a>

        <form asp-controller="Task" asp-action="Delete" asp-
route-id="@task.Id" method="post" class="d-inline">

            <button type="submit" class="btn btn-sm btn-
danger">🗑️</button>

        </form>

    </div>

</div>

<p class="card-text">@task.Description</p>
<p class="card-text">@task.DueDate</p>
</div>

}
</div>

<!-- Форма створення нової задачі -->
<div class="card-footer d-none" id="form-@status">

    <form asp-controller="Task" asp-action="CreateInColumn"
method="post">

        <input type="hidden" name="ProjectId" value="@Model.Id" />
        <input type="hidden" name="Status" value="@status" />
        <input type="hidden" name="CreatedByUserId"
value="@UserManager.GetUserId(User)" />
        <div class="form-group">

            <input type="text" name="Title" class="form-control mb-2"
placeholder="Назва задачі" required />

            <textarea name="Description" class="form-control mb-2"
placeholder="Onuc"></textarea>

            <button type="submit" class="btn btn-sm btn-
primary">Створити</button>

        </div>

```

```
        </form>
      </div>
    </div>
  </div>
}
</div>
</div>

<script>
  function toggleForm(status) {
    const form = document.getElementById(`form-${status}`);
    form.classList.toggle("d-none");
  }
</script>
```

Лістинг коду класу TaskServiceTests

```
public class TaskServiceTests
{
    private TaskManagerContext GetInMemoryContext()
    {
        var options = new DbContextOptionsBuilder<TaskManagerContext>()
            .UseInMemoryDatabase(databaseName: Guid.NewGuid().ToString())
            .Options;
        return new TaskManagerContext(options);
    }

    [Fact]
    public async Task CreateTaskAsync_AddsTaskToDatabase()
    {
        // Arrange
        using var context = GetInMemoryContext();
        var service = new TaskService(context);
        var dto = new TaskDto
        {
            Title = "Test Task",
            Description = "Testing",
            Status = "New",
            Priority = "High",
            ProjectId = 1,
            AssignedUserId = 2,
            TagIds = new List<int> { 10, 20 }
        };
        // Act
```

```

var result = await service.CreateTaskAsync(dto);

// Assert
var taskInDb = await context.Tasks.Include(t => t.TaskTags).FirstOrDefaultAsync();
Assert.NotNull(taskInDb);
Assert.Equal("Test Task", taskInDb.Title);
Assert.Equal(2, taskInDb.AssignedUserId);
Assert.Equal(2, taskInDb.TaskTags.Count);
}

[Fact]
public async Task GetTaskByIdAsync_ReturnsCorrectTask()
{
    using var context = GetInMemoryContext();
    var task = new ProjectTask
    {
        Title = "Task A",
        Status = "New",
        Priority = "Low",
        ProjectId = 1,
        TaskTags = new List<TaskTag> { new TaskTag { TagId = 1 } }
    };
    context.Tasks.Add(task);
    await context.SaveChangesAsync();
    var service = new TaskService(context);
    var result = await service.GetTaskByIdAsync(task.TaskId);

    Assert.NotNull(result);
    Assert.Equal("Task A", result!.Title);

```

```
Assert.Single(result.TagIds!);
}
```

[Fact]

```
public async Task UpdateTaskAsync_UpdatesFieldsCorrectly()
{
    using var context = GetInMemoryContext();

    var task = new ProjectTask { Title = "Old Title", ProjectId = 1, Status = "New",
        Priority = "Low" };
    context.Tasks.Add(task);
    await context.SaveChangesAsync();

    var service = new TaskService(context);
    var dto = new TaskDto
    {
        Id = task.TaskId,
        Title = "Updated Title",
        Description = "Updated Desc",
        Status = "In Progress",
        Priority = "High",
        DueDate = DateTime.UtcNow,
        ProjectId = 1
    };

    var success = await service.UpdateTaskAsync(dto);
    var updated = await context.Tasks.FindAsync(task.TaskId);
    Assert.True(success);
    Assert.Equal("Updated Title", updated!.Title);
    Assert.Equal("High", updated.Priority);
}
```

[Fact]

```
public async Task DeleteTaskAsync_RemovesTask()
{
    using var context = GetInMemoryContext();
    var task = new ProjectTask { Title = "To Delete", ProjectId = 1 };
    context.Tasks.Add(task);
    await context.SaveChangesAsync();

    var service = new TaskService(context);
    var result = await service.DeleteTaskAsync(task.TaskId);

    Assert.True(result);
    Assert.Empty(context.Tasks);
}
```

[Fact]

```
public async Task GetTasksByProjectAsync_ReturnsOnlyProjectTasks()
{
    using var context = GetInMemoryContext();
    context.Tasks.AddRange(
        new ProjectTask { Title = "P1-Task1", ProjectId = 1 },
        new ProjectTask { Title = "P1-Task2", ProjectId = 1 },
        new ProjectTask { Title = "P2-Task1", ProjectId = 2 }
    );
    await context.SaveChangesAsync();

    var service = new TaskService(context);
    var result = (await service.GetTasksByProjectAsync(1)).ToList();
}
```

```
Assert.Equal(2, result.Count);  
Assert.All(result, t => Assert.Equal(1, t.ProjectId));  
}  
}
```

Лістинг коду класу UserControllerTests

```
public class UserControllerTests
{
    private Mock<IUserStore<User>> userStoreMock = new
    Mock<IUserStore<User>>();

    private Mock<UserManager<User>> GetMockUserManager()
    {
        return new Mock<UserManager<User>>(userStoreMock.Object, null, null, null, null,
        null, null, null, null);
    }

    private Mock<RoleManager<Role>> GetMockRoleManager()
    {
        var roleStoreMock = new Mock<IRoleStore<Role>>();
        return new Mock<RoleManager<Role>>(roleStoreMock.Object, null, null, null,
        null);
    }

    [Fact]
    public async Task Index_ReturnsViewWithUsersAndRoles()
    {
        var userManagerMock = GetMockUserManager();
        var roleManagerMock = GetMockRoleManager();
        var serviceMock = new Mock<IUserService>();

        var users = new List<User>
        {
            new User { Id = 1, FullName = "User One", Email = "user1@test.com" },
        }
```

```

new User { Id = 2, FullName = "User Two", Email = "user2@test.com" }
}.AsQueryable();

userManagerMock.Setup(u => u.Users).Returns(users);

userManagerMock.Setup(u
u.GetRolesAsync(It.IsAny<User>())).ReturnsAsync(new List<string> { "Admin" });

var controller = new UserController(serviceMock.Object, roleManagerMock.Object,
userManagerMock.Object);

var result = await controller.Index();

var viewResult = Assert.IsType<ViewResult>(result);
var model
Assert.IsAssignableFrom<List<UserWithRolesViewModel>>(viewResult.Model);
Assert.Equal(2, model.Count);
Assert.All(model, item => Assert.Equal("Admin", item.RoleName));
}

[Fact]
public async Task Edit_Get_ReturnsUserWithRoles()
{
var userManagerMock = GetMockUserManager();
var roleManagerMock = GetMockRoleManager();
var serviceMock = new Mock<IUserService>();
var user = new User { Id = 1, FullName = "User One", Email = "user1@test.com" };
var roles = new List<Role> { new Role { Name = "Admin" }, new Role { Name =
"User" } };

userManagerMock.Setup(u => u.FindByIdAsync("1")).ReturnsAsync(user);

```

Продовження додатка Л

```
userManagerMock.Setup(u => u.GetRolesAsync(user)).ReturnsAsync(new
List<string> { "Admin" });
```

```
roleManagerMock.Setup(r => r.Roles).Returns(roles.AsQueryable());
```

```
var controller = new UserController(serviceMock.Object, roleManagerMock.Object,
userManagerMock.Object);
```

```
var result = await controller.Edit(1);
```

```
var viewResult = Assert.IsType<ViewResult>(result);
```

```
var model = Assert.IsAssignableFrom<UserWithRolesViewModel>(viewResult.Model);
```

```
Assert.Equal("User One", model.FullName);
```

```
Assert.Equal("Admin", model.RoleName);
```

```
Assert.Equal(2, model.AvailableRoles.Count);
```

```
}
```

```
[Fact]
```

```
public async Task Edit_Post_UpdatesUserRole()
```

```
{
```

```
var userManagerMock = GetMockUserManager();
```

```
var roleManagerMock = GetMockRoleManager();
```

```
var serviceMock = new Mock<IUserService>();
```

```
var user = new User { Id = 1, FullName = "User One", Email = "user1@test.com" };
userManagerMock.Setup(u => u.FindByIdAsync("1")).ReturnsAsync(user);
```

```
userManagerMock.Setup(u => u.GetRolesAsync(user)).ReturnsAsync(new
List<string> { "User" });
```

Продовження додатка Л

```
userManagerMock.Setup(u => u.RemoveFromRolesAsync(user,
It.IsAny<IEnumerable<string>>())).ReturnsAsync(IdentityResult.Success);
```

```
userManagerMock.Setup(u => u.AddToRoleAsync(user,
"Admin")).ReturnsAsync(IdentityResult.Success);
```

```
userManagerMock.Setup(u =>
u.UpdateAsync(user)).ReturnsAsync(IdentityResult.Success);
```

```
var controller = new UserController(serviceMock.Object, roleManagerMock.Object,
userManagerMock.Object);
```

```
var model = new UserWithRolesViewModel
```

```
{
```

```
Id = 1,
```

```
FullName = "User One",
```

```
Email = "user1@test.com",
```

```
RoleName = "Admin"
```

```
};
```

```
var result = await controller.Edit(model);
```

```
var redirect = Assert.IsType<RedirectToActionResult>(result);
```

```
Assert.Equal("Index", redirect.ActionName);
```

```
}
```

```
[Fact]
```

```
public async Task Details_ReturnsCorrectUser()
```

```
{
```

```
var userManagerMock = GetMockUserManager();
```

```
var roleManagerMock = GetMockRoleManager();
```

```
var serviceMock = new Mock<IUserService>();
```

Продовження додатка Л

```
var user = new User { Id = 1, FullName = "User One", Email = "user1@test.com" };
userManagerMock.Setup(u => u.FindByIdAsync("1")).ReturnsAsync(user);
userManagerMock.Setup(u => u.GetRolesAsync(user)).ReturnsAsync(new
List<string> { "User" });

var controller = new UserController(serviceMock.Object, roleManagerMock.Object,
userManagerMock.Object);

var result = await controller.Details(1);
var viewResult = Assert.IsType<ViewResult>(result);
var model = Assert.IsType<UserDto>(viewResult.Model);
Assert.Equal("User One", model.FullName);
Assert.Equal("user1@test.com", model.Email);
Assert.Equal("User", model.RoleName);
}
}
```