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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Система для управління задачами і відстеження помилок проекту у хмарному середовищі

Виконав : студент 4 курсу, групи ІП-93
(шифр групи)

Трембач Анастасія Дмитрівна
(прізвище, ім’я, по батькові) (підпис)

Керівник ст. викладач, доктор філософії, Таран В. І.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) доцент, к.т.н. Волокита А. М.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент доцент, к.т.н. Шимкович В. М.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

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

Київ – 2023 р.

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

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Трембач Анастасія Дмитрівна

1. Тема проєкту *Система для управління задачами і відстеження помилок проекту у хмарному середовищі*
керівник проєкту Таран Владислав Ігорович, ст. викладач, доктор філософії ,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 31 травня 2023 року №2101-с

2. Термін здачі студентом закінченого проєкту *5 червня 2023 р.*

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

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Огляд систем для управління задачами проекту і відстеження помилок проекту

Розділ 2. Вибір та обґрунтування технологій для розробки системи

Розділ 3. Архітектурний аналіз та проектування системи

Розділ 4. Розгортання і тестування системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структура взаємодії компонентів (структурна схема), діаграма класів (функціональна схема), алгоритми системи (принципова схема).

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

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А. М.		

7. Дата видачі завдання «24» грудня 2022 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*20.12.2022-25.12.2022*	
2.	*Вивчення та аналіз завдання*	*25.12.2022-15.03.2023*	
3.	*Розробка архітектури та загальної структури системи*	*15.03.2023-25.03.2023*	
4.	*Розробка структур окремих підсистем*	*25.03.2023-5.04.2023*	
5.	*Програмна реалізація системи*	*5.04.2023-15.04.2023*	
6.	*Оформлення пояснювальної записки*	*15.04.2023-24.05.2023*	
7.	*Захист програмного продукту*	*25.04.2023*	
8.	*Передзахист*	*31.05.2023*	
9.	*Захист*	*24.06.2023*	

Студент-дипломник _____ Анастасія ТРЕМБАЧ
(підпис)

Керівник проєкту _____ Владислав ТАРАН
(підпис)

АНОТАЦІЯ

Дипломна робота присвячена розробці системи для управління задачами і відстеження помилок проекту з використанням хмарного середовища. Метою дослідження є створення масштабованої системи, яка дозволяє ефективно керувати задачами, відстежувати та вирішувати помилки, що виникають під час проектної діяльності. У роботі було проведено аналіз існуючих рішень для управління задачами та відстеження помилок у проектах. З метою вдосконалення системи було запропоновані покращення, які враховують ці недоліки. У роботі були досліджені сучасні технології розробки. Основним результатом роботи є розроблена та розгорнута система для управління задачами і відстеження помилок у проекту у хмарному середовищі, яка відповідає сучасним вимогам та потребам організацій, що займаються проектною діяльністю.

ANNOTATION

The diploma thesis is dedicated to the development of a system for task management and error tracking in a project using cloud environment. The research aims to create a scalable system that efficiently manages tasks and tracks and resolves errors that occur during project activities. The thesis includes an analysis of existing solutions for task management and error tracking in projects. In order to enhance the system, improvements addressing these shortcomings have been proposed. The study explores modern development technologies. The main outcome of the thesis is the developed and deployed system for task management and error tracking in a project within a cloud environment, which meets the modern requirements and needs of organizations engaged in project activities.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Система для управління	4		
			задачами і відстеження			
			помилок проекту у			
			хмарному середовищі			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Система для управління	88		
			задачами і відстеження			
			помилок проекту у			
			хмарному середовищі			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Система для управління	1		
			задачами і відстеження			
			помилок проекту у			
			хмарному середовищі			
			Структура взаємодії компо-			
			нентів (структурна схема)			
	А4	***ІАЛЦ.4672008.005 Д2***	Система для управління	1		
			задачами і відстеження			
			помилок проекту у			
			хмарному середовищі			
			Діаграма класів			
			(функціональна схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Система для управління	1		
			задачами і відстеження			
			помилок проекту у			
			хмарному середовищі			
			Алгоритми системи			
			(принципова схема)			
	А4	***ІАЛЦ.467200.007 Д4***	Система для управління	36		
			задачами і відстеження			
			помилок проекту у			
			хмарному середовищі			
			Текст програмного коду			

Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	

ІАЛЦ.467200.001 ОА

Розроб		Трембач А.Д.			*Система для управління задачами і відстеження помилок проекту у хмарному середовищі* Опис альбому	Літ.	Аркуш 1 / Аркушів 1
Перев		Таран В.І.					
						КПІ ім. Ігоря Сікорського ФІОТ ІП-93	

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система для управління задачами і відстеження помилок проекту у хмарному середовищі*»

Київ – 2023

ЗМІСТ

					ІАЛЦ.467200.002 ТЗ		
		№ докум.	Підпис	Дата			
Розробив		Трембач А. Д.			**Система для управління задачами і відстеження помилок проекту у хмарному середовищі** Технічне завдання	Літ.	Аркуш 1 / Аркушів 4
Перевірив		Таран В.І.					
						КПІ ім. Ігоря Сікорського ФІОТ ІП-93	
Н. Контр.		Волокита А. М.					
Затвердив							

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Це технічне завдання стосується розробки системи для управління задачами і відстеження помилок у хмарному середовищі. Крім того, передбачається надання подальшої підтримки розробленої системи.

Система призначена для використання в проектній діяльності організацій, що потребують ефективного управління задачами та відстеження та вирішення помилок, що виникають під час виконання проектів. Вона може бути застосована у будь-якій сфері, де важлива організація робочих процесів, керування завданнями та ефективне вирішення проблем.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

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

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням розробки є створення ефективної та масштабованої системи для управління задачами і відстеження помилок у проекті, з використанням хмарного середовища. Система повинна надавати зручні інструменти для керування задачами, відстеження помилок та аналізу даних.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерела розробки для системи для управління задачами і відстеження помилок у проекту у хмарному середовищі включають література та

документацію, офіційні документації, онлайн-ресурси, такі як блоги, форуми та спільноти розробників, для отримання додаткової інформації, порад та рекомендацій щодо розробки систем, вирішення проблем та вдосконалення робочих процесів.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Система повинна забезпечувати зручний та інтуїтивно зрозумілий інтерфейс для користувачів.
- Система повинна бути масштабованою та ефективною, здатною обробляти великий обсяг даних та велику кількість користувачів.
- Система повинна забезпечувати безпеку даних, включаючи захист від несанкціонованого доступу та резервне копіювання даних.
- Система повинна бути доступною через хмарне середовище з різних пристроїв з підключенням до Інтернету.
- Система повинна бути легко розширюваною та модульною, з можливістю додавання нових функціональних можливостей в майбутньому.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Visual Studio 2019 IDE версії або вище.

5.3. Вимоги до апаратної частини

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

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	20.12.2022-25.12.2022
Вивчення та аналіз завдання	25.12.2022-15.03.2023
Розробка архітектури та загальної структури системи	15.03.2023-25.03.2023
Розробка структур окремих частин системи	25.03.2023-5.04.2023
Програмна реалізація системи	5.04.2023-15.04.2023
Виправлення помилок	15.04.2023-20.05.2023
Оформлення пояснювальної записки	25.04.2023

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система для управління задачами і відстеження помилок проекту у хмарному середовищі*»

Київ – 2023

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

DDD	Domain-Driven Design
CQRS	Command Query Responsibility Segregation
DTO	Data Transfer Object
ORM	Object-Relational Mapping
SQL	Structured Query Language
API	Application Programming Interface
CI/CD	Continuous Integration and Continuous Delivery

ВСТУП

Сьогоднішній світ проектного менеджменту вимагає не тільки активного керування завданнями, але й вміння швидко реагувати на помилки, що виникають під час реалізації проектів. На допомогу в цьому приходять системи управління задачами та відстеження помилок.

Системи управління задачами допомагають організовувати роботу над проектом, розподіляти завдання між учасниками команди, встановлювати терміни виконання та відстежувати прогрес. Вони допомагають створювати чіткий план, слідкувати за завданнями, щоб не загубити орієнтацію та завжди бути в курсі всіх робіт.

Ці системи є незамінним помічником у виявленні, реєстрації та вирішенні проблем, що з'являються під час проектування. Вони дозволяють зберігати повний звіт про всі помилки, їх причини і шляхи виправлення. А це дуже корисно для поліпшення роботи, уникнення повторних помилок і забезпечення більш ефективного управління ризиками.

Та, що особливо цікаво, ці системи створюють спільну платформу для спілкування та співпраці всієї команди проекту. З їхньою допомогою команда може обмінюватись інформацією, коментарями, документами та вести дискусії про завдання та проблеми. А це призводить до покращення комунікації, зменшення непорозумінь і підвищення ефективності спільної роботи.

Ну і на додаток до всього цього, системи управління задачами та відстеження помилок допомагають збирати і аналізувати дані про прогрес проекту, продуктивність команди та результативність роботи. Вони надають необхідну інформацію для прийняття рішень та постійного покращення.

Отже, метою даної дипломної роботи є розробка ефективної та масштабованої системи, яка дозволить організаціям ефективно керувати задачами, відстежувати та вирішувати помилки, що виникають під час

проектної діяльності. Головним завданням є розробка комплексного рішення, яке поєднає в собі зручний інтерфейс для користувачів, потужні механізми управління завданнями, відстеження помилок та аналізу проектної діяльності. У процесі розробки системи було проведено аналіз існуючих рішень для управління задачами та відстеження помилок у проектах.

Також, ціль роботи розробити систему з розгортанням у хмарному середовищі. Хмарні середовища надають безліч переваг, зокрема, масштабованість, доступність, надійність та зручність у розгортанні та керуванні системами. У цьому контексті розробка системи для управління задачами і відстеження помилок у хмарному середовищі стає актуальною задачею.

Розроблена система сприяла підвищенню продуктивності та якості проектної діяльності, спрощенню процесу спільної роботи та забезпеченню ефективного контролю над проектами.

РОЗДІЛ 1. ОГЛЯД СИСТЕМ ДЛЯ УПРАВЛІННЯ ЗАДАЧАМИ І ВІДСТЕЖЕННЯ ПОМИЛОК ПРОЕКТУ

1.1 Визначення і призначення систем для управління задачами

Системи для управління завданнями проектів це фактично цілий напрямок у програмному забезпеченні, який включає комплекс додатків для планування і відстеження завдань, управління матеріальними і часовими ресурсами, організацію командної роботи та спільного управління проектами.

Це можуть бути як звичайні списки або календарі для організації невеликих проектів, так і більш потужні інструменти, які дають змогу керувати повним життєвим циклом проекту та всією командою [1]. Але при такому розмаїтті продуктів основна мета систем для організації завдань проектів одна і та ж. І полягає вона в тому, щоб надати інструмент, що дозволяє ефективно та швидко здійснювати управління, контролювати робочий процес, роблячи його прозорим, упорядкованим та зрозумілим.

Вже далеко в минулому ті часи, коли термін від ідеї до продукту вимірювався кількома роками та реалізовувався індивідуально або групами з двох, трьох, рідко більше співробітників.

У сьогоднішньому інформаційному просторі, що стрімко розвивається, в умовах жорсткої конкурентної боротьби, лише максимально ефективне використання ресурсів, та застосування найсучасніших технологій може дозволити успішно реалізувати проект.

У таких умовах, система для управління завданнями є незамінним інструментом для ефективної та продуктивної організації роботи проекту. Що дозволяє членам команди працювати швидше і більш організовано, залишатися

на вірному шляху, виконувати завдання вчасно, з меншим ризиком помилок та упущень.

Команди, що користуються такими системами, зазвичай показують більш швидкі та якісні результати, що, у свою чергу, сприяє їхньому успішному розвитку та конкурентоспроможності.

Як згадувалося, таких програм досить багато, але незважаючи на загальні принципи, набір і реалізація функцій відрізняється. Тому обрати вірну програму для організації роботи проекту може бути складно. Перед тим, як вибрати необхідне програмне забезпечення необхідно врахувати різні фактори, такі як: розмір команди, бюджет та потреби проекту.

1.2 Огляд існуючих систем для управління задачами проекту

1.2.1 Asana

«Asana є лідером у сфері програмного забезпечення для колективного управління проектами для команд будь-якого розміру та географічного розташування. Asana допомагає вам, вашій команді та зацікавленим сторонам швидше та ефективніше співпрацювати для досягнення бізнес-цілей.» [2] – Так описують цю платформу її розробники на офіційному сайті.

Asana – популярна платформа для управління проектами та задачами. Нижче представлені основні можливості і особливості цієї платформи.

Asana надає можливість створювати необмежену кількість проектів для зручності команди. Користувачі можуть створювати необмежену кількість завдань в межах кожного проекту. Платформа дозволяє встановлювати дедлайни для кожної задачі та задавати їй пріоритет, щоб забезпечити якісне і своєчасне виконання. Кожен учасник команди може фільтрувати та сортувати

задачі за різними параметрами, щоб швидко знайти необхідну інформацію і вчасно нею скористатися.

Зручним і важливим інструментом є інформаційна панель(Dashboard) (див. рисунок 1.1) для наочної демонстрації прогресу виконання задач проекту у вигляді узагальненої інформації щодо кількості виконаних і запланованих завдань, а також у вигляді графіків і діаграм.

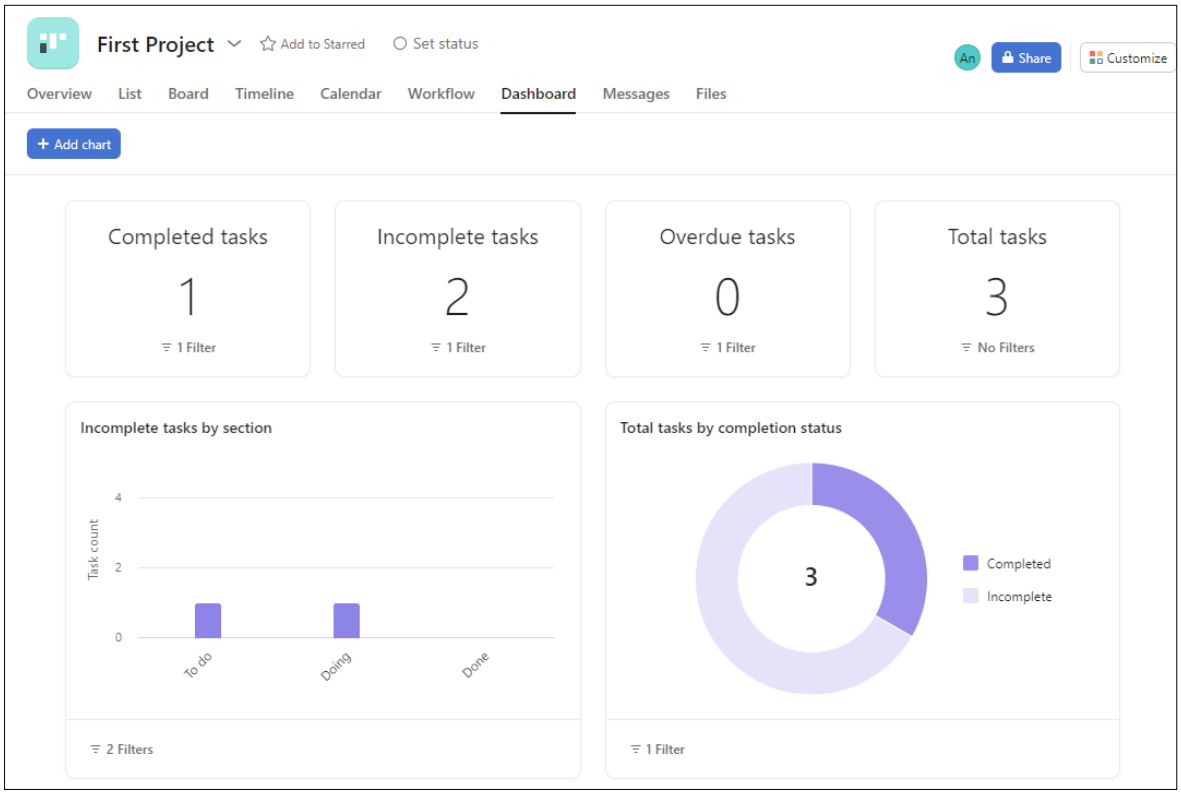


Рисунок 1.1 – Інформаційна панель платформи Asana

Система надає можливість відображати список задач у різних форматах:

- у вигляді списку(List), де задачі розділені на секції і є колонки для відображення додаткової інформації про задачу, наприклад її пріоритет та прогрес;
- у вигляді дошки(Board), де задачі розділені на три колонки: зробити, в процесі, зроблено;
- у вигляді часової шкали(Timeline), де задачі розміщені на шкалі за часовим станом, що добре показує початок, кінець і тривалість завдання;

- у вигляді календаря(Calendar), де показуються час виконання задачі відповідно до дати.

Система Asana має вбудований месенджер для зручної і швидкої комунікації команди. До того ж, під будь якою задачею можна залишити коментар чи задати уточнююче питання [3]. Крім того, у платформи Asana є можливість інтегрувати у роботу інші застосунки.

Asana не є безкоштовним застосунком. Система пропонує пробний період роботи для всіх користувачів. Після якого користувачам пропонується купити розширений платний тариф використання системи.

Отже, Asana має зручний і інтуїтивно зрозумілий інтерфейс, дає можливість використовувати різноманітні шаблони задач, а також містить багато інтеграції з іншими застосунками. Однак, для отримання доступу до деякого функціоналу необхідно придбати платний план. Також, можливість відслідковування часу, витраченого на завдання, обмежена.

1.2.2 Jira

Jira - це платформа для управління проектами та задачами, яка надає широкий спектр інструментів для зручності командної роботи [4]. Основні можливості та особливості платформи включають:

1. Створення проектів: Jira дозволяє створювати необмежену кількість проектів для зручності команди. Користувачі можуть створювати необмежену кількість завдань в межах кожного проекту.

2. Дедлайни та пріоритети: Платформа дозволяє встановлювати дедлайни для кожної задачі та задавати їй пріоритет, щоб забезпечити якісне та своєчасне виконання.

3. Інформаційна панель: Jira має інформаційну панель для наочної демонстрації прогресу виконання задач проекту у вигляді узагальненої

інформації, щодо кількості виконаних та запланованих завдань, а також у вигляді графіків та діаграм.

4. Різні формати відображення: Система надає можливість відображення списку задач у різних форматах, включаючи список(List), дошку(Board) (на рисунку 1.2), часову шкалу(Timeline) та календар(Calendar).

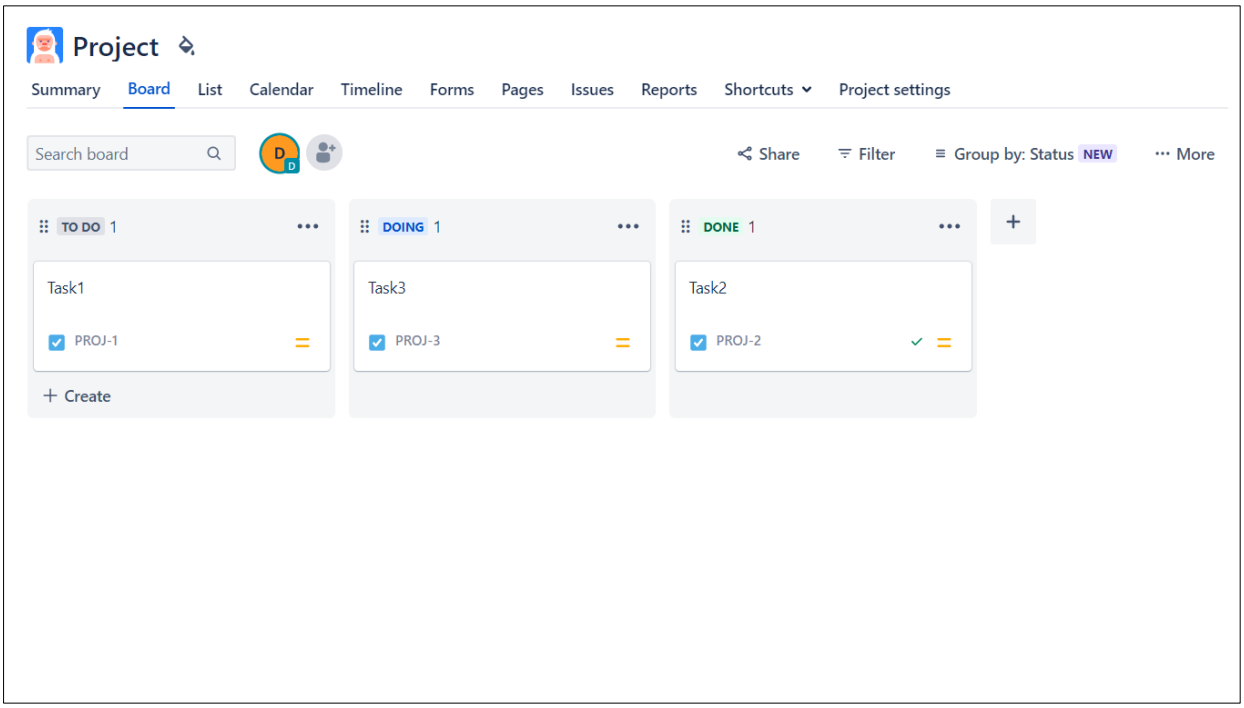


Рисунок 1.2 – Відображення задач у вигляді дошки на платформі Jira

5. Фільтрування та сортування: Кожен учасник команди може фільтрувати та сортувати задачі за різними параметрами, щоб швидко знайти необхідну інформацію.

6. Комунікації команди: Jira містить коментарі, що дозволяє швидко та зручно обговорювати різні питання. Також, користувачі можуть залишати коментарі та запитувати уточнення під будь-якою задачею.

7. Платний тариф: Щоб користуватися Jira, необхідно мати платний тариф. Проте, система надає всім користувачам можливість скористатися пробним періодом, після якого користувачі можуть купити розширений тариф використання системи.

8. Інтеграції: Jira має широкий вибір застосунків для інтеграції. Наприклад, Microsoft Teams, що дозволить легко підключити комунікацію команди проекту до виконуваних задач в системі [5]. Також є інтеграція з Google Drive, що дає можливість легко прикріплювати файли до задач та отримувати сповіщення про коментарі для організації командної роботи. Інтеграції з цими застосунками зможуть підвищити ефективність і значно спростити роботу команди в платформі Jira, роблячи її більш зручною та доступною для командної співпраці.

Отже, JIRA є потужною та багатофункціональною системою, яка підходить для великих команд та проектів з багатьма завданнями. Ця система має широкі можливості налаштування, високий рівень безпеки та зручний інструмент для відстеження часу. Однак, використання системи вимагає додаткових поглиблених знань та навичок в користуванні системою і управлінні проектами, що може зробити її складною для новачків.

1.2.3 Redmine

Одним з існуючих рішень у сфері систем управління проектами є Redmine. Redmine є відкритим програмним забезпеченням (opensource) і надає широкий набір функцій та можливостей для ефективного управління проектами, завданнями та відстеження помилок.

Redmine надає можливість створювати користувачів з різними ролями та правами доступу. Це дозволяє обмежувати доступ до певних функцій і даних системи в залежності від ролі користувача.

Також, Redmine має вбудовані засоби для створення звітів і аналітики, що дозволяє відстежувати прогрес проектів, використання ресурсів, тренди та іншу статистику. Redmine має велику спільноту розробників, яка активно працює над розширеннями та плагінами. Це дозволяє налаштувати систему під власні потреби та розширити її функціональність.

Redmine надає зручний інтерфейс для створення та керування проектами, який наведений на рисунку 1.3. Користувач може визначати терміни виконання, призначати відповідальних користувачів, стежити за прогресом та встановлювати пріоритети завдань. Крім того, Redmine дозволяє створювати та відстежувати завдання, присвоювати їм статуси, встановлювати пріоритети, визначати відповідальних користувачів та вести обговорення.

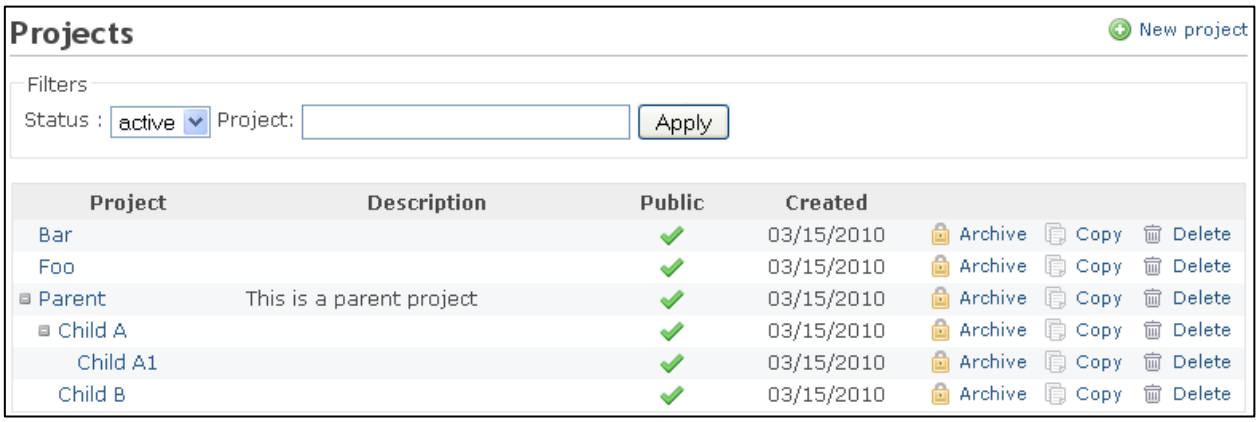


Рисунок 1.3 – Інтерфейс Redmine для керування проектами

Redmine, як opensource-варіанту системи управління проектами, має ряд переваг, таких як відкритий код і економічна цінність. Redmine має відкритий вихідний код, що дозволяє вам налаштовувати систему під свої потреби та вносити зміни за необхідності. А використання такого відкритого програмного забезпечення може значно знизити витрати на ліцензії та розробку в порівнянні з комерційними системами управління проектами.

1.3 Порівняння існуючих систем для управління задач

Asana, Jira і Redmine - це популярні системи для управління проектами, які пропонують набір інструментів для керування та відстеження робочих завдань. Ці продукти мають деякі спільні особливості, такі як досить легке використання та можливість користувачів створювати власні інформаційні панелі. Крім того, користувачі можуть керувати своїми робочими процесами з дошки у стилі Канбан або переглядати їх у вигляді календаря чи графіку.

У таблиці 1.1 наведено порівняння між Asana, Jira і Redmine, яке відображає різноманітні аспекти цих програм для управління проектами. Зокрема, детально описані наступні характеристики: UX (користувацький інтерфейс), інформаційна панель, управління проектами, звітність, інтеграції та ціноутворення.

Таблиця 1.1 – Порівняння платформ Asana, Jira і Redmine [6]

Платформи / Критерії	Asana	Jira	Redmine
UX	Asana є більш простим у використанні, має більш привабливий дизайн та більш інтуїтивну структуру.	Jira, як правило, дуже проста у використанні, але не приділяє уваги до дизайну та організації, яку має Asana.	UX в Redmine може бути описаний як приємний, але дещо застарілий.
Інформаційна панель	Asana пропонує загальну панель керування та звітну панель, але на даний момент не має варіантів віджетів для панелі.	Jira дозволяє створювати кілька панелей та має широкий вибір гаджетів для додавання до них.	Redmine дозволяє відображати на панелі прогрес проектів, поточні завдання та інше.

Продовження таблиці 1.1

Платформи / Критерії	Asana	Jira	Redmine
Управління проектами	Загальне управління проектами є простішим та краще організованим в Asana.	Щодо чистого управління проектами, то Jira трохи відстає, але вона краще розроблена для відстеження проблем	Redmine має розширений набір функцій для управління проектами
Звіти	Звіти добре розроблені, детальні та дозволяють настроювання.	Jira пропонує менше зразків звітів та дозволяє настроювання тільки на облікових записах, які керуються компанією.	Redmine має вбудовані засоби для створення звітів та аналітики.
Інтеграції	Asana має ринок додатків, що становить близько третини від Jira, проте користувачі можуть знайти те, що їм потрібно у вбудованій утиліті.	Jira має великий ринок додатків та входить у кілька категорій продуктів.	Redmine має можливості для інтеграції з іншими інструментами і сервісами.

Кінець таблиці 1.1

Платформи / Критерії	Asana	Jira	Redmine
Складність опанування	Навчання зазвичай займає мало часу, інтерфейс є досить інтуїтивним.	Jira більш складна для використання, вона менш інтуїтивна, особливо для новачків.	Опанування Redmine може зайняти деякий час, особливо для новачків, оскільки він має велику кількість функцій та налаштувань.
Ціноутворення	Безкоштовний пакет Asana доволі щедрий, але він не має варіанту Enterprise, а платні пакети коштують майже вдвічі дорожче, ніж у Jira.	Jira помітно дешевша, має міцний безкоштовний пакет і поставляється у трьох платних варіантах.	Redmine є відкритим програмним забезпеченням і доступний безкоштовно.

Оцінка користувацького інтерфейсу (UX) показала, що програмне забезпечення Asana має привабливий дизайн та інтуїтивно зрозумілу структуру, що робить його більш доступним для користувачів. Jira також є досить простим у використанні, але може виглядати менш сучасним порівняно з Asana. У свою чергу, Redmine має простий, але застарілий дизайн, який може вимагати деякого часу для звикання.

Щодо інформаційної панелі, Asana і Jira надають змогу створювати різні види панелей та додавати до них віджети для відображення різних типів інформації. У Redmine можна налаштовувати інформаційну панель, але він може бути обмежений у функціональності порівняно з Asana та Jira.

Управління проектами в Asana, Jira і Redmine надає можливості організації та відстеження завдань, але вони можуть відрізнятися за зручністю та гнучкістю. Asana має простий і ефективний інтерфейс для створення та організації проектів. Jira надає широкі можливості для керування проектами. Redmine також надає функціональність для управління проектами, але може вимагати більше налаштувань для досягнення потрібного функціоналу.

Щодо звітності, Asana, Jira і Redmine надають різні можливості для створення звітів і аналізу проектів. Asana має потужну систему звітів з великим вибором налаштувань та гнучкістю. Jira надає шаблони звітів та аналітичні інструменти для відстеження продуктивності проектів. Redmine також має можливості для створення звітів, але вони можуть бути менш розширеними порівняно з Asana та Jira.

Крім того, Asana та Jira надають багато можливостей для інтеграції з іншими інструментами і сервісами, такими як системи контролю версій, електронна пошта, календарі та інші. У Redmine також є можливість інтеграції, але вибір доступних інтеграцій може бути меншим.

Щодо ціноутворення, Asana, Jira і Redmine мають різні моделі ціноутворення. Asana та Jira пропонують різні платні пакети залежно від потреб користувача, а також безкоштовні версії з обмеженими можливостями [7]. Redmine є відкритим програмним забезпеченням і доступний безкоштовно, але вартість підтримки та налаштування може бути присутньою.

Загалом, Asana, Jira і Redmine є популярними інструментами для управління проектами, кожен з яких має свої переваги та обмеження. Вибір між ними залежить від конкретних потреб користувача, розміру проекту та бюджету.

1.4 Недоліки існуючих систем для управління задачами

Програмні додатки для керування завданнями мають низький рівень функціональності та якості. Розглянемо ситуацію, коли через місяць або більше введення проекту в системі для управління задач, користувач має список завдань, який безперервно розширюється, проте виконує з нього всього лише три-чотири пункти на день. Список продовжує надмірно наростати, поки користувач не вирішить виключити з нього завершені завдання, які були забуті або невірно внесені до системи з початку. Перед користувачем постає питання: чи доцільно розбити список на групи, включити його до проекту, позначити певними мітками, додати до сьогоднішніх завдань або запланувати на пізніший час? Найімовірніше, користувачеві буде необхідно створити нове завдання, для того, щоб розібратися з уже наявними задачами. Насправді, основна причина, чому так стається, полягає у низькій якості та функціональності більшості програмних застосунків для організації завдань.

Однією з причин, чому це стається, є те, що більшість систем для управління задачами не призначені для підвищення продуктивності користувача. Замість цього, вони пропонують можливості перетягування та додавання, категоризації, позначення та інші функції, щоб допомогти вам організувати свої завдання. Однак, в цих системах не передбачено вбудованого робочого процесу, який допоміг би користувачам досягти більшої продуктивності. Користувачу необхідно самостійно зрозуміти, як бути більш продуктивним з усіма цими інструментами. Щодо виконання великих проектів або цілей, процес досить простий: користувач зберігає список всіх великих проектів або цілей, які хоче виконати, і кожен день бере одне-два маленькі завдання або кроки, які рухають процес вперед. Проблема більшості систем для управління задачами сьогодні полягає в тому, що вони не мають вбудованого способу допомогти користувачу у цьому.

1.5 Рішення недоліків існуючих систем

Для того, щоб вирішити недоліки існуючих систем для управляння задачами, я пропонує додати два удосконалення до системи. Це завдання “у фокусі” і список задач “зараз чи потім”.

Основна ідея завдання “у фокусі”, полягає у спрощенні виконання проектів шляхом фокусу на одному завданні замість перегляду всього списку завдань, які потрібно виконати для завершення проекту. Функціонал системи буде доступний в рамках проекту, де буде створено список завдань, які потрібно виконати. Задача, яка буде в фокусі, буде вибиратися за певним алгоритмом, з урахуванням дедлайну, статусу, пріоритету та складності всіх завдань проекту. Після завершення поточного завдання, система автоматично відображатиме наступне завдання в списку, яке потрібно виконати.

Зосередження на невеликих кроках допомагатиме користувачу зосередитися на виконанні завдання та зберегти ефективність в процесі роботи над проектом. Крім того, система відображатиме прогрес проекту, що допоможе користувачу бачити, скільки вже зроблено і відчути відчуття досягнення. Це може бути мотивуючим фактором для продовження роботи та досягнення поставлених цілей.

Окрім цього, система може мати можливість відстежувати час, витрачений на кожне завдання, що дасть користувачу більш детальну інформацію про свій прогрес та ефективність. Також, можуть бути встановлені попередження для дедлайнів або завдань з високим пріоритетом, щоб допомогти користувачеві виконувати завдання вчасно та ефективно.

Результатом використання такої системи може бути поліпшення ефективності та продуктивності виконання проектів за рахунок спрощення процесу та зосередження на важливих завданнях.

Список завдань "зараз чи потім" є ефективним способом планування часу та організації робочих завдань. Основна мета цього списку полягає в тому, щоб показати на завданнях, які можна виконати швидко та ефективно. Вони зазвичай не потребують значної кількості часу та зусиль для їх виконання, не є дуже важливими та не є складними в реалізації.

Цей підхід дозволяє підвищити продуктивність та ефективність роботи, оскільки дозволяє виконувати більшу кількість завдань протягом дня. Крім того, цей список може бути корисним для тих випадків, коли користувач чекає на відповідь чи уточнення щодо основного завдання, що дозволяє виконувати дрібніші завдання на час очікування.

Наприклад, можна скористатися списком "зараз чи потім" для виконання коротких завдань, таких як відправлення електронної пошти, відповідь на повідомлення в соціальних мережах, перегляд короткого відео, тощо. Всі ці завдання можуть бути виконані під час перерви між більш важливими завданнями, та не вимагатимуть багато часу і зусиль.

Отже, список завдань "зараз чи потім" може бути важливим інструментом для ефективного планування часу та організації робочих завдань, що дозволяє підвищити продуктивність та ефективність роботи.

Як можна побачити, це прості і ефективні способи покращити системи для управління задачами і допомогти користувачу поступово рухатися вперед організованим способом, що не перевантажує користувача великою кількістю завдань, на які не потрібно звертати увагу в даний момент.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було розглянуто визначення і призначення систем для управління задачами і відстеження помилок проектів. Системи для управління задачами проектів - це широкий термін, який використовується для опису програмних застосунків, що допомагають в організації та керуванні проектами. Головна мета систем управління завданнями полягає в тому, щоб допомогти команді планувати, керувати, виконувати та оцінювати прогрес проекту, для досягнення цілей. Вони включають функції керування проектами та окремими завданнями, розподіл ресурсів та інструменти для комунікації членів команди. Системи управління задачами є важливими інструментами для будь-якого проекту, який хоче збільшити ефективність та продуктивність, вони можуть допомогти людині та команді залишатися на вірному шляху та виконувати завдання вчасно, зменшуючи ризик помилок та упущень. Проте, перед вибором програмного забезпечення для організації роботи проекту, необхідно враховувати різні фактори, такі як розмір команди, бюджет і потреби.

Крім того, було розглянуто існуючі рішення для систем управління задачами проектів та проаналізовано їхні схожі та відмінні риси.

У галузі інформаційних технологій, багато компаній надають програмне забезпечення для керування та моніторингу робочих проектів. Asana та Jira є двома популярними варіантами, які мають свої переваги та недоліки. Ці продукти мають спільні риси, такі як легкість використання та можливість користувачів створювати власні інформаційні панелі. Крім того, користувачі можуть управляти своїми робочими процесами через дошку у стилі Канбан або переглядати їх у вигляді календаря чи графіку.

Однак, Asana та Jira відрізняються за кількістю та якістю своїх інструментів. Наприклад, Asana пропонує більше можливостей для створення звітів та інтеграцію з різними інструментами. З іншого боку, Jira пропонує

ширший вибір гаджетів для інформаційних панелей та більш глибоку інтеграцію з іншими продуктами Atlassian. Крім того, ці продукти відрізняються за цінами та функціоналом платних версій, які можуть відповідати різним потребам користувачів.

Користувачі часто стикаються з проблемами організації та виконання завдань, що може призвести до збільшення списку завдань та зниження продуктивності. Однією з причин цього є відсутність вбудованого робочого процесу, що допоміг би користувачам досягти більшої продуктивності. Для виконання великих проектів або цілей, користувачі мають самостійно зрозуміти, як використовувати інструменти системи для досягнення більшої ефективності. Тому, важливо додати до системи вбудований процес для підвищення продуктивність користувачів.

Я пропоную вдосконалити систему для управління завданнями, шляхом додавання нових функцій: завдання "у фокусі" та список завдань "зараз чи потім". Завдання "у фокусі" дозволяє сконцентруватися на важливих завданнях та забезпечує їх послідовне виконання, що полегшує процес виконання проектів. Система також надає користувачу можливість відстежувати час виконання завдань та встановлювати попередження для дедлайнів та завдань з високим пріоритетом. Результатом використання такої системи може бути покращення ефективності та продуктивності виконання проектів.

Також, для покращення систем для управління задачами і відстеження помилок у проектах було розглянуто переваги та можливості використання списку завдань "зараз чи потім" для ефективного планування часу та організації робочих завдань. Цей список дозволяє швидко виконувати завдання, які не потребують багато часу та зусиль, підвищує продуктивність та ефективність роботи. Крім того, він корисний для виконання завдань під час очікування основного завдання. Користувач може поступово рухатися вперед організованим способом, не перевантажуючи себе великою кількістю завдань.

РОЗДІЛ 2. ВИБІР ТА ОБҐРУНТУВАННЯ ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ

Розробка системи для управління задачами і відстеження помилок у хмарному середовищі вимагає уваги до аспектів доступності, мобільності та функціональності. Веб-застосунки є ідеальним вибором для таких систем з кількох причин.

По-перше, вони забезпечують доступність системи з будь-якого пристрою з підтримкою веб-браузера. Це означає, що користувачі можуть легко отримати доступ до системи зі своїх персональних комп'ютерів, ноутбуків, планшетів або смартфонів, незалежно від їх місця розташування та операційної системи.

По-друге, веб-застосунки забезпечують мобільність, оскільки їх можна використовувати на різних пристроях та з'єднуватися з системою з різних місць. Це особливо важливо для системи управління задачами, де користувачі можуть мати потребу відстежувати та оновлювати задачі навіть під час відряджень або з віддалених робочих місць.

По-третє, веб-застосунки можуть надати широкий функціонал і можливості для взаємодії з іншими сервісами та системами. Вони можуть бути легко інтегровані з іншими хмарними сервісами, такими як хмарні сховища або сервіси для спільної роботи. Крім того, веб-застосунки можуть надавати можливості спільної роботи та комунікації між користувачами, що є важливим аспектом для системи управління задачами.

Таким чином , з огляду на доступність, мобільність та функціональність, веб-застосунок є оптимальним варіантом для розробки системи для управління задачами і відстеження помилок у хмарному середовищі.

Веб-застосунок складається з двох основних частин:

- Бекенд (backend) - це серверна частина веб-застосунку, яка забезпечує обробку запитів, роботу з базою даних та інші функції, які відбуваються на стороні сервера [8].
- Фронтенд (frontend) - це клієнтська частина веб-застосунку, яка відповідає за відображення інтерфейсу користувача, обробку взаємодії з користувачем та передачу запитів на сервер.

Кожна з них розробляється окремо і використовує власні технології.

2.1 Технології для розробки бекенд частини системи

2.1.1 Вибір мови програмування

При розробці бекенду будь-якого програмного забезпечення, вибір мови програмування грає важливу роль і формує основний стек технологій. При виборі необхідно враховувати наступні чинники:

- Мета проекту: в залежності від мети проекту вибір мови може відрізнятися.
- Досвід розробника. це зменшить час, який потрібен для навчання нової мови та допоможе уникнути помилок, пов'язаних з незнанням мови.
- Доступність ресурсів: важливо мати на увазі, що для кожної мови програмування існують різні засоби розробки, бібліотеки та фреймворки. Важливо переконатися, що необхідні ресурси доступні для обраної мови.
- Швидкість виконання: якщо проект потребує швидкого виконання складних обчислень або маніпуляцій з великими обсягами даних, то бажано використовувати мови, що забезпечують високу швидкість виконання.

- Масштабованість: якщо проект потребує масштабування, тобто здатність працювати з великим обсягом даних та користувачів, то бажано використовувати мови, які дозволяють легко масштабувати систему.
- Спільнота розробників: важливо визначити, чи є активна спільнота розробників для обраної мови програмування. Якщо так, то це забезпечить доступ до знань та допомоги.

Для розробки бекенд частини системи управління задачами та відстеження помилок я обрала мову програмування C#, базуючи свій вибір на таких критеріях як розширюваність, продуктивність, доступність інструментів для розробки та тестування, та інших.

C# є мовою програмування, яка підтримує об'єктно-орієнтований підхід, що дозволяє розбити систему на окремі об'єкти, що можуть бути легко модифіковані та перевикористані в інших частинах системи [9]. Це сприяє структуруванню коду та робить систему більш підтримуваною і дає можливість легко розширювати проект.

C# має багату функціональність та широкі можливості розширення, що дозволяє розробникам ефективно створювати складні системи з багатьма функціями. Обрана мова програмування вважається однією з найпродуктивнішою. C# виконується в середовищі .NET, що забезпечує швидку роботу програм. Також, C# має можливість компіляції в нативний код для підвищення продуктивності [10].

Ця мова пропонує широкий вибір доступних інструментів для розробки та тестування. C# має велику кількість інструментів для розробки та тестування, таких як Visual Studio, ReSharper, NUnit, xUnit, тощо.

Крім того, C# - це дуже популярна мова програмування, що має велику спільноту розробників, яка завжди готова надавати допомогу та відповісти на

запитання. Це дозволяє розробникам швидко знайти відповіді на питання, які можуть виникнути.

Нарешті, у даної мови програмування є підтримка від Microsoft. Microsoft, яка розробляє C# та .NET, є надійним партнером для бізнесу та надає широкий спектр інструментів для розробки та підтримки рішень на її платформах. Це дозволяє забезпечити надійність та стабільність системи.

2.1.2 Вибір платформи для розробки програмного забезпечення

.NET Core та .NET Framework - це дві різні реалізації платформи .NET для розробки програмного забезпечення. .NET Framework був розроблений в першу чергу для ОС Windows та має обмежену підтримку на інших операційних системах. З іншого боку, .NET Core є більш універсальною платформою, яка ефективно працює на різних операційних системах, в тому числі на Windows, Linux та MacOS [11].

Для бекенду системи у хмарному середовищі, .NET Core є кращим вибором порівняно з .NET Framework. Це пов'язано з тим, що хмарні сервіси все частіше використовують Linux-сервери, на яких .NET Framework має обмежену підтримку та може виникнути проблема з сумісністю. З іншого боку, .NET Core має підтримку для більшої кількості операційних систем та може працювати на будь-якій інфраструктурі хмарних сервісів, що робить його більш універсальним та гнучким для використання у хмарному середовищі.

Коли розробляється система, важливо мати на увазі можливість розширення її функціоналу в майбутньому. Однією з переваг .NET Core є можливість розширення функціоналу шляхом використання різноманітних бібліотек і пакетів NuGet, а також написання власних розширень.

Також, мова програмування C# є розширюваною, що дозволяє розширювати стандартну бібліотеку класів за допомогою розширень (extension

methods) [12] та створювати власні класи, що успадковують функціонал від інших класів.

Крім того, .NET Core має кращу продуктивність та масштабованість порівняно з .NET Framework, що дозволяє створювати більш потужні та швидкодіючі системи для управління задачами та відстеження помилок проекту у хмарному середовищі. Наприклад, .NET Core забезпечує більш швидкий запуск та виконання додатків, що дозволяє досягати кращої продуктивності та відповідності вимогам користувачів. .NET Core дозволяє забезпечити високу продуктивність завдяки багатопоточній обробці запитів, оптимізації швидкодії, використанню пам'яті та ефективному управлінню ресурсами.

Крім того, .NET Core має більш широкий вибір інструментів для розробки та тестування, таких як Entity Framework Core, AutoMapper, MediatR, DI, MS SQL Server та xunit, що забезпечують зручнішу розробку та тестування системи.

.NET Core має відкритий код та велику спільноту розробників, що дозволяє створювати власні розширення та бібліотеки для роботи з різними сервісами, що сприяє зменшенню часу розробки та забезпечує широкі можливості для розширення функціональності системи.

Узагальнюючи, .NET Core та мова програмування C# є оптимальним вибором для розробки бекенду системи для управління задачами і відстеження помилок проекту у хмарному середовищі з огляду на їх продуктивність, ефективність, безпеку, підтримку розширюваності та доступність інструментів для розробки та тестування.

2.1.3 Вибір веб-фреймворку для розробки системи

ASP.NET є одним з найбільш популярних веб-фреймворком для веб-розробки, розробленим компанією Microsoft для створення веб-додатків, веб-сервісів та інтернет-сторінок на мовах програмування, таких як C#, VB.NET, та F# [13]. Фреймворк базується на платформі .NET та надає розробникам

можливість створювати високопродуктивні, масштабовані, та безпечні веб-додатки та веб-сервіси. ASP.NET містить в собі різноманітні компоненти, які допомагають у забезпеченні безпеки, обробки запитів, роботи з базами даних, тестуванні та налагодженні додатків. ASP.NET використовується багатьма компаніями та розробниками для створення високопродуктивних веб-додатків та веб-сервісів.

Для розробки бекенд частини системи управління задачами та відстеження помилок я обрала веб-фреймворк ASP.NET через такі його переваги, як висока продуктивність, висока масштабованість, легкість розробки та розширення, а також широкий набір інструментів для тестування та налагодження додатків. Крім того, ASP.NET має безпечну архітектуру, що дозволяє забезпечити захист від різноманітних атак та зловмисних дій.

ASP.NET дозволяє легко масштабувати систему від невеликих до великих проектів. ASP.NET підтримує різні методи масштабування, такі як вертикальне та горизонтальне масштабування. Також ASP.NET є відкритою платформою, що дозволяє використовувати різноманітні розширення та сторонні бібліотеки для поліпшення функціональності системи.

Більше того, ASP.NET забезпечує високий рівень безпеки завдяки вбудованим механізмам автентифікації, авторизації та шифруванню даних. ASP.NET надає потужні засоби для забезпечення безпеки веб-додатків, включаючи підтримку протоколу HTTPS, автентифікацію та авторизацію, обмеження доступу до ресурсів за допомогою ролей та дозволів [14]. Це дозволяє знизити ризик втрати даних та підвищити надійність системи.

Додатково, ASP.NET підтримується компанією Microsoft, що забезпечує активний розвиток та підтримку платформи. Це забезпечує стабільну та безпечну роботу системи протягом тривалого часу. ASP.NET дозволяє швидко розробляти веб-додатки завдяки використанню стандартних компонентів та готових шаблонів. Це знижує час на розробку та зменшує кількість помилок.

Крім того, ASP.NET забезпечує широкі можливості для розробки веб-додатків, які відповідають вимогам сучасних стандартів веб-розробки. Завдяки вбудованим бібліотекам та компонентам, розробникам не потрібно писати код з нуля, що дозволяє значно скоротити час розробки та знизити ймовірність помилок.

З використанням ASP.NET також можливе розгортання додатків в хмарних сервісах, таких як Microsoft Azure, що дозволяє забезпечити високу доступність та масштабованість системи.

Отже, ASP.NET в поєднанні з C# та .NET Core дозволяє забезпечити швидку та безпечну розробку веб-додатків з високою масштабованістю та доступністю в хмарному середовищі.

2.1.4 Вибір бази даних для системи

Реляційні та нереляційні бази даних є двома основними типами баз даних, використовуваними для зберігання та організації даних. Вони відрізняються у своїх моделях даних, підходах до структури даних та характеристиках використання.

Реляційні бази даних використовують реляційну модель даних, де дані представлені у вигляді таблиць зі зв'язками між ними за допомогою ключів. Це дозволяє забезпечити структурованість і співвідношення між даними. Проте нереляційні бази даних використовують різні моделі даних, такі як ключ-значення, документ, стовпчикова, графова тощо. Ці моделі дозволяють уникнути жорсткого прив'язування до таблиць і зв'язків.

Нереляційні бази даних мають більш гнучку структуру, де кожен об'єкт може мати свою власну структуру даних. Це дозволяє зберігати інформацію з різними схемами без потреби в регулярних оновленнях схеми бази даних. А реляційні бази даних мають жорстку структуру, оскільки дані мають

відповідати певній схемі, визначеній таблицями, стовпцями та зв'язками [15]. Це дозволяє забезпечити цілісність і консистентність даних.

Реляційні бази даних зазвичай мають оптимізовані операції для зчитування та запису даних, а також для виконання складних запитів. Вони підтримують індекси, оптимізатор запитів та інші механізми для поліпшення продуктивності. Однак, при великому обсязі даних і складних операціях можуть виникати проблеми продуктивності. Проте нереляційні бази даних зазвичай мають високу продуктивність, особливо при операціях зчитування та запису великого обсягу даних. Вони використовують спеціалізовані структури даних та алгоритми для швидкого доступу до даних. Проте, складні запити можуть бути обмеженими або менш ефективними у порівнянні з реляційними базами даних.

Реляційна база даних надає підтримку транзакцій, що дозволяє групувати операції з базою даних у логічні блоки, що виконуються атомарно, консистентно, ізольовано та незмінно (ACID властивості). Це дозволяє забезпечити цілісність даних та відновлення до попереднього стану у випадку помилок або відмов. Крім того, реляційна база даних, зазвичай підтримує стандарти SQL, що робить її сумісною з іншими реляційними базами даних. Це означає, що можна легко інтегрувати систему управління задачами з іншими додатками та системами, які використовують реляційну базу даних.

Для розробки системи для управління задачами і відстеження помилок у проекту краще обрати саме реляційну систему, бо система буде працювати зі структурованими даними, які мають складні зв'язки та потребують сильної консистентність. Вона надасть сильну структуру даних, можливість виконувати складні запити та забезпечити цілісність даних.

Однією з найпопулярніших реляційних баз даних є MS SQL Server. Для розробки дипломного проекту я обрала цю базу даних через такі її переваги, як масштабованість, надійність, безпека й можливість інтеграція з іншими продуктами Microsoft [16].

MS SQL Server може масштабуватися як горизонтально (шляхом додавання нових серверів) так і вертикально (збільшення обсягу ресурсів на існуючих серверах). Це дозволяє легко розширювати вашу систему в разі збільшення обсягу даних або навантаження. Крім того, MS SQL Server є відомим і визнаним продуктом, який має довгу історію розробки та тестування. Він надійний і забезпечує високий рівень доступності та стабільності. Це особливо важливо для систем управління задачами, де надійність даних і відстеження помилок є критичними факторами.

MS SQL Server надає широкий спектр можливостей для оптимізації та керування базою даних. Він підтримує різні типи індексів, запити і процедури збереження, транзакції, реплікацію та багато іншого. Це дозволяє ефективно працювати з великим обсягом даних та забезпечити швидкий доступ до них. Додатково, MS SQL Server добре інтегрується з іншими продуктами Microsoft, такими як .NET Framework, Azure, SharePoint та інші. Якщо ви використовуєте інші продукти Microsoft у вашому хмарному середовищі, ви можете отримати додаткові переваги від використання MS SQL Server.

MS SQL Server повністю підтримує стандарти реляційних баз даних, такі як SQL і ACID (Atomicity, Consistency, Isolation, Durability), що робить його сумісним з багатьма іншими додатками і системами, які також використовують ці стандарти.

Зважаючи на ці фактори, MS SQL Server може бути відмінним вибором для системи управління задачами і відстеження помилок проекту в хмарному середовищі.

2.1.5 Вибір механізмів для роботи з базою даних

Для роботи з базою даних необхідний уніфікований спосіб доступу до даних для забезпечення більш зручного та продуктивного програмування. З цим може допомогти ORM (Object-Relational Mapping) система. Вона перетворює

дані з реляційної бази даних у об'єкти, з якими розробник може працювати безпосередньо в своєму коді [17].

ORM системи впроваджують принципи об'єктно-орієнтованого програмування та забезпечують високу абстракцію бази даних, спрощуючи процес розробки та збереження даних. Вони пропонують зручний і продуктивний спосіб роботи з базами даних, знижуючи зусилля, пов'язані з ручним написанням SQL-запитів та взаємодією з базою даних. Крім того, вони сприяють переносимості системи та забезпечують безпеку даних. Таким чином, використання ORM системи, такої як Entity Framework Core, є обґрунтованим вибором для системи для управління задачами і відстеження помилок у хмарному середовищі.

Entity Framework Core (EF Core) є сучасним об'єктно-реляційним мапером, який надає зручні інструменти для роботи з базами даних у дотнет-екосистемі [18].

Я обрала саме EF Core як механізму доступу до даних для системи через такі його переваги, як:

- Універсальність: Entity Framework Core підтримує різні бази даних, що дозволяє системі бути гнучкою щодо вибору платформи бази даних. Технологія дає можливість легко перенести систему на іншу базу даних без необхідності великих змін у коді.
- Швидкість розробки: EF Core автоматично генерує SQL-запити на основі моделі даних. Це спрощує процес створення запитів і зменшує кількість коду, що потрібно написати [19]. Крім того, механізм міграцій дозволяє автоматично оновлювати схему бази даних, що зменшує зусилля, пов'язані з ручним оновленням бази даних.
- Строга типізація: мапер підтримує строгу типізацію, що дозволяє виявляти помилки в процесі компіляції. Це сприяє надійності та

стабільності системи, оскільки багато помилок можна виявити на ранніх етапах розробки.

- Підтримка запитів і фільтрації: Entity Framework Core надає потужні можливості для складання складних запитів та фільтрації даних. Додатково, можна використовувати LINQ-запити або методи розширення для створення запитів, що спрощує роботу з базою даних.

Таким чином, користання Entity Framework Core для доступу до бази даних у системі для управління задачами і відстеження помилок у хмарному середовищі пропонує безліч переваг, включаючи універсальність, швидкість розробки та зручність управління даними.

Крім ORM системи, для зручної і ефективної роботи з маніпуляціями даних з бази даних необхідний механізм для мапінгу об'єктів. Для цієї задачі я обрала бібліотеку для мапування об'єктів в .NET-додатках – AutoMapper. Вона дозволяє зручно виконувати відображення (mapping) даних між об'єктами різних типів, зокрема між об'єктами домену і об'єктами доступу до даних (DTO) [20]. AutoMapper автоматично знаходить відповідності між властивостями об'єктів і виконує їх копіювання або трансформацію.

AutoMapper зменшує зусилля розробки шляхом автоматизації процесу мапування між об'єктами домену і об'єктами доступу до даних. Розробникам не потрібно писати власний код для копіювання властивостей, вони можуть скористатись можливостями AutoMapper для автоматичного виконання мапування. Це дозволяє ефективно використовувати час та зусилля розробників і зосередитися на більш важливих аспектах розробки системи.

Більше того, використання AutoMapper допомагає забезпечити консистентність даних між об'єктами домену і об'єктами доступу до даних. Він автоматично виконує перенесення даних між цими об'єктами, що дозволяє уникнути можливих помилок в даних.

Таким чином, використання AutoMapper обґрунтоване для спрощення роботи з об'єктами домену та їх відображенням на об'єкти, що зберігаються у базі даних. Він допомагає знизити зусилля розробки, забезпечує консистентність даних і пропонує зручний підхід до роботи з DTO.

2.2 Технології для розробки фронтенд частини системи

2.2.1 Вибір мови програмування

Вибір мови програмування для фронтенд частини системи є важливим рішенням, яке впливає на продуктивність, якість коду та зручність розробки. У контексті системи для управління задачами і відстеження помилок, обґрунтованим вибором є використання TypeScript замість JavaScript. Далі наведено кілька аргументів, чому TypeScript може бути кращим вибором.

TypeScript надає статичну типізацію, що дозволяє виявляти помилки на етапі розробки [21]. Це дозволяє знизити кількість помилок та полегшити рефакторинг коду. У системі для управління задачами і відстеження помилок, де точність та надійність даних є критичними, типізація TypeScript може значно поліпшити якість і безпеку коду.

Крім того, TypeScript є розширенням JavaScript і надає додаткові можливості, які полегшують розробку і підтримку великих проектів. Він має ряд особливостей, таких як підтримка класів, модулів, інтерфейсів, декораторів та інших сучасних конструкцій мови. Це дозволяє розробникам створювати більш структурований та легко зрозумілий код.

TypeScript підтримує широкий спектр інструментів для розробки, таких як популярні редактори коду (Visual Studio Code, WebStorm), фреймворки (Angular, React, Vue.js) та інші. Це спрощує розробку, налагодження та підтримку системи для управління задачами і відстеження помилок, оскільки

TypeScript надає більшу підтримку засобів розробки та розширює їх можливості.

TypeScript є популярною мовою програмування з активною та широкою спільнотою розробників [21]. Це означає, що є багато доступних ресурсів, документації, пакетів та розширень, які сприяють швидкій розробці та вирішенню проблем.

Додатково, TypeScript дозволяє створювати розширені абстракції, робити перевірку типів та рефакторинг коду, що полегшує роботу з великими та складними проектами. Він сприяє підтримці масштабованості системи, дозволяючи розробникам швидко зрозуміти структуру та взаємодію компонентів.

Узагальнюючи, використання TypeScript для фронтенд частини системи для управління задачами і відстеження помилок принесе переваги в плані типізації, розширеної функціональності, зручних інструментів розробки, розширеної екосистеми та підтримки масштабованості. Всі ці фактори сприятимуть розробці високоякісного, безпечного та легко підтримуваного коду для системи.

2.2.2 Вибір веб-фреймворку для розробки системи

Для розробки фронтенд частини системи для управління задачами і відстеження помилок важливо обрати правильний веб-фреймворк. Два найпопулярніші на сьогоднішній день це Angular і React. І для розробки системи я обрала саме React, далі наведено аргументи, чому це кращий вибір.

React пропонує простоту використання. Він зосереджується на створенні компонентів, які є повторно використовуваними та легко керованими. React базується на концепції компонентів, що дозволяє створювати ієрархію повторно використовуваних та модульних компонентів [22]. Це спрощує організацію коду, покращує його читабельність та підтримку. Компонентна

архітектура робить розробку та тестування окремих частин системи більш простими та ефективними.

Крім того, React надає гнучкість та розширюваність у розробці функціоналу. Він не нав'язує жорстких правил архітектури, що дає можливість розробникам використовувати підхід, який найкраще відповідає потребам проекту. Це особливо важливо у системі для управління задачами і відстеження помилок, де можуть виникати унікальні вимоги та потреби користувачів.

Що важливо, React пропонує високу продуктивність завдяки застосуванню віртуального DOM (Document Object Model) і алгоритму поділу та злиття (diffing) для ефективного оновлення і відображення компонентів. Це забезпечує швидку реакцію на зміни даних та оптимізує процес оновлення інтерфейсу користувача.

Узагальнюючи, вибір фреймворка React для фронтенд частини системи для управління задачами і відстеження помилок є обґрунтованим через його простоту вивчення, гнучкість, швидкодію та підтримку великого співтовариства розробників. React дозволяє розробляти ефективний та модульний фронтенд, забезпечуючи широкі можливості управління станом додатку, повторне використання компонентів та зручну роботу з віртуальним DOM. Крім того, наявність багатої екосистеми та інструментарію робить React потужним інструментом для розробки функціонального та привабливого інтерфейсу.

2.2.3 Вибір бібліотек для розробки дизайну системи

Перше на що користувач звертає свою увагу на веб сторінці – це зовнішній вигляд системи, тому створення дизайну є важливою частиною всієї розробки. Для розробки дизайну я обрала такі бібліотеки, як Material UI для створення загального дизайну, і nivo – для стилізації графіків.

Material UI надає готові компоненти для стильного дизайну, що сприяє швидкій розробці професійного інтерфейсу користувача [23]. Використання Material UI забезпечує єдиний та консистентний вигляд у всій системі. Крім того, Material UI підтримує респонсивний дизайн, що дозволяє системі коректно відображатися на різних пристроях і розмірах екрану. Це важливо для системи, яка може бути доступною як на десктопних комп'ютерах, так і на мобільних пристроях. Респонсивний дизайн Material UI допомагає забезпечити зручну та зрозумілу взаємодію з системою на будь-якому пристрої.

nivo, у свою чергу, дозволяє створювати різноманітні графіки та налаштовувати їх. nivo працює на основі векторної графіки та оптимізована для високої продуктивності. Вона використовує SVG та Canvas для візуалізації графіків, що дозволяє забезпечити швидке та плавне відображення навіть при великій кількості даних [24]. Більш того, бібліотека дозволяє відображати великий обсяг даних та забезпечує інтерактивність. Використання nivo дозволяє візуалізувати дані та забезпечити користувачів зручним способом взаємодії з графіками.

Обидві бібліотеки мають документацію та активну спільноту розробників, що дозволяє легко засвоїти їхні можливості та отримати підтримку. Використання цих бібліотек сприяє швидкому розробленню функціонального та привабливого інтерфейсу користувача з динамічними та зрозумілими графіками.

Таким чином, вибір Material UI та nivo є обґрунтованим для розробки системи для управління задачами і відстеження помилок, оскільки вони спрощують розробку, забезпечують привабливий та функціональний інтерфейс користувача та дозволяють відображати дані у зручному та інтерактивному форматі.

2.3 Вибір хмарного середовища для системи

Розгортання системи у хмарному середовищі відкриває безліч переваг і можливостей, які сприяють ефективності, гнучкості та масштабованості системи.

Одна з найвагоміших переваг хмарного середовища - це масштабованість. Хмарні середовища дають можливість збільшувати або зменшувати потужність інфраструктури залежно від потреб проекту [25]. Це означає, що система може адаптуватися та масштабуватися відповідно до розміру та потреб проекту. Більше немає необхідності купувати та обслуговувати власне обладнання, можна просто використовувати існуючу інфраструктуру хмарного провайдера та платити лише за те, що використовуєте.

Ще одна перевага – це глобальна доступність. Хмарні платформи розподілені по всьому світу і з'єднують користувачів з системою з будь-якого куточка планети. Будь-який користувач, будь-де в світі, може легко отримати доступ до системи без затримок та перешкод, що розширює коло можливих користувачів і забезпечує їм максимальне задоволення.

Хмарні платформи вкладають неймовірні зусилля в забезпечення безпеки даних. Вони використовують передові методи шифрування, системи контролю доступу та захисту від зламу, щоб дані залишалися цілими та конфіденційними. Так що можна спокійно спати, знаючи, що система знаходиться під надійним захистом.

З хмарним середовищем можна забути про непотрібне турбування щодо обслуговування та оновлення апаратного забезпечення. Всі ці клопоти покладаються на плечі хмарного провайдера, який постійно вдосконалює свою інфраструктуру та пропонує користувачам оновлення без перебоїв. Розробники можуть спокійно зосередитися на розвитку системи та задоволенні потреб бізнесу, знаючи, що завжди буде доступ до найновіших можливостей.

Крім того, хмарні рішення дозволяють економити фінансові ресурси. Необхідно платити лише за те, що використовуєте, без необхідності великих початкових інвестицій або витрат на обслуговування обладнання. Це дозволяє оптимізувати витрати і спрямувати ресурси на розвиток системи.

Отже, розгортання системи у хмарному середовищі є стратегічним рішенням, яке дозволяє сконцентруватися на розробці продукту, забезпечуючи при цьому масштабованість, доступність, безпеку та ефективність. Це важлива перевага, особливо у сучасному швидкозмінному світі, де ділові вимоги змінюються, а компанії шукають швидке та надійне рішення для своїх проектів.

Для розгортання системи для управління задачами і відстеження помилок я обрала хмарне середовище Microsoft Azure. Microsoft Azure є одним з провідних хмарних платформ, яка забезпечує широкі можливості для розгортання, масштабування та керування додатками [26]. Далі наведено кілька причин, чому Microsoft Azure може бути оптимальним вибором для моєї системи:

Azure має глибоку підтримку для .NET-технологій, зокрема .NET Core. Це дає можливість легко розгорнути бекенд системи, використовуючи Azure App Service або Azure Functions, які підтримують виконання .NET Core додатків.

Крім того, Azure надає широкі можливості для масштабування системи в залежності від потреб. Можна гнучко налаштувати ресурси та забезпечити високу доступність та надійність системи завдяки автоматичному масштабуванню, резервному копіюванню та іншим функціям, доступним в Azure.

Azure надає багато інструментів для розробки, тестування, моніторингу та управління додатками. Наприклад, можна використовувати Visual Studio або Azure DevOps для розробки та CI/CD процесів. Крім того, Azure надає широкий вибір сервісів для моніторингу та логування, таких як Azure Monitor, Azure Application Insights і Azure Log Analytics. Це дозволяє відстежувати

продуктивність, виявляти проблеми та аналізувати дані, щоб покращити роботу системи.

Додатково, Azure забезпечує високий рівень безпеки для системи. Він має широкий набір заходів безпеки, таких як контроль доступу, захист мережі, шифрування даних та інші механізми захисту. Можна легко налаштувати правила безпеки, використовувати ідентифікацію та автентифікацію Azure Active Directory і використовувати інші інструменти безпеки, щоб забезпечити захист системи та даних.

Узагальнюючи, вибір хмарного середовища для розгортання системи для управління задачами і відстеження помилок проекту є важливим кроком у процесі розробки. Microsoft Azure виявляється оптимальним варіантом, оскільки вона забезпечує широкий спектр можливостей та інтеграцію з .NET Core, що використовуються у моїй системі. Azure надає зручні інструменти розробки, масштабованість, безпеку та підтримку, що допоможуть розгорнути і керувати системою ефективно. Вибір Azure дозволить сконцентруватися на розробці системи, маючи надійну та потужну хмарну інфраструктуру.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі було проведено вибір технологій для розробки системи для управління задачами і відстеження помилок у проекті. Кожен вибір був обґрунтований з метою забезпечення ефективності, гнучкості та масштабованості системи.

Для розробки бекенд частини системи було розглянуто низку сучасних технологій. Вибір мови програмування, платформи розробки, веб-фреймворку, бази даних та механізмів для роботи з базою даних був здійснений з урахуванням низки факторів, що впливають на ефективність, гнучкість та надійність системи.

Обґрунтування вибору мови програмування було зроблено на користь .NET Core та C#. Ці технології забезпечують широкий функціонал, масштабованість, високу продуктивність та підтримку спільноти розробників.

Вибір платформи для розробки програмного забезпечення був здійснений на основі особливостей проекту і вибору .NET як основної платформи, яка надає надійність, розширюваність та широкий інструментарій. ASP.NET в поєднанні з C# та .NET Core дозволяє забезпечити швидку та безпечну розробку веб-додатків з високою масштабованістю та доступністю в хмарному середовищі.

Обґрунтування вибору бази даних було зроблено на користь Microsoft SQL Server, який забезпечує стабільну та надійну роботу з даними, високу продуктивність та багатий функціонал. Вибір механізмів для роботи з базою даних базується на використанні Entity Framework Core, що надає зручність у роботі з базою даних, мапування об'єктів та підтримку легкої міграції.

У даному розділі також було проведено обґрунтування вибору технологій для розробки фронтенд частини системи управління задачами і відстеження помилок у проекті. Вибір мови програмування, веб-фреймворку та бібліотек

для розробки дизайну був здійснений з урахуванням важливих аспектів, що впливають на користувальницький досвід, швидкість розробки та функціональні можливості системи.

Обґрунтування вибору мови програмування було зроблено на користь TypeScript, який є широко використовуваною мовою для розробки фронтенд додатків, забезпечує широку підтримку браузерів та багатий вибір інструментів і бібліотек.

Для розробки веб-фреймворку був обраний React, що є потужним та популярним фреймворком, забезпечує компонентний підхід, швидку реактивну відповідь та зручну управління станом додатку.

Щодо бібліотек для розробки дизайну було обрано Material UI та nivo. Оскільки вони спрощують розробку, забезпечують привабливий та функціональний інтерфейс користувача та дозволяють відображати дані у зручному та інтерактивному форматі.

Обрані технології відповідають сучасним вимогам розробки фронтенд частини, забезпечують зручний інтерфейс, високу продуктивність та можливість легкого розширення функціоналу системи.

Для розгортання системи було обрано хмарне середовище Microsoft Azure. Це обґрунтовано широким спектром послуг, масштабованістю, надійністю та безпекою, які надає Microsoft Azure.

В результаті вибору цих технологій та хмарного середовища, очікується створення ефективної, масштабованої та надійної системи для управління задачами і відстеження помилок у проекті, яка відповідає сучасним вимогам та потребам організацій, зайнятих проектною діяльністю.

РОЗДІЛ 3. АРХІТЕКТУРНИЙ АНАЛІЗ ТА ПРОЕКТУВАННЯ СИСТЕМИ

3.1 Застосування архітектурного патерну Domain-Driven Design

Архітектурний патерн DDD (Domain-Driven Design), що перекладається як "проектування з фокусом на домен", є підходом до розробки програмного забезпечення, який ставить акцент на моделювання та розуміння доменної проблематики. DDD дозволяє розробникам та бізнес-експертам спільно працювати над складними системами, фокусуючись на їх специфічних бізнес-правилах та концепціях.

Архітектура DDD передбачає розділення системи на різні шари (layers), кожен з яких відповідає за свою функціональність та має свої відповідальності. На рисунку 3.1 наведено шари і залежності між ними для моєї системи.

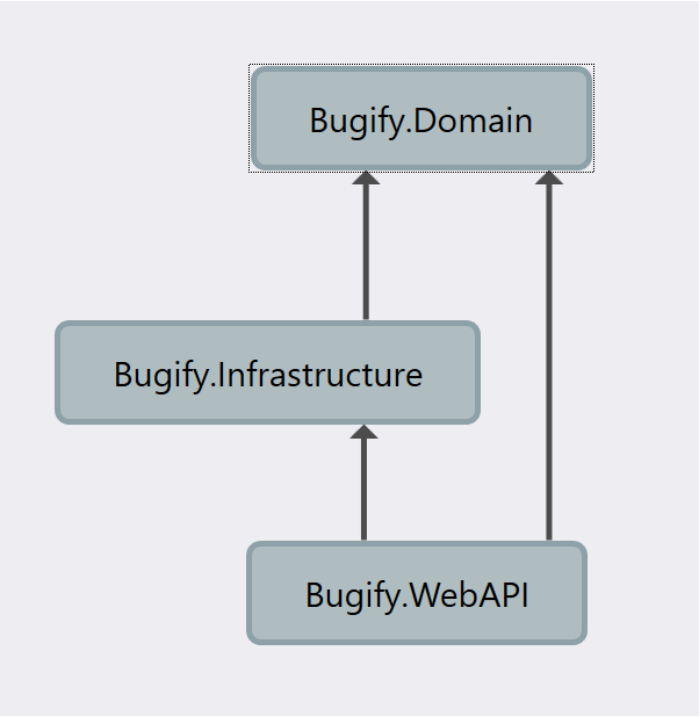


Рисунок 3.1 - Залежності між шарами в DDD

DDD сприяє розбиттю складної системи на менші, доменно орієнтовані контексти, що полегшує розуміння та моделювання проблемної області. Використання мови домену та бізнес-термінології дозволяє команді розробників та бізнес-експертів ефективно спілкуватися та вирішувати проблеми разом.

Окрім цього, DDD впроваджує концепцію розглядати в першу чергу доменні об'єкти та правила. Це означає, що доменна модель займає центральне місце в системі, а інфраструктура та інші технічні аспекти вибудовуються навколо неї. Такий підхід дозволяє підтримувати гнучкість та розширюваність системи, а також полегшує тестування та підтримку коду.

Основні шари в архітектурі DDD включають:

1. Доменний шар (Domain Layer): Це найважливіший шар, який містить доменну модель - представлення бізнес-проблематики та правил. В цьому шарі зосереджена основна логіка системи, а також сутності, агрегати, значення та сервіси, що представляють ключові концепції домену. Доменний шар визначає, як система працює з даними та як відбувається обробка бізнес-процесів. В моїй системі цей шар називається Bugify.Domain.

2. Інфраструктурний шар (Infrastructure Layer): Цей шар забезпечує підтримку технічних аспектів системи, таких як збереження даних, комунікація з зовнішніми системами, логування, кешування тощо. В інфраструктурному шарі розміщуються репозиторії, фабрики, постачальники, служби доступу до даних та інші технічні компоненти. Він взаємодіє з зовнішніми ресурсами та забезпечує інтеграцію з ними. В моїй системі цей шар називається Bugify.Infrastructure.

3. Шар застосунку (Application Layer): Цей шар виступає в якості посередника між користувачем та системою. Він містить логіку застосунку, яка оркеструє виконання бізнес-операцій. Застосунок приймає запити від користувача, координує виконання дій в доменному шарі та повертає результати назад до користувача. Він також може включати додаткову логіку,

пов'язану з безпекою, авторизацією, транзакціями тощо. В моїй системі цей шар називається Bugify.WebAPI.

Отже, застосування патерну DDD для системи управління задачами і відстеження помилок у хмарному середовищі допоможе створити більш гнучку, розширювану та бізнес-орієнтовану систему, яка відповідає потребам користувачів та забезпечує високу якість роботи з даними та обробку бізнес-процесів. DDD не є просто набором технік або патернів, але це цілісний підхід до розробки програмного забезпечення, який змінює спосіб мислення проектувальників та розробників.

3.2 Проектування доменного шару системи

Доменний шар є ключовою складовою архітектури DDD і включає наступні складові:

1. Модель домену (Domain entity model): Модель домену є центральною частиною доменного шару. Вона представляє собою внутрішнє відображення бізнес-процесів, правил та концепцій, що використовуються в системі.

2. Доменні сутності з даними та поведінкою: Доменні сутності включають дані та поведінку, які відображають ключові аспекти домену. Вони можуть мати властивості для збереження даних та методи для здійснення бізнес-операцій та виконання правил домену.

3. Папка SeedWork: У папці SeedWork знаходяться загальні класи, які містять базову логіку, методи і властивості, що є спільними для багатьох доменних об'єктів. Використання SeedWork допомагає уникнути дублювання коду та забезпечує однорідність та згуртованість в доменному шарі.

4. Контракти/інтерфейси репозиторіїв: Репозиторії визначають контракти/інтерфейси для збереження та витягування доменних сутностей з бази даних або іншого джерела даних. Це дозволяє розділити логіку доступу до

даних від логіки домену, підтримуючи принцип високого рівня абстракції та розширюваності. На рисунку 3.2 наведено інтерфейси репозиторіїв для моєї системи.

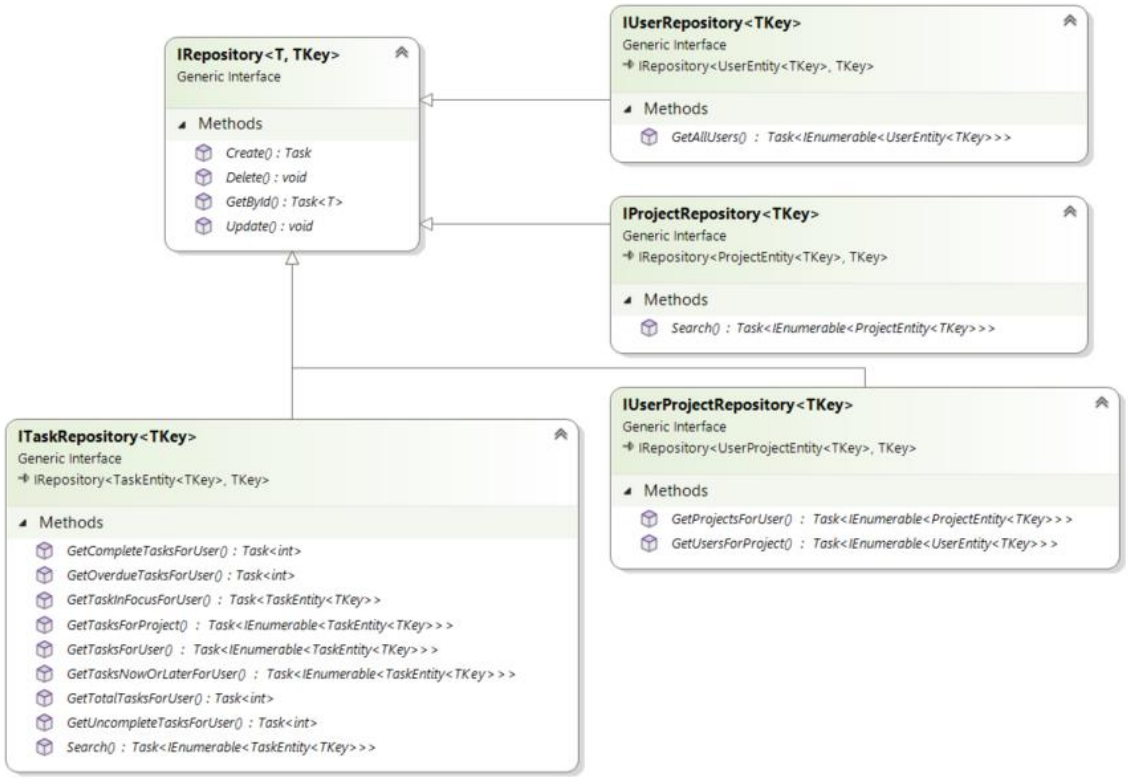


Рисунок 3.2 – Інтерфейси репозиторіїв системи

На рисунку 3.3 наведено структуру проекту для доменного шару системи для управління і відстеження помилок.

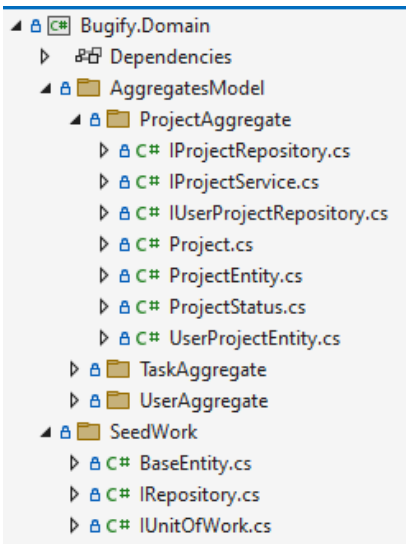


Рисунок 3.3 – Структура доменного шару системи

Доменний шар DDD використовує патерни, такі як доменна сутність (domain entity), агрегат (aggregate), корінь агрегату (aggregate root) та контракти/інтерфейси репозиторіїв. Ці патерни допомагають організувати та моделювати комплексність домену та взаємодію з ним.

Патерн "Domain Entity" (доменна сутність) є одним із основних патернів в архітектурі Domain-Driven Design (DDD). Він допомагає моделювати та втілювати ключові поняття домену у вигляді об'єктів. Основна ідея патерну "Domain Entity" полягає у тому, що доменні сутності повинні мати унікальну ідентичність, яка визначається через їх атрибути. Це дозволяє розрізняти різні сутності та встановлювати зв'язки між ними. На рисунку 3.4 розглянута структури класів основних сутностей системи.

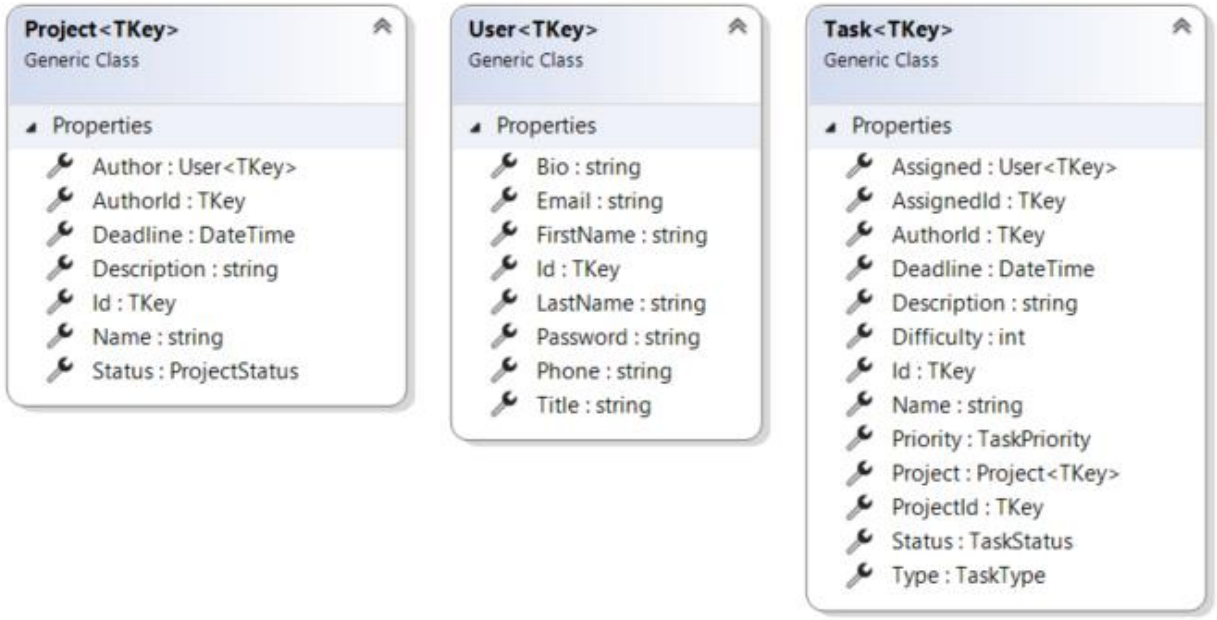


Рисунок 3.4 – Структура класів основних сутностей

Патерн "Aggregate" (агрегат) допомагає управляти консистентністю та зв'язками між доменними об'єктами. Агрегат є групою пов'язаних доменних сутностей, які взаємодіють як один цілісний об'єкт.

Основна ідея патерну "Aggregate" полягає у тому, що він визначає кореневий об'єкт агрегату, який виступає як точка входу для доступу до всіх

інших об'єктів в агрегаті. Кореневий об'єкт відповідає за забезпечення цілісності та управління змінами всередині агрегату. Наприклад, на рисунку 3.5 наведено агрегат для сутності Task, де

- ITaskRepository – інтерфейс репозиторію;
- Task, TaskEntity – класи, які реалізують доменну сутність;
- TaskPriority, TaskStatus, TaskType – типи перерахування, які зберігають відповідні дані.

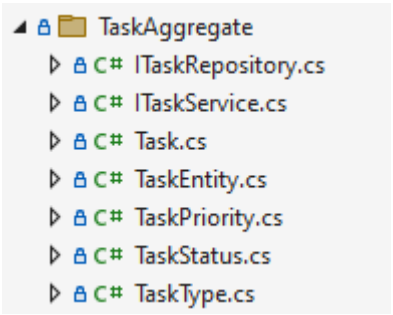


Рисунок 3.5 – Структура агрегату для сутності Task

Отже, у результаті проектування доменного шару було визначено основні складові, які включають доменну модель, доменні сутності, агрегати та підходи DDD. Доменна модель визначає основні поняття та бізнес-правила системи, а доменні сутності представляють конкретні об'єкти та їх взаємодіють. Агрегати служать для групування доменних сутностей та забезпечення їх консистентності.

Проектування доменного шару відіграє важливу роль у створенні масштабованих, гнучких та сильно зв'язаних систем управління задачами та відстеження помилок. Правильне визначення складових, використання патернів та підходів DDD допомагають забезпечити чистоту коду, модульність та легкість розширення системи в майбутньому.

3.3 Проектування схеми бази даних

При проектуванні бази даних для системи управління задачами та відстеження помилок у хмарному середовищі, одним із ключових етапів є визначення сутностей, атрибутів і зв'язків, що відображають основні компоненти цієї системи. Цей процес допомагає створити структуру бази даних, яка відповідає потребам проекту і забезпечує зручний та ефективний доступ до інформації.

На першому етапі проектування, необхідно визначити основні сутності, які існують у системі. У випадку системи управління задачами та відстеження помилок, такими сутностями можуть бути "Задача", "Проект" та "Користувач". Кожна з цих сутностей відображає конкретний об'єкт або елемент, з яким ви будете працювати.

На наступному етапі для кожної сутності необхідно визначити атрибути, які описують її характеристики та властивості. Кожен атрибут має свій тип даних і визначається його обмеженнями.

Визначення атрибутів для сутності "Tasks" наведено в таблиці 3.1.

Таблиця 3.1 - Атрибути сутності "Tasks"

Атрибут	Тип атрибута	Призначення
Id	цілочисельний тип	унікальний ідентифікатор задачі
Name	рядковий тип, максимальна довжина 255 символів	назва задачі
Description	рядковий тип, максимальна довжина 255 символів	опис задачі
ProjectId	цілочисельний тип	ідентифікатор проекту, до якого належить задача
AuthorId	цілочисельний тип	ідентифікатор автора задачі

Кінець таблиці 3.1.

Атрибут	Тип атрибута	Призначення
AssignedId	цілочисельний тип	ідентифікатор користувача, якому призначено задачу
Status	цілочисельний тип	статус задачі
Type	цілочисельний тип	тип задачі
Priority	цілочисельний тип	пріоритет задачі
Difficulty	цілочисельний тип	складність задачі
Deadline	тип datetime	крайній термін виконання задачі
Created	тип datetime	дата та час створення задачі

Визначення атрибутів для сутності "Projects" наведено в таблиці 3.2.

Таблиця 3.2 - Атрибути сутності "Projects"

Атрибут	Тип атрибута	Призначення
Id	цілочисельний тип	унікальний ідентифікатор проекту
Name	рядковий тип, максимальна довжина 255 символів	назва проекту
Description	рядковий тип, максимальна довжина 255 символів	опис проекту
AuthorId	цілочисельний тип	ідентифікатор автора проекту
Status	цілочисельний тип	статус проекту

Кінець таблиці 3.2.

Атрибут	Тип атрибута	Призначення
Deadline	тип datetime	крайній термін виконання проекту
Created	тип datetime	дата та час створення проекту

Визначення атрибутів для сутності "Users" наведено в таблиці 3.3.

Таблиця 3.3 - Атрибути сутності "Users"

Атрибут	Тип атрибута	Призначення
Id	цілочисельний тип	унікальний ідентифікатор користувача
FirstName	рядковий тип, максимальна довжина 255 символів	ім'я користувача
LastName	рядковий тип, максимальна довжина 255 символів	прізвище користувача
Email	рядковий тип, максимальна довжина 255 символів	електронна адреса користувача
Phone	рядковий тип, максимальна довжина 255 символів	номер телефону користувача
Title	рядковий тип, максимальна довжина 255 символів)	посада користувача
Bio	рядковий тип, максимальна довжина 255 символів	біографія користувача
Created	тип datetime	дата та час створення користувача
Password	рядковий тип, максимальна довжина 255 символів	пароль користувача

Останнім етапом є визначення зв'язків між сутностями. Зв'язки вказують на взаємозв'язок між двома сутностями і можуть мати різні типи, такі як один до багатьох, один до одного, або багато до багатьох. Наприклад, в моїй системі "Задача" може мати зв'язок "багато до одного" з "Проектом", оскільки кожна задача пов'язана з певним проектом. Також, "Задача" може мати зв'язок "багато до одного" з "Користувачем", оскільки кожна задача може бути призначена конкретному користувачеві. Зв'язки визначаються за допомогою зовнішніх ключів, які забезпечують цілісність даних і зв'язків між таблицями. Далі наведено зв’язки між таблицями та встановлені зовнішні ключі:

- Зв'язок "Tasks" до "Projects":
    - Зовнішній ключ "ProjectId" у таблиці "Tasks" посилається на поле "Id" у таблиці "Projects".
- Зв'язок "Tasks" до "Users":
    - Зовнішній ключ "AuthorId" у таблиці "Tasks" посилається на поле "Id" у таблиці "Users".
    - Зовнішній ключ "AssignedId" у таблиці "Tasks" посилається на поле "Id" у таблиці "Users".
- Зв'язок "Projects" до "Users" (AuthorId):
    - Зовнішній ключ "AuthorId" у таблиці "Projects" посилається на поле "Id" у таблиці "Users".
- Зв'язок багато-багато "Users" до "Projects" через проміжну таблицю "UserProject":
    - Зовнішній ключ "UserId" у таблиці "UserProject" посилається на поле "Id" у таблиці "Users".
    - Зовнішній ключ "ProjectId" у таблиці "UserProject" посилається на поле "Id" у таблиці "Projects".

Щоб забезпечити ефективне та організоване зберігання і доступ до даних системи необхідно розробити нормалізовану схему бази даних. Для цього

необхідно, щоб дані були коректно структуровані, не містили зайвого дублювання і могли легко оновлюватись та запитуватись. Це означає, що дані повинні бути розподілені по таблицям таким чином, щоб не було зайвого дублювання і виразів залежності. Необхідно, щоб кожна таблиця містила лише пов'язані з нею атрибути. Кожна таблиця повинна мати первинний ключ, який унікально ідентифікує кожен запис у таблиці. Наступним кроком є встановлення зв'язків між таблицями. Також, кожна таблиця повинна містить дані тільки про один тип об'єкта, а зв'язки між таблицями відображають правильні залежності між ними.

Отже, визначення сутностей дозволяє ідентифікувати ключові об'єкти, з якими будемо працювати, такі як задачі, проекти та користувачі. Атрибути надають деталізовану інформацію про кожну сутність, їх характеристики та властивості. Зв'язки вказують на взаємозв'язок між сутностями і визначають структуру взаємодії. Проектування бази даних з урахуванням цих аспектів забезпечує належну організацію даних та їх ефективне зберігання. Це впливає на продуктивність системи, спрощує доступ до необхідної інформації та забезпечує її цілісність.

Правильно визначена структура бази даних допомагає забезпечити ефективну роботу системи управління задачами та відстеження помилок, спрощує розробку функціональності та підтримку системи у майбутньому. Важливо приділити належну увагу цьому етапу проектування, щоб забезпечити успішну реалізацію системи та задоволення потреб її користувачів.

3.4 Проектування інфраструктурного шару системи

Інфраструктурний шар включає в себе реалізацію інфраструктури, необхідної для ефективної роботи системи управління задачами та відстеження помилок. Основною метою цього шару є забезпечення доступу до даних, робота

з зовнішніми сервісами та інші технічні аспекти, які не пов'язані безпосередньо з бізнес-логікою.

На рисунку 3.6 наведено структуру проекту для інфраструктурного шару системи для управління і відстеження помилок.

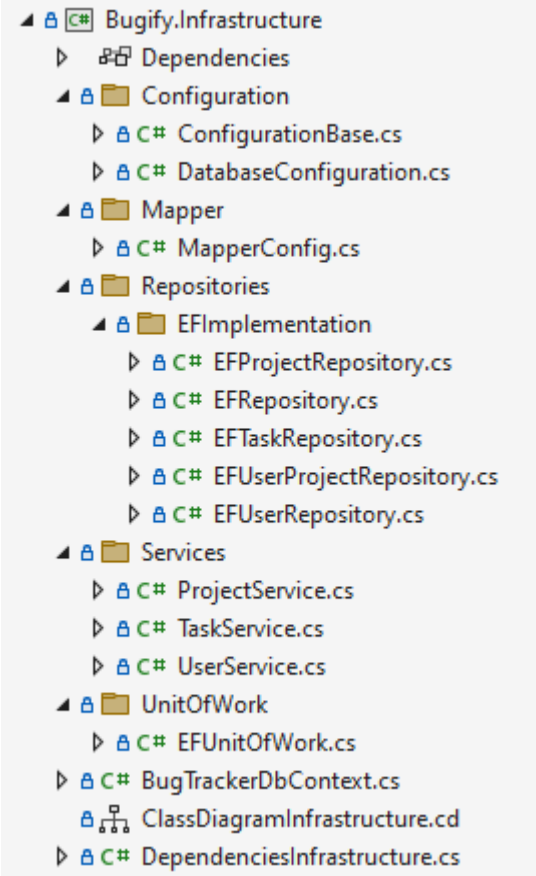


Рисунок 3.6 – Структура інфраструктурного шару системи

Основні складові інфраструктурного шару включають:

1. Інфраструктура збереження даних - реалізація репозиторіїв: Ця складова відповідає за забезпечення доступу до бази даних та збереження даних, пов'язаних з системою управління задачами та відстеження помилок. Використання репозиторіїв дозволяє абстрагуватися від конкретної бази даних і забезпечити однаковий інтерфейс для роботи з даними незалежно від використовуваної технології збереження.
2. Використання ORM або API доступу до даних - Entity Framework Core: ORM система Entity Framework Core використовується для

зручного та ефективного доступу до даних. Вона дозволяє легко працювати з базою даних і мапувати об'єкти домену на таблиці бази даних.

3. Інші інфраструктурні реалізації, використовувані з рівня додатку - логування: Крім доступу до даних, інфраструктурний шар може включати інші реалізації, такі як система логування. Логування дозволяє відстежувати події та помилки, що виникають у системі, що є важливим для налагодження, виявлення проблем та покращення продуктивності.

Для забезпечення надійності, масштабованості та ефективності інфраструктури я використовую такі патерни як Unit of Work і Repository.

Патерн Unit of Work (Одиниця роботи) є архітектурним патерном, який використовується для управління транзакціями та змінами в базі даних. Він дозволяє групувати операції збереження та модифікації даних в одну одиницю, яка може бути здійснена атомарно. На рисунку 3.7 наведено структуру класу, який реалізую цей патерн.

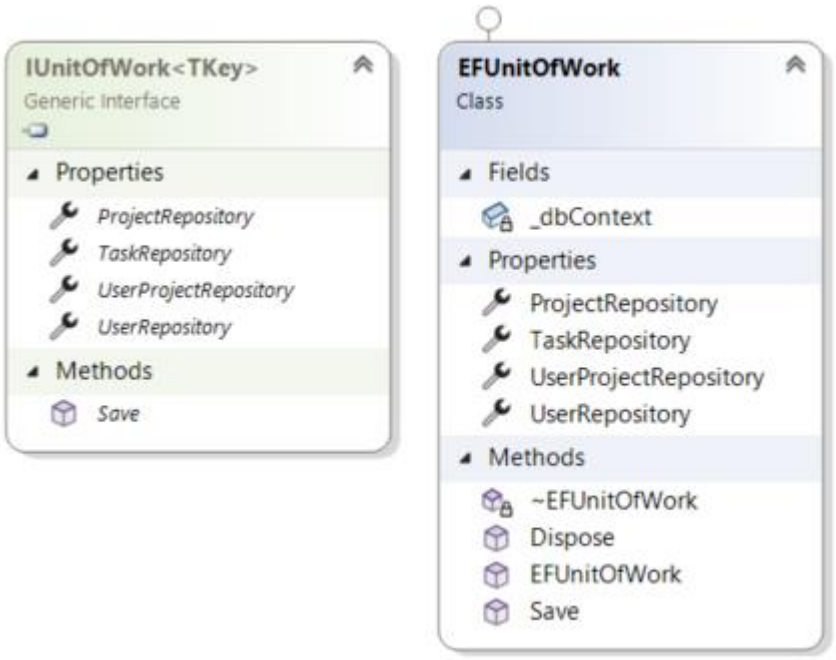


Рисунок 3.7 – Структура класа EFUnitOfWork

Основна ідея Unit of Work полягає в тому, що він визначає контекст для виконання декількох операцій з базою даних як єдину транзакцію. Він забезпечує механізм відстеження змін, що відбуваються з об'єктами домену під час виконання операцій, і здійснює збереження цих змін у базі даних після завершення транзакції.

Unit of Work може бути реалізований шляхом створення спеціального класу, який містить методи для реєстрації, збереження та відкату змін в базі даних. Він може також включати логіку для керування транзакціями та відновленням стану об'єктів домену після відкату.

Переваги використання шаблону Unit of Work полягають в тому, що він забезпечує консистентність даних і гарантує, що зміни будуть виконані атомарно. Він спрощує керування транзакціями та збереженням даних, забезпечуючи єдиний механізм для взаємодії з базою даних.

Шаблон репозиторію (Repository pattern) є одним з ключових архітектурних патернів, який використовується для розділення бізнес-логіки від механізмів доступу до даних. Він надає єдиний інтерфейс для взаємодії з даними, приховуючи деталі роботи з базою даних або іншими джерелами даних.

Основна ідея репозиторію полягає в тому, що він представляє собою абстракцію над збереженням, вибіркою та модифікацією даних. Він надає методи для створення, читання, оновлення та видалення об'єктів домену, приховуючи деталі доступу до бази даних. Це дозволяє розробникам працювати з даними на вищому рівні абстракції, не прив'язуючись до конкретної реалізації збереження.

Репозиторій може бути реалізований шляхом створення інтерфейсу або абстрактного класу, який визначає методи для роботи з даними, такі як додавання, вибірка, оновлення та видалення. Конкретна реалізація репозиторію включає в себе механізми доступу до бази даних, включаючи SQL-запити або

використання ORM-інструментів. На рисунку 3.8 наведена структура розроблених репозиторіїв системи.

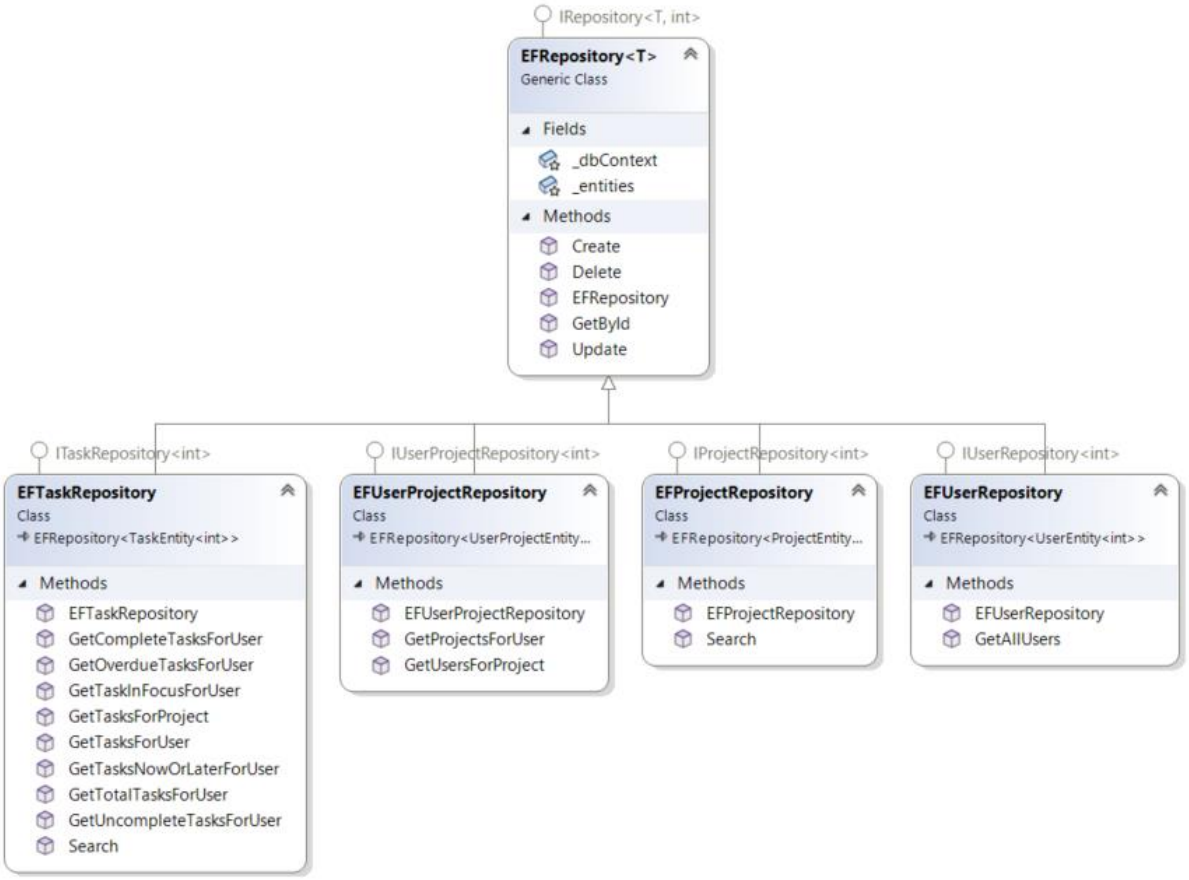


Рисунок 3.8 – Структура репозиторіїв системи

Переваги використання шаблону репозиторію полягають в тому, що він дозволяє відокремити бізнес-логіку від деталей доступу до даних. Це спрощує тестування, покращує зрозуміння коду та забезпечує гнучкість системи, оскільки зміни в механізмах доступу до даних не впливають на бізнес-логіку.

Узагальнюючи, проектування інфраструктурного шару системи має важливе значення, оскільки він визначає основні компоненти і механізми, які забезпечують функціонування системи в цілому. Він дозволяє розподілити відповідальності та забезпечити модульність системи, що сприяє покращенню її супроводу, розширюваності та масштабованості.

Правильне проектування інфраструктурного шару в системі для управління задачами та відстеження помилок у хмарному середовищі є важливим кроком у створенні стабільної та ефективної системи, яка задовольняє потреби користувачів і забезпечує успішну роботу бізнесу.

3.5 Реалізація алгоритмів системи

Для поліпшення ефективності та продуктивності виконання проектів я розробила два алгоритми. Схеми цих алгоритмів наведені у додатку 3. Один з них це алгоритм для визначення завдання “у фокусі”, його реалізація `GetTaskInFocusForUser` наведена на рисунку 3.9.

```
public async Task<TaskEntity<int>> GetTaskInFocusForUser(
    int userId)
{
    return await _entities.AsQueryable()
        .Where(task => task.AssignedId.Equals(userId)
                       && (task.Status == TaskStatus.New ||
                           task.Status == TaskStatus.InProgress)
                       && task.Project.Status != ProjectStatus.Closed)// IQueryable<TaskEntity<…>>
        .OrderBy(task => task.Deadline)
        .ThenByDescending(task => task.Priority)
        .ThenByDescending(task => task.Type)// IOrderedQueryable<TaskEntity<…>>
        .FirstAsync();// Task<TaskEntity<…>>
}
```

Рисунок 3.9 – Реалізація функції `GetTaskInFocusForUser`

Далі наведено покроковий алгоритм для функції `GetTaskInFocusForUser`:

1. Початок функції `GetTaskInFocusForUser` з передачею `userId` в якості аргументу.

2. Створення запиту до бази даних для отримання списку завдань за допомогою `_entities.AsQueryable()`.

3. Застосування фільтрів до запиту:

- Перевірка, чи поле `AssignedId` завдання співпадає з переданим `userId`.

- Перевірка, чи статус завдання є `New` або `InProgress`.

- Перевірка, чи статус проекту, пов'язаного з завданням, не є `Closed`.

4. Сортування результатів за кінцевим терміном `Deadline` в порядку зростання за допомогою `.OrderBy(task => task.Deadline)`.

5. У випадку, якщо кінцеві терміни однакові, сортування за спаданням за пріоритетом завдання `.ThenByDescending(task => task.Priority)`.

6. У випадку, якщо пріоритети також однакові, сортування за спаданням за типом завдання `.ThenByDescending(task => task.Type)`.

7. Виконання запиту до бази даних з використанням `.FirstAsync()`, яке повертає перше завдання зі відповідного списку, що задовольняє всі умови.

8. Повернення результату виконання функції як об'єкта `TaskEntity<int>`, який упаковує значення першого завдання.

Узагальнюючи, цей алгоритм виконує запит до бази даних, фільтрує та сортує результати, а потім повертає перше завдання, що задовольняє умови, в якості об'єкта `TaskEntity<int>`.

Інший алгоритм, який я додала це алгоритм для створення списку задач “зараз чи потім”, його реалізація `GetTasksNowOrLaterForUser` наведена на рисунку 3.10

```
public async Task<IEnumerable<TaskEntity<int>>> GetTasksNowOrLaterForUser(
    int userId)
{
    return await _entities.AsQueryable()
        .Where(task => task.AssignedId.Equals(userId)
                       && (task.Status == TaskStatus.New ||
                           task.Status == TaskStatus.InProgress)
                       && task.Project.Status != ProjectStatus.Closed) // IQueryable<TaskEntity<…>>
        .OrderBy(task => task.Difficulty)
        .ThenByDescending(task => task.Priority)
        .ThenBy(task => task.Deadline)
        .ThenBy(task => task.Type) // IOrderedQueryable<TaskEntity<…>>
        .Take(4) // IQueryable<TaskEntity<…>>
        .ToListAsync(); // Task<List<…>>
}
```

Рисунок 3.10 – Реалізація функції `GetTasksNowOrLaterForUser`

Ось покроковий алгоритм для функції `GetTasksNowOrLaterForUser`:

1. Початок функції `GetTasksNowOrLaterForUser` з передачею `userId` в якості аргументу.

2. Створення запиту до бази даних для отримання списку завдань за допомогою `_entities.AsQueryable()`.

3. Застосування фільтрів до запиту:

- Перевірка, чи поле `AssignedId` завдання співпадає з переданим `userId`.

- Перевірка, чи статус завдання є `New` або `InProgress`.

- Перевірка, чи статус проекту, пов'язаного з завданням, не є `Closed`.

4. Сортування результатів за складністю завдання `Difficulty` в порядку зростання за допомогою `.OrderBy(task => task.Difficulty)`.

5. У випадку, якщо складності однакові, сортування за спаданням за пріоритетом завдання `.ThenByDescending(task => task.Priority)`.

6. У випадку, якщо пріоритети також однакові, сортування за кінцевим терміном `Deadline` в порядку зростання `.ThenBy(task => task.Deadline)`.

7. У випадку, якщо кінцеві терміни та пріоритети однакові, сортування за типом завдання `.ThenBy(task => task.Type)`.

8. Вибір перших 4 завдань з результатів запиту за допомогою `.Take(4)`.

9. Виконання запиту до бази даних з використанням `.ToListAsync()`, яке повертає список перших 4 завдань, що задовольняють всі умови.

10. Повернення результату виконання функції як колекції об'єктів `TaskEntity<int>`.

Узагальнюючи, цей алгоритм виконує запит до бази даних, фільтрує та сортує результати, а потім повертає перші 4 завдання, які задовольняють умови, у вигляді колекції об'єктів `TaskEntity<int>`.

3.6 Проектування шару застосунку системи

При проектуванні шару застосунку системи для управління задачами та відстеження помилок у хмарному середовищі, я зосередилась на розробці

функціональності, яка взаємодіє з користувачем і надає зручний та інтуїтивно зрозумілий інтерфейс для роботи з системою. Цей шар відповідає за прийом та обробку запитів користувача, валідацію введених даних, виконання дій у відповідності до бізнес-правил і забезпечення взаємодії з доменним шаром. Він також включає компоненти для автентифікації, авторизації, обробки сесій, забезпечення безпеки та інші функції, які не пов'язані безпосередньо з бізнес-логікою, але важливі для коректної роботи системи.

На рисунку 3.11 наведена структура шару застосунку для систем для управління і відстеження помилок.

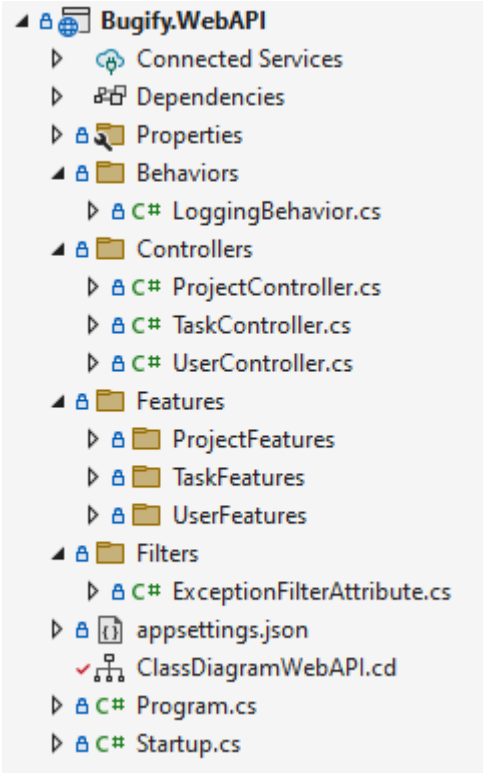


Рисунок 3.11 - Структура шару застосунку системи

У складі шару застосунку можна виділити наступні компоненти:

1. ASP.NET Web API: Це фреймворк, який дозволяє визначати ендпоінти, роути і контролери, які обробляють запити та повертають відповіді.

2. API контракти/реалізація: У цьому компоненті визначаються контракти для взаємодії з системою. Це класи, які описують доступні операції та дані, що можуть бути передані чи отримані через API.

3. Команди та обробники команд (Commands and Command Handlers): Команди представляють запити від користувача на виконання певної дії в системі. Обробники команд (Command Handlers) відповідають за прийом та обробку цих команд, виконання необхідних операцій і спілкування з доменним шаром для здійснення бізнес-логіки.

4. Запити (Queries): Запити представляють запити користувача на отримання даних з системи. Їх обробка здійснюється за допомогою відповідних обробників запитів (Query Handlers), які отримують запит, взаємодіють з доменним шаром для отримання необхідних даних і повертають результат користувачу.

Компоненти шару застосунку взаємодіють зі змінними даними, передаючи необхідну інформацію до доменного шару для обробки та виконання бізнес-логіки. Для розробки шару застосунку я використовувала патерни CQRS і Mediator.

Патерн CQRS (Command Query Responsibility Segregation) - це архітектурний патерн, який розділяє операції команд (Command) та запитів (Query) в системі. За допомогою цього патерну, команди та запити обробляються окремо, використовуючи різні моделі та механізми.

Основна ідея патерну CQRS полягає в тому, що операції зміни стану (команди) та операції отримання стану (запити) вимагають різних підходів. Замість того, щоб мати одну модель, яка відповідає як за команди, так і за запити, я розділила їх на дві окремі моделі.

У контексті системи управління задачами та відстеження помилок у хмарному середовищі, можна застосувати патерн CQRS таким чином:

1. Команди (Commands): Команди відповідають за модифікацію стану системи. Наприклад, створення нової задачі або оновлення існуючої. Команди передаються до доменного шару, де вони обробляються відповідними обробниками команд, які виконують необхідні дії та змінюють стан системи.

2. Запити (Queries): Запити відповідають за отримання даних з системи без їх модифікації. Наприклад, отримання списку всіх задач або детальної інформації про конкретну задачу. Запити передаються до відповідних обробників запитів, які взаємодіють з читаючими моделями та повертають результат користувачу.

На рисунку 3.12 наведено перелік запитів і команд для сутності Task.

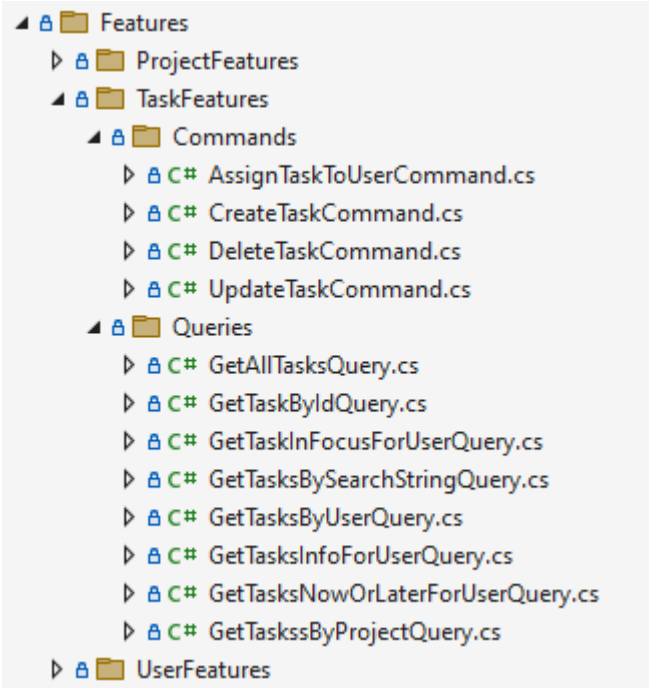


Рисунок 3.12 – Запити і команди для сутності Task

Застосування патерну CQRS дозволяє оптимізувати систему, використовуючи різні підходи до команд та запитів. Це дозволяє досягти більшої швидкодії та масштабованості, а також полегшує розробку та тестування окремих компонентів системи.

У підході CQRS використовується патерн "Посередник" (Mediator) у пайплайні команд для обробки запитів в пам'яті. Цей розумний посередник схожий на шину в пам'яті і вміє перенаправляти команди до відповідних обробників на основі їх типу або DTO.

У контролері ASP.NET Core команда передається до пайплайну команд MediatR, щоб вона потрапила до відповідного обробника. Використання патерну "Посередник" має сенс у додатках, де обробка запитів може бути

складною. На рисунку 3.13 наведено приклад застосування цього патерну у системі для управління задачами.

```
namespace Bugify.WebAPI.Controllers;

[Route( template: "/[controller]")]
[ApiController]
[TypeFilter(typeof(ExceptionFilter))]
public class ProjectController : ControllerBase
{
    private readonly IMediator _mediator;

    public ProjectController(IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpGet( template: "{id}")]
    [ProducesResponseType(typeof(Project<int>), statusCode: StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async System.Threading.Tasks.Task<IActionResult> GetById(int id)
    {
        return Ok(await _mediator.Send( request: new GetProjectByIdQuery
            { ProjectId = id }));
    }
```

Рисунок 3.13 – Приклад використання посередника в ProjectController

Отже, посередник - це об'єкт, який виконує координацію процесу обробки команд на основі стану, способу виклику обробників команд або переданої інформації. За допомогою посередника можна легко додавати суттєві функції із забезпеченням їх централізованого та прозорого застосування за допомогою декораторів.

У результаті проектування шару застосунку було розроблено структуровану та організовану систему, що включає Asp.net Web API, API контракти/реалізації, команди та обробники команд, а також запити. Використання патерну CQRS дозволило ефективно розділити команди та запити, сприяючи кращій організації логіки системи. Результатом є гнучкий та високопродуктивний шар застосунку, який відповідає вимогам і забезпечує зручний доступ до функціональності системи.

3.7 Проектування фронтенд частини системи

Фронтенд частина системи для управління задачами і відстеження помилок у розробляється на основі фреймворка React і компонентної архітектури.

Компонентна архітектура організовує програму у набір незалежних та повторно використовуваних компонентів, які можуть бути складені разом для створення складніших інтерфейсів. На рисунку 3.14 наведено файлову структуру фронтенд частини системи.

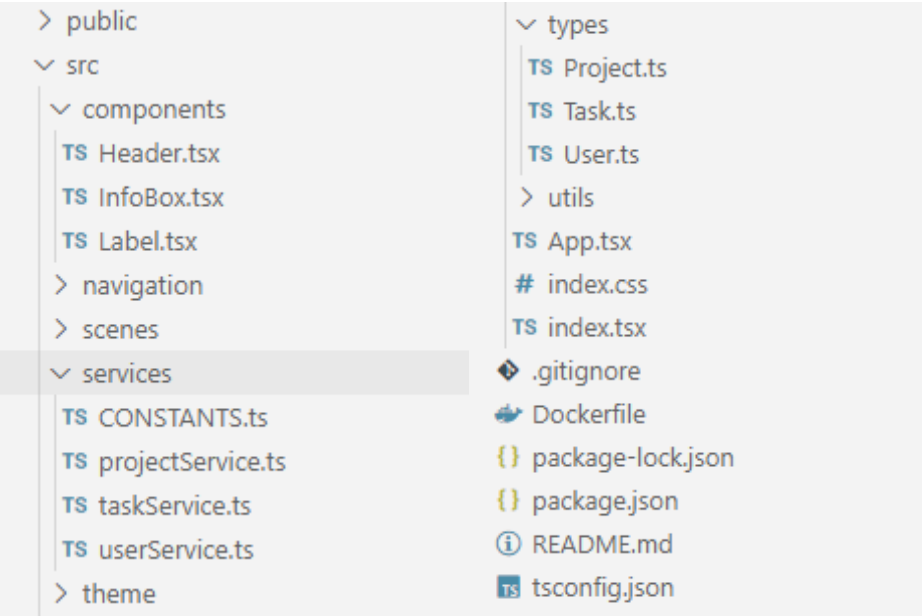


Рисунок 3.14 – Файлова структура фронтенд частини системи

У проекті компоненти знаходяться у каталозі `components` і вони розділяються на різні частини, такі як `Header`, `InfoBox` та `Label`. Кожен компонент відповідає за конкретну функціональність і відображення певної частини інтерфейсу користувача. Це дозволяє легко керувати окремими компонентами, розширювати їх та повторно використовувати у різних частинах вашого додатку.

Наприклад, `Header` компонент відповідає за відображення заголовку системи на кожній сторінці. Він містить всі елементи, які потрібні для

заголовку, реалізація цього компонента наведена на рисунку 3.15. Завдяки розділенню цієї частини інтерфейсу в окремий компонент, можна змінювати або розширювати заголовок, не впливаючи на інші частини додатку.

src > components > TS Header.tsx > ...

```tsx
import { FC } from "react";
import { Typography, Box } from "@mui/material";
import { useTheme, Theme } from "@mui/material/styles";
import { tokens } from "../theme/theme";

interface HeaderProps {
  title?: string;
  subtitle?: string;
}

const Header: FC<HeaderProps> = ({ title, subtitle }) => {
  const theme: Theme = useTheme();
  const { grey, greenAccent } = tokens(theme.palette.mode);
  return (
    <Box mb={2}>
      <Typography variant="h2" color={grey[1]} fontWeight="bold" sx={{ mb: 1 }}>
        {title}
      </Typography>
      {subtitle && (
        <Typography variant="h5" color={greenAccent[4]}>
          {subtitle}
        </Typography>
      )}
    </Box>
  );
};

export default Header;
```

Рисунок 3.15 – Реалізація компоненти `Header`

Також, у проекті є сцени (scenes) у каталозі `scenes`, які також використовують компонентну архітектуру. Наприклад, `Dashboard` сцена відповідає за відображення головної інформаційної панелі, де користувач може переглядати загальну інформацію про проекти, завдання та користувачів. Ця сцена включає компоненти, такі як списки проектів, завдань та користувачів, а також графіки або інші елементи, які необхідні для відображення головної панелі. Також, користувач може перглядати поточний прогрес його роботи та бачити наступну задачу.

Файлова структура каталогу `scenes` наведена на рисунку 3.16.

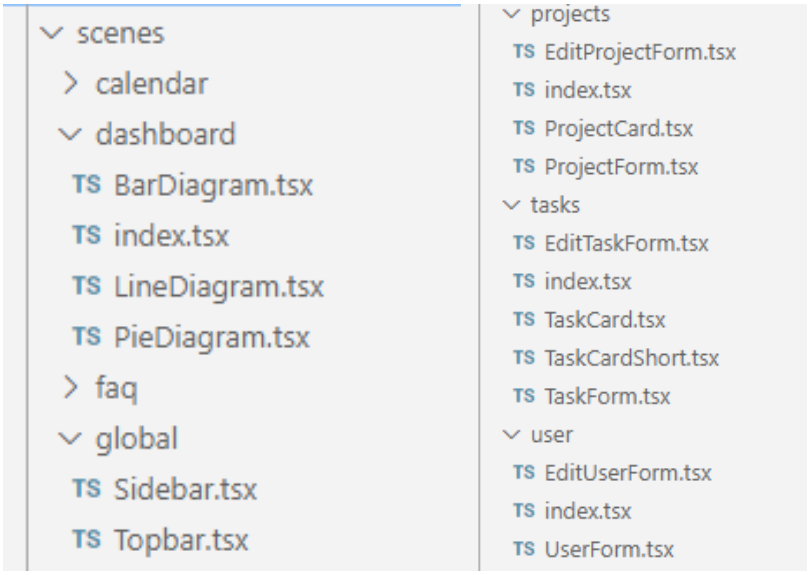


Рисунок 3.16 - Файлова структура каталогу `scenes`

Компоненти можуть бути вкладеними один в одного, що дозволяє побудувати складніші інтерфейси. Наприклад, компонент `Projects`, який включає компонент `ProjectItem`, який в свою чергу містить компоненти `TaskList` та інші. Така ієрархічна структура дозволяє логічно організувати компоненти та зручно керувати їх станом та поведінкою.

Окрім компонентів, у проекті також використовуються сервіси у каталозі `services`. Ці сервіси відповідають за взаємодію з сервером та виконання різних операцій, таких як отримання даних про проекти, завдання та користувачів. Використання сервісів дозволяє розділити логіку взаємодії з сервером від компонентів, що сприяє більшій чистоті коду та його повторному використанню.

Загалом, компонентна архітектура проекту дозволяє ефективно організовувати код, спрощує розробку та підтримку системи, а також покращує перевикористання компонентів. Вона забезпечує модульність, гнучкість та легкість розширення фронтенд додатку.

ВИСНОВОК ДО РОЗДІЛУ 3

У цьому розділу проведеного архітектурний аналізу та проектування системи, на основі аналізу вимог до системи.

Основою для архітектури системи був використаний архітектурний патерн Domain-Driven Design (DDD), який виявився вельми корисним. DDD надає розробникам потужні інструменти для моделювання складних доменних проблем і забезпечує збільшення розуміння бізнес-логіки системи. Застосування патерну DDD для системи управління задачами і відстеження помилок у хмарному середовищі допомогло створити більш гнучку, розширювану та бізнес-орієнтовану систему, яка відповідає потребам користувачів та забезпечує високу якість роботи з даними та обробку бізнес-процесів.

Проектування доменного шару системи було зосереджене на визначенні сутностей, атрибутів та зв'язків, а також використанні DDD-патернів, таких як доменні сутності, агрегати, агрегатний корінь та значущий об'єкт. Це дозволило створити гнучку та модульну систему, яка відображає ділову логіку та вимоги до домену.

Проектування інфраструктурного шару системи включало розробку компонентів для зберігання даних, таких як репозиторії та інтерфейси доступу до даних. Використання ORM, такого як Entity Framework Core, сприяло швидкому та ефективному доступу до бази даних. Завдяки правильному проектуванню та реалізації складових інфраструктурного шару, система управління задачами та відстеження помилок може працювати ефективно та надійно, забезпечуючи швидкий доступ до даних і виконання технічних завдань, що вимагаються для її функціонування.

Проектування схеми бази даних було проведене з урахуванням нормалізації та ефективності запитів. Були визначені таблиці, стовпці та зв'язки,

які відображають бізнес-сутності та їх взаємодію. Правильно визначена структура бази даних допомогла забезпечити ефективну роботу системи управління задачами та відстеження помилок, спростила розробку функціональності та підтримку системи у майбутньому.

Проектування шару застосунку системи включало розробку API, контролерів, команд та запитів. Використання патерну CQRS дозволило розділити команди і запити, що сприяє кращій організації логіки системи та підвищує її продуктивність. Проектування шару застосунку має велике значення для створення зручного та зрозумілого інтерфейсу між користувачем та системою. В результаті проектування шару застосунку було досягнуто балансу між функціональністю та простотою в управлінні.

В цілому, архітектурний аналіз та проектування системи дозволили створити гнучку, модульну та масштабовану систему для управління і відстеження помилок, яка відповідає бізнес-вимогам і забезпечує ефективну роботу з даними та логікою домену.

РОЗДІЛ 4. РОЗГОРТАННЯ І ТЕСТУВАННЯ СИСТЕМИ

4.1 Розгортання системи у хмарному середовищі

Розгортання системи для управління задачами і відстеження помилок у хмарному середовищі є важливим етапом в процесі розробки та впровадження програмного забезпечення. В цьому пункті буде розглянуто розгортання системи на платформі Azure, від створення ресурсів до налаштування взаємодії в системі.

Перший крок у розгортанні системи в хмарному середовищі Azure - створення Resource Group. Resource Group є логічним контейнером, який групує всі ресурси, пов'язані зі системою. Для системи я створила Resource Group під назвою BugifyAPIRecourceGroup в регіоні Germany West Central, створений контейнер наведений на рисунку 4.1. Вибір регіону має важливе значення, оскільки він визначає фізичну локацію серверів Azure, на яких будуть розгорнуті ресурси. Рекомендується вибрати регіон, що найближчий до місця розташування користувачів для забезпечення низької затримки та кращої продуктивності.

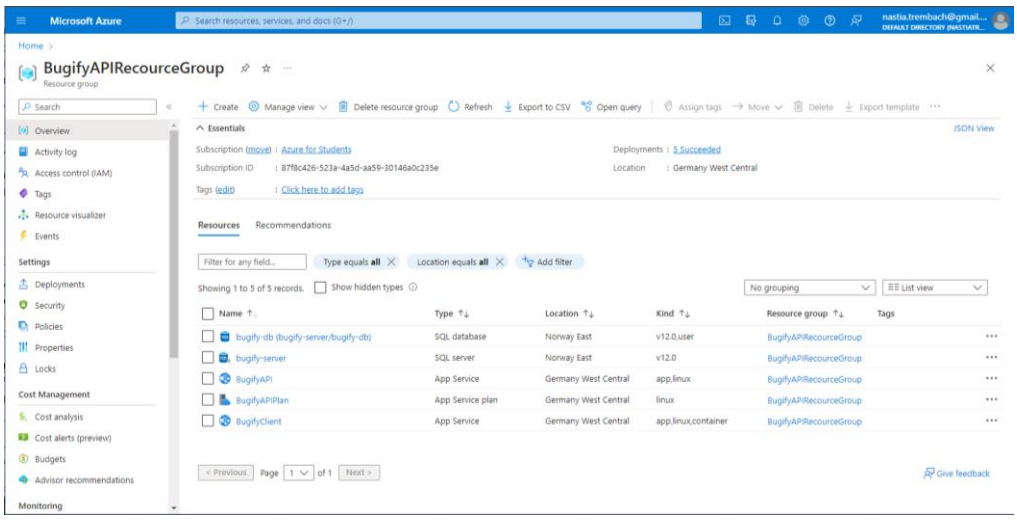


Рисунок 4.1 – Створений Resource Group

На рисунку 4.2 зображена структура взаємодії всіх ресурсів у створеному контейнері.

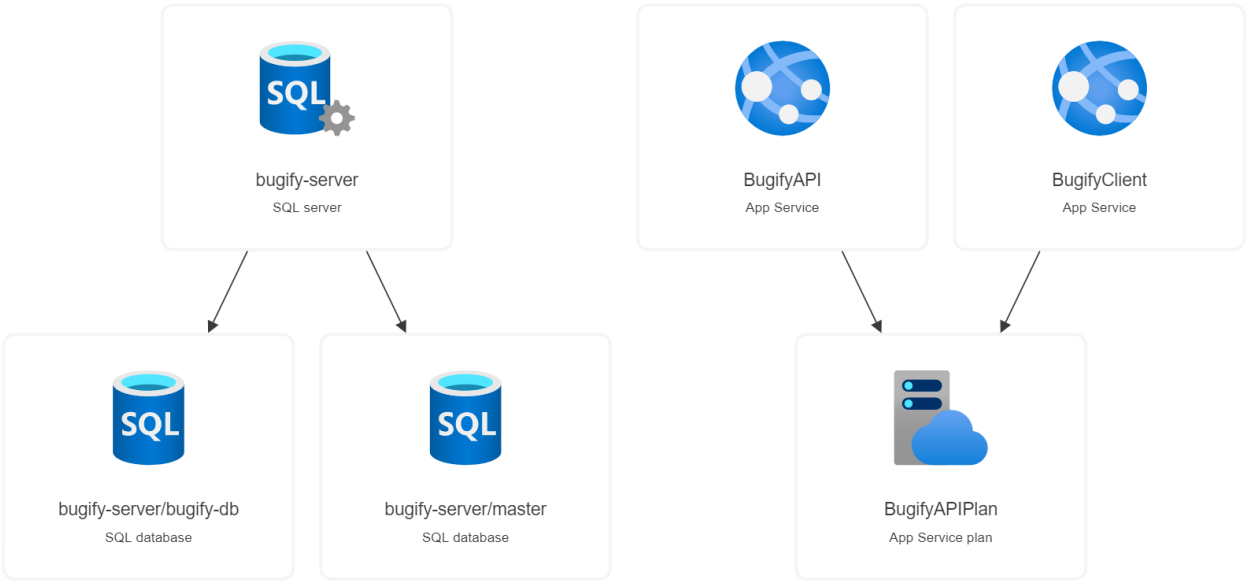


Рисунок 4.2 – Структура взаємодії створених ресурсів в Resource Group

Для системи управління задачами і відстеження помилок використовується база даних SQL Server в хмарному середовищі Azure. Після створення ресурсу SQL Server необхідно налаштувати його параметри, такі як версія, аутентифікація та інші. На рисунку 4.3 наведений створений сервер bugify-server.

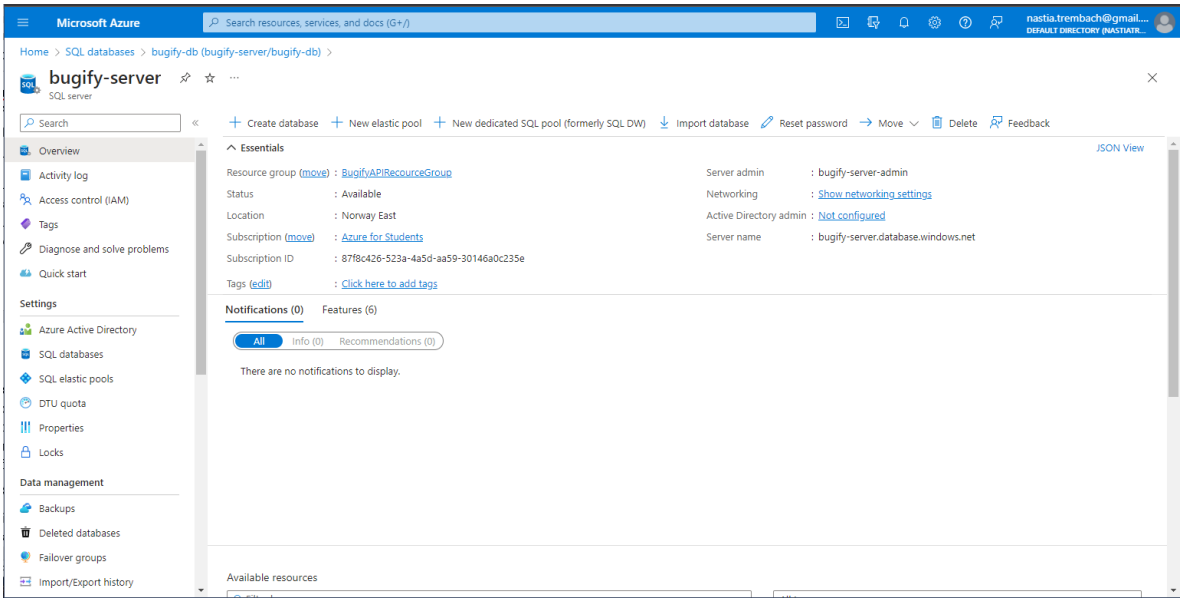


Рисунок 4.3 – Створений SQL Server в Azure

Після цього створюється SQL база даних на розгорнутому SQL Server і налаштовується доступ до неї. Налаштування параметрів бази даних включає отримання рядка підключення, який наведений на рисунку 4.4.

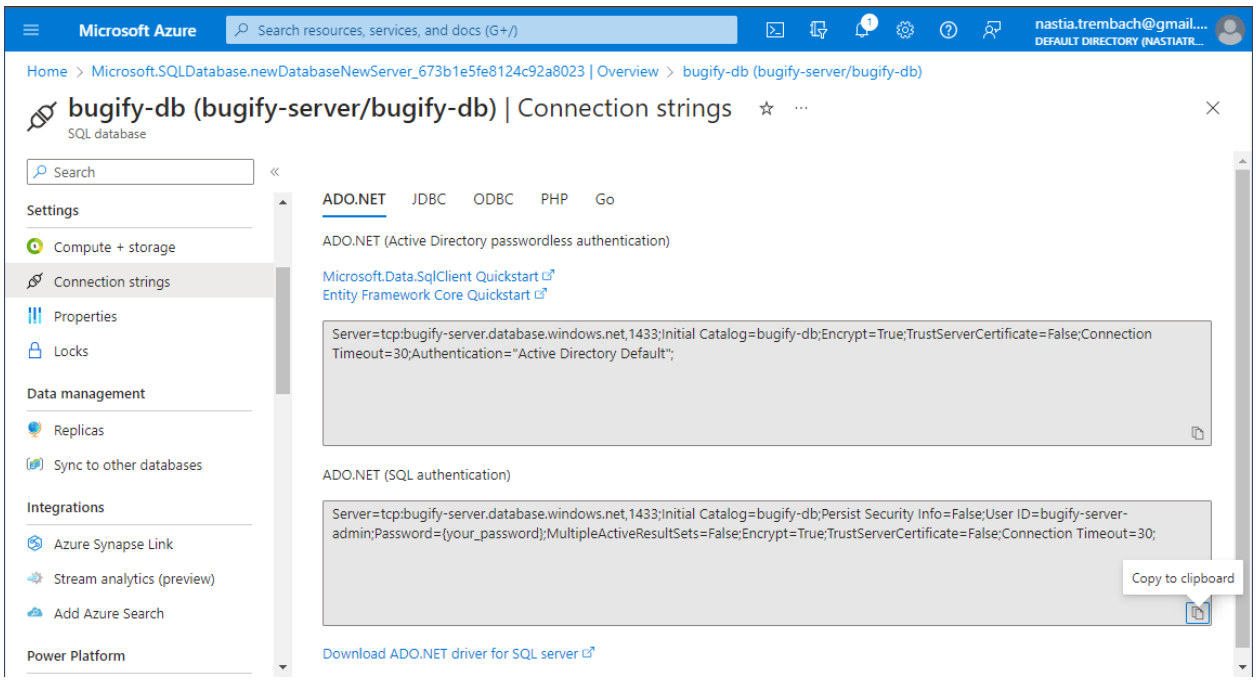


Рисунок 4.4 – Отримання рядка підключення до бази даних з Azure

Після створення бази даних необхідно налаштувати взаємодію бекенд частини з базою даних. Наступний етап це створення Azure App Services для розгортання бекенд частини системи. Створений App Services під назвою BugifyAPI наведений на рисунку 4.5.

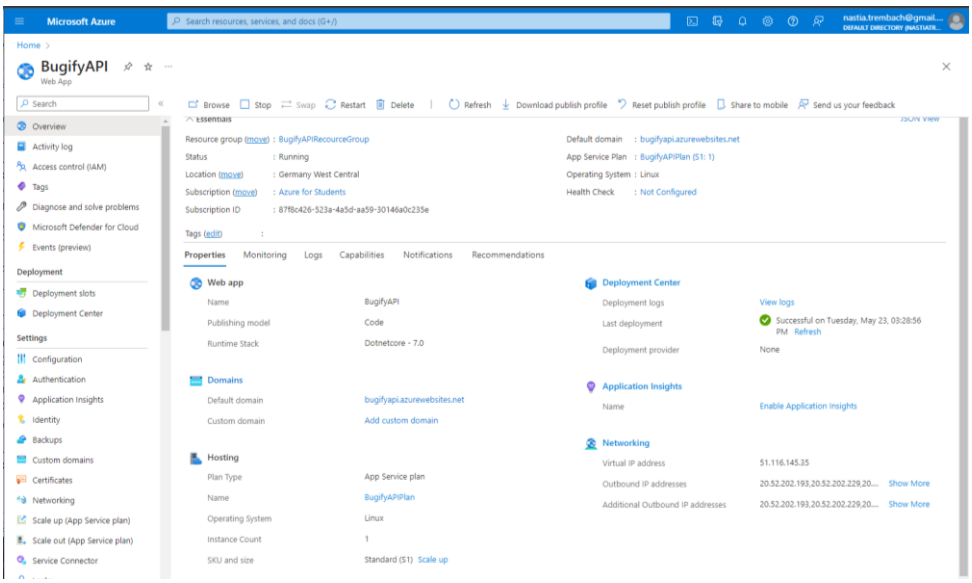


Рисунок 4.5 – Створений Azure App Services для бекенду системи.

Для наступного етапу, а саме розгортання бекенду на Azure App Services, рекомендується використовувати процес Continuous Integration/Continuous Deployment (CI/CD) засобами, такими як GitHub Actions. Спочатку необхідно завантажити вихідний код бекенду до репозиторію GitHub. Потім налаштуйте інтеграцію з GitHub Actions, щоб автоматично розгортати бекенд на App Services при кожному оновленні коду. Налаштування залежностей, конфігураційних параметрів та скриптів для розгортання допоможуть забезпечити успішну і надійну постачання бекенду в хмарному середовищі.

Для розгортання фронтенду на Azure App Services рекомендується використовувати Docker-контейнери. Спочатку необхідно зібрати фронтенд-код та створити Docker-контейнер, який міститиме фронтенд додаток. Потім контейнер можна опублікувати на Docker Hub або в репозиторій контейнерів Azure. Розгорнутий фронтенд наведений на рисунку 4.6 під назвою BugifyClient.

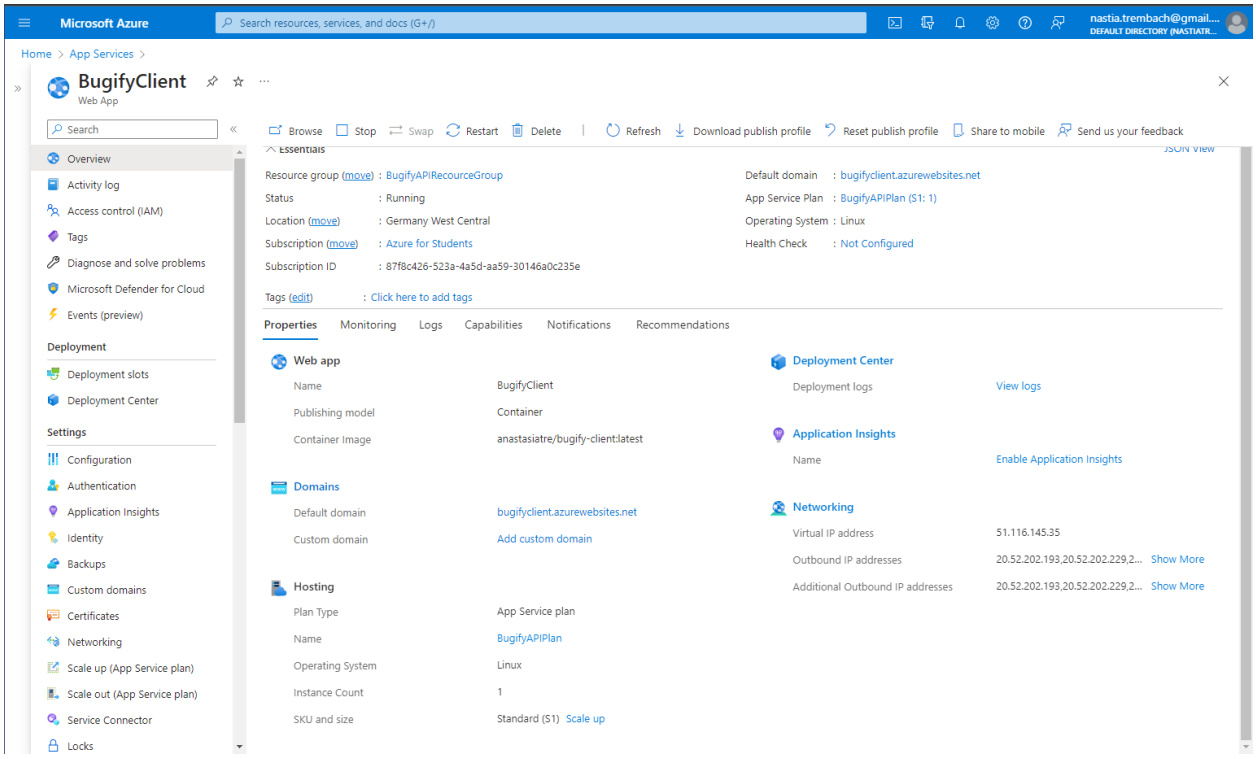


Рисунок 4.6 - Створений Azure App Services для фронтенду системи

Для успішної взаємодії між бекендом і фронтендом необхідно налаштувати API-інтерфейс бекенду. Це включає конфігурацію доступу до

функцій управління задачами та відстеження помилок, які будуть доступні з фронтенду. Крім того, важливо забезпечити коректну маршрутизацію та обробку запитів з фронтенду до бекенду. Це можна досягти шляхом налаштування маршрутів API та встановлення правил маршрутизації для перенаправлення запитів до відповідних ресурсів бекенду.

Отже, розгортання системи для управління задачами і відстеження помилок у хмарному середовищі Azure, є процесом, що включає створення ресурсів Azure, налаштування бази даних, конфігурацію підключення до бази даних , розгортання бекенду та фронтенду, налаштовування їх взаємодії. В цілому, розгортання системи у хмарному середовищі дозволяє забезпечити гнучкість, масштабованість та доступність системи для користувачів.

4.2 Тестування системи

4.2.1 Юніт-тестування системи

Юніт-тестування є процесом, в рамках якого розробляються і виконуються тести для перевірки правильності роботи окремих компонентів системи на найнижчому рівні, таких як функції, класи або модулі. Це означає, що кожна окрема частина системи перевіряється незалежно від інших компонентів, з фокусом на їх коректну функціональність та взаємодію з іншими компонентами.

Для юніт-тестування системи для управління задачами і відстеження помилок я використовувала наступні бібліотеки:

- Xunit: Xunit є популярним фреймворком для написання юніт-тестів у .NET. Він надає набір атрибутів і методів, які дозволяють визначити тести, їх вхідні дані та очікувані результати.

- Moq: Moq є бібліотекою для створення імітацій об'єктів (mock objects) у .NET. Вона дозволяє замінити залежності, такі як зовнішні сервіси або бази даних, на імітовані об'єкти для забезпечення контролю та тестування поведінки системи.
- FluentAssertions: FluentAssertions - це розширення для написання зрозумілих та експресивних тверджень (assertions) у юніт-тестах. Воно надає декларативний синтаксис, який дозволяє виразно визначати очікувані результати та порівнювати їх з отриманими значеннями.
- AutoBogus: AutoBogus - це бібліотека для автоматичного генерації тестових даних.

На рисунку 4.7 наведено приклад юніт-тесту для TaskController.

```
public class TaskControllerTests
{
    private readonly Mock<IMediator> _mediator;
    private readonly TaskController _target;

    public TaskControllerTests()
    {
        _mediator = new Mock<IMediator>();
        _target = new TaskController(_mediator.Object);
    }

    [Fact]
    public async ThreadTasks.Task GetTaskById_ReturnsOk()
    {
        // arrange
        var expected = AutoFaker.Generate<Domain.AggregatesModel.TaskAggregate.Task<int>>();
        _mediator.Setup(expression: x :IMediator => x.Send(
                request: new GetTaskByIdQuery { TaskId = expected.Id },
                cancellationToken: It.IsAny<CancellationToken>())) // ISetup<IMediator,Task<…>>
            .ReturnsAsync(expected);

        // act
        var actual :IActionResult = await _target.GetById(expected.Id);
        var okResult = actual as ObjectResult;

        // assert
        okResult.Should().NotBeNull();
        okResult.Should().BeOfType<OkObjectResult>();
        okResult!.StatusCode.Should().Be(StatusCodes.Status200OK);
    }
```

Рисунок 4.7 – Приклад юніт-тесту для TaskController

Метою юніт-тестування є виявлення помилок, які можуть виникнути на ранніх етапах розробки. Це дозволяє розробникам виправляти проблеми в окремих компонентах до їх інтеграції з іншими частинами системи. Виконання юніт-тестів допомагає забезпечити якість коду, зменшити можливість появи помилок і полегшити процес налагодження. На рисунку 4.8 наведено загальні результати юніт-тестування системи.

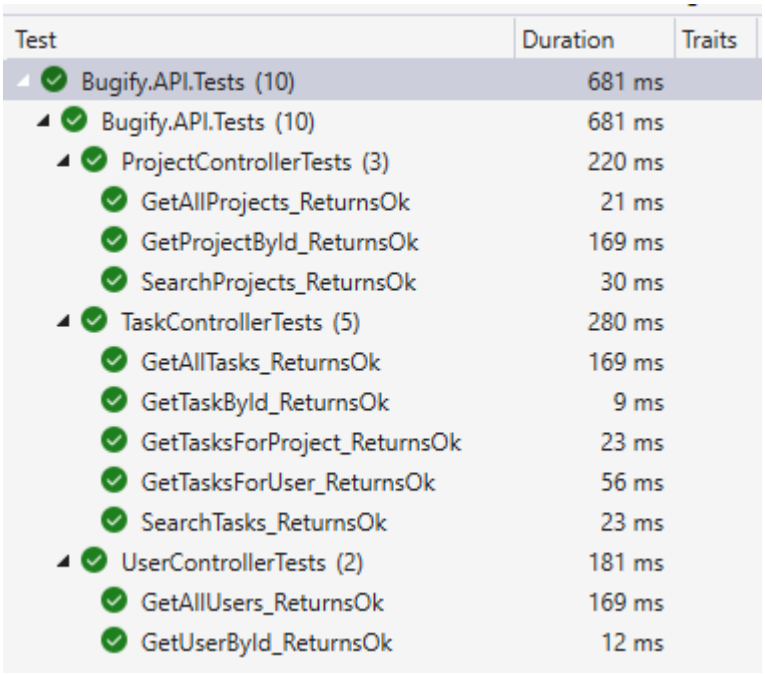


Рисунок 4.8 – Результати юніт-тестування системи

Отже, юніт-тестування допомагає забезпечити стабільність, ефективність та масштабованість системи, знижуючи ризик появи помилок та полегшуючи розширення та підтримку в майбутньому. За результати загального тестування системи всі тести успішно пройдені.

4.2.2 Тестування функціональності системи

Тестування функціональності системи є важливим етапом розробки та впровадження будь-якої програмної системи. Цей процес дозволяє перевірити, чи працює система згідно з очікуваннями та вимогами, які були встановлені перед її розробкою. Тестування функціональності забезпечує визначення

правильності реалізації функцій та можливостей системи, а також виявлення потенційних помилок, дефектів та несумісностей.

Під час тестування функціональності системи для управління задачами було перевірено загальні можливості системи. Наприклад, чи мали користувачі можливість вибирати тему інтерфейсу (світла або темна) відповідно до їхніх вподобань. На рисунку 4.9 наведено темну теми.

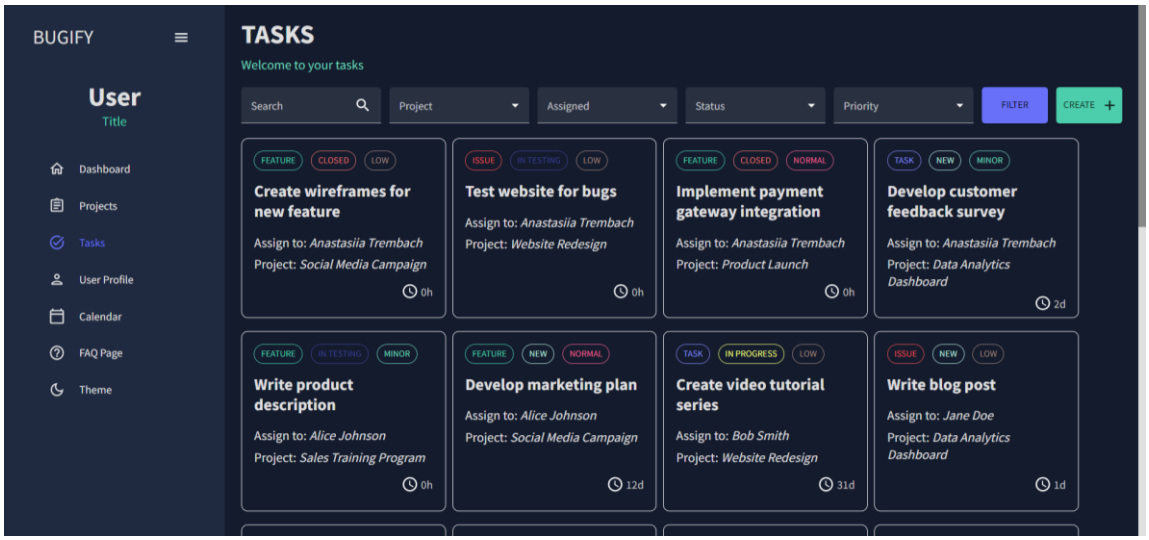


Рисунок 4.9 – Темна тема інтерфейсу системи

Також було перевірено, що користувачі змогли змінювати свій профіль, редагувати основну інформацію та налаштування облікового запису без жодних проблем. На рисунку 4.10 наведено веб-сторінку для редагування інформацію про користувача.

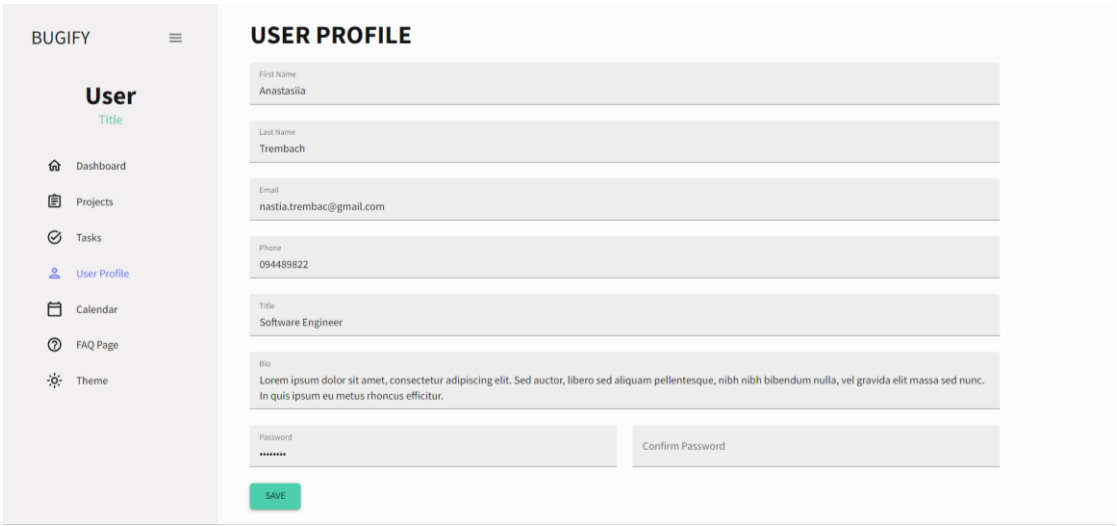


Рисунок 4.10 – Веб-сторінка профілю користувача

Крім того, під час тестування функціональності системи було перевірено можливості інформаційної панелі, яка наведена на рисунку 4.11. А саме, що зображення поточної задачі, яка перебуває у фокусі користувача, правильно відображалося. Також було перевірено, що панель містила список задач "Зараз чи потім?", який відображав задачі відповідно до їх статусу. Додатково, було перевірено, чи правильно відображалися діаграми та інфобокси, які надавали інформацію про поточний прогрес роботи користувача.

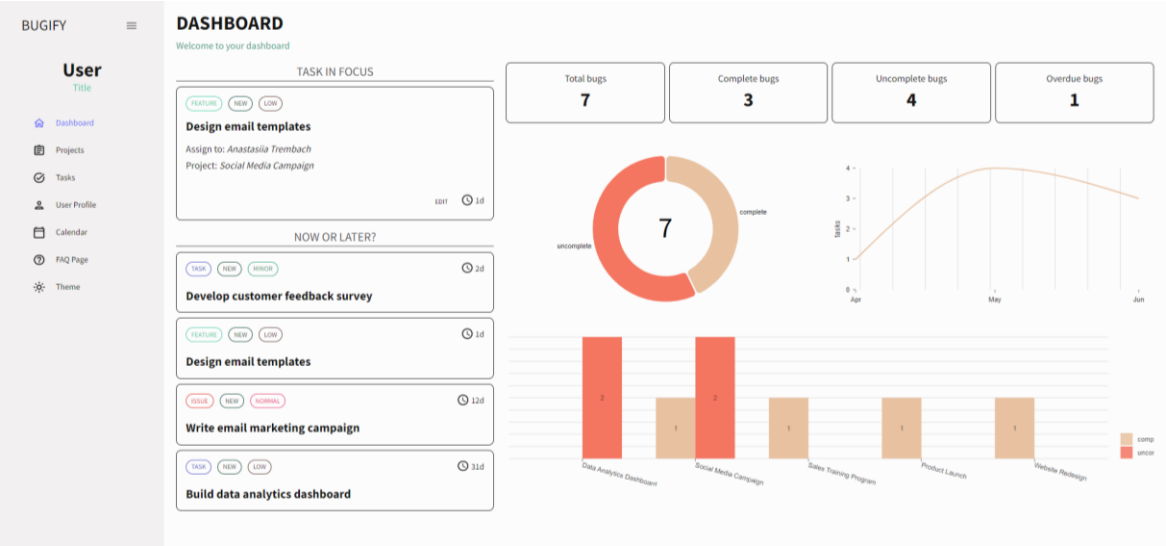


Рисунок 4.11 – Інформаційна панель системи

Більше того, було перевірено всі операції над проектами системи, на рисунку 4.12 наведено веб-сторінку для цього.

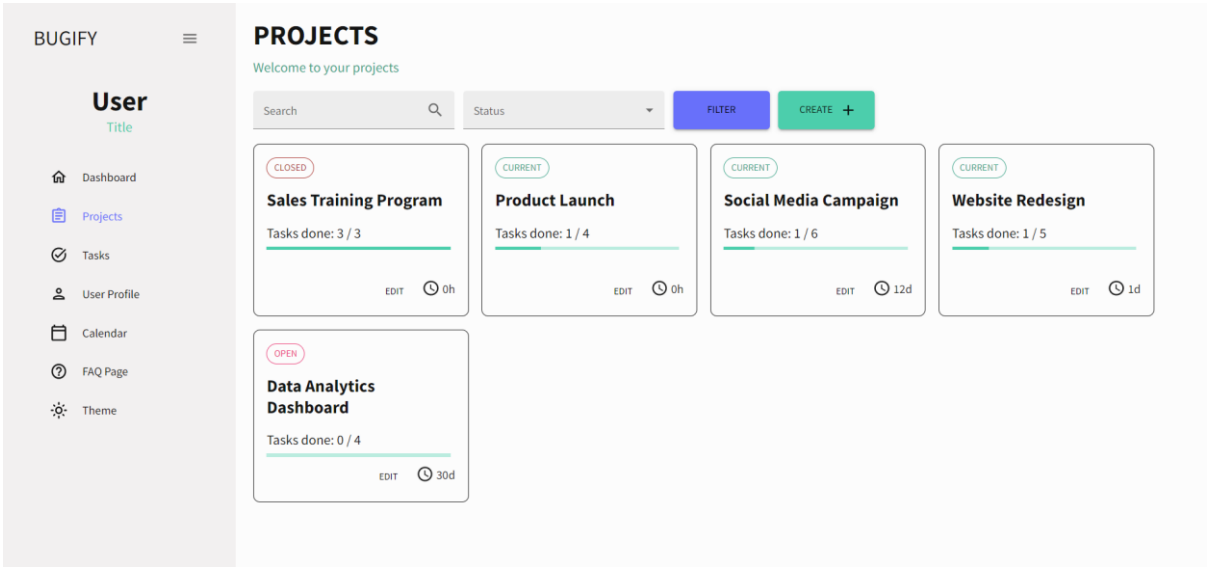


Рисунок 4.12 – Веб-сторінка для роботи з проектами системи

Було перевірено, що користувачі змогли успішно виконувати операції фільтрації, створення, редагування та видалення проектів. Було перевірено, що фільтри працювали правильно, користувачі могли легко створювати та редагувати проекти, а також видаляти їх з системи без проблем.

Нарешті, було здійснено тестування всіх операцій з задачами, на веб-сторінці, що наведено на рисунку 4.13. Під час тестування функціональності системи було перевірено, що користувачі змогли успішно виконувати операції фільтрації, створення, редагування та видалення задач. Було перевірено, що функції фільтрації працювали правильно, користувачі могли легко створювати нові задачі або редагувати існуючі. Також було перевірено, що задачі видалялися коректно і повністю з системи без непередбачених наслідків.

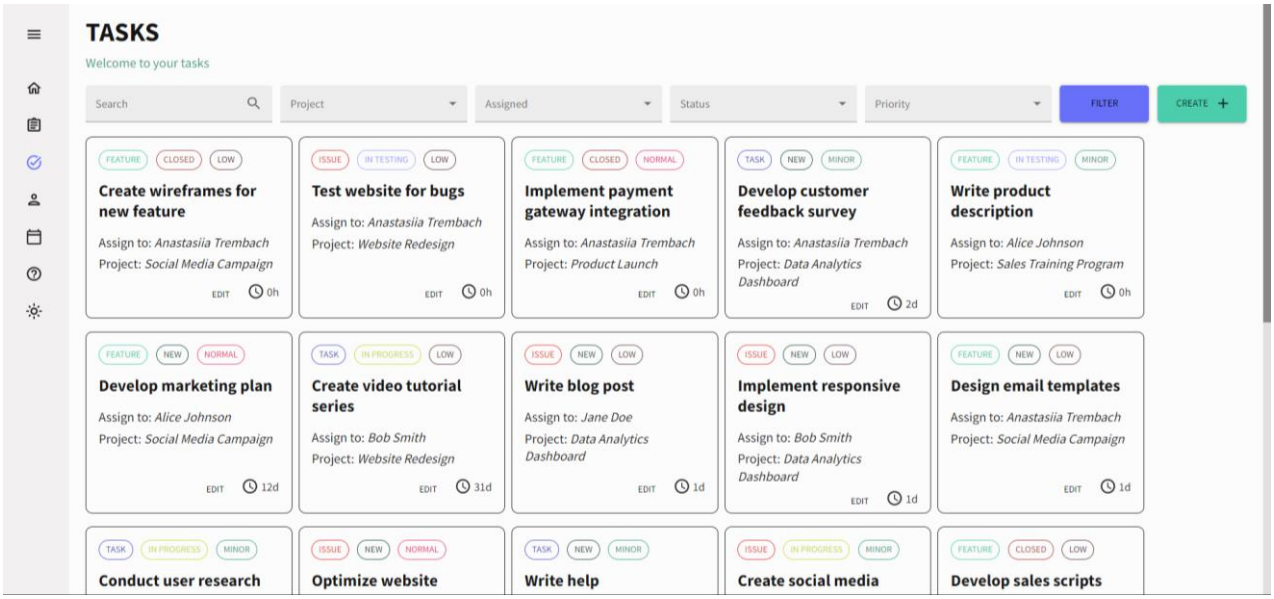


Рисунок 4.13 – Веб-сторінка для роботи з задачами системи

Отже, під час тестування цих можливостей було перевірено, що система працює без помилок і відповідає функціональним вимогам. Також було перевірено коректність взаємодії між різними компонентами системи та їхню відповідність очікуваній поведінці.

4.3 Рекомендацій щодо подальшого розвитку і вдосконалення системи

У цьому пункті я пропоную рекомендації щодо подальшого розвитку і вдосконалення системи для управління задачами і відстеження помилок у хмарному середовищі. З урахуванням зростаючих потреб користувачів та швидкого розвитку технологій, постійне покращення системи є важливим завданням, щоб забезпечити зручну та ефективну роботу з завданнями, а також захистити дані від несанкціонованого доступу.

Однією з найважливіших рекомендацій є забезпечення безпеки даних у системі. Необхідно ретельно попрацювати над впровадженням механізмів аутентифікації та авторизації, щоб гарантувати, що тільки правомірні користувачі матимуть доступ до системи. Резервне копіювання даних є необхідним для запобігання втраті інформації та відновлення даних в разі виникнення проблем. Крім того, важливо розробити ефективні механізми захисту від несанкціонованого доступу, включаючи шифрування даних та контроль доступу до них.

Ще однією рекомендацією є розширення функціональності системи. Більше різних видів візуалізації списку задач може сприяти зручній і швидкій навігації між завданнями та забезпечити краще розуміння стану роботи. Наприклад, використання графіків, діаграм або кольорових міток може допомогти користувачам швидко зорієнтуватися в нагальних завданнях або пріоритетах.

Важливим аспектом подальшого розвитку системи є впровадження інструментів для комунікації команди. Система може надавати можливості сповіщень та повідомлень, що дозволить користувачам ефективно спілкуватися та спільно працювати над завданнями. Наприклад, введення системи сповіщень

про нагадування, оновлення або коментарі до задач може покращити спілкування та сприяти більшій згуртованості команди.

Не менш важливою рекомендацією є покращення інтерфейсу користувача системи. Чистий, інтуїтивно зрозумілий та привабливий інтерфейс може значно полегшити роботу з системою та забезпечити приємний користувацький досвід. Розробники повинні звернути увагу на ергономіку, розміщення елементів, використання кольорів та шрифтів, щоб зробити взаємодію з системою максимально комфортною та зрозумілою.

Ці рекомендації щодо подальшого розвитку та вдосконалення системи допоможуть створити потужний та привабливий інструмент для управління задачами і відстеження помилок у хмарному середовищі. Впровадження заходів безпеки, розширення функціональності, інструментів для комунікації команди та покращення інтерфейсу допоможуть забезпечити зручну роботу користувачів та досягти високої продуктивності.

ВИСНОВОК ДО РОЗДІЛУ 4

У цьому розділі було детально розглянуто кілька ключових аспектів, пов'язаних з розгортанням, аналізом, тестуванням системи. Такою було надано рекомендації щодо подальшого розвитку і вдосконалення системи для управління задачами у хмарному середовищі.

Розгортання системи у хмарному середовищі було важливим етапом, де було визначено відповідну інфраструктуру та забезпечено готовність системи до роботи. Аналіз розробленої системи надав всеохоплюючий огляд її функцій, можливостей та потенційних вдосконалень.

Юніт-тестування дозволило перевірити окремі компоненти системи на належну роботу та виявити можливі помилки або недоліки. Це сприяло покращенню якості та надійності системи в цілому.

Тестування функціональності системи було направлене на перевірку загальних можливостей, операцій над проектами та задачами. Протягом тестування функціональності було перевірено загальні можливості системи, зосередившись на виборі теми інтерфейсу та редагуванні профілю користувача. Також було досліджено можливості інформаційної панелі, зокрема зображення поточної задачі, списку задач "Зараз чи потім?" та візуалізацію прогресу користувача. Операції над проектами та задачами також були перевірені, з увагою на їх фільтрацію, створення, редагування та видалення. Загалом, було перевірено, що система відповідає вимогам та забезпечує зручний та ефективний інтерфейс для користувача.

Завершуючи розділ, було надано рекомендації щодо подальшого розвитку і вдосконалення системи. Забезпечення безпеки даних, розширення функціональності, використання інструментів для комунікації команди та покращення інтерфейсу користувача є ключовими аспектами, які сприятимуть подальшому розвитку і успішному функціонуванню системи.

В цілому, розгортання і тестування системи є важливим етапом в розвитку будь-якого програмного забезпечення. Вдале розгортання та ефективне тестування допомагають забезпечити якість, стабільність та надійність системи. Застосування правильних стратегій, технік та інструментів є вирішальним для досягнення успіху. Професійний підхід до розгортання і тестування допомагає впевнитися, що система відповідає вимогам та очікуванням користувачів, а також готова до подальшого розвитку та вдосконалення.

ВИСНОВКИ

В даній дипломній роботі була проведена ретельна аналітична робота з вивчення систем управління задачами та відстеження помилок у проектному менеджменті. Метою роботи було розробити ефективну та масштабовану систему, що забезпечує організаціям і окремим користувачам ефективне керування задачами, відстеження та вирішення помилок під час проектної діяльності.

У процесі дослідження було виявлено, що системи управління задачами є необхідним помічником у створенні чіткого плану, розподілі завдань, контролі прогресу та збереженні орієнтації в проекті. Вони також забезпечують реєстрацію та аналіз помилок, що дозволяє уникати повторних помилок та покращувати роботу з управління ризиками.

Розроблена система також забезпечує збір і аналіз даних про прогрес проекту, продуктивність команди та результативність роботи. Це надає необхідну інформацію для прийняття рішень та постійного покращення.

Крім того, розгортання системи у хмарному середовищі дозволяє скористатися безліччю переваг, таких як масштабованість, доступність, надійність та зручність у керуванні. Це робить розроблену систему актуальною та конкурентоздатною.

У першому розділі було проведено огляд існуючих систем для управління задачами і відстеження помилок проекту. Зроблено висновок про необхідність розробки нової системи, яка враховуватиме особливості проектного менеджменту та забезпечуватиме ефективне керування завданнями та виявлення помилок.

У другому розділі були розглянуті різні технології для розробки системи управління задачами і відстеження помилок. Зроблено вибір технологій, обґрунтовано їх переваги та відповідність поставленим цілям проекту.

У третьому розділі було проведено архітектурний аналіз системи управління задачами і відстеження помилок. Розроблено архітектурну структуру системи, визначено основні компоненти і їх взаємодію. Це забезпечить ефективне функціонування системи та задоволення вимог користувачів.

У четвертому розділі було проведено розгортання розробленої системи та її тестування. Під час розгортання системи були враховані особливості хмарних середовищ для забезпечення масштабованості, доступності та надійності. Результати тестування підтвердили правильну роботу системи та її відповідність вимогам та очікуванням.

Очікується, що розроблена система сприятиме підвищенню продуктивності та якості проектної діяльності, спрощенню процесу спільної роботи та забезпечить ефективний контроль над проектами. Її інтуїтивний інтерфейс, потужні механізми управління завданнями, відстеження помилок та аналізу проектної діяльності роблять її незамінним інструментом для будь-якої організації, що працює над проектами.

Результати дипломної роботи свідчать про великий потенціал використання систем управління задачами та відстеження помилок у сучасному проектному менеджменті. Розроблена система є важливим кроком у напрямку покращення ефективності та якості проектної діяльності, а також сприяє покращенню комунікації та співпраці в команді.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What Is Project Management Software? [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://www.accelo.com/resources/blog/what-is-project-management-software/.
2. Asana [Електронний ресурс] – Режим доступу до ресурсу: https://asana.com/product.
3. 5 Reasons Why We Love Asana [Електронний ресурс] – Режим доступу до ресурсу: https://designdough.co.uk/five-reasons-why-we-love-asana/.
4. Welcome to Jira Software [Електронний ресурс] – Режим доступу до ресурсу: https://www.atlassian.com/software/jira/guides/getting-started/introduction.
5. Поган О. Що таке Jira і як з нею працювати [Електронний ресурс] / Олексій Поган. – 2023. – Режим доступу до ресурсу: https://iampm.club/ua/blog/shho-take-jira-i-yak-z-neyu-praczyuvati/.
6. Asana vs. Jira: Which is best? [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://zapier.com/blog/asana-vs-jira/.
7. Asana Vs. Jira Comparison [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://www.forbes.com/advisor/business/software/asana-vs-jira-comparison/.
8. Back-End Web Architecture [Електронний ресурс] – Режим доступу до ресурсу: https://www.codecademy.com/article/back-end-architecture.
9. Ryder P. Buggy C# Code: The 10 Most Common Mistakes in C# Programming [Електронний ресурс] / Patrick Ryder – Режим доступу до ресурсу: https://www.toptal.com/c-sharp/top-10-mistakes-that-c-sharp-programmers-make.
10. Rodenburg J. Code like a Pro in C# / Jort Rodenburg., 2021. – pp. 15-32.

11. .NET Core vs .NET Framework [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://www.interviewbit.com/blog/net-core-vs-net-framework/.
12. Skeet J. C# in Depth, Fourth Edition / Jon Skeet., 2019. – pp. 103-106.
13. ASP.NET Core Advantages and Disadvantages [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://redwerk.com/blog/asp-net-core-pros-and-cons/.
14. Lock A. ASP.NET Core in Action, Second Edition / Andrew Lock., 2021. – pp. 436-502.
15. What's the Difference? Relational vs Non-Relational Databases [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://insightsoftware.com/blog/whats-the-difference-relational-vs-non-relational-databases/.
16. SQL Server 2022 [Електронний ресурс] – Режим доступу до ресурсу: https://www.microsoft.com/en-us/sql-server/sql-server-2022.
17. Abba I. What is an ORM – The Meaning of Object Relational Mapping Database Tools [Електронний ресурс] / Ihechikara Abba. – 2022. – Режим доступу до ресурсу: https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/.
18. Entity Framework Core [Електронний ресурс] – Режим доступу до ресурсу: https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx.
19. Smith J. Entity Framework Core in Action / Jon Smith., 2018. – pp. 27-58.
20. AutoMapper in C# [Електронний ресурс] – Режим доступу до ресурсу: https://dotnettutorials.net/lesson/automapper-in-c-sharp/.
21. Raval N. TypeScript vs JavaScript: The Difference You Should Know [Електронний ресурс] / Nihar Raval. – 2023. – Режим доступу до ресурсу: https://radixweb.com/blog/typescript-vs-javascript.

22. Angular Vs React: Difference Between Angular and React [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://www.interviewbit.com/blog/angular-vs-react/.
23. MATERIAL-UI [Електронний ресурс] – Режим доступу до ресурсу: https://v4.mui.com/.
24. Maldonado L. Building charts in React with Nivo [Електронний ресурс] / Leonardo Maldonado. – 2021. – Режим доступу до ресурсу: https://blog.logrocket.com/building-charts-in-react-with-nivo/.
25. What Is Cloud Computing? Definition, Benefits, Types, and Trends [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://www.spiceworks.com/tech/cloud/articles/what-is-cloud-computing/.
26. Get to know Azure [Електронний ресурс] – Режим доступу до ресурсу: https://azure.microsoft.com/en-us/explore/.

ДОДАТОК 1

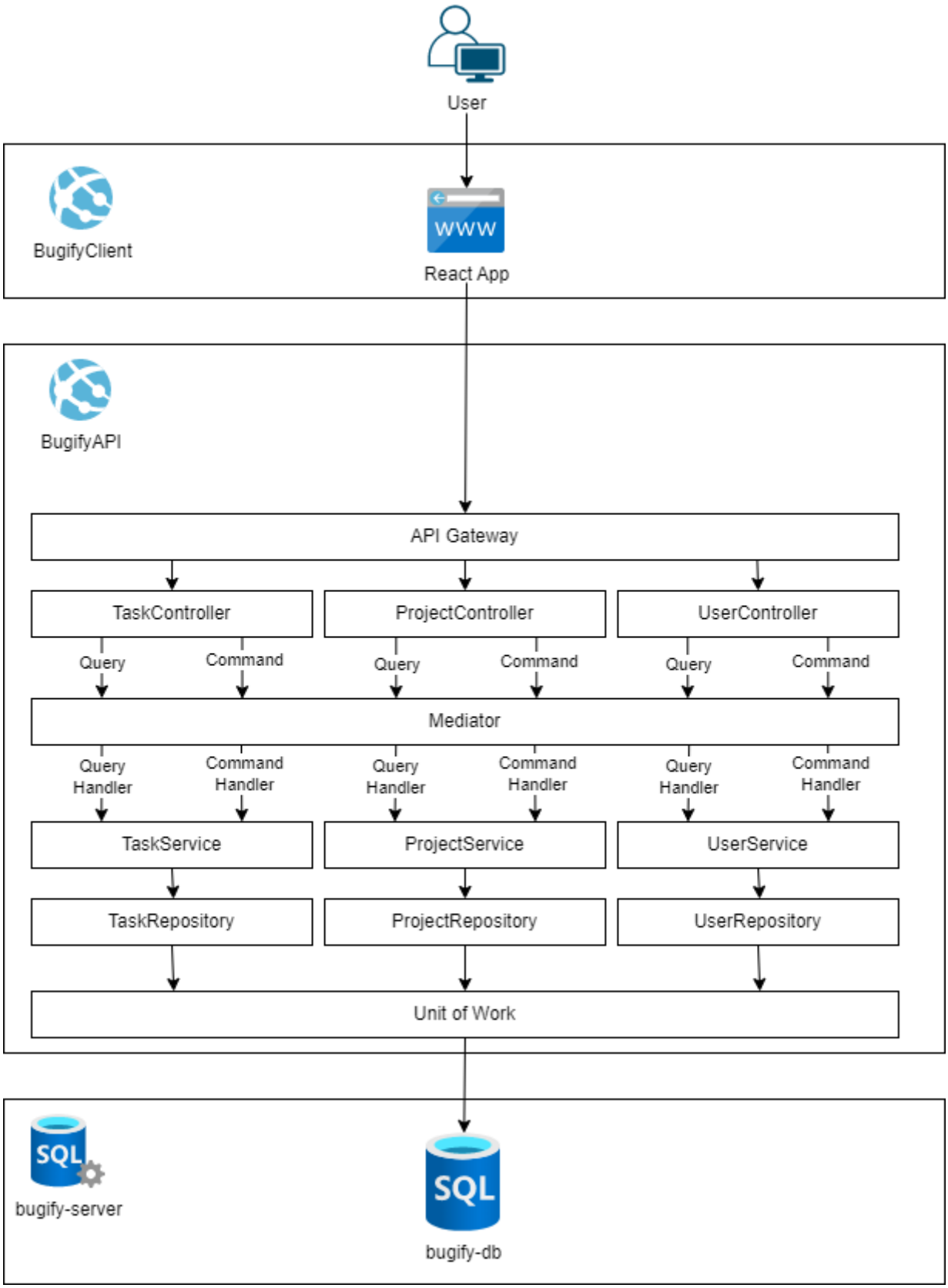
Система для управління задачами і відстеження помилок проекту у хмарному середовищі

Структура взаємодії компонентів (структурна схема)

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Трембач А. Д.			**Система для управління задачами і відстеження помилок проекту у хмарному середовищі** Структура взаємодії компонентів (структурна схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Таран В.І.				КПІ ім. Ігоря Сікорського ФІОТ ІП-93
Н. Контр.		Волокита А. М.				
Затвердив						

ДОДАТОК 2

Система для управління задачами і відстеження помилок проекту у хмарному середовищі

Діаграма класів (функціональна схема)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2023 р

					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Трембач А. Д.			**Система для управління задачами і відстеження помилок проекту у хмарному середовищі** Діаграма класів (функціональна схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Таран В.І.				КПІ ім. Ігоря Сікорського ФІОТ ІП-93
Н. Контр.		Волокита А. М.				
Затвердив						

ДОДАТОК З

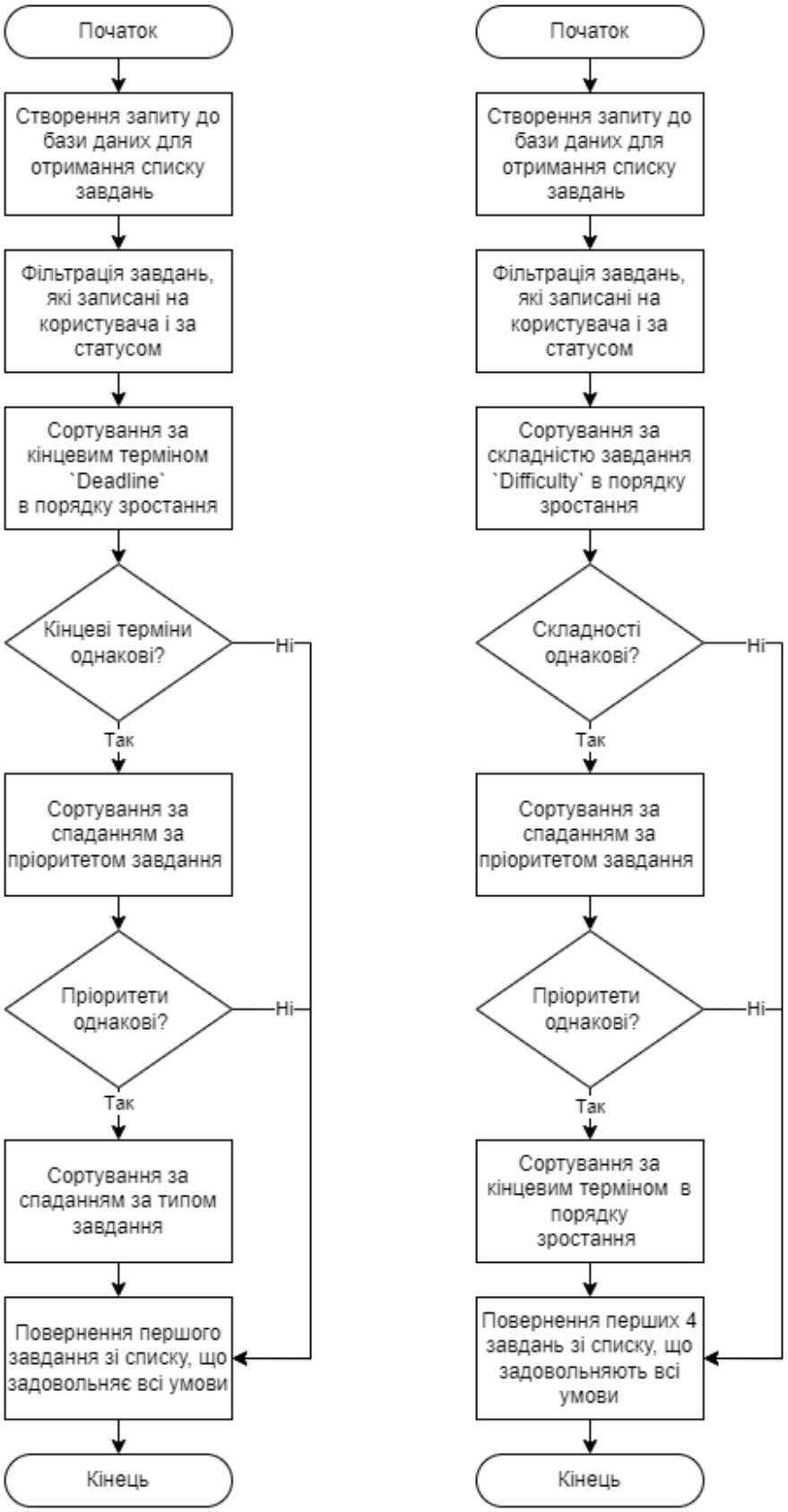
Система для управління задачами і відстеження помилок проекту у хмарному середовищі

Алгоритми системи (принципова схема)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Трембач А. Д.			**Система для управління задачами і відстеження помилок проекту у хмарному середовищі** Алгоритми системи (принципова схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Таран В. І.				
Н. Контр.		Волокита А. М.				КПІ ім. Ігоря Сікорського ФІОТ ІП-93
Затвердив						

ДОДАТОК 4

Система для управління задачами і відстеження помилок проекту у хмарному середовищі

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 36

Київ 2023 р

```
using System;
using System.Collections.Generic;
using System.Text;
using Bugify.Domain.AggregatesModel.ProjectAggregate;
using Bugify.Domain.AggregatesModel.UserAggregate;

namespace Bugify.Domain.AggregatesModel.TaskAggregate;

public class Task<TKey>
{
    public TKey Id { get; set; }

    public string Name { get; set; }
    public string Description { get; set; }

    public TaskStatus Status { get; set; }
    public TaskType Type { get; set; }
    public TaskPriority Priority { get; set; }
    public int Difficulty { get; set; }

    public TKey AssignedId { get; set; }
    public User<TKey> Assigned { get; set; }

    public TKey AuthorId { get; set; }

    public TKey ProjectId { get; set; }
    public Project<TKey> Project { get; set; }

    public DateTime Deadline { get; set; }
}

namespace Bugify.Domain.AggregatesModel.TaskAggregate;

public interface ITaskService<TKey>
{
    public System.Threading.Tasks.Task<Task<TKey>> GetTaskById(TKey id);

    public System.Threading.Tasks.Task<IEnumerable<Task<TKey>>>
        SearchTasks(string searchString = "");

    public System.Threading.Tasks.Task<IEnumerable<Task<TKey>>>
        GetTasksForProject(TKey projectId);

    public System.Threading.Tasks.Task<IEnumerable<Task<TKey>>> GetTasksForUser(
        TKey userId);

    public System.Threading.Tasks.Task<Task<TKey>> AssignTaskToUser(TKey taskId,
        TKey userId);

    public System.Threading.Tasks.Task<Task<TKey>>
        CreateTask(Task<TKey> task);

    public System.Threading.Tasks.Task<Task<TKey>>
        UpdateTask(Task<TKey> task);

    public System.Threading.Tasks.Task<Task<TKey>>
        DeleteTask(Task<TKey> task);
```

```
    public System.Threading.Tasks.Task<Task<TKey>> GetTaskInFocusForUser(
        TKey userId);

    public System.Threading.Tasks.Task<IEnumerable<Task<TKey>>>
        GetTasksNowOrLaterForUser(TKey userId);

    public System.Threading.Tasks.Task<int> GetTotalTasksForUser(int userId);

    public System.Threading.Tasks.Task<int> GetCompleteTasksForUser(int userId);

    public System.Threading.Tasks.Task<int> GetUncompleteTasksForUser(int userId);

    public System.Threading.Tasks.Task<int> GetOverdueTasksForUser(int userId);
}

using Bugify.Domain.AggregatesModel.ProjectAggregate;
using Bugify.Domain.AggregatesModel.UserAggregate;
using Bugify.Domain.SeedWork;

namespace Bugify.Domain.AggregatesModel.TaskAggregate;

public class TaskEntity<TKey> : BaseEntity<TKey>
{
    public string Name { get; set; }
    public string Description { get; set; }

    public TaskStatus Status { get; set; }
    public TaskType Type { get; set; }
    public TaskPriority Priority { get; set; }
    public int Difficulty { get; set; }

    public TKey AssignedId { get; set; }
    public UserEntity<TKey> Assigned { get; set; }

    public TKey AuthorId { get; set; }

    public TKey ProjectId { get; set; }
    public ProjectEntity<TKey> Project { get; set; }

    public DateTime Deadline { get; set; }
}

using Bugify.Domain.SeedWork;

namespace Bugify.Domain.AggregatesModel.TaskAggregate;

public interface ITaskRepository<TKey> : IRepository<TaskEntity<TKey>, TKey>
{
    public System.Threading.Tasks.Task<IEnumerable<TaskEntity<TKey>>> Search(string
searchString);

    public System.Threading.Tasks.Task<IEnumerable<TaskEntity<TKey>>> GetTasksForProject(
        TKey projectId);

    public System.Threading.Tasks.Task<IEnumerable<TaskEntity<TKey>>> GetTasksForUser(TKey
userId);

    public System.Threading.Tasks.Task<TaskEntity<TKey>> GetTaskInFocusForUser(TKey userId);

    public System.Threading.Tasks.Task<IEnumerable<TaskEntity<TKey>>>
GetTasksNowOrLaterForUser(
        TKey userId);

    public System.Threading.Tasks.Task<int> GetTotalTasksForUser(int userId);
```

```
    public System.Threading.Tasks.Task<int> GetCompleteTasksForUser(int userId);

    public System.Threading.Tasks.Task<int> GetUncompleteTasksForUser(int userId);

    public System.Threading.Tasks.Task<int> GetOverdueTasksForUser(int userId);
}

namespace Bugify.Domain.AggregatesModel.TaskAggregate;

public enum TaskType
{
    Issue = 1,
    Feature,
    Task,
    Bug
}

namespace Bugify.Domain.AggregatesModel.TaskAggregate;

public enum TaskStatus
{
    New = 1,
    InProgress,
    InTesting,
    Closed
}


namespace Bugify.Domain.AggregatesModel.TaskAggregate;

public enum TaskPriority
{
    Minor = 1,
    Low,
    Normal,
    High
}

using System.ComponentModel.DataAnnotations;

namespace Bugify.Domain.SeedWork;

public abstract class BaseEntity<TKey>
{
    [Key] public TKey Id { get; set; }

    public DateTime Created { get; set; } = DateTime.Now;
}


namespace Bugify.Domain.SeedWork;

public interface IRepository<T, in TKey> where T : BaseEntity<TKey>
{
    Task<T> GetById(TKey id);

    Task Create(T item);
    void Update(T item);
    void Delete(T item);
}

using Bugify.Domain.AggregatesModel.ProjectAggregate;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using Bugify.Domain.AggregatesModel.UserAggregate;
```

```
namespace Bugify.Domain.SeedWork;

public interface IUnitOfWork<TKey>
{
    public ITaskRepository<TKey> TaskRepository { get; set; }
    public IUserRepository<TKey> UserRepository { get; set; }
    public IProjectRepository<TKey> ProjectRepository { get; set; }
    public IUserProjectRepository<TKey> UserProjectRepository { get; set; }

    public Task Save();
}

using System.Collections.Generic;
using AutoMapper;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using Bugify.Domain.SeedWork;

namespace Bugify.Infrastructure.Services;

public class TaskService : ITaskService<int>
{
    private readonly IMapper _mapper;
    private readonly IUnitOfWork<int> _unitOfWork;

    public TaskService(IUnitOfWork<int> unitOfWork, IMapper mapper)
    {
        _unitOfWork = unitOfWork;
        _mapper = mapper;
    }

    public async System.Threading.Tasks.Task<Task<int>> GetTaskById(int id)
    {
        var task = await _unitOfWork.TaskRepository.GetById(id);
        return _mapper.Map<Task<int>>(task);
    }

    public async System.Threading.Tasks.Task<IEnumerable<Task<int>>>
        SearchTasks(string searchString)
    {
        var tasks = await _unitOfWork.TaskRepository.Search(searchString);
        return _mapper.Map<IEnumerable<Task<int>>>(tasks);
    }

    public async System.Threading.Tasks.Task<IEnumerable<Task<int>>>
        GetTasksForProject(
            int projectId)
    {
        var tasks =
            await _unitOfWork.TaskRepository.GetTasksForProject(projectId);
        return _mapper.Map<IEnumerable<Task<int>>>(tasks);
    }

    public async System.Threading.Tasks.Task<IEnumerable<Task<int>>>
        GetTasksForUser(int userId)
    {
        var tasks = await _unitOfWork.TaskRepository.GetTasksForUser(userId);
        return _mapper.Map<IEnumerable<Task<int>>>(tasks);
    }

    public async System.Threading.Tasks.Task<Task<int>> AssignTaskToUser(
        int taskId, int userId)
    {
        var task = await _unitOfWork.TaskRepository.GetById(taskId);
        task.AssignedId = userId;
```

```
    _unitOfWork.TaskRepository.Update(task);
    await _unitOfWork.Save();
    return _mapper.Map<Task<int>>(task);
}

public async System.Threading.Tasks.Task<Task<int>> CreateTask(
    Task<int> task)
{
    var mappedTask = _mapper.Map<TaskEntity<int>>(task);
    await _unitOfWork.TaskRepository.Create(mappedTask);
    await _unitOfWork.Save();
    return task;
}

public async System.Threading.Tasks.Task<Task<int>> UpdateTask(
    Task<int> task)
{
    var mappedTask = _mapper.Map<TaskEntity<int>>(task);
    _unitOfWork.TaskRepository.Update(mappedTask);
    await _unitOfWork.Save();
    return task;
}

public async System.Threading.Tasks.Task<Task<int>> DeleteTask(
    Task<int> task)
{
    var mappedTask = _mapper.Map<TaskEntity<int>>(task);
    _unitOfWork.TaskRepository.Delete(mappedTask);
    await _unitOfWork.Save();
    return task;
}

public async System.Threading.Tasks.Task<Task<int>> GetTaskInFocusForUser(
    int userId)
{
    var tasks =
        await _unitOfWork.TaskRepository.GetTaskInFocusForUser(userId);
    return _mapper.Map<Task<int>>(tasks);
}

public async System.Threading.Tasks.Task<IEnumerable<Task<int>>>
    GetTasksNowOrLaterForUser(int userId)
{
    var tasks =
        await _unitOfWork.TaskRepository.GetTasksNowOrLaterForUser(userId);
    return _mapper.Map<IEnumerable<Task<int>>>(tasks);
}

public System.Threading.Tasks.Task<int> GetTotalTasksForUser(int userId)
{
    return _unitOfWork.TaskRepository.GetTotalTasksForUser(userId);
}

public System.Threading.Tasks.Task<int> GetCompleteTasksForUser(int userId)
{
    return _unitOfWork.TaskRepository.GetCompleteTasksForUser(userId);
}

public System.Threading.Tasks.Task<int> GetUncompleteTasksForUser(int userId)
{
    return _unitOfWork.TaskRepository.GetUncompleteTasksForUser(userId);
}

public System.Threading.Tasks.Task<int> GetOverdueTasksForUser(int userId)
{
```

```
        return _unitOfWork.TaskRepository.GetOverdueTasksForUser(userId);
    }
}

using Bugify.Domain.AggregatesModel.ProjectAggregate;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using Bugify.Domain.AggregatesModel.UserAggregate;
using Bugify.Domain.SeedWork;
using Bugify.Infrastructure.Repositories.EFImplementation;
using Bugify.Infrastructure.Services;
using Bugify.Infrastructure.UnitOfWork;
using Microsoft.Extensions.DependencyInjection;

namespace Bugify.Infrastructure;

public static class DependenciesInfrastructure
{
    public static IServiceCollection SetEFDataDependencies(
        this IServiceCollection services)
    {
        services.AddDbContext<BugTrackerDbContext>();
        services.AddScoped<ITaskRepository<int>, EFTaskRepository>();
        services.AddScoped<IUserRepository<int>, EFUserRepository>();
        services.AddScoped<IProjectRepository<int>, EFProjectRepository>();
        services
            .AddScoped<IUserProjectRepository<int>,
                EFUserProjectRepository>();
        services.AddScoped<IUnitOfWork<int>, EFUnitOfWork>();

        return services;
    }

    public static IServiceCollection SetServices(
        this IServiceCollection services)
    {
        services.AddScoped<ITaskService<int>, TaskService>();
        services.AddScoped<IUserService<int>, UserService>();
        services.AddScoped<IProjectService<int>, ProjectService>();

        return services;
    }
}

using Bugify.Domain.AggregatesModel.ProjectAggregate;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using Bugify.Domain.AggregatesModel.UserAggregate;
using Bugify.Infrastructure.Configuration;
using Microsoft.EntityFrameworkCore;

namespace Bugify.Infrastructure;

public sealed class BugTrackerDbContext : DbContext
{
    private readonly string _connectionString;

    public BugTrackerDbContext(
        DbContextOptions<BugTrackerDbContext> options,
        string connectionString = null) : base(options)
    {
        _connectionString = connectionString ?? DbDefaultConnectionString;
        //Database.Migrate();
    }

    private static string DbDefaultConnectionString =>
        new DatabaseConfiguration()
```

```
            .GetDatabaseConnectionString();

    public DbSet<TaskEntity<int>> Tasks { get; set; }
    public DbSet<UserEntity<int>> Users { get; set; }
    public DbSet<ProjectEntity<int>> Projects { get; set; }
    public DbSet<UserProjectEntity<int>> UserProjects { get; set; }

    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder
            .UseSqlServer(_connectionString);
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {

        modelBuilder.Entity<ProjectEntity<int>>()
            .Navigation(p => p.Author)
            .AutoInclude();

        modelBuilder.Entity<TaskEntity<int>>()
            .Navigation(p => p.Assigned)
            .AutoInclude();
        modelBuilder.Entity<TaskEntity<int>>()
            .Navigation(p => p.Project)
            .AutoInclude();
    }
}

using System.Threading.Tasks;
using Bugify.Domain.AggregatesModel.ProjectAggregate;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using Bugify.Domain.AggregatesModel.UserAggregate;
using Bugify.Domain.SeedWork;

namespace Bugify.Infrastructure.UnitOfWork;

public class EFUnitOfWork : IUnitOfWork<int>
{
    private readonly BugTrackerDbContext _dbContext;

    public EFUnitOfWork(BugTrackerDbContext dbContext,
        ITaskRepository<int> taskRepository,
        IUserRepository<int> userRepository,
        IProjectRepository<int> projectRepository,
        IUserProjectRepository<int> userProjectRepository)
    {
        _dbContext = dbContext;
        TaskRepository = taskRepository;
        UserRepository = userRepository;
        ProjectRepository = projectRepository;
        UserProjectRepository = userProjectRepository;
    }

    public ITaskRepository<int> TaskRepository { get; set; }
    public IUserRepository<int> UserRepository { get; set; }
    public IProjectRepository<int> ProjectRepository { get; set; }
    public IUserProjectRepository<int> UserProjectRepository { get; set; }

    public async Task Save()
    {
        await _dbContext.SaveChangesAsync();
    }
```

```
    public async void Dispose()
    {
        await _dbContext.DisposeAsync();
    }

    ~EFUnitOfWork()
    {
        _dbContext.DisposeAsync();
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Bugify.Domain.AggregatesModel.ProjectAggregate;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using Microsoft.EntityFrameworkCore;
using TaskStatus = Bugify.Domain.AggregatesModel.TaskAggregate.TaskStatus;

namespace Bugify.Infrastructure.Repositories.EFImplementation;

public class EFTaskRepository : EFRepository<TaskEntity<int>>,
    ITaskRepository<int>
{
    public EFTaskRepository(BugTrackerDbContext dbContext) : base(dbContext,
        dbContext.Tasks)
    {
    }

    public async System.Threading.Tasks.Task<IEnumerable<TaskEntity<int>>> Search(
        string searchString)
    {
        var result = _entities.AsQueryable();
        var isSearchStringEmpty = string.IsNullOrEmpty(searchString);

        if (!isSearchStringEmpty)
        {
            searchString = searchString.ToLower();
            result = await Task.Run(() => result.Where(task =>
                task.Name.ToLower().Contains(searchString)
                || task.Description.Contains(searchString)));
        }

        return await result.OrderByDescending(task => task.Created).ToListAsync();
    }

    public async System.Threading.Tasks.Task<IEnumerable<TaskEntity<int>>>
GetTasksForProject(
        int projectId)
    {
        return await _entities.AsQueryable()
            .Where(task => task.ProjectId.Equals(projectId))
            .ToListAsync();
    }

    public async System.Threading.Tasks.Task<IEnumerable<TaskEntity<int>>> GetTasksForUser(
        int userId)
    {
        return await _entities.AsQueryable()
            .Where(task => task.AssignedId.Equals(userId))
            .ToListAsync();
    }

    /*
```

```
     * -- task in focus
       select top 1 * from Tasks
       where AssignedId = 1
       and status <= 2
       and ProjectId in (select Id from Projects
       where Status != 3)
       order by Deadline, Priority desc, Type desc
    */
    public async System.Threading.Tasks.Task<TaskEntity<int>> GetTaskInFocusForUser(
        int userId)
    {
        return await _entities.AsQueryable()
            .Where(task => task.AssignedId.Equals(userId)
                           && (task.Status == TaskStatus.New ||
                               task.Status == TaskStatus.InProgress)
                           && task.Project.Status != ProjectStatus.Closed)
            .OrderBy(task => task.Deadline)
            .ThenByDescending(task => task.Priority)
            .ThenByDescending(task => task.Type)
            .FirstAsync();
    }

    /*
     * -- now or later
       select top 4 * from Tasks
       where AssignedId = 1
       and status <= 2
       and ProjectId in (select Id from Projects
       where Status != 3)
       order by Difficulty, Priority desc, Deadline, Type
     */
    public async System.Threading.Tasks.Task<IEnumerable<TaskEntity<int>>>
GetTasksNowOrLaterForUser(
        int userId)
    {
        return await _entities.AsQueryable()
            .Where(task => task.AssignedId.Equals(userId)
                           && (task.Status == TaskStatus.New ||
                               task.Status == TaskStatus.InProgress)
                           && task.Project.Status != ProjectStatus.Closed)
            .OrderBy(task => task.Difficulty)
            .ThenByDescending(task => task.Priority)
            .ThenBy(task => task.Deadline)
            .ThenBy(task => task.Type)
            .Take(4)
            .ToListAsync();
    }

    public async System.Threading.Tasks.Task<int> GetTotalTasksForUser(int userId)
    {
        return await _entities.AsQueryable()
            .Where(task => task.AssignedId.Equals(userId)
                           && task.Project.Status != ProjectStatus.Closed)
            .CountAsync();
    }

    public async System.Threading.Tasks.Task<int> GetCompleteTasksForUser(int userId)
    {
        return await _entities.AsQueryable()
            .Where(task => task.AssignedId.Equals(userId)
                           && (task.Status == TaskStatus.InTesting ||
                               task.Status == TaskStatus.Closed)
                           && task.Project.Status != ProjectStatus.Closed)
            .CountAsync();
    }
```

```
    public async System.Threading.Tasks.Task<int> GetUncompleteTasksForUser(int userId)
    {
        return await _entities.AsQueryable()
            .Where(task => task.AssignedId.Equals(userId)
                           && (task.Status == TaskStatus.New ||
                               task.Status == TaskStatus.InProgress)
                           && task.Project.Status != ProjectStatus.Closed)
            .CountAsync();
    }

    public async System.Threading.Tasks.Task<int> GetOverdueTasksForUser(int userId)
    {
        return await _entities.AsQueryable()
            .Where(task => task.AssignedId.Equals(userId)
                           && (task.Status == TaskStatus.New ||
                               task.Status == TaskStatus.InProgress ||
                               task.Status == TaskStatus.InTesting)
                           && task.Project.Status != ProjectStatus.Closed)
            .Where(task => task.Deadline < DateTime.Now)
            .CountAsync();
    }
}


using System.Threading.Tasks;
using Bugify.Domain.SeedWork;
using Microsoft.EntityFrameworkCore;

namespace Bugify.Infrastructure.Repositories.EFImplementation;

public class EFRepository<T> : IRepository<T, int> where T : BaseEntity<int>
{
    protected readonly BugTrackerDbContext _dbContext;
    protected readonly DbSet<T> _entities;

    public EFRepository(BugTrackerDbContext dbContext, DbSet<T> dbSet)
    {
        _dbContext = dbContext;
        _entities = dbSet;
    }

    public async Task Create(T entity)
    {
        await _entities.AddAsync(entity);
    }

    public void Update(T entity)
    {
        _entities.Update(entity);
    }

    public void Delete(T entity)
    {
        _entities.Remove(entity);
    }

    public async Task<T> GetById(int id)
    {
        return await _entities.FindAsync(id);
    }
}
```

```
using Bugify.Domain.AggregatesModel.ProjectAggregate;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using Bugify.Domain.AggregatesModel.UserAggregate;
using Microsoft.Extensions.DependencyInjection;

namespace Bugify.Infrastructure.Mapper;

public static class MapperConfig
{
    public static IServiceCollection SetMapperConfig(
        this IServiceCollection services)
    {
        services.AddAutoMapper(cfg =>
            cfg.CreateMap<Task<int>, TaskEntity<int>>().ReverseMap());
        services.AddAutoMapper(cfg =>
            cfg.CreateMap<Project<int>, ProjectEntity<int>>()
                .ReverseMap());
        services.AddAutoMapper(cfg =>
            cfg.CreateMap<User<int>, UserEntity<int>>().ReverseMap());

        return services;
    }
}


namespace Bugify.Infrastructure.Configuration;

internal class DatabaseConfiguration : ConfigurationBase
{
    public string GetDatabaseConnectionString()
    {
        var configuration =
            GetConfiguration().GetSection("DatabaseConnection");
        var server = configuration["DbServer"];
        var user = configuration["DbUser"];
        var password = configuration["DbPassword"];
        var database = configuration["DbName"];

        return
            $"Server={server};Initial Catalog={database};User
ID={user};Password={password};" +
            "MultipleActiveResultSets=true;Connect Timeout=30;Encrypt=False;" +

"TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False";
    }
}

using System;
using Microsoft.Extensions.Configuration;

namespace Bugify.Infrastructure.Configuration;

public abstract class ConfigurationBase
{
    protected IConfigurationRoot GetConfiguration()
    {
        return new ConfigurationBuilder()
            .SetBasePath(AppDomain.CurrentDomain.BaseDirectory)
            .AddJsonFile("appsettings.json", true, true)
            .AddJsonFile(

$"appsettings.{Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT")}.json",
                true, true)
            .Build();
    }
```

```
    protected void RaiseValueNotFoundException(string configurationKey)
    {
        throw new Exception(
            $"appsettings key ({configurationKey}) could not be found.");
    }
}

using System.Collections.Generic;
using System.Threading;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using MediatR;

namespace Bugify.WebAPI.Features.TaskFeatures.Queries;

public class GetTasksByProjectQuery : IRequest<IEnumerable<Task<int>>>
{
    public int ProjectId { get; set; }

    public class GetTasksByProjectQueryHandler : IRequestHandler<
        GetTasksByProjectQuery, IEnumerable<Task<int>>>
    {
        private readonly ITaskService<int> _service;

        public GetTasksByProjectQueryHandler(ITaskService<int> service)
        {
            _service = service;
        }

        public async System.Threading.Tasks.Task<IEnumerable<Task<int>>> Handle(
            GetTasksByProjectQuery request,
            CancellationToken cancellationToken)
        {
            var result =
                await _service.GetTasksForProject(request.ProjectId);
            return result;
        }
    }
}


using System.Collections.Generic;
using System.Threading;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using MediatR;

namespace Bugify.WebAPI.Features.TaskFeatures.Queries;

public class GetTasksNowOrLaterForUser : IRequest<IEnumerable<Task<int>>>
{
    public int UserId { get; set; }

    public class GetTasksNowOrLaterForUserHandler : IRequestHandler<
        GetTasksNowOrLaterForUser, IEnumerable<Task<int>>>
    {
        private readonly ITaskService<int> _service;

        public GetTasksNowOrLaterForUserHandler(ITaskService<int> service)
        {
            _service = service;
        }

        public async System.Threading.Tasks.Task<IEnumerable<Task<int>>> Handle(
            GetTasksNowOrLaterForUser request,
            CancellationToken cancellationToken)
```

```
        {
            var result =
                await _service.GetTasksNowOrLaterForUser(request.UserId);
            return result;
        }
    }
}

using System.Threading;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using MediatR;

namespace Bugify.WebAPI.Features.TaskFeatures.Queries;

public class GetTaskInFocusForUser : IRequest<Task<int>>
{
    public int UserId { get; set; }

    public class GetTaskInFocusForUserHandler : IRequestHandler<
        GetTaskInFocusForUser, Task<int>>
    {
        private readonly ITaskService<int> _service;

        public GetTaskInFocusForUserHandler(ITaskService<int> service)
        {
            _service = service;
        }

        public async System.Threading.Tasks.Task<Task<int>> Handle(
            GetTaskInFocusForUser request, CancellationToken cancellationToken)
        {
            var result =
                await _service.GetTaskInFocusForUser(request.UserId);
            return result;
        }
    }
}


using System.Threading;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using MediatR;

namespace Bugify.WebAPI.Features.TaskFeatures.Queries;

public class GetTaskByIdQuery : IRequest<Task<int>>
{
    public int TaskId { get; set; }

    public class
        GetTaskByIdQueryHandler : IRequestHandler<GetTaskByIdQuery, Task<int>>
    {
        private readonly ITaskService<int> _service;

        public GetTaskByIdQueryHandler(ITaskService<int> service)
        {
            _service = service;
        }

        public async System.Threading.Tasks.Task<Task<int>> Handle(
            GetTaskByIdQuery request,
            CancellationToken cancellationToken)
        {
            var result = await _service.GetTaskById(request.TaskId);
            return result;
```

```
        }
    }
}

using System.Collections.Generic;
using System.Threading;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using MediatR;

namespace Bugify.WebAPI.Features.TaskFeatures.Queries;

public class GetAllTasksQuery : IRequest<IEnumerable<Task<int>>>
{
    public class GetAllTasksQueryHandler : IRequestHandler<GetAllTasksQuery,
        IEnumerable<Task<int>>>
    {
        private readonly ITaskService<int> _service;

        public GetAllTasksQueryHandler(ITaskService<int> service)
        {
            _service = service;
        }

        public async System.Threading.Tasks.Task<IEnumerable<Task<int>>> Handle(
            GetAllTasksQuery request, CancellationToken cancellationToken)
        {
            var result = await _service.SearchTasks();
            return result;
        }
    }
}


using System.Threading;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using MediatR;

namespace Bugify.WebAPI.Features.TaskFeatures.Commands;

public class UpdateTaskCommand : IRequest<Task<int>>
{
    public Task<int> Task { get; set; }

    public class
        UpdateTaskCommandHandler : IRequestHandler<UpdateTaskCommand,
            Task<int>>
    {
        private readonly ITaskService<int> _service;

        public UpdateTaskCommandHandler(ITaskService<int> service)
        {
            _service = service;
        }

        public async System.Threading.Tasks.Task<Task<int>> Handle(
            UpdateTaskCommand request, CancellationToken cancellationToken)
        {
            var result = await _service.UpdateTask(request.Task);
            return result;
        }
    }
}


using System.Threading;
```

```
using Bugify.Domain.AggregatesModel.TaskAggregate;
using MediatR;

namespace Bugify.WebAPI.Features.TaskFeatures.Commands;

public class DeleteTaskCommand : IRequest<Task<int>>
{
    public Task<int> Task { get; set; }

    public class
        DeleteTaskCommandHandler : IRequestHandler<DeleteTaskCommand,
            Task<int>>
    {
        private readonly ITaskService<int> _service;

        public DeleteTaskCommandHandler(ITaskService<int> service)
        {
            _service = service;
        }

        public async System.Threading.Tasks.Task<Task<int>> Handle(
            DeleteTaskCommand request, CancellationToken cancellationToken)
        {
            var result = await _service.DeleteTask(request.Task);
            return result;
        }
    }
}

using System.Threading;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using MediatR;

namespace Bugify.WebAPI.Features.TaskFeatures.Commands;

public class CreateTaskCommand : IRequest<Task<int>>
{
    public Task<int> Task { get; set; }

    public class
        CreateTaskCommandHandler : IRequestHandler<CreateTaskCommand,
            Task<int>>
    {
        private readonly ITaskService<int> _service;

        public CreateTaskCommandHandler(ITaskService<int> service)
        {
            _service = service;
        }

        public async System.Threading.Tasks.Task<Task<int>> Handle(
            CreateTaskCommand request, CancellationToken cancellationToken)
        {
            var result = await _service.CreateTask(request.Task);
            return result;
        }
    }
}

using System.Threading;
using Bugify.Domain.AggregatesModel.TaskAggregate;
using MediatR;

namespace Bugify.WebAPI.Features.TaskFeatures.Commands;
```

```
public class AssignTaskToUserCommand : IRequest<Task<int>>
{
    public int TaskId { get; set; }
    public int UserId { get; set; }

    public class
        AssignTaskToUserCommandHandler : IRequestHandler<
            AssignTaskToUserCommand, Task<int>>
    {
        private readonly ITaskService<int> _service;

        public AssignTaskToUserCommandHandler(ITaskService<int> service)
        {
            _service = service;
        }

        public async System.Threading.Tasks.Task<Task<int>> Handle(
            AssignTaskToUserCommand request,
            CancellationToken cancellationToken)
        {
            var result =
                await _service.AssignTaskToUser(request.TaskId,
                    request.UserId);
            return result;
        }
    }
}

using System.Collections.Generic;
using Bugify.WebAPI.Features.TaskFeatures.Commands;
using Bugify.WebAPI.Features.TaskFeatures.Queries;
using Bugify.WebAPI.Filters;
using MediatR;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ThreadTasks = System.Threading.Tasks;

namespace Bugify.WebAPI.Controllers;

[Route("/[controller]")]
[ApiController]
[TypeFilter(typeof(ExceptionFilter))]
public class TaskController : ControllerBase
{
    private readonly IMediator _mediator;

    public TaskController(IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpGet("{id}")]
    [ProducesResponseType(typeof(Domain.AggregatesModel.TaskAggregate.Task<int>),
StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> GetById(int id)
    {
        return Ok(
            await _mediator.Send(new GetTaskByIdQuery { TaskId = id }));
    }

    [HttpGet]

[ProducesResponseType(typeof(IEnumerable<Domain.AggregatesModel.TaskAggregate.Task<int>>),
        StatusCodes.Status200OK)]
```

```
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> GetAll()
    {
        return Ok(await _mediator.Send(new GetAllTasksQuery()));
    }

    [HttpGet]
    [Route("search")]

[ProducesResponseType(typeof(IEnumerable<Domain.AggregatesModel.TaskAggregate.Task<int>>),
        StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> SearchTasks(
        [FromQuery] string searchString)
    {
        return Ok(await _mediator.Send(
            new GetTasksBySearchStringQuery { SearchString = searchString }));
    }

    [HttpGet]
    [Route("forUser")]

[ProducesResponseType(typeof(IEnumerable<Domain.AggregatesModel.TaskAggregate.Task<int>>),
        StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> GetTasksForUser([FromQuery] int userId)
    {
        return Ok(await _mediator.Send(
            new GetTasksByUserQuery { UserId = userId }));
    }

    [HttpGet]
    [Route("forProject")]

[ProducesResponseType(typeof(IEnumerable<Domain.AggregatesModel.TaskAggregate.Task<int>>),
        StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> GetTasksForProject(
        [FromQuery] int projectId)
    {
        return Ok(await _mediator.Send(
            new GetTasksByProjectQuery { ProjectId = projectId }));
    }

    [HttpPut]
    [Route("assign")]
    [ProducesResponseType(typeof(Domain.AggregatesModel.TaskAggregate.Task<int>),
StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> AssignTaskToUser([FromQuery] int bugId,
        int userId)
    {
        return Ok(await _mediator.Send(
            new AssignTaskToUserCommand { TaskId = bugId, UserId = userId }));
    }


    [HttpPost]
    [Route("create")]
    [ProducesResponseType(typeof(Domain.AggregatesModel.TaskAggregate.Task<int>),
        StatusCodes.Status201Created)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult>
CreateTask(Domain.AggregatesModel.TaskAggregate.Task<int> task)
    {
```

```
        var actionName = nameof(CreateTask);
        return CreatedAtAction(actionName, await _mediator.Send(
            new CreateTaskCommand
                { Task = task }));
    }

    [HttpPost]
    [Route("{id}")]
    [ProducesResponseType(typeof(Domain.AggregatesModel.TaskAggregate.Task<int>),
        StatusCodes.Status201Created)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult>
UpdateTask(Domain.AggregatesModel.TaskAggregate.Task<int> task)
    {
        var actionName = nameof(UpdateTask);
        return CreatedAtAction(actionName, await _mediator.Send(
            new UpdateTaskCommand
                { Task = task }));
    }

    [HttpDelete]
    [Route("{id}")]
    [ProducesResponseType(typeof(Domain.AggregatesModel.TaskAggregate.Task<int>),
        StatusCodes.Status201Created)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult>
DeleteTask(Domain.AggregatesModel.TaskAggregate.Task<int> task)
    {
        return Ok(await _mediator.Send(new DeleteTaskCommand
            { Task = task }));
    }

    [HttpGet]
    [Route("inFocus")]
    [ProducesResponseType(typeof(Domain.AggregatesModel.TaskAggregate.Task<int>),
        StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> GetTaskInFocusForUser(
        [FromQuery] int userId)
    {
        return Ok(await _mediator.Send(
            new GetTaskInFocusForUser { UserId = userId }));
    }

    [HttpGet]
    [Route("nowOrLater")]

[ProducesResponseType(typeof(IEnumerable<Domain.AggregatesModel.TaskAggregate.Task<int>>),
        StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> GetTasksNowOrLaterForUser(
        [FromQuery] int userId)
    {
        return Ok(await _mediator.Send(
            new GetTasksNowOrLaterForUser { UserId = userId }));
    }

    [HttpGet]
    [Route("total")]
    [ProducesResponseType(typeof(int),
        StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> GetTotalTasksForUser(
        [FromQuery] int userId)
    {
```

```
        return Ok(await _mediator.Send(
            new GetTotalTasksForUser { UserId = userId }));
    }

    [HttpGet]
    [Route("complete")]
    [ProducesResponseType(typeof(int),
        StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> GetCompleteTasksForUser(
        [FromQuery] int userId)
    {
        return Ok(await _mediator.Send(
            new GetCompleteTasksForUser { UserId = userId }));
    }

    [HttpGet]
    [Route("uncomplete")]
    [ProducesResponseType(typeof(int),
        StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> GetUncompleteTasksForUser(
        [FromQuery] int userId)
    {
        return Ok(await _mediator.Send(
            new GetUncompleteTasksForUser { UserId = userId }));
    }

    [HttpGet]
    [Route("overdue")]
    [ProducesResponseType(typeof(int),
        StatusCodes.Status200OK)]
    [ProducesDefaultResponseType]
    public async ThreadTasks.Task<IActionResult> GetOverdueTasksForUser(
        [FromQuery] int userId)
    {
        return Ok(await _mediator.Send(
            new GetOverdueTasksForUser { UserId = userId }));
    }
}


import { Typography, useTheme } from "@mui/material";
import { Box } from "@mui/material";
import { tokens } from "../theme/theme";


const InfoBox = (props: { label: string; num: number }) => {
  const theme = useTheme();
  const colors = tokens(theme.palette.mode);


  return (
    <Box
      sx={{
        height: 100,
        display: "flex",
        flexDirection: "column",
        justifyContent: "space-between",
```

```
        alignItems: "center",
        padding: "15px",
        border: 1,
        borderRadius: "8px",
      }}
    >
      <Typography variant="h5" color={colors.grey[1]} sx={{ m: "0px 0 5px 0" }}>
        {props.label}
      </Typography>
      <Typography
        variant="h2"
        color={colors.grey[1]}
        fontWeight="bold"
        sx={{ m: "0px 0 5px 0" }}
      >
        {props.num}
      </Typography>
    </Box>
  );
};

export default InfoBox;

import axios from "axios";
import { Serie } from "@nivo/line";
import { Task } from "../types/Task";
import {
  GET_ALL_TASKS,
  CREATE_TASK,
  GET_TASK,
  GET_TASKS_FOR_PROJECT,
  GET_TASK_IN_FOCUS,
  GET_TASKS_FOR_NOW_OR_LATER,
  GET_TASK_TOTAL,
  GET_TASK_COMPLETE,
  GET_TASK_UNCOMPLETE,
  GET_TASK_OVERDUE,
  GET_TASKS_FOR_USER,
} from "./CONSTANTS";

export class TaskService {
  private axiosInstance = axios.create({
    headers: {
```

```
    Accept: "application/json",
    "Content-Type": "application/json",
  },
});

private convertToTaskModel(item: any): Task {
  return new Task(item);
}

private convertToTaskModels(data: any[]): Task[] {
  return data.map(this.convertToTaskModel);
}

public async getTaskById(id: number): Promise<Task> {
  try {
    const { data } = await this.axiosInstance.get<Task>(`${GET_TASK()}${id}`);
    return this.convertToTaskModel(data);
  } catch (error) {
    console.log("Failed getTaskById to get Task:", error);
    throw new Error("Failed getTaskById to get Task");
  }
}

public async getAllTasks(): Promise<Task[]> {
  try {
    const { data } = await this.axiosInstance.get<Task[]>(GET_ALL_TASKS());
    return this.convertToTaskModels(data);
  } catch (error) {
    console.log("Failed getAllTasks to get tasks:", error);
    throw new Error("Failed getAllTasks to get tasks");
  }
}

public async filterTasks(
  name?: string,
  status?: string,
  type?: string,
  priority?: string,
  project?: string,
  assigned?: string
): Promise<Task[]> {
  const tasks = await this.getAllTasks();
  let filteredTasks = [...tasks];
```

```
  if (name) {
    filteredTasks = filteredTasks.filter((task) =>
      task.name.toLowerCase().includes(name.toLowerCase())
    );
  }

  if (status) {
    filteredTasks = filteredTasks.filter(
      (task) => task.getStatus() === status
    );
  }

  if (type) {
    filteredTasks = filteredTasks.filter((task) => task.getType() === type);
  }

  if (priority) {
    filteredTasks = filteredTasks.filter(
      (task) => task.getPriority() === priority
    );
  }

  if (project) {
    filteredTasks = filteredTasks.filter(
      (task) => task.project?.name === project
    );
  }

  if (assigned) {
    filteredTasks = filteredTasks.filter(
      (task) => task.assigned?.getFullName() === assigned
    );
  }

  return filteredTasks;
}

public async createTask(task: Task) {
  try {
    const { data, status: responseStatus } =
      await this.axiosInstance.post<Task>(
        CREATE_TASK(),
```

```
        JSON.stringify(task)
      );
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.log("error message filterTasks: ", error.message);
    } else {
      console.log("unexpected error filterTasks: ", error);
    }
  }
}

public async updateTask(task: Task) {
  try {
    const { data, status: responseStatus } =
      await this.axiosInstance.post<Task>(
        `${GET_TASK()}${task.id}`,
        JSON.stringify(task)
      );
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.log("error message updateTask: ", error.message);
    } else {
      console.log("unexpected error updateTask: ", error);
    }
  }
}

public async deleteTask(task: Task) {
  try {
    const { data, status: responseStatus } =
      await this.axiosInstance.delete<Task>(`${GET_TASK()}${task.id}`, {
        data: JSON.stringify(task),
      });
  } catch (error) {
    if (axios.isAxiosError(error)) {
      console.log("error message deleteTask: ", error.message);
    } else {
      console.log("unexpected error deleteTask: ", error);
    }
  }
}

public async getAllTasksForProject(projectId: number): Promise<Task[]> {
```

```
  try {
    const { data } = await this.axiosInstance.get<Task[]>(
      GET_TASKS_FOR_PROJECT(),
      { params: { projectId } }
    );
    return this.convertToTaskModels(data);
  } catch (error) {
    console.log("Failed getAllTasksForProject to get tasks:", error);
    throw new Error("Failed getAllTasksForProject to get tasks");
  }
}

public async getAllTasksForUser(userId: number): Promise<Task[]> {
  try {
    const { data } = await this.axiosInstance.get<Task[]>(
      GET_TASKS_FOR_USER(),
      { params: { userId } }
    );
    return this.convertToTaskModels(data);
  } catch (error) {
    console.log("Failed getAllTasksForUser to get tasks:", error);
    throw new Error("Failed getAllTasksForUser to get tasks");
  }
}

public async getTaskInFocus(userId: number): Promise<Task> {
  try {
    const { data } = await this.axiosInstance.get<Task>(GET_TASK_IN_FOCUS(), {
      params: { userId },
    });
    return this.convertToTaskModel(data);
  } catch (error) {
    console.log("Failed getTaskInFocus to get tasks:", error);
    throw new Error("Failed getTaskInFocus to get tasks");
  }
}

public async getAllTasksForNowOrLater(userId: number): Promise<Task[]> {
  try {
    const { data } = await this.axiosInstance.get<Task[]>(
      GET_TASKS_FOR_NOW_OR_LATER(),
      { params: { userId } }
    );
```

```
    return this.convertToTaskModels(data);
  } catch (error) {
    console.log("Failed getAllTasksForNowOrLater to get tasks:", error);
    throw new Error("Failed getAllTasksForNowOrLater to get tasks");
  }
}

public async getTaskTotal(userId: number): Promise<number> {
  try {
    const { data } = await this.axiosInstance.get<number>(GET_TASK_TOTAL(), {
      params: { userId },
    });
    return data;
  } catch (error) {
    console.log("Failed getTaskTotal to get tasks:", error);
    throw new Error("Failed getTaskTotal to get tasks");
  }
}

public async getTaskComplete(userId: number): Promise<number> {
  try {
    const { data } = await this.axiosInstance.get<number>(
      GET_TASK_COMPLETE(),
      {
        params: { userId },
      }
    );
    return data;
  } catch (error) {
    console.log("Failed getTaskComplete to get tasks:", error);
    throw new Error("Failed getTaskComplete to get tasks");
  }
}

public async getTaskUncomplete(userId: number): Promise<number> {
  try {
    const { data } = await this.axiosInstance.get<number>(
      GET_TASK_UNCOMPLETE(),
      {
        params: { userId },
      }
    );
    return data;
```

```
  } catch (error) {
    console.log("Failed getTaskUncomplete to get tasks:", error);
    throw new Error("Failed getTaskUncomplete to get tasks");
  }
}

public async getTaskOverdue(userId: number): Promise<number> {
  try {
    const { data } = await this.axiosInstance.get<number>(
      GET_TASK_OVERDUE(),
      {
        params: { userId },
      }
    );
    return data;
  } catch (error) {
    console.log("Failed getTaskOverdue to get tasks:", error);
    throw new Error("Failed getTaskOverdue to get tasks");
  }
}

public prepareDataForLineDigram(tasks: Task[]): Serie[] {
  const countsByMonth = tasks.reduce((counts, task) => {
    const month = task.deadline.getMonth() + 1;
    const year = task.deadline.getFullYear();
    const key = `${year}-${month}-01`;

    counts[key] = (counts[key] || 0) + 1;
    return counts;
  }, {} as Record<string, number>);

  const data: { x: string; y: number }[] = Object.entries(countsByMonth).map(
    ([key, count]) => ({
      x: key,
      y: count,
    })
  );

  data.sort((a, b) => new Date(a.x).getTime() - new Date(b.x).getTime());

  const result: Serie = {
    id: "tasks",
    data,
```

```
    };

    return [result];
  }

  public prepareDataForBarDiagram(
    tasks: Task[]
  ): { project: string; complete: number; uncomplete: number }[] {
    const countsByProject: Record<
      string,
      { complete: number; uncomplete: number }
    > = {};

    tasks.forEach((task) => {
      const key = task.project?.name || "";
      const isComplete = task.status > 2;

      if (!countsByProject[key]) {
        countsByProject[key] = { complete: 0, uncomplete: 0 };
      }

      if (isComplete) {
        countsByProject[key].complete += 1;
      } else {
        countsByProject[key].uncomplete += 1;
      }
    });

    const data: { project: string; complete: number; uncomplete: number }[] =
      Object.entries(countsByProject).map(([key, count]) => ({
        project: key,
        complete: count.complete,
        uncomplete: count.uncomplete,
      }));

    return data;
  }
}

import { Project } from "./Project";
import { User } from "./User";

export class Task {
```

```
public id: number | undefined;
public name: string = "";
public description: string = "";
public projectId: number = 1;
public project: Project | null = new Project();
public deadline: Date = new Date();
public status: number = 1; //string = "NEW"; // "NEW" | "PROGRESS" | "TESTING" | "CLOSED"
public type: number = 1; // string = "TASK"; // "ISUEE" | "FEATURE" | "TASK" | "BUG"
public assignedId: number = 1;
public assigned: User | null = new User();
public authorId: number = 1; //string = "";
public priority: number = 3; // string = "NORMAL"; // "MINOR" | "LOW" | "NORMAL" | "HIGH"
public difficulty: number = 0;

constructor(initializer?: any) {
  if (!initializer) return;
  if (initializer.id) this.id = initializer.id;
  if (initializer.name) this.name = initializer.name;
  if (initializer.description) this.description = initializer.description;
  if (initializer.projectId) this.projectId = initializer.projectId;
  if (initializer.project) this.project = new Project(initializer.project);
  if (initializer.deadline) this.deadline = new Date(initializer.deadline);
  if (initializer.status) this.status = initializer.status;
  if (initializer.type) this.type = initializer.type;
  if (initializer.assignedId) this.assignedId = initializer.assignedId;
  if (initializer.assigned) this.assigned = new User(initializer.assigned);
  if (initializer.authorId) this.authorId = initializer.authorId;
  if (initializer.priority) this.priority = initializer.priority;
  if (initializer.difficulty) this.difficulty = initializer.difficulty;
}

getStatus(): string {
  const statuses = ["NEW", "PROGRESS", "TESTING", "CLOSED"];
  return statuses[this.status - 1];
}

getType(): string {
  const types = ["ISSUE", "FEATURE", "TASK", "BUG"];
  return types[this.type - 1];
}

getPriority(): string {
  const priorities = ["MINOR", "LOW", "NORMAL", "HIGH"];
```

```
    return priorities[this.priority - 1];
  }
}

import { Box, Button, Typography, useTheme } from "@mui/material";
import { tokens } from "../../theme/theme";
import AccessTimeIcon from "@mui/icons-material/AccessTime";
import GetLabel from "../../components/Label";
import { Task } from "../../types/Task";
import { getTimeLeftTo } from "../../utils/TimeDate";
import { TASKS } from "../../navigation/CONSTANTS";
import { Link } from "react-router-dom";

const TaskCard = (props: { task: Task }) => {
  const task = props.task;
  const theme = useTheme();
  const colors = tokens(theme.palette.mode);

  return (
    <div>
      <Box
        sx={{
          height: 220,
          width: "100%",
          display: "flex",
          flexDirection: "column",
          justifyContent: "space-between",
          padding: "15px",
          border: 1,
          borderRadius: "8px",
          margin: "8px 8px 0 0",
        }}
      >
        <Box display="flex" flexDirection="column">
          {/* LABELS BOX */}
          <Box display="flex" mb="3px">
            {GetLabel(task.getType().toLowerCase())}
            {GetLabel(task.getStatus().toLowerCase())}
            {GetLabel(task.getPriority().toLowerCase())}
          </Box>

          <Typography
            variant="h4"
```

```
      color={colors.grey[1]}
      fontWeight="bold"
      sx={{ m: "10px 0 15px 0" }}
    >
      {task.name}
    </Typography>

    <Typography
      variant="h5"
      color={colors.grey[1]}
      sx={{ m: "0px 0 5px 0" }}
    >
      Assign to: <i>{task.assigned?.getFullName()}</i>
    </Typography>

    <Typography
      variant="h5"
      color={colors.grey[1]}
      sx={{ m: "0px 0 5px 0" }}
    >
      Project: <i>{task.project?.name}</i>
    </Typography>
  </Box>

  {/* TIME ICON */}
  <Box
    display="flex"
    justifyContent="flex-end"
    sx={{ m: "0px 0 0px 3px" }}
  >
    <Link to={TASKS + "/" + task.id} state={{ stateTask: task }}>
      <Button size="small">Edit</Button>
    </Link>
    <Box display="flex" justifyContent="flex-end">
      <AccessTimeIcon />
      <Typography
        variant="h6"
        color={colors.grey[2]}
        sx={{ m: "0px 0 0px 3px" }}
      >
        {getTimeLeftTo(task.deadline)}
      </Typography>
    </Box>
```

```
        </Box>
      </Box>
    </div>
  );
};

export default TaskCard;

import { useLocation } from "react-router-dom";
import { Task } from "../../types/Task";
import TaskForm from "./TaskForm";

const EditTaskPage = () => {
  const location = useLocation();
  const task = location.state?.stateTask;
  console.log(task);

  if (!task) {
    return <TaskForm task={new Task()} />;
  }

  return <TaskForm task={task} />;
};

export default EditTaskPage;

import {
  Box,
  Button,
  MenuItem,
  TextField,
  useMediaQuery,
  useTheme,
} from "@mui/material";
import { DatePicker } from "@mui/x-date-pickers/DatePicker";
import Header from "../../components/Header";
import { Formik } from "formik";
import * as yup from "yup";
import { Task } from "../../types/Task";
import { tokens } from "../../theme/theme";
import { useNavigate } from "react-router-dom";
import { TaskService } from "../../services/taskService";
import dayjs from "dayjs";
```

```
import { useEffect, useState } from "react";
import { Project } from "../../types/Project";
import { User } from "../../types/User";
import { ProjectService } from "../../services/projectService";
import { UserService } from "../../services/userService";

const TaskForm = (props: { task: Task }) => {
  const navigate = useNavigate();
  const theme = useTheme();
  const colors = tokens(theme.palette.mode);

  const service = new TaskService();
  const projectService = new ProjectService();
  const userService = new UserService();
  const [projects, setProjects] = useState<Project[]>([]);
  const [users, setUsers] = useState<User[]>([]);

  useEffect(() => {
    async function loadData() {
      try {
        const projects = await projectService.getAllProjects();
        setProjects(projects);
        const users = await userService.getAllUsers();
        setUsers(users);
      } catch (e) {
        console.log("Error in load data " + e);
      }
    }
    loadData();
  }, []);

  const task = props.task;
  const isNonMobile = useMediaQuery("(min-width:600px)");

  const handleFormSave = async (values: Task) => {
    if (!values.id) {
      values.project = null;
      values.assigned = null;
      await service.createTask(values);
    } else {
      values.project = projects[values.projectId - 1];
      values.assigned = users[values.assignedId - 1];
      await service.updateTask(values);
```

```
  }
  console.log("Save " + values.name);
  navigate(-1);
};

const handleFormDelete = async (values: Task) => {
  await service.deleteTask(values);
  console.log("Delete " + values.name);
  navigate(-1);
};

return (
  <Box m="20px">
    <Header title="TASK FORM" />

    <Formik
      onSubmit={handleFormSave}
      initialValues={task}
      validationSchema={checkoutSchema}
    >
      {({
        values,
        errors,
        touched,
        handleBlur,
        handleChange,
        handleSubmit,
        isSubmitting,
      }) => (
        <form onSubmit={handleSubmit}>
          <Box
            width="100%"
            display="grid"
            gap="20px"
            gridTemplateColumns="repeat(2, minmax(0, 1fr))"
            sx={{
              "& > div": { gridColumn: isNonMobile ? undefined : "span 4" },
            }}
          >
            <Box
              width="100%"
              display="grid"
              gap="20px"
```

```
          gridTemplateColumns="repeat(4, minmax(0, 1fr))"
          sx={{
            "& > div": { gridColumn: isNonMobile ? undefined : "span 4" },
          }}
        >
          <TextField
            fullWidth
            variant="filled"
            type="text"
            label="Name"
            onBlur={handleBlur}
            onChange={handleChange}
            value={values.name}
            name="name"
            error={!!touched.name && !!errors.name}
            helperText={touched.name && errors.name}
            sx={{ gridColumn: "span 4" }}
          />
          <TextField
            fullWidth
            variant="filled"
            type="text"
            label="Priority"
            select
            onBlur={handleBlur}
            onChange={handleChange}
            value={values.priority}
            name="priority"
            error={!!touched.priority && !!errors.priority}
            helperText={touched.priority && errors.priority}
            sx={{ gridColumn: "span 4" }}
          >
            <MenuItem value={1}>Minor</MenuItem>
            <MenuItem value={2}>Low</MenuItem>
            <MenuItem value={3}>Normal</MenuItem>
            <MenuItem value={4}>High</MenuItem>
          </TextField>
          <DatePicker
            label="Deadline"
            value={dayjs(values.deadline)}
            onChange={(value) => {
              handleChange({
                target: {
```

```
              name: "deadline",
              value: value,
            },
          });
        }}
        slotProps={{ textField: { variant: "filled" } }}
        sx={{ gridColumn: "span 4" }}
      />
      <TextField
        fullWidth
        variant="filled"
        type="number"
        label="Difficulty"
        onBlur={handleBlur}
        onChange={handleChange}
        value={values.difficulty}
        name="difficulty"
        error={!!touched.difficulty && !!errors.difficulty}
        helperText={touched.difficulty && errors.difficulty}
        sx={{ gridColumn: "span 4" }}
      />
    </Box>
    <Box
      width="100%"
      display="grid"
      gap="20px"
      gridTemplateColumns="repeat(4, minmax(0, 1fr))"
      gridTemplateRows="repeat(7, minmax(0, 1fr))"
      sx={{
        "& > div": { gridColumn: isNonMobile ? undefined : "span 4" },
      }}
    >
    </Box>
  </Box>
  {/* Buttons */}
  <Box display="flex" mt="20px">
    <Button type="submit" color="secondary" variant="contained">
      Save
    </Button>
    <Button
      onClick={(e) => handleFormDelete(values)}
      variant="contained"
      sx={{ ml: "10px" }}
```

```
                style={{
                  backgroundColor: colors.redAccent[5],
                  color: "black",
                }}
              >
                Delete
              </Button>
            </Box>
          </form>
        )}
      </Formik>
    </Box>
  );
};

const checkoutSchema = yup.object().shape({
  name: yup.string().required("required"),
  description: yup.string().required("required"),
  deadline: yup.string().required("required"),
  status: yup.number().required("required"),
  type: yup.number().required("required"),
  priority: yup.number().required("required"),
  difficulty: yup.number().required("required"),
});

export default TaskForm;

export const getTimeLeftTo = (date: Date): string => {
  const today = new Date();

  const millisecondsPerDay = 1000 * 60 * 60 * 24;
  const daysLeft = Math.ceil(
    (date.getTime() - today.getTime()) / millisecondsPerDay
  );
  if (daysLeft > 0) return daysLeft + "d";

  const millisecondsPerHour = 1000 * 60 * 60;
  let hoursLeft = Math.ceil(
    (date.getTime() - today.getTime()) / millisecondsPerHour
  );
  if (hoursLeft < 0) hoursLeft = 0;
  return hoursLeft + "h";
};
```