ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка програмного забезпечення для управління нормативно регульованими проєктами»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Дмитро МАКСИМЕНКО
(підпис)

Виконав: здобувач вищої освіти групи ПД-41
Дмитро МАКСИМЕНКО

Керівник: Віталій ЗАЛИВА
доктор філософії (PhD)

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Максименку Дмитру Сергійовичу

1. Тема кваліфікаційної роботи: «Розробка програмного забезпечення для управління нормативно регульованими проєктами»
керівник кваліфікаційної роботи старший викладач кафедри, доктор філософії (PhD) Віталій ЗАЛИВА,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про принципи управління проєктами та особливості нормативно регульованих процесів, вимоги до інформаційної безпеки, аудиту та контролю доступу, підходи до шаблонізації даних і автоматизації бізнес-процесів.

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

1. Огляд та аналіз існуючих рішень для керування проєктами.
2. Проєктування вимог до програмного забезпечення.
3. Огляд засобів реалізації програмного забезпечення для керування нормативно регульованими проєктами.

4. Проєктування і розробка програмного забезпечення для керування нормативно регульованими проєктами.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Опис ключових структур для роботи з шаблонами.
6. Діаграма послідовностей завантаження сторінки проєкту.
7. Екранні форми
8. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Огляд та аналіз існуючих рішень для керування проєктами	14.03-28.03.2025	
4	Проєктування вимог до програмного забезпечення	29.03-06.04.2025	
5	Огляд засобів реалізації програмного забезпечення для керування нормативно регульованими проєктами	07.04-18.04.2025	
6	Проєктування та розробка програмного забезпечення для керування нормативно регульованими проєктами	19.04-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Дмитро МАКСИМЕНКО
(підпис)

Керівник
кваліфікаційної роботи _____ Віталій ЗАЛИВА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: __52__ стор., __3__ табл., _16_ рис., _21_ джерел.

Мета роботи – автоматизація процесу виконання нормативно регульованих проєктів.

Об'єкт дослідження – процес керування та виконання нормативно регульованих процесів.

Предмет дослідження – програмне забезпечення для управління нормативно регульованими проєктами.

Короткий зміст роботи: В роботі проаналізовано існуючі підходи та методи управління проєктами. Розглянуті та проаналізовані платформи для керування проєктами та виявлено ключові недоліка та обмеження їхньої функціональності. Розроблено архітектуру системи, варіанти використання, програмно реалізовані ключові функціональні можливості, зокрема: створення та керування шаблонами проєктів та задач, створення та керування користувацькими ролями, керування проєктами та система аудиту дій користувачів. В роботі використано сучасні підходи до проєктування програмного забезпечення, фреймворк NestJS для розроблення серверної частини та бібліотеку React.js для клієнтської частини.

Сферою використання застосунку є керування проєктами.

КЛЮЧОВІ СЛОВА: УПРАВЛІННЯ ПРОЄКТКАМИ, РОЗРОБКА, ШАБЛОНІЗАЦІЯ, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, АВТОМАТИЗАЦІЯ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

MVP - Мінімально життєздатний продукт (Minimum viable product)

СУБД - Система управління базою даних

JSON - JavaScript Object Notation

SPA - Single-Page Application

DOM - Document Object Model

REST - Representational State Transfer

API - Application Programming Interface

CLI - Command-line interface

ВСТУП

Для комерційних організацій та компаній основою їх діяльності є безпосередньо комерціалізація продукту, та кожна з них обирає свій підхід до збільшення прибутку. Один з них, скорочення витрат та прискорення процесів. Винайшовши чи удосконаливши процес, що він вимагає менше ресурсів, компанія суворо слідкує за його виконанням, оскільки це основний спосіб збільшення частки прибутків. Однак якщо відповідальність за контроль дотримання вимог падає на співробітника, якому в ручному режимі потрібно слідкувати за відповідністю виконання процесу, що слугує передумовою виникнення наступних проблем:

- велику кількість рутинних задач, які потребують часу та людських ресурсів;
- підвищену ймовірність виникнення помилок через людський фактор;
- низька прозорість та контроль.

Створення програмного забезпечення, яка буде дозволяти створювати шаблони та додавати перевірки відповідно до нормативного документу, політик чи стандартів організації, може значно покращити ситуацію стосовно кожної з наведених вище проблем. Також, важливим аспектом такої системи є централізоване адміністрування, яке буде підвищувати безпеку стабільності та відповідності шаблонів та правил, та простота інтеграції нових виконавців та користувачів до програмного забезпечення, що зменшить час навчання нових співробітників та дозволить невеликим організаціям чи групам простіше автоматизувати робочі процеси.

Наразі є велика кількість перевірених часом програмних рішень, які покривають частину з наведених потреб, однак не в єдиному рішенні, або лише при залученні сторонніх користувацьких плагінів чи прямої модифікації спеціалістом.

Розроблене програмне забезпечення являє собою MVP де наведена частина функціоналу, виділена для безпосередньо проєктів, фокус в яких стоїть над виконанням стандартизованих процесів з максимізацією гнучкості системи керування та мінімізацією можливостей припуститися помилки в їх виконанні.

Робота пройшла апробацію: Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ» за темою «Забезпечення цілісності та безпеки внутрішніх процесів у системі управління нормативно регульованими проєктами»[1] та XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025» за темою «Порівняння засобів захисту інформації в системах управління проєктами при хмарному та локальному розгортанні»[2].

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Особливості керування проєктами

Керування проєктами – це комплекс заходів, спрямованих на досягнення визначених цілей у межах конкретного проєкту у межах визначених обмежень у часі, бюджету та якості. Загальною характеристикою проєкту[3, с. 15-25] є його тимчасовість, унікальність результату та обмеженість ресурсами, тому управління проєктом передбачає систематичний підхід до координації багатьох взаємопов’язаних елементів.

На Рисунку 1.1 зображений загальний цикл проведення проєкту.

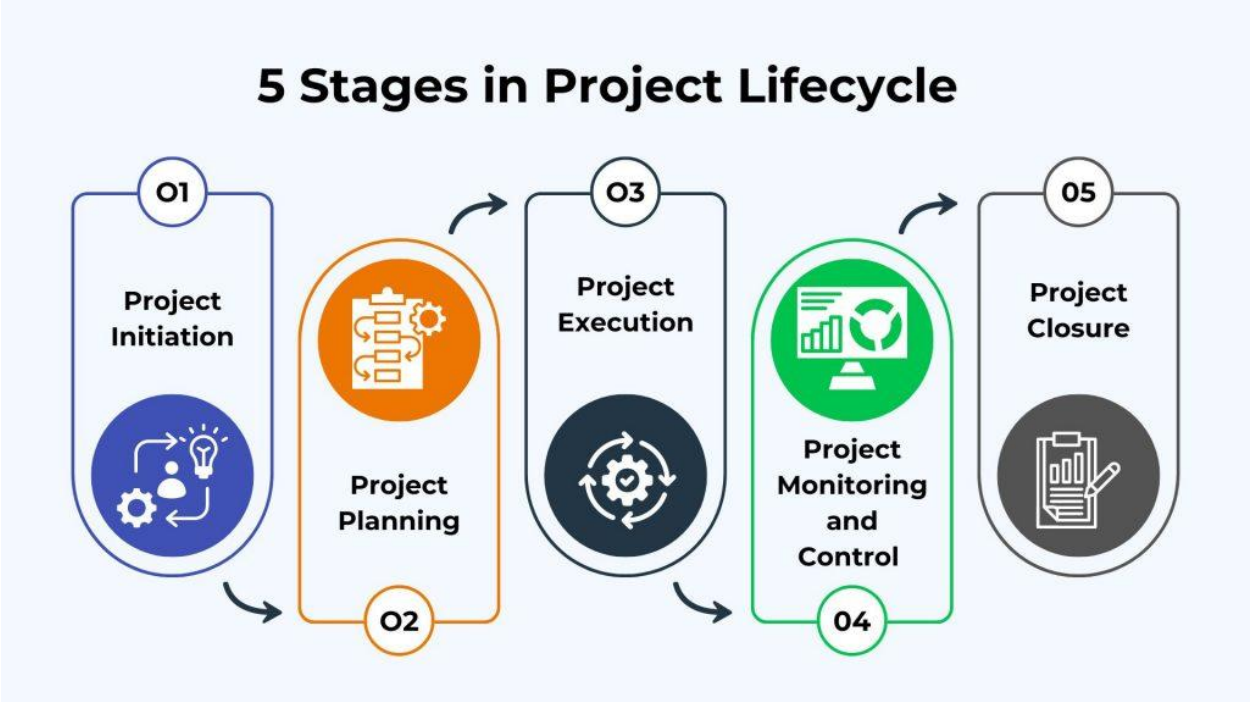


Рис. 1.1 Загальний життєвий цикл керування проєктом

В контексті даної роботи, варто звернути особливу увагу на етап моніторингу і контролю. Після успішного планування та запуску проєкту, в нього повинні бути визначенні певні критерії успіху, стандарти та нормативні документи, які будуть вести проєкт до його успішного завершення.

Саме на етапі моніторингу і контролю виникає потреба в постійній перевірці відповідності дій учасників проєкту визначеним політикам, внутрішнім регламентам та зовнішнім стандартам. У випадку відхилення або помилок, важливо мати механізм їх оперативного виявлення та внесення коригувань у хід робіт, або взагалі обмежити виконання аби позбутися можливості виникнення помилок. Це дозволяє підтримувати якість та контроль за ресурсами та результатами на належному рівні.

Таким чином, ключовими викликами на цих етапах є забезпечення прозорості, стабільності, а також можливість легко перевірити відповідність кожного кроку проєкту встановленим вимога. Для цього важливо мати розширений аудит дій, для подальшого аналізу, чітку та непорушну структуру проєкту, яка буде описана в регламентах та реалізована в незмінних шаблонах, та інструменти швидкого та зручного адміністрування та реагування на окремі інциденти під час процесу керування проєктом.

Ще одним важливим аспектом побудови ефективного процесу керування проєктами є мінімізація часу навчання нового персоналу та адаптація співробітників до нових вимог та стандартів.[4] У цьому контексті, система повинна обслуговувати інтереси трьох рівнів персоналу. По-перше, адміністратор, який володіє глибокими знаннями нормативних документів, політик, стандартів та специфікацій, потребує інструменти для точного відображення цих вимог у вигляді шаблонів та правил. Для цього важливі гнучкість і повнота функціоналу, а не спрощення. По-друге, керівник проєкту зацікавлений у швидкій адаптації та зборі даних – надлишкова складність та необхідність розбиратись у деталях стандартів лише заважатимуть його роботі. Тому важливо мати доступ до готових, перевірених шаблонів, які гарантують відповідність без глибокого занурення у регламенти. По-третє, виконавець, який може бути новачком або мати обмежене уявлення про вимоги, повинен мати можливість негайно приступити до виконання завдань без ризику порушити встановлені правила. На Рисунку 1.2 зображене спрощене візуальне представлення основних потреб користувачів при використанні такого підходу до організації процесу.

Завдяки заздалегідь визначеним сценаріям дій та обмеженню можливості відхилення від шаблонів, система мінімізує ризик помилок, забезпечує контроль якості та значно скорочує час на введення нового персоналу в робочий процес.

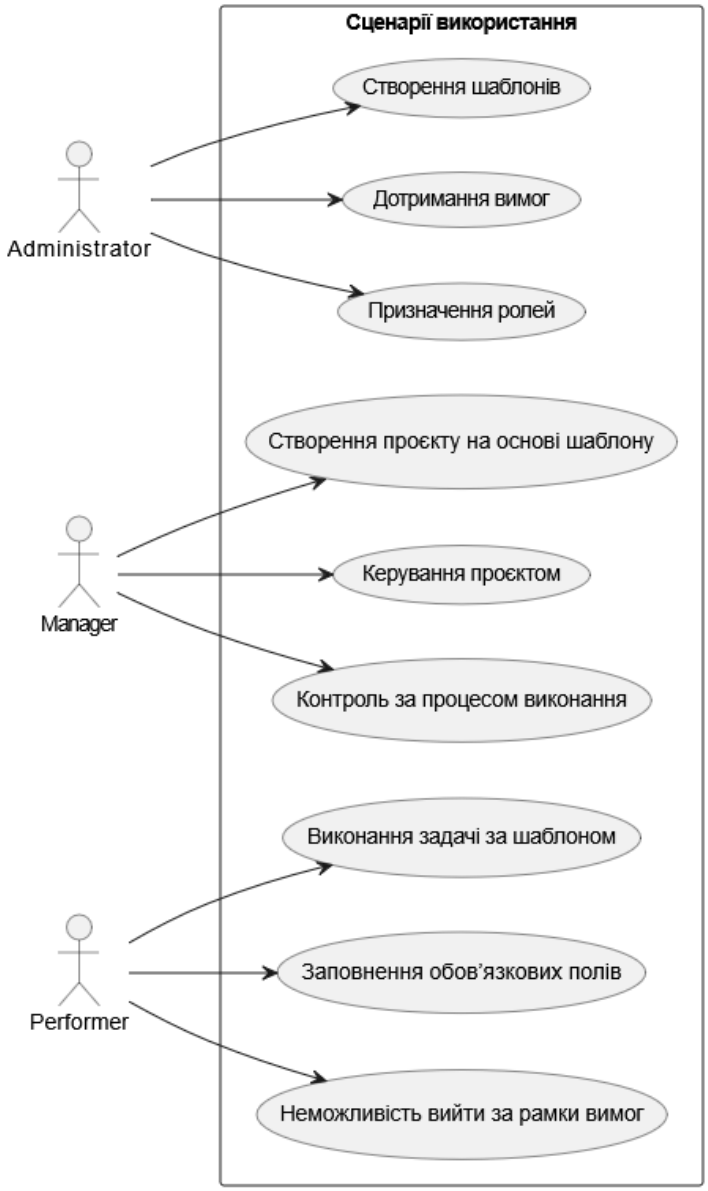


Рис. 1.2 Спрощене представлення основних потреб користувачів

1.2 Підходи до організації процесу керування проєктами

Організація процесу керування проєктами відрізняється в різних компаній, залежно від їхніх підходів до організації всіх внутрішніх процесів, таким чином,

можна виділити два граничних підходи, з жорстоко регламентованим підходом, гнучкі підходи та спектр між ними.

Жорстоко регламентовані підходи, такі як Waterfall та PRINCE2 передбачають суворе планування, фіксацію вимог на початковому етапі проєкту та поступове проходження всіх фаз життєвого циклу проєкту, від ініціації до завершення. На Рисунку 1.3 зображена візуалізація моделі Waterfall з якої видно чітку послідовність життєвого циклу проєкту. Перевага таких підходів у передбачуваності, чіткості та відповідності стандартам, що особливо важливо в проєктах з суворою документацією. З точки зору бізнесу це дозволяє ефективно контролювати бюджет та терміни, але вимагає високої стійкості до змін, через непристосованість такого підходу до нестабільних проєктів.

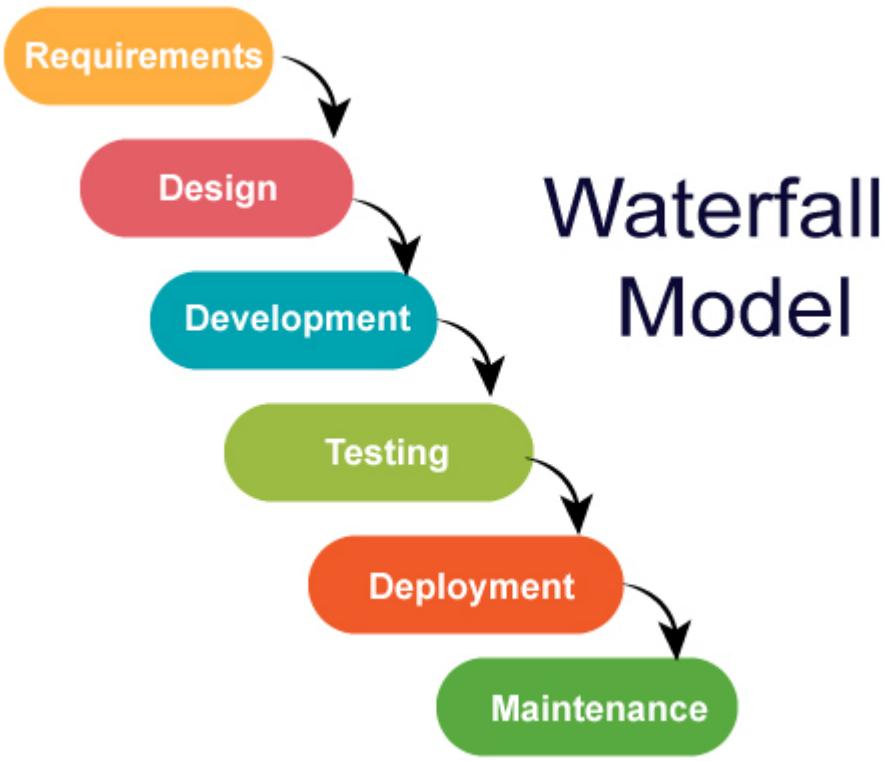


Рис. 1.3 Модель Waterfall

Гнучкі ж підходи, такі як Agile, Scrum та Kanban ґрунтуються на ітеративній моделі з постійною адаптацією до зміни вимог. Вони жертвують частиною ефективності в моменті, щоб мінімізувати ризики, що проєкт буде повністю

провальним, через постійні перевірки та перегляди обставин. Бізнес отримує гнучкість, швидку реакцію на ринок та вищу задоволеність замовника, але втрачає частину передбачуваності щодо термінів та витрат. На Рисунку 1.4 зображено модель методології Agile, з якої видно, що чіткого обмеження в кількості ітерацій проєкту, немає, що робить проєкт гнучкішим, але менш визначеним.[5]

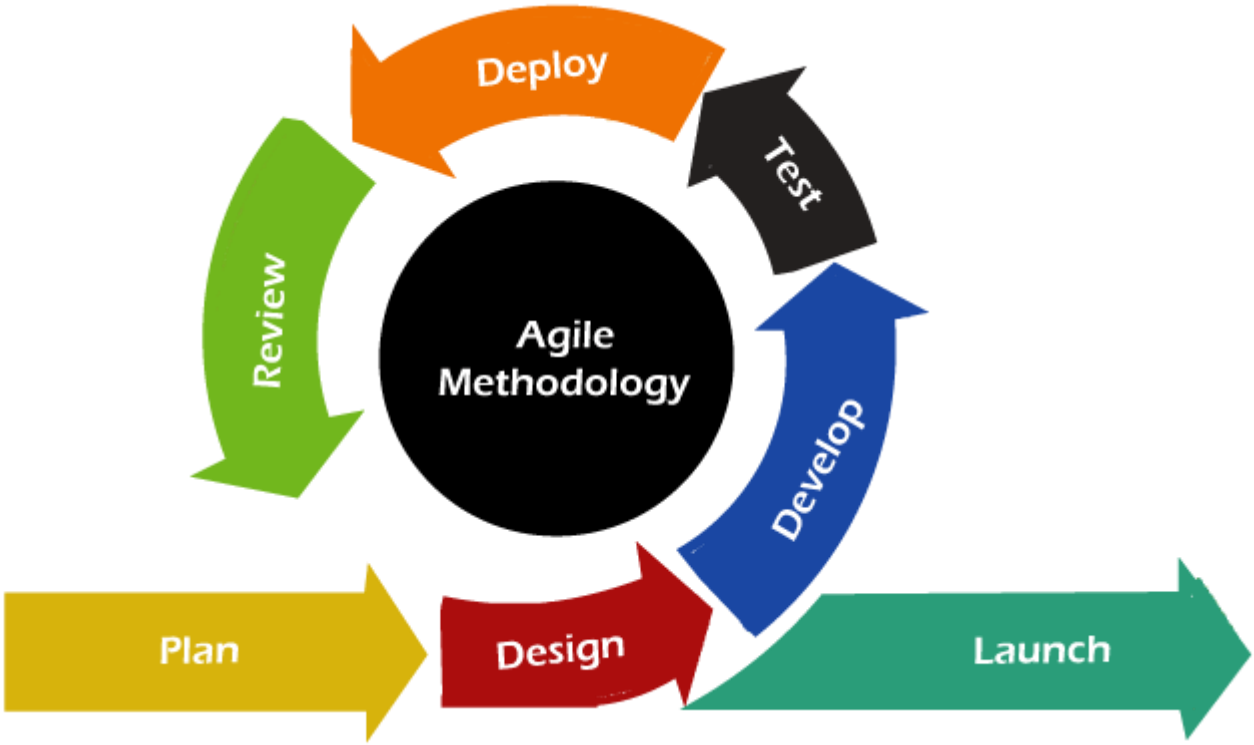


Рис. 1.4 Методологія Agile

Варіативність комбінацій та проміжних форм керування проєктами може бути безліч та вони можуть видозмінюватися залежно від конкретного керівника.

1.3 Особливості нормативно регульованих проєктів

Нормативно регульовані проєкти реалізуються в умовах суворого контролю та відповідності встановленим стандартам. Найзрозумілішим прикладом сфери де такі проєкти виконуються це державний сектор, в якому кожен процес регламентується в нормативних документах, як наприклад написання кваліфікаційної роботи студента в державних навчальних закладах. Студент не

може пропустити декілька етапів написання, узгодження, перевірок та уточнень, оскільки вони гарантують відповідність кінцевого результату цього проєкту до зразка.

Основні особливості таких проєктів:

- Адміністрування: кожен етап роботи проходить перевірку відповідним органом чи відповідальною особою, якщо результат конкретного кроку не відповідає вимогам, він не вважається виконаним, а на перевірку всіх попередніх вимог, час не витрачається, оскільки вважається, що після успішної перевірки, результат є відповідним до вимог.
- Мінімізація помилок при виконання: якщо частину виконання вимог можна автоматизувати створенням шаблонів, заготовок та форм, цей процес автоматизується, що дозволяє зменшити кількість помилок на етапі виконання та спростити процес виконання для нових виконавців.
- Прозорість та контроль: За кожним етапом є відповідальна особа або орган, який гарантує що його етап був пройдений успішно. У разі виявлення помилки, повинна бути визначена відповідальна особа, для подальшого виправлення та усунення можливості припуститися такої помилки знову.
- Фіксація вимог: У таких проєктах вимоги не повинні змінюватися без проходження відповідних процедур впровадження. Це знижує гнучкість, але забезпечує передбачуваність та відповідність очікуваним результатам.

1.4 Аналіз аналогів

Notion

Notion[6] - це застосунок орієнтований на дві категорії задач. Перша задача - ведення нотаток, створення власної бібліотеки баз даних та знань. Друга -

організація задач та проєктів. Ці, а також фактор високої гнучкості та поширеності серед користувачів, роблять Notion популярним вибором при першому огляді наявних інструментів для організації власного навчання чи часу, та створення невеликих проєктів.

На Рисунку 1.5 зображений приклад оформлення системи контролю за особистими задачами в застосунку Notion.

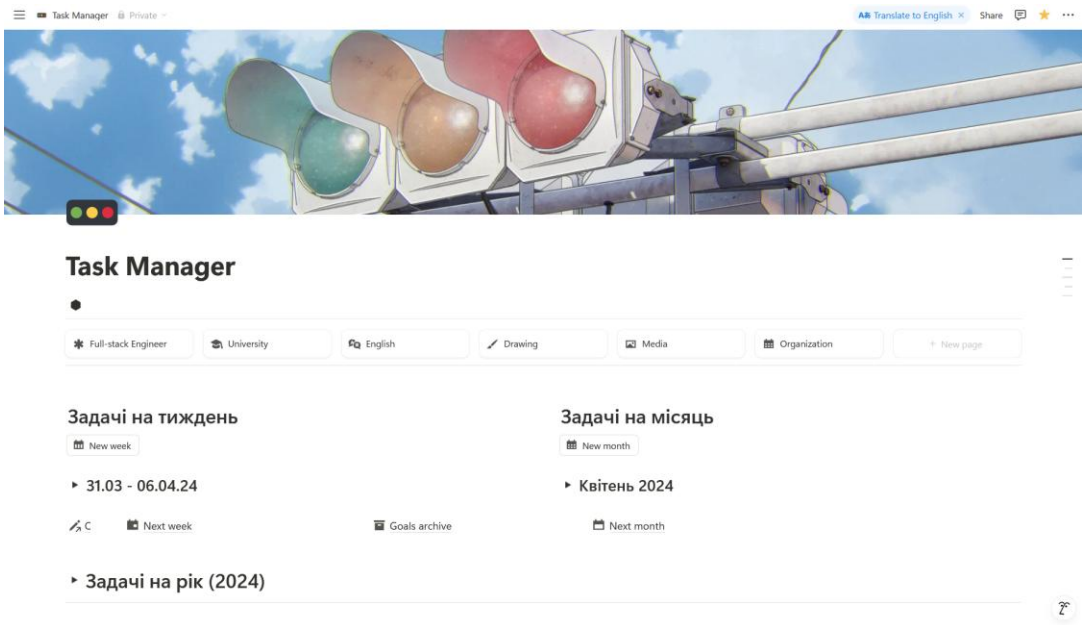


Рис. 1.5 Приклад реалізації сторінки в Notion

На сьогоднішній день, у Notion можна виділити наступні особливості:

- Блочна система організації сторінки, що дозволяє створювати індивідуальні сторінки.

- Власна система баз даних та можливість переглядати її у різних візуальних відображеннях.

- Формули, які надають зручний та широкий функціонал обробки даних. Це дозволяє створювати спеціалізовані сторінки, які будуть відповідати специфічним вимогам користувача.

- Шаблони. Широкий спектр вже існуючих користувацьких шаблонів та зручна система створення власних з можливістю додавання динамічних даних та планування регулярного створення сторінок за цим шаблоном.

- Notion AI. Помічник з використанням технологій штучного інтелекту, який може допомогти у обробці даних або генерації сторінок.

- Інтеграція з дочірнім сервісом Notion Calendar та декількома зовнішніми сервісами, як GitHub, Google Drive, Asana, Zoom.

Переваги:

- Наявність шаблонів. Цей функціонал дозволяє автоматизувати комплексі та повторювані задачі, що може економити велику кількість часу та зменшувати ймовірність помилки.

- Високий рівень кастомізації. З комбінацією всіх наведених особливостей, Notion дає можливість створити майже будь-яку систему, яка користувачеві тільки може знадобитися.

- Easy to learn. Освоєння базового функціоналу не займає багато часу, оскільки є близьким до інтуїтивного, багато реалізацій запозичено з вже класичних та широковикористовуваних застосунків.

Недоліки:

- Hard to master. На освоєння функціоналу, для створення ідеальної для себе системи може піти багато часу, від декількох днів, для основного розуміння, до місяця, для повного розуміння всіх особливостей.

- Неоднорідність. Через високий рівень кастомізації, система яку зробив користувач для себе, скоріше за все, не підійде іншому користувачеві, що створює проблеми під час налагодження робочого процесу в команді. Також, відсутність блокування зміни оформлення сторінки, для окремих користувачів, дає їм можливість порушувати однорідність процесу.

Notion насправді є дуже потужним інструментом, як для організації особистої роботи, так і для командної. Але через його комплексність, для великих проєктів будуть складності в організації процесу та необхідність глибокого розуміння ще більше ускладнює ситуацію.

1.4.1 Redmine

Redmine[7] - це одне з класичних рішень по управлінню проєктами, яке можна зустріти у великих компаніях та корпораціях, оскільки існує вже 18 років, та підтримується по сьогодні. Це безкоштовне та open source рішення, яке націлене на розгортання в корпоративній мережі компанії. Оскільки це проєкт з відкритим вихідним, його можна легко модифікувати власними розширеннями та створювати відгалуження (fork) з власними модифікаціями. Наразі, вже існує велика кількість популярних відгалужень, які додають багато нового функціоналу, поліпшують візуальну складову та надають підтримку компаніям, які використовують їх версію. Наприклад OpenProject[8], Easy Redmine, Redmine X та ChiliProject. На Рисунку 1.6 зображений візуальний інтерфейс застосунку Redmine.

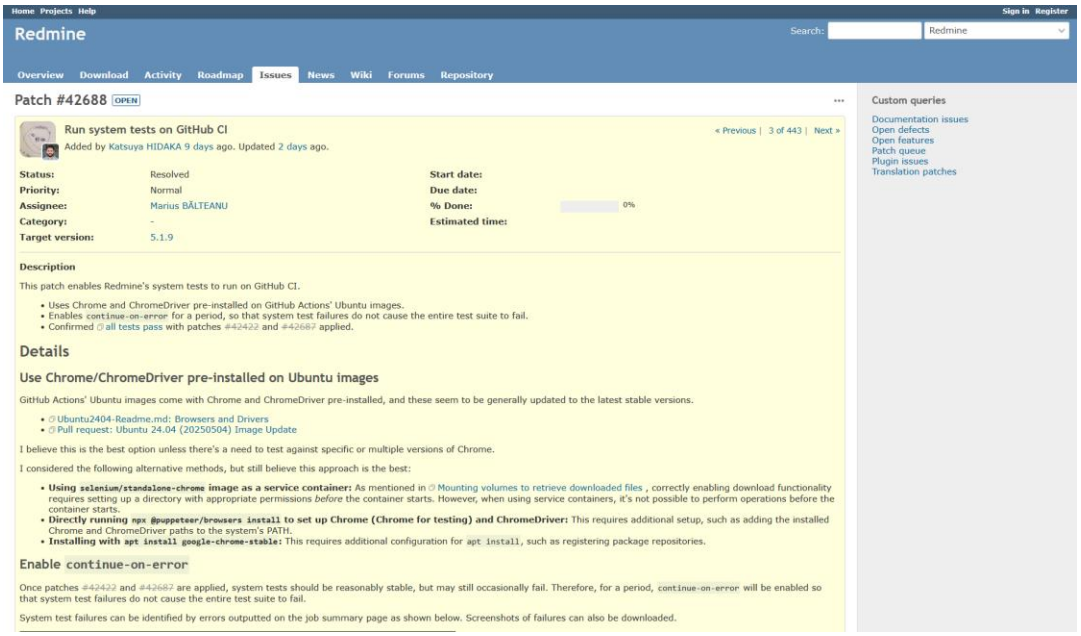


Рис. 1.6 Візуальний інтерфейс застосунку Redmine

Переваги:

- Відкритий програмний код. Компанія яка використовує цю платформу, може перевірити чи немає в застосунку шкідливого програмного коду чи витоку інформації.

- Локальне розгортання платформи. Відповідальність за запуск проєкту, перекладається на користувача, що дає користувачеві розуміння стабільності

роботи проєкту, оскільки він сам обирає апаратне забезпечення де буде розгортатися платформа.

- Контроль над версіями. Користувач особисто обирає яку версію платформи ставити та коли її оновлювати до наступної. Це дозволяє захистити систему від випадкових помилок в нових версіях.

Недоліки:

- Складність розгортання. Для того щоб розгортати, налаштовувати та підтримувати платформу, потрібна буде окрема людина, що може бути проблемою для маленьких проєктів.

- Вузький функціонал "з коробки". Базова функціональність платформи є достатньою для більшості проєктів, але у порівнянні з тим, що можна отримати з використанням користувацьких плагінів, виглядає недостатньою.

- Застарілий UI/UX дизайн. У порівнянні з сучасними застосунками, візуальна частина платформи застаріла.

Redmine це класична, але застаріла платформа, яку обирати для нових проєктів та компаній, буде не найкращим рішенням, оскільки вона потребуватиме сильно більше уваги до себе, ніж сучасні платформи. Але все ще залишається потужним інструментом для проєктного менеджменту.

Однак, пропоную розглянути одне з відгалужень, кожна з яких виправляє недоліки цієї платформи.

1.4.2 OpenProject

OpenProject - це відгалуження від Redmine, яке надає розширений функціонал, оновлений дизайн та безпосередню підтримку партнерам. Основна частина платформи залишається безкоштовною, однак підтримка та доповнення (add-ons) вже є платними послугами. На Рисунку 1.7 зображений візуальний інтерфейс застосунку.

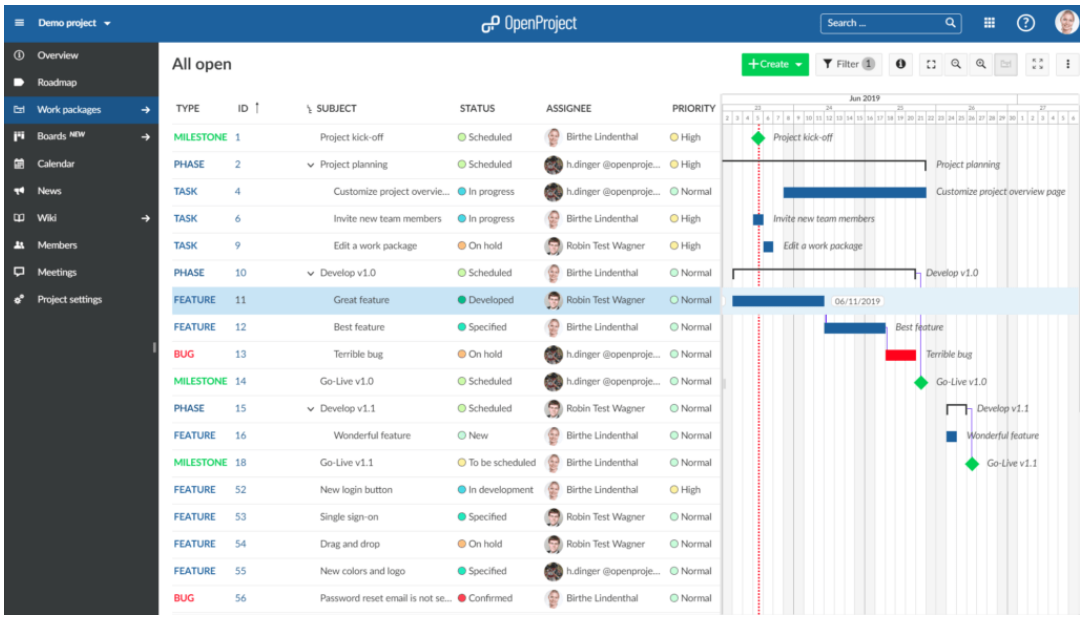


Рис. 1.7 Візуальний інтерфейс застосунку OpenProject

Переваги:

- Оновлений UI/UX. OpenProject пропонує досить строгий корпоративний сучасний дизайн.

- Можливість локального та хмарного розгортання. Зберігається можливість мати повний контроль над процесом розгортання на власному сервері, але також пропонується розгортання та підтримка на серверах команди OpenProject.

- Розширений функціонал. В стандартну версію додано більше функціоналу для аналітики та візуалізації даних.

- Підтримка зі сторони команди OpenProject. Команда проєкту надає постійну підтримку під час робочого тижня, щодо роботи з платформою. Міра підтримки визначається розміром компанії.

Недоліки:

- Платність. Спірний недолік, але варто зазначити, що більшість додаткового функціоналу та підтримка доступні лише у платній підписці.

OpenProject це ідеальне рішення для компаній, які зацікавлені в повному контролі над системою проєктного менеджменту, але не хоче навчати або шукати співробітника який буде займатися виключно підтримкою Redmine.

1.4.3 Asana

Asana[9] - це сучасний та один з найпопулярніших застосунків для командного менеджменту, який активно використовується як малими командами, так і великими організаціями. Серед головних переваг вирізняються простота кастомізації, висока функціональність для управління та аналізу, та низьким порогом входу для новачків. Крім цього. Asana іпдтримує широкі можливості інтеграції з іншими сервісами, що дозволяє адаптувати її під специфічні потреби команди чи проєкту.. На Рисунку 1.8 зображений візуальний інтерфейс застосунку.

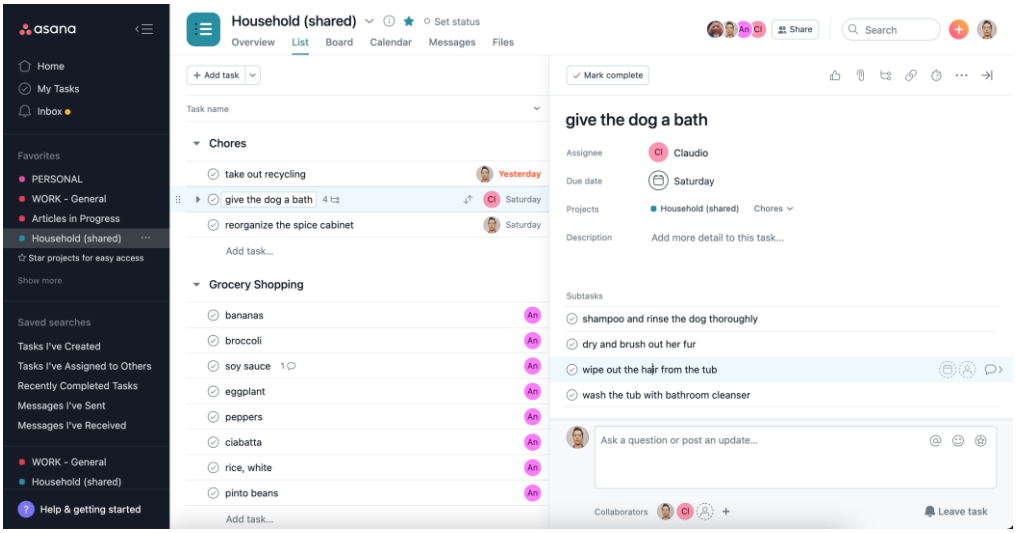


Рис. 1.8 Візуальний інтерфейс застосунку Asana

Переваги:

- Максимальна простота. UI/UX платформи вражає своєю простотою та інтуїтивністю.

- Легкість розуміння стану проєкту. Реалізовано багато різних способів відображень всієї необхідної інформації по проєкту, що допомагає у відслідковуванні прогресу. Також, є можливість виводу потрібних вам інформаційних вікон щодо окремих проєктів та всіх проєктів у цілому на домашню сторінку.

- Відкрите API. Asana створили, гарно документоване та покриваюче весь функціонал застосунку, API, яке дозволяє кастомізувати, автоматизувати

управління, або створювати шаблони та скрипти ініціалізації навіть для повністю нових проєктів. Також, відкрите API дозволяє іншим застосункам легко інтегруватися в систему Asana, через що її функціонал можна сильно розширювати.

Недоліки:

- Обмеженість у безкоштовному плані. Обмежуються дуже важливі інструменти для отримання аналітики по проєкту та можливість керуванням ролями та правами доступу. Ці функції є критичними.

- Вартість. У порівнянні з іншими застосунками, Asana вважається дорогою.

На мій погляд, Asana кращий вибір серед застосунків для проєктного та командного менеджменту будь якого рівня, якщо оцінювати за зручністю та функціоналом, але для стартапів та невеликих компаній вартість може бути зависокою.

1.5 Порівняння аналогів

Зведені результати аналізу характеристик розглянутих додатків наведено у таблиці 1.1.

Цей аналіз виявляє, що в окремих системах, необхідний функціонал вже існує, але відсутній аналог, який поєднує в собі всі його аспекти. Запропоноване рішення “NormalProject” реалізує всі необхідні можливості для керування нормативно регульованими проєктами.

Основними показниками важливими для проєкту, були визначені можливості шаблонізації проєктів, їх однорідність, можливості по контролю та кастомізації ролей та доступів, можливість контролю та відслідковуванню змін на проєкті та простота освоєння для нових учасників проєкту.

Саме ці частини є найважливішими для програмного забезпечення, яке спрямоване на керування нормативно регульованими проєктами.

Таблиця 1.1

Зведені результати аналізу характеристик додатків для керування проєктами

Показник	**Notion**	**Redmine**	**OpenProject**	**Asana**	**NormalProject**
Шаблонізація процесів	+, шаблони сторінок, автоматизація створення	-, лише через сторонні плагіни	+, шаблони сторінок	+, шаблони задач	+, шаблони проєктів і задач
Однорідність	-, високий рівень доступної кастомізації створює неоднорідність	+, повна однорідність всіх проєктів	+, повна однорідність всіх проєктів	+, повна однорідність всіх проєктів	+, повна однорідність в межах кожного окремого проєкту
Ролі та права доступу	-, базова система ролей	+, гнучка система ролей	+, гнучка система ролей	+, гнучка система ролей	+, гнучка система ролей
Контроль змін	+, але лише в межах кожної окремої сторінки	+, повний лог	+, повний лог	-, часткови й аудит	+, логування змін, аудит доступу
Простота освоєння	+, простий інтерфейс, низький поріг входу, але хаотична вкладеність	-, застарілий інтерфейс, висока вкладеність	-, висока вкладеність	+, сучасний інтерфейс, низька вкладеність	+, сучасний інтерфейс, низька вкладеність

2 ПРОЄКТУВАННЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Після проведення аналізу предметної галузі та огляду наявних аналогів, наступним ключовим етапом у процесі створення програмного забезпечення є формалізація вимог до системи. Саме чітке розуміння очікувань майбутніх користувачів та бізнес-цілей проєкту дозволяє визначити, яким має бути функціонал, інтерфейс та поведінка системи в різних умовах експлуатації.

Розробка вимог починається з ідентифікації основних категорій користувачів та їх потреб у межах взаємодії з системою. Це дозволяє сформулювати ключові сценарії використання (user stories), які згодом трансформуються у функціональні та нефункціональні вимоги. Задокументовані вимоги слугують основою для побудови архітектури застосунку, планування етапів розробки, а також забезпечують узгодженість між технічною реалізацією й очікуваннями зацікавлених сторін.

2.1 Визначення ролей та потреб користувачів

Організація ефективної колективної роботи у проєктній діяльності передбачає чітке визначення ролей учасників, розподіл відповідальностей і визначення їхніх потреб. У межах проєкту — незалежно від його галузі — завжди постає питання узгодженості дій, координації, підзвітності та забезпечення контролю якості виконання.

Для моделювання структури користувачів у програмному забезпеченні було здійснено узагальнення типових проєктних сценаріїв, зокрема в середовищі малих і середніх організацій, освітніх ініціатив, внутрішніх корпоративних процесів, а також у межах регламентованих процедур. У результаті аналізу було виокремлено чотири ключові ролі, що відображають базову ієрархію та логіку функціонування проєкту як структурної одиниці.

Керівник проєкту — центральна фігура у проєктній структурі, відповідальна за ініціацію, планування та координацію всіх етапів виконання. Ця особа формує склад команди, розподіляє завдання, встановлює дедлайни, а також здійснює моніторинг прогресу. Вона взаємодіє з усіма іншими учасниками проєкту та відповідає за відповідність результатів поставленим цілям. У контексті більш складних організацій може також виступати як посередник між вищим керівництвом або замовником і виконавцями.

Учасник проєкту (виконавець) — особа, яка безпосередньо реалізує завдання, поставлені в рамках проєкту. Учасники можуть мати різні спеціалізації та рівні залученості, але об'єднує їх активна участь у щоденній роботі над проєктом. У багатьох випадках саме від учасників залежить якість та своєчасність реалізації технічної чи інформаційної частини проєкту.

Спостерігач — роль, що передбачає непряму, але цілеспрямовану участь у проєкті. Це може бути ментор, ревізор, аудитор, представник замовника або інша зацікавлена сторона, яка не виконує завдань безпосередньо, але слідкує за прогресом, дотриманням стандартів і проміжними результатами. Присутність цієї ролі особливо важлива у формальних або регламентованих середовищах, де контроль є частиною вимог до проєкту.

Адміністратор організації — хоча ця роль не є частиною конкретного проєкту, вона має важливе значення в межах усієї системи керування. Адміністратор відповідає за загальну структуру, підтримку стандартів і методологій, які використовуються при запуску та реалізації проєктів. У певних організаціях це може бути методист, фахівець із забезпечення якості, координатор навчального процесу тощо. Його діяльність спрямована на забезпечення узгодженості та повторюваності процесів у межах усієї організації.

Варто наголосити, що такі ролі можуть реалізовуватись у різний спосіб залежно від масштабу й типу організації. В одних випадках одна особа може поєднувати кілька ролей (наприклад, бути і керівником, і виконавцем), в інших — навпаки, кожна роль може бути детально подрібнена на підролі. Проте

запропонований поділ охоплює базову логіку функціонування типової проєктної одиниці, яка є універсальною для більшості прикладних сценаріїв.

На основі цих ролей, були розроблені користувацькі історії (user stories), щоб покрити їх основні потреби. На Таблиці 2.1 наведено перелік головних потреб, важливих для початкової версії застосунку.

Таблиця 2.1

Основні користувацькі історії

Роль	**Користувацька історія**
Адміністратор	• Як адміністратор, я хочу створювати, редагувати та керувати шаблонами для проєктів. • Як адміністратор, я хочу переглядати каталог шаблонів. • Як адміністратор, я хочу створювати тимчасові копії шаблонів, щоб випускати цілісні оновлення. • Як адміністратор, я хочу дублювати шаблони, щоб не витрачати час на створення оновленого шаблону та не змінювати старі проєкти. • Як адміністратор, я хочу оновлювати шаблони та створювати нові версії. • Як адміністратор, я хочу керувати та створювати ролі користувачів. • Як адміністратор, я хочу бачити аудит дій користувачів. • Як адміністратор, я хочу редагувати загальну інформаційну сторінку, щоб публікувати загальні оголошення. • Як адміністратор, я хочу налаштувати SSO, щоб користувачі могли входити через корпоративний обліковий запис.

Продовження таблиці 2.1

Основні користувацькі історії

Роль	**Користувацька історія**
Адміністратор	• Як адміністратор, я хочу редагувати дозволи ролей, щоб забезпечити гнучкість у політиках доступу. • Як адміністратор, я хочу мати доступ до журналу змін шаблонів, щоб мати контроль над версіями. • Як адміністратор, я хочу отримувати звіти по проєктах, щоб аналізувати загальну продуктивність.
Керівник	• Як керівник, я хочу створювати проєкти за шаблонами. • Як керівник, я хочу залучати користувачів до проєкту. • Як керівник, я хочу створювати задачі, призначати виконавців і відслідковувати виконання. • Як керівник, я хочу оновлювати свій проєкт під нову версію шаблону. • Як керівник, я хочу переглядати аналітику та прогрес. • Як керівник, я хочу редагувати інформаційну сторінку проєкту, щоб підтримувати її актуальність. • Як керівник, я хочу коментувати завдання, щоб пояснювати вимоги або давати фідбек. • Як керівник, я хочу переглядати загальні оголошення, щоб бути в курсі оновлень на платформі. • Як керівник, я хочу переглядати права учасників проєкту, щоб краще орієнтуватися в задачах проєкту.

Продовження таблиці 2.1

Основні користувацькі історії

Роль	**Користувацька історія**
Керівник	• Як керівник, я хочу отримувати сповіщення про зміни в задачах, щоб швидко реагувати. • Як керівник, я хочу отримувати нагадування про наближення дедлайнів, щоб контролювати терміни. • Як керівник, я хочу бачити історію змін у проєкті, щоб відслідковувати динаміку. • Як керівник, я хочу мати змогу відновити попередню версію проєкту, якщо була помилка в останніх змінах. • Як керівник, я хочу експортувати звіти для презентації керівництву або зовнішнім сторонам.
Виконавець	• Як виконавець, я хочу бачити задачі, призначені мені, та відмічати виконання. • Як виконавець, я хочу залишати коментарі під задачами, щоб ставити уточнення або ділитися статусом. • Як виконавець, я хочу переглядати проєкт, до якого мене залучено. • Як виконавець, я хочу переглядати інформаційну сторінку проєкту, щоб мати доступ до важливих документів і посилань. • Як виконавець, я хочу бачити загальні оголошення, щоб не пропускати важливу інформацію.

Продовження таблиці 2.1

Основні користувацькі історії

Роль	**Користувацька історія**
Виконавець	• Як виконавець, я хочу отримувати сповіщення, коли мені призначили задачу або її оновили. • Як виконавець, я хочу бачити нагадування про дедлайни, щоб не забути про виконання задач.
Спостерігач	• Як спостерігач, я хочу переглядати всі проєкти та шаблони. • Як спостерігач, я хочу залишати коментарі з зауваженнями. • Як наглядач, я хочу переглядати інформаційну сторінку, щоб оцінювати відповідність проєкту до вимог. • Як наглядач, я хочу мати доступ до перегляду історії дій в проєкті, щоб забезпечити дотримання правил.

Далі, для спрощеного групування та виділення основних компонентів системи, я застосував підхід який використовують в Agile методологіях підходу до розробки по розділенню задач[10], що зображено на Рисунку 2.1. Основними компонентами такого підходу, є Користувацькі історії (User stories) – короткі вимоги або запити, написані з точки зору користувача, Епіки (Epics) – великі частини роботи, які можна розбити на низку менших задач, та Ініціативи (Initiatives) – колекції епіків, які рухають проєкт до певної мети. Такий підхід до розділення проєкту, дозволяє розбити проєкт на менші, цілісні та змістовні елементи, що спрощує розуміння системи в цілому та допомагає визначити пріоритети.

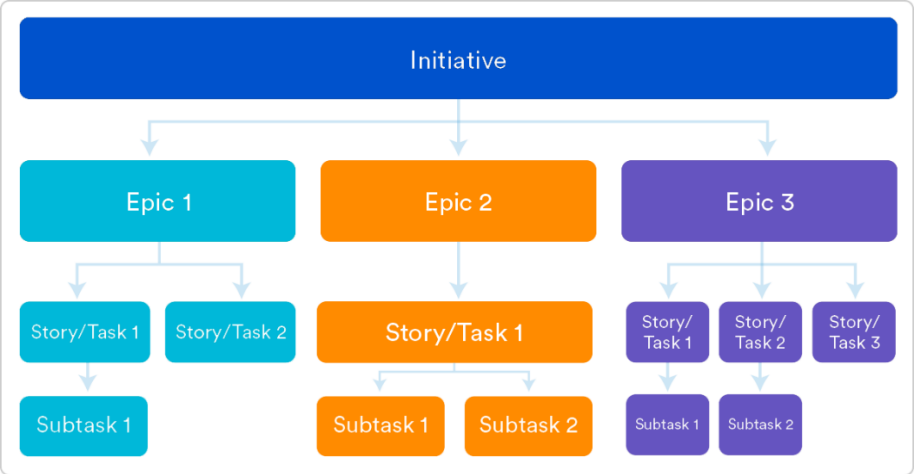


Рис. 2.1 Схема структури компонентів задачі в Agile методології

Таблиця 2.2

Ініціативи та Епіки

Ініціатива	**Епік**
Шаблонізація проєктів	• Створення шаблонів. • Створення проєктів за шаблонами. • Оновлення шаблонів та проєктів.
Керування проєктами	• Управління проєктами. • Управління ролями. • Управління задачами.
Комунікація	• Інформаційна сторінка проєкту. • Спілкування між менеджером та виконавцем. • Загальна інформаційна сторінка.
Безпека та аутентифікація	• Аутентифікація (JWT, SSO). • Управління доступами та дозволами. • Аудит та логування дій користувачів.
Нотифікація	• Центр повідомлень та сповіщень. • Сповіщення про зміни в проєкті. • Сповіщення про дедлайни.

Продовження таблиці 2.2

Ініціативи та Епіки

Ініціатива	**Епік**
Версіонування	• Відстеження змін у шаблонах та проєктах. • Перегляд історії редагування. • Відновлення попередніх версій.
Аналітика та звітність	• Перегляд статистики проєкту. • Візуалізація прогресу. • Генерація звітів.

Однак, в сучасних підходах до розробки, хоча і розуміються всі потреби користувачів, але для зменшення ризиків використовують прототипування та визначення MVP проєкту. В нашому випадку, основним MVP проєкту є Керування проєктами та процесами, Шаблонування, Керування ролями та доступом та Аудит дій.[11]

2.2 Функціональні та нефункціональні вимоги

Функціональні вимоги описують поведінку системи з точки зору очікуваної взаємодії з користувачем. Вони були сформовані на основі визначених ролей та користувацьких історій. Основні функціональні вимоги до програмного забезпечення “NormalProject” згруповані за ключовими функціональними напрямами:

Шаблонування:

- Можливість створення та використання шаблонів для всього проєкту, окремих сторінок і задач.
- Підтримка повторного використання структур, заповнення даних за шаблоном та збереження шаблонів в бібліотеці організації.

Керування ролями та доступом:

- Розмежування прав доступу на основі ролей (наприклад, адміністратор організації, керівник проєкту, виконавець, спостерігач).
- Контроль доступу до функцій створення, редагування, перегляду шаблонів та задач.
- Можливість обмеження редагування окремих полів або блоків даних.

Управління проєктами та процесами:

- Призначення відповідальних осіб за задачі та підпроцеси.
- Встановлення дедлайнів, пріоритетів та етапів виконання.
- Підтримка нормативних обмежень, передбачених регульованими процесами.

Аудит дій:

- Повне логування дій користувачів, включаючи створення, зміну, видалення та перегляд елементів.
- Перегляд історії змін для кожної задачі або шаблону.

Нефункціональні вимоги визначають характеристики програмного забезпечення, що впливають на його продуктивність, масштабованість, безпеку та зручність використання. Вони забезпечують відповідність системи очікуванням замовника, користувачів і стандартам розробки.

Нефункціональні вимоги:

Безпека:

- Шифрування всіх запитів за допомогою HTTPS.
- Системне логування та аудит дій користувачів з можливістю фільтрації подій за часом, типом та користувачем.

Зручність використання (UI/UX):

- Адаптивний інтерфейс, оптимізований під різні розміри екранів (десктоп, планшет, смартфон).
- Мінімальна вкладеність елементів для забезпечення інтуїтивного навігаційного досвіду.

- Підтримка зрозумілої структури навігації, інтерактивних підказок та візуального позначення статусів задач.

Масштабованість і підтримуваність:

- Компонентна архітектура, що дозволяє незалежне масштабування основних сервісів.
- Використання контейнеризації (Docker) для ізольованого розгортання.

Таким чином, сформульовані функціональні та нефункціональні вимоги визначають основні можливості та характеристики програмного забезпечення “NormalProject”. Вони охоплюють ключові аспекти — від гнучкої роботи з шаблонами, детального управління ролями та проєктами, до забезпечення безпеки, зручності інтерфейсу та масштабованості системи. Такий підхід дозволяє створити рішення, що відповідає потребам нормативно регульованого середовища, забезпечує прозорість процесів і зручність для різних категорій користувачів.

3 ЗАСОБИ РЕАЛІЗАЦІЇ

У межах реалізації програмного забезпечення для керування нормативно регульованими проєктами було обрано низку сучасних інструментів, фреймворків та бібліотек, які дозволяють забезпечити високу продуктивність, масштабованість і зручність у розробці. Основу серверної частини скаладає стек на базі Node.js із використанням TypeScript та фреймворку NestJS, а клієнтська частина реалізована за допомогою React.js і Tailwind CSS. Для зберігання даних використовується нереляційна база даних MongoDB, а для контейнеризації та розгортання – Docker.

У цьому розділі буде проведено огляд обраних технологій та їх альтернатив.

3.1 Огляд технологій для серверної частини програмного забезпечення

3.1.1 Серверне середовище виконання

Одним з ключових рішень під час побудови серверної частини програмного забезпечення є вибір середовища виконання – платформи, яка дозволяє обробляти запити клієнта, керувати логікою застосунку та взаємодіяти з базою даних. На сьогоднішній день існує низка перевірених та широко використовуваних серверних середовищ, кожне з яких має свої переваги та підходить до різних сценаріїв використання.

Node.js є середовищем виконання JavaScript, що дозволяє запускати код поза межами браузера. Воно побудоване на основі рушія V8 від Google і підтримує неблокуючу модель вводу/виводу, що робить його ефективним при розробці великої кількості одночасних підключень.[12] Цикл подій зображено на Рисунку 3.1 Node.js активно використовується у веброзробці, особливо у випадках коли потрібна висока швидкість обміну даними, наприклад у чатах, стримінгу або REST API. Платформа має розвинену екосистему пакетів через менеджер npm та підтримує інтеграцію з TypeScript, що дозволяє підвищити надійність коду завдяки статичній типізації.

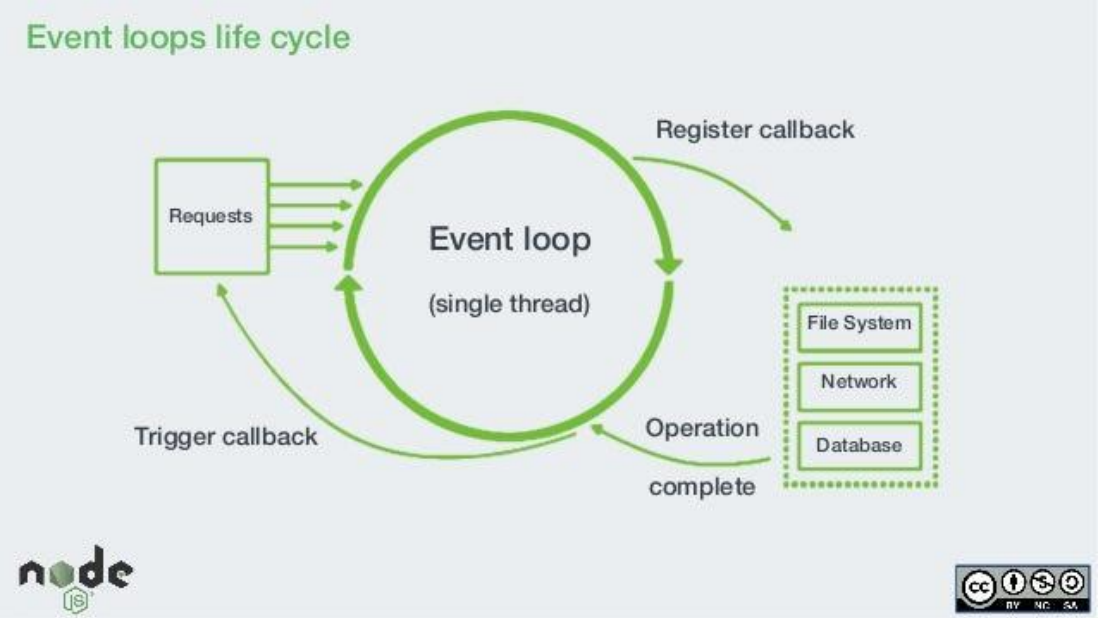


Рис. 3.1 Цикл подій в Node.js

Python є популярною мовою програмування загального призначення, яка широко застосовується і в веброзробці. Для побудови серверної частини застосунків використовують, зокрема, фреймворки Django та Flask. Django орієнтований на швидку розробку складних вебзастосунків і пропонує багато функціоналу "з коробки": ORM, аутентифікацію, панель адміністратора. Flask, навпаки, є мікрофреймворком, що дозволяє збудувати більш легку архітектуру з можливістю поступового додавання компонентів. Python-програми добре читаються, мають багату документацію та активну спільноту.

Java залишається основою для розробки масштабованих корпоративних застосунків. Одним із найпоширеніших фреймворків є Spring Boot, який дозволяє спростити конфігурацію та швидко запускати вебзастосунки. Java відома стабільністю, багатопотоковістю, інструментами для безпеки та можливістю реалізації складної бізнес-логіки.[13] Завдяки JVM (віртуальній машині Java), застосунки можуть працювати на різних платформах без змін у коді. Також існує розвинена інфраструктура для тестування, розгортання та моніторингу Java-застосунків.

Go — це мова програмування від Google, яка орієнтована на високу продуктивність, просту синтаксисну модель і легку паралелізацію. Вона має

вбудовані засоби для роботи з HTTP, що дозволяє створювати швидкі вебсервери з мінімальною залежністю від зовнішніх бібліотек. Go часто обирають для побудови мікросервісів та систем, які мають обробляти велику кількість запитів при мінімальній затримці. Попри це, екосистема Go менш орієнтована на швидку веброзробку в порівнянні з JavaScript чи Python.

PHP вже давно є частиною веброзробки, і хоча сьогодні він менш популярний серед нових проєктів, сучасні фреймворки на кшталт Laravel забезпечують високу продуктивність і приємний досвід для розробника. Laravel пропонує шаблони, ORM, маршрутизацію, аутентифікацію та багато іншого. Він добре підходить для класичних CRUD-застосунків, але має нижчий поріг масштабованості в порівнянні з іншими сучасними платформами.

Node.js було обрано як основне серверне середовище завдяки його неблокуючій архітектурі, яка забезпечує ефективну обробку паралельних запитів і дозволяє досягти високої продуктивності та масштабованості для сучасних веб-додатків. Значною перевагою є широка підтримка в індустрії та наявність великої кількості бібліотек, що значно прискорює розробку та спрощує інтеграцію з іншими технологіями. Крім того, можливість використовувати одну мову програмування (JavaScript або TypeScript) як на клієнті, так і на сервері, спрощує логіку взаємодії, обмін типами даних і сприяє кращому взаєморозумінню між різними частинами системи. Це робить Node.js оптимальним вибором для створення сучасних, масштабованих і продуктивних веб-рішень.

3.1.2 Мова програмування

Середовище Node.js підтримує виконання JavaScript-коду, що зумовлює використання відповідних мов програмування. Проте навіть у цьому контексті існує вибір між кількома підходами до організації коду — як базовий JavaScript, так і надмножини, що розширюють його можливості.

JavaScript є основною мовою для роботи з Node.js. Вона є динамічно типізованою, має величезну екосистему бібліотек і підтримується усіма сучасними інструментами розробки. JavaScript дозволяє швидко створювати вебзастосунки,

однак його динамічна типізація може призводити до помилок, які виявляються лише під час виконання. З іншого боку, саме ця гнучкість дозволяє розробникам швидко будувати прототипи та виконувати ітеративну розробку.

TypeScript — це надмножина JavaScript, що представлено на Рисунку 3.2, яка додає статичну типізацію, підтримку інтерфейсів, класів та строгішу перевірку коду на етапі компіляції. Це особливо корисно у великих проєктах, де важливо гарантувати відповідність структури даних, а також полегшити командну розробку за рахунок чіткої документації через типи.[14] TypeScript підтримується всіма основними бібліотеками Node.js, легко компілюється у звичайний JavaScript, а також інтегрується з системами зборки коду та редакторами. Його використання значно підвищує надійність та масштабованість серверної частини.

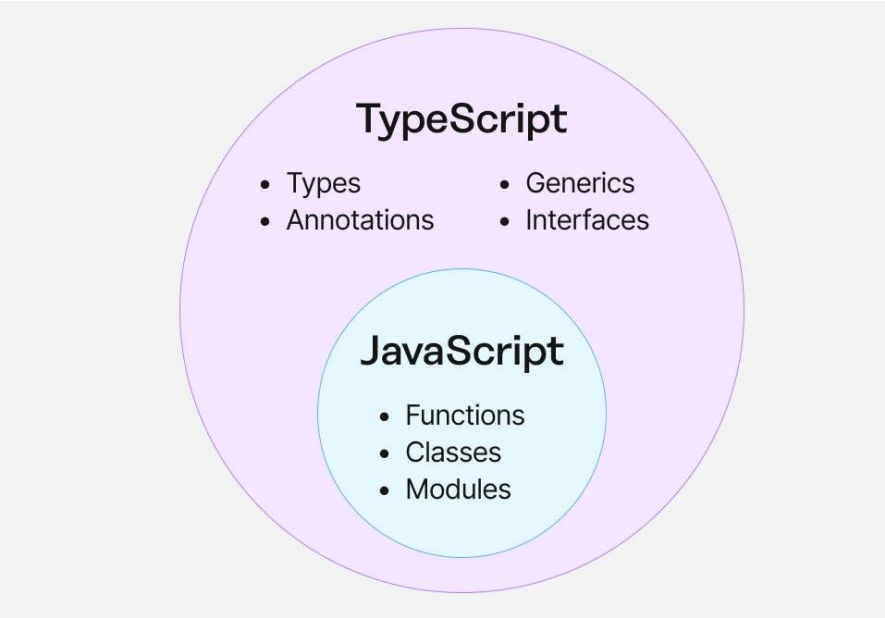


Рис. 3.2 Візуальне представлення надмножити TypeScript

У межах обраного середовища Node.js було прийнято рішення використовувати TypeScript як основну мову програмування для серверної частини. Його підтримка статичної типізації дозволяє виявляти помилки на ранніх етапах розробки, покращує читаємість та підтримуваність коду, а також полегшує командну роботу над проєктом. Завдяки тому, що TypeScript є надмножиною JavaScript, розробники мають доступ до всієї екосистеми npm-пакетів без втрати

сумісності. Використання TypeScript є оптимальним вибором для побудови масштабованого та стабільного серверного застосунку на базі Node.js.

3.1.3 Серверний фреймворк

У межах середовища Node.js існує велика кількість фреймворків, які полегшують розробку серверної логіки, організовують структуру проєкту та забезпечують підтримку шаблонів архітектури. Вибір фреймворку визначає, наскільки легко масштабувати, тестувати та підтримувати програмне забезпечення у майбутньому. У цьому підрозділі розглянемо найбільш популярні рішення для побудови серверної частини на Node.js.

Express.js — це мінімалістичний і гнучкий веб-фреймворк для Node.js, який забезпечує базові можливості для створення серверів, маршрутизації запитів і обробки HTTP. Він має невеликий розмір, надзвичайно велику спільноту, простий API та величезну кількість плагінів. Його гнучкість дозволяє будувати будь-яку архітектуру, але водночас вимагає від розробника самостійно приймати багато рішень щодо структури застосунку, розподілу залежностей, обробки помилок, тощо. Express чудово підходить для невеликих або середніх проєктів, де потрібна швидка реалізація.

Fastify — це сучасний високопродуктивний фреймворк для Node.js, який фокусується на швидкості, мінімізації накладних витрат і підтримці плагінної архітектури. Він підтримує схожий на Express синтаксис, однак забезпечує значно кращу продуктивність. Fastify має вбудовану валідацію схем запитів і відповідей через JSON Schema, що підвищує безпеку API. Завдяки меншій популярності порівняно з Express, кількість готових прикладів та рішень у спільноті обмеженіша.

NestJS — це прогресивний фреймворк для створення масштабованих серверних застосунків на базі Node.js і TypeScript.[15] Він використовує модульну архітектуру, підтримує об’єктно-орієнтоване програмування, інверсію контролю, декоратори, а також шаблони, подібні до Angular. NestJS інтегрується з Express або Fastify на низькому рівні, дозволяючи розробникам скористатися перевагами цих фреймворків, не відмовляючись від зручної структури. Крім того, NestJS має

підтримку мікросервісів, GraphQL, WebSocket, а також зручні механізми для тестування, логування та обробки помилок.[16] Він є особливо ефективним для командної роботи над середніми і великими проєктами.

Зважаючи на вимоги до масштабованості, чіткості архітектури та зручності підтримки, для реалізації серверної частини було обрано NestJS. Його модульна структура і сувора підтримка TypeScript дозволяють створювати добре структурований код, який легко підтримувати й масштабувати. Завдяки схожості з Angular, фреймворк забезпечує швидку адаптацію для розробників із досвідом фронтенду, а також надає багаті можливості інтеграції з базами даних, зовнішніми сервісами та внутрішніми модулями. NestJS є логічним вибором для створення складного серверного застосунку з чітко організованими ролями, API та бізнес-логікою.

3.1.4 Системи управління базами даних

База даних є фундаментальним компонентом будь-якого сучасного вебзастосунку. Її вибір залежить від характеру даних, вимог до узгодженості, масштабування, швидкості доступу, підтримки складних зв'язків, а також — гнучкості структури. У контексті серверного застосунку на базі Node.js розглянемо найпоширеніші рішення, які добре інтегруються з цим середовищем.

PostgreSQL — це потужна об’єктно-реляційна СУБД, яка підтримує ACID-транзакції[17], складну реляційну логіку, збережені процедури на PL/pgSQL, розширену типізацію (наприклад, JSONB, ENUM, масиви), CTE (common table expressions), індексацію GIN/GiST, window-функції та інші можливості для реалізації складної бізнес-логіки.[18]

Окремої уваги заслуговує підтримка формату JSONB — бінарного представлення JSON, який дозволяє зберігати напівструктуровані дані. Це робить PostgreSQL частково придатним для сценаріїв, притаманних NoSQL системам. Однак при цьому PostgreSQL все ще зберігає реляційну природу: немає можливості встановлювати вкладені посилання між JSONB-об'єктами на рівні бази (наприклад, аналогів $lookup у MongoDB), а типізація та індексація вкладених структур

обмежена. Робота з глибоко вкладеними структурами або частими змінами схеми в JSONB є складнішою, ніж у нативних документно-орієнтованих СУБД.

Інтеграція з Node.js здійснюється через драйвер pg або ORM-рішення, такі як Prisma, TypeORM, Objection.js, що забезпечують абстракцію над SQL-запитами та покращують читабельність коду.

MySQL — одна з найвідоміших реляційних СУБД із відкритим кодом, яка підтримує базовий набір SQL-функціональності, транзакції (через InnoDB), реплікацію, тригери, представлення (views) тощо. MariaDB є її вдосконаленим форком, орієнтованим на відкритість і швидкість розвитку.

У контексті Node.js ці бази даних підтримуються численними ORM та query builder'ами (наприклад, Sequelize, Prisma, Knex.js), що робить їх зручними для швидкого старту. Однак, порівняно з PostgreSQL, вони мають дещо обмеженішу підтримку складних типів даних і менш виразну аналітичну потужність.

MongoDB — це документно-орієнтована СУБД, яка працює за принципом зберігання об’єктів у вигляді BSON (Binary JSON). Вона не потребує попереднього визначення жорсткої схеми — структура кожного документа може змінюватися динамічно.[19] Це ідеально підходить для систем, де моделі даних змінюються з часом, або в яких існує велика кількість сутностей із непостійною структурою (наприклад, системи із шаблонами, розширюваними формами, гнучкими правами доступу).

MongoDB підтримує вкладені документи, масиви, посилання між колекціями через агрегацію ($lookup), а також має потужну мову запитів для фільтрації та обробки даних. Хоча вона не гарантує жорстку узгодженість між пов’язаними об'єктами, використання транзакцій (з версії 4.0+) дозволяє забезпечити атомарність навіть при кількох записах. Крім того, MongoDB масштабована горизонтально (через шардінг), що вигідно при роботі з великим обсягом нереляційних даних.

Для Node.js існує офіційний драйвер, а також ODM-бібліотека Mongoose, яка дозволяє створювати схеми з валідацією, типами, middleware-підтримкою,

індексацією тощо — що робить MongoDB зручнішою в керуванні у великих застосунках.

З огляду на специфіку проєкту — розробку системи керування проєктами з можливістю роботи з гнучкими шаблонами, структурованими правами доступу, динамічними ролями та формами — було обрано MongoDB. Гнучкість структури даних дозволяє швидко адаптуватися до змін у логіці застосунку без потреби складних міграцій. Вбудована підтримка вкладених документів, масивів і запитів до пов’язаних даних через $lookup забезпечує достатню функціональність для більшості зв’язків, при цьому зберігаючи масштабованість і продуктивність.

Крім того, MongoDB легко інтегрується з NestJS через Mongoose, що забезпечує декларативний підхід до опису моделей і валідації. Підтримка транзакцій, індексації та реплікації робить її безпечною й надійною для використання у виробничих середовищах, включно з нормативно-регульованими системами, де потрібна стабільність збереження даних

3.1.5 Контейнеризація та розгортання

У сучасній розробці програмного забезпечення важливу роль відіграють засоби контейнеризації та автоматизації розгортання. Вони дозволяють створювати стандартизовані та відтворювані середовища для розробки, тестування і продакшн, що позитивно впливає на гнучкість, масштабованість і підтримку застосунку.

Docker — це платформа, яка дозволяє упаковувати застосунок разом із залежностями у легкі, ізольовані контейнери. Контейнери працюють однаково у будь-якому середовищі — від локальної машини до хмарних серверів. Docker забезпечує ізоляцію середовища, відтворюваність, гнучке масштабування та підтримку різних операційних систем. Для стеку Node.js/NestJS і React/Tailwind Docker дозволяє розділити клієнтську і серверну частини у окремі контейнери, що спрощує розробку, тестування та деплоймент.

Серед альтернатив Docker можна виділити Podman, який сумісний із Docker CLI, але не потребує окремого демонського процесу, що підвищує безпеку. LXC/LXD — це системні контейнери, які ближчі до віртуальних машин і

забезпечують більшу ізоляцію, але складніші у налаштуванні. Віртуальні машини на кшталт VMware, Hyper-V чи VirtualBox дають глибшу ізоляцію, але споживають більше ресурсів і менш зручні для швидких розгортань. Також існують serverless-платформи, як-от AWS Lambda чи Azure Functions, які дозволяють запускати код без управління сервером, але мають певні обмеження.

Docker був обраний через його популярність, розвинену екосистему, простоту налаштування і легку інтеграцію з CI/CD. Він дозволяє швидко переходити від локальної розробки до тестування і продакшн без зміни конфігурацій. Крім того, Docker підтримується багатьма хмарними провайдерами та оркестраторами, що забезпечує масштабованість проєкту. Для проєктів, де важлива відтворюваність середовищ і контроль версій, Docker є оптимальним вибором.

3.2 Огляд технологій для користувацької частини програмного забезпечення

3.2.1 Бібліотеки та фреймворки для побудови інтерфейсу користувача

Побудова клієнтського інтерфейсу сучасного вебзастосунку зазвичай базується на використанні фреймворків або бібліотек JavaScript, які спрощують процес створення інтерактивних, реактивних та масштабованих інтерфейсів. Серед найбільш популярних інструментів у цій категорії — React, Vue.js, Angular та Svelte.

React.js — це бібліотека для створення користувацьких інтерфейсів, що розробляється та підтримується компанією Meta.[20] Вона базується на концепції компонентів, що дозволяє розбивати інтерфейс на незалежні модулі з власною логікою та станом. React широко підтримується спільнотою, має гнучку екосистему і дозволяє використовувати різні підходи до управління станом та маршрутизації. Його віртуальний DOM забезпечує ефективне оновлення елементів інтерфейсу.

Vue.js — прогресивний фреймворк для побудови інтерфейсів, який фокусується на простоті використання та плавному навчанні. Vue має вбудовані інструменти для маршрутизації, управління станом та побудови компонентів, а також підтримує реактивність “із коробки”. Його архітектура дозволяє поступово інтегруватися в проєкти різної складності, а синтаксис близький до класичного HTML/CSS, що знижує поріг входу.

Angular — повноцінний фреймворк від Google для розробки SPA (single-page applications), що включає рішення для маршрутизації, форм, HTTP-запитів, тестування, DI (ін’єкції залежностей) та ін. Angular використовує TypeScript як основну мову та суворо визначену архітектуру, що підходить для великих проєктів. Його підхід може вимагати більше часу на вивчення, але забезпечує високу структурованість і стабільність коду.

Svelte — сучасний компілятор, який перетворює декларативні компоненти в оптимізований JavaScript-код на етапі збірки. Завдяки цьому Svelte не потребує віртуального DOM і має високу продуктивність у рендерінгу. Він має мінімалістичний синтаксис і просту реактивну модель, але меншу екосистему порівняно з іншими фреймворками.

Для реалізації клієнтської частини програмного забезпечення було обрано React.js. Такий вибір зумовлений гнучкістю цього фреймворку, адже він дозволяє самостійно обирати інструменти для маршрутизації, управління станом та стилізації, що дає змогу точково налаштовувати архітектуру застосунку. Компонентний підхід React забезпечує повторне використання коду, полегшує тестування та підтримку масштабованих інтерфейсів. Крім того, широка екосистема, потужна спільнота, розвинена документація та підтримка великими компаніями гарантують довгострокову стабільність проєкту. Важливо й те, що React добре поєднується з Tailwind CSS для стилізації та легко інтегрується із сучасними бекенд-рішеннями через REST API або GraphQL. Усе це робить React.js оптимальним вибором для розробки клієнтської частини програмного забезпечення.

3.2.2 Системи стилізації інтерфейсу

Сучасні вебзастосунки вимагають не лише функціональності, але й зручного, адаптивного та естетично привабливого інтерфейсу. Для цього використовуються різноманітні підходи та інструменти стилізації — від класичного CSS до утилітарних фреймворків. Найбільш поширеними є CSS-препроцесори (Sass, Less), UI-фреймворки (Bootstrap, Material UI, Ant Design), а також утилітарні CSS-фреймворки (Tailwind CSS).[21]

Bootstrap — один із найстаріших і найпопулярніших фреймворків для швидкої розробки адаптивного інтерфейсу. Він пропонує набір готових компонентів (кнопок, форм, сіток, модалок) та використовує класичну сітку на Flexbox. Bootstrap добре підходить для прототипування, однак може обмежувати унікальність дизайну через типовий "bootstrap-стиль".

Material UI (MUI) — реалізація дизайну Google Material Design для React. Забезпечує велику кількість компонентів, які відповідають загальним гайдлайнам дизайну Google. Перевагою MUI є глибока інтеграція з React та можливість кастомізації тем. Однак, як і Bootstrap, він нав’язує певну стилістичну концепцію, що може бути обмеженням у створенні унікального вигляду.

Ant Design — бібліотека компонентів, орієнтована переважно на розробку бізнес-застосунків. Має великий набір елементів інтерфейсу, розроблений за принципами дизайну Alibaba. Вона підтримує локалізацію, темізацію та інтегрується з багатьма React-інструментами. Водночас, компоненти мають виразну стилізацію, що також потребує значних зусиль для зміни стандартного вигляду.

Tailwind CSS — утилітарний CSS-фреймворк, що базується на концепції використання набору низькорівневих класів для стилізації елементів. Tailwind не містить готових UI-компонентів, але дозволяє швидко створювати інтерфейси з повним контролем над їхнім виглядом. Він добре масштабується, підтримує адаптивність, dark mode, кастомізацію теми і інтегрується з такими інструментами, як PostCSS, Headless UI тощо.

Sass / Less — CSS-препроцесори, які дозволяють використовувати змінні, вкладеність, міксіни та інші конструкції, що полегшують підтримку стилів у великих проєктах. Вони часто застосовуються як доповнення до інших фреймворків, особливо у випадках повністю кастомного дизайну без сторонніх бібліотек.

У цьому проєкті було прийнято рішення використовувати Tailwind CSS як основний інструмент стилізації. Такий вибір зумовлений гнучкістю дизайну, оскільки Tailwind не нав’язує готових стилів і дозволяє створювати унікальний вигляд без необхідності переписувати існуючі стилі. Крім того, утилітарний підхід цієї бібліотеки сприяє підвищенню продуктивності розробки, оскільки стилі можна створювати безпосередньо в JSX-компонентах, що зменшує кількість контекстних перемикань і прискорює роботу. Tailwind також забезпечує масштабованість і зручну підтримку темізації, включаючи адаптивність і темну тему без складних конфігурацій. Окремо варто відзначити ідеальну інтеграцію з React, що дозволяє швидко створювати гнучкі та легко настроювані UI-компоненти.

4 ПРОЄКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Архітектура програмного забезпечення

Архітектура програмного забезпечення вебзастосунку “NormalProject” побудована за принципами багаторівневої клієнт-серверної моделі, з чітким поділом відповідальностей між фронтендом, бекендом та базою даних, що зображено на Рисунку 4.1. Такий підхід забезпечує гнучкість, масштабованість і розширюваність системи, що є критично важливим у контексті підтримки великої кількості користувачів, проєктів та шаблонів. Застосунок орієнтований на корпоративне використання, з можливістю адаптації до організацій з регламентованими процесами, що зумовлює додаткові вимоги до структури даних, контролю доступу та аудиту дій.

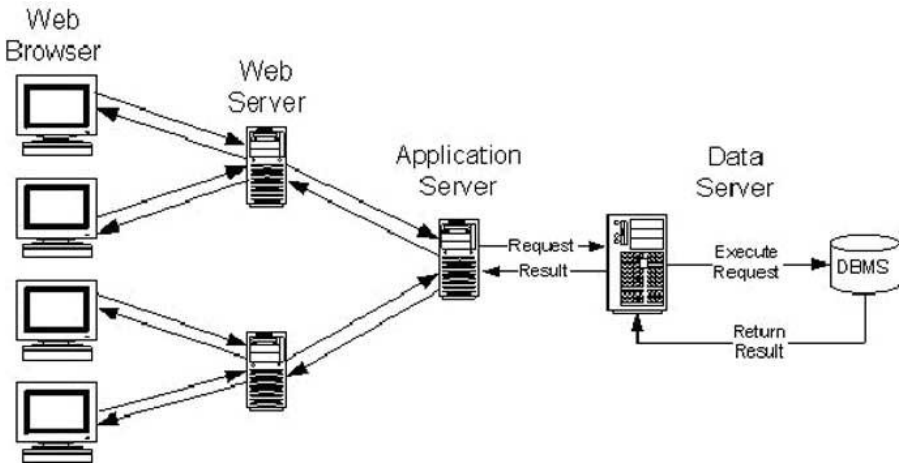


Рисунок 4.1 Схема роботи Клієнт-серверної архітектури

Фронтенд-частина реалізована як односторінковий застосунок (SPA) з використанням React.js. Це дозволяє досягти високої продуктивності при взаємодії користувача з інтерфейсом завдяки динамічному оновленню DOM та взаємодії з API без повного перезавантаження сторінки. Для стилізації застосовується Tailwind CSS — утилітарний CSS-фреймворк, який спрощує створення адаптивного та

послідовного інтерфейсу. Фронтенд взаємодіє з бекендом через RESTful API, що дозволяє чітко відділити клієнтську та серверну логіку.

Бекенд реалізовано з використанням фреймворку NestJS — прогресивного серверного фреймворку для Node.js, що базується на TypeScript та підтримує модульну архітектуру. Використання NestJS дозволяє структурувати код за допомогою чітко визначених модулів, сервісів, контролерів і middleware, що сприяє повторному використанню компонентів, зручному тестуванню та підтримці. Основні функції бекенду включають обробку запитів від клієнта, перевірку автентичності та авторизації, управління шаблонами, задачами, користувачами, ролями, а також логування дій.

Збереження даних відбувається у MongoDB — документо-орієнтованій базі даних, яка забезпечує гнучке зберігання складно структурованих об'єктів у форматі BSON. MongoDB є оптимальним рішенням у випадках, коли структура об’єктів може змінюватися з часом або мати глибоко вкладені зв’язки, що є типовим для системи шаблонів, реалізованої в “NormalProject”. Завдяки цьому можливо зберігати шаблони з ієрархічною структурою полів, метаданих та зв’язків між задачами у межах одного документа, що спрощує запити та зменшує навантаження на систему.

Комунікація між фронтендом і бекендом забезпечується через HTTP-запити з використанням протоколу HTTPS для забезпечення конфіденційності та цілісності даних. Для автентифікації користувачів використовується механізм JSON Web Token (JWT), що дозволяє захищати маршрути і забезпечувати доступ до ресурсів відповідно до ролі користувача.

4.2 Структури даних та бази даних

Для ефективного зберігання та обробки інформації у вебзастосунку “NormalProject” було розроблено систему логічних структур даних, які відображають ключові сутності системи та їхні взаємозв’язки. Уся інформація зберігається в документо-орієнтованій базі даних MongoDB, що дозволяє гнучко

оперувати вкладеними структурами, адаптуватися до змін бізнес-вимог і мінімізувати надлишкові зв’язки.

Одна з ключових функцій системи — це робота з шаблонами, що дозволяють уніфікувати та стандартизувати введення даних у проєктах. Основними сутностями цього модуля є:

- Template — об'єкт, який містить визначення шаблону, включаючи назву, тип, версію, автора, дату створення та оновлення. Ключовим атрибутом є структура шаблону — масив полів, що описують очікувані дані.
- Field — структура, яка описує окреме поле шаблону: назву, тип (наприклад, текст, число, дата), обов’язковість, доступність лише для читання та опис. Ці поля використовуються як метамодель для форм.
- TemplateInstance — екземпляр шаблону, який застосовується у конкретному проєкті. Містить посилання на шаблон, ідентифікатор проєкту, автора створення та словник значень полів (у форматі ключ-значення).
- Project — сутність, яка об’єднує користувачів, шаблони та робочі процеси. Вона містить загальні атрибути, такі як назва, опис, автор, дати створення та оновлення, а також статус проєкту.

Ці сутності пов’язані між собою однозначними зв’язками: кожен проєкт використовує певний шаблон, кожен екземпляр шаблону належить конкретному проєкту та шаблону, а шаблон, у свою чергу, містить набір полів.

Враховуючи орієнтацію застосунку на командну роботу, важливим аспектом є контроль доступу до інформації. Для цього реалізовано гнучку систему ролей і прав:

- User — основна сутність, що представляє користувача системи. Містить атрибути автентифікації (електронну пошту, пароль, токени), а також посилання на ролі в межах проєктів та організацій.

- Organization — об'єднання користувачів, яке є основним простором для створення проєктів. Організація має власного адміністратора, що керує доступом та ролями в межах організації.
- Role — абстракція, яка задає набір дозволів (наприклад, створення задач, редагування шаблонів, перегляд звітів). Кожен користувач отримує ролі в контексті конкретної організації або проєкту.
- Permission — базова одиниця доступу, що описує право на виконання певної дії. Система дозволяє як попередньо визначені ролі (адміністратор, виконавець, спостерігач), так і конфігуровані, створені адміністратором.

Прив’язка користувачів до проєктів здійснюється через рольові зв’язки. Це забезпечує гнучкість у керуванні доступом та дозволяє точно обмежити можливості учасників відповідно до їхніх функціональних обов’язків.

Керування проєктами базується на наборі сутностей, які визначають основні елементи життєвого циклу задач та взаємодій у команді:

- Task (Задача) — ключова одиниця роботи в проєкті. Включає опис, виконавця, терміни, статус, пріоритет, зв'язок із шаблоном (якщо задача базується на конкретному типі даних).
- Stage / Workflow — опціональна структура для проєктів із фіксованими етапами виконання. Дозволяє впровадити канбан-підхід або лінійні процеси.
- Comment / Log — підтримують ведення історії змін, залишення коментарів та здійснення аудиту дій, що критично важливо для нормативно регульованих проєктів.

Завдяки такій моделі даних система підтримує сценарії спільної роботи з високим ступенем контролю, адаптовані як для динамічних команд, так і для організацій із суворими вимогами до безпеки та простежуваності дій.

4.3 Реалізація ключових функцій

У цьому розділі подано огляд реалізації основних функціональних модулів програмного забезпечення. Кожен підрозділ детально описує окрему функціональність, яка забезпечує повноцінну роботу системи керування проєктами: від побудови шаблонів та організації доступів до роботи з проєктами, логування дій та особливостей інфраструктури.

Оскільки проєкт реалізовано з використанням модульної архітектури фреймворку NestJS, кожна функціональність винесена в окремий модуль, що взаємодіє з іншими компонентами через контролери, сервіси та middleware. Такий підхід дозволяє ефективно масштабувати систему, спрощує підтримку та подальший розвиток.

Основною особливістю реалізації є гнучка система шаблонів, на основі яких будуються сторінки проєктів. Вона забезпечує можливість створення динамічних форм із довільною структурою полів, яка адаптується під потреби конкретного проєкту. Разом з гнучким керуванням доступом на рівні ролей і дозволів, це дозволяє точно налаштовувати права користувачів у межах проєктної діяльності.

4.3.1 Шаблонування

Шаблони є основою побудови інтерфейсу користувача в системі. Вся структура сторінки проєкту формується на основі шаблону — заздалегідь визначеної схеми, яка описує поля, їхні типи, властивості та розташування. Це дозволяє створювати гнучкі динамічні сторінки, які адаптуються під конкретні потреби різних проєктів без необхідності внесення змін у код.

При завантаженні сторінки клієнтська частина (Frontend App) надсилає запит на отримання шаблону за його унікальним ідентифікатором GET /templates/{id}. Backend API спочатку перевіряє наявність шаблону у кеші. Якщо шаблон знайдено — він негайно повертається клієнту. У випадку відсутності шаблону в кеші, API витягує його з бази даних (MongoDB), після чого зберігає в кеш для майбутніх

запитів. Як результат, клієнт отримує структуру metadata + mock-сторінку, яка містить тільки опис полів без жодного заповненого значення.

Після відображення шаблону відбувається другий етап — завантаження конкретних даних для відкритого проєкту. Кожне поле, що потребує заповнення, пов’язане з відповідним елементом структури даних, який запитується окремо за допомогою GET /projects/{projectId}/data. Це дає змогу використовувати одну й ту саму структуру (шаблон) для різних наборів даних, зберігаючи при цьому строгий поділ між представленням і змістом.

На Рисунку 4.2 представлена діаграма послідовностей цього процесу.

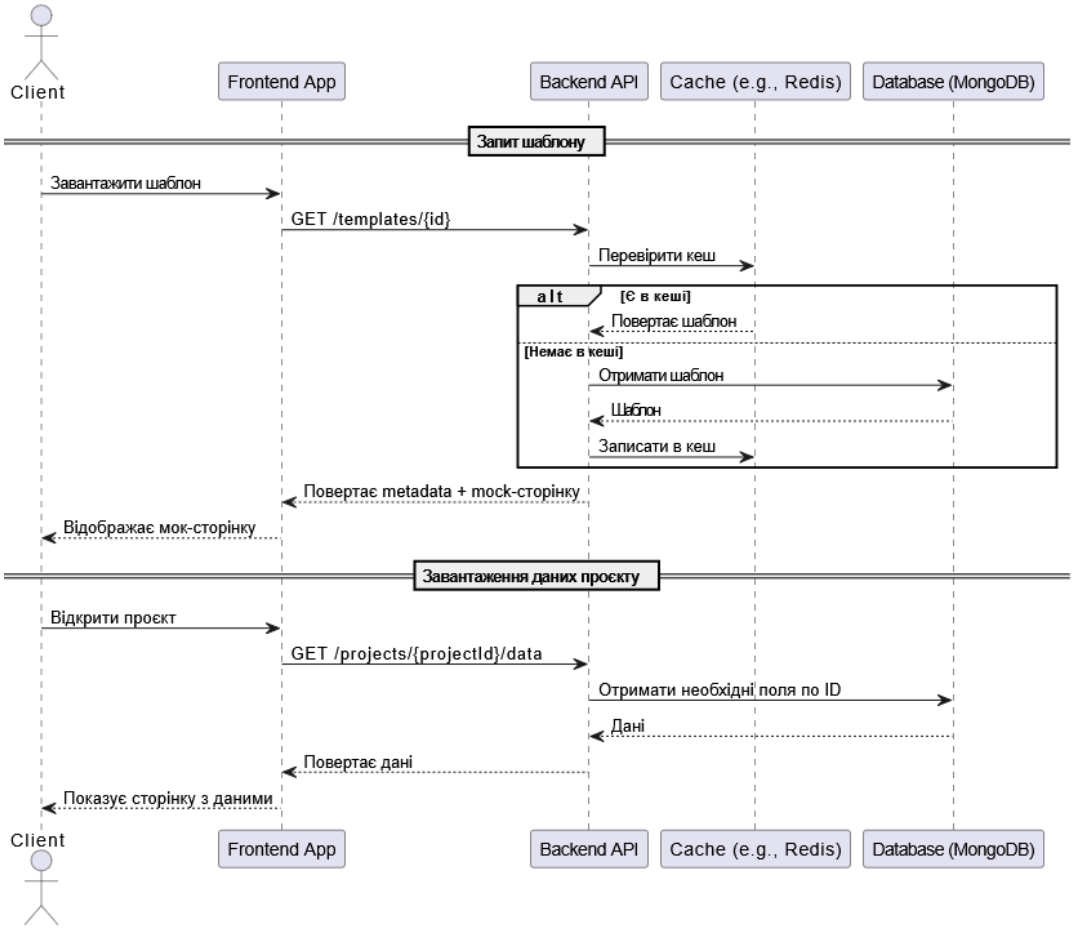


Рис. 4.2 Діаграма послідовностей роботи з шаблонами

Для такого потоку даних в програмі, були реалізовані моделі, представлені на Рисунку 4.3, які дозволяють розділити повністю процес завантаження сторінки та завантаження даних до відповідних полів на цій сторінці.

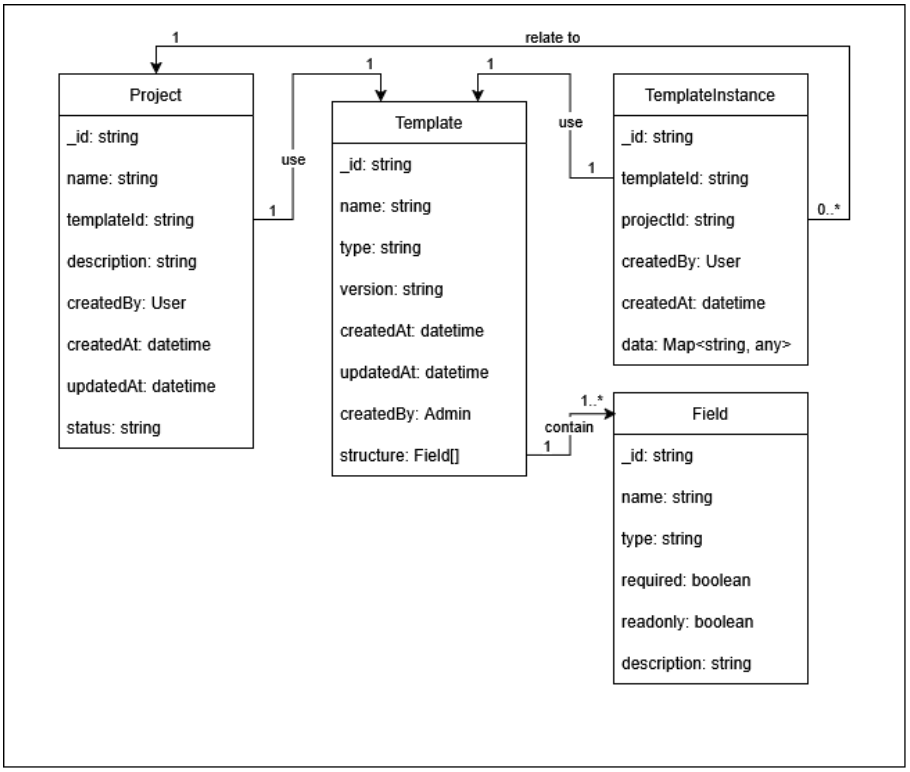


Рис. 4.3

4.3.2 Керування ролями та доступом

Система керування доступом у застосунку реалізована на основі ролей, що призначаються користувачам у межах конкретного проєкту. Такий підхід забезпечує гнучке та масштабоване обмеження прав відповідно до потреб кожного проєкту.

Кожен проєкт має власний набір ролей, які створюються адміністраторами організації. Ролі є кастомізованими: адміністратор визначає їхню назву та набір дозволів. Кожна роль, створюється адміністратором, відповідно до кожного шаблону, для неї обирається список дозволів з переліку та після цього, адміністратор призначає ролі до кожного учасника проєкту. Узагальнене представлення дозволів користувачів представлене на Рисунку 4.4.

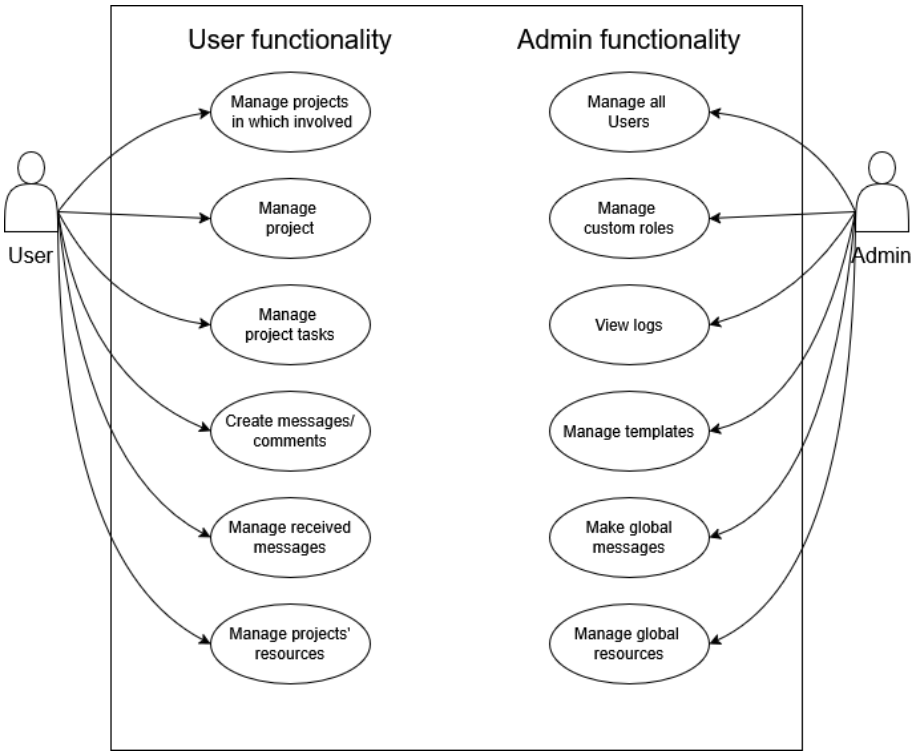


Рис. 4.4 Діаграма варіантів використання

Контроль доступу до функцій системи реалізований через middleware на рівні бекенду. При кожному запиті користувача система визначає його роль у межах відповідного проєкту та перевіряє, чи має вона необхідні дозволи для виконання запиту.

Життєвий цикл доступу:

1. Користувач надсилає запит до ресурсу PATCH /projects/{id}/fields/values.
2. Middleware визначає роль користувача у цьому проєкті.
3. Згідно з конфігурацією ролі, перевіряється наявність права edit_field_values.
4. У разі відсутності дозволу — користувач отримує відповідь 403 Forbidden.

Таким чином, система ролей забезпечує не лише безпеку, але й чітку організацію співпраці учасників у проєкті, даючи змогу розмежувати повноваження відповідно до їх функцій.

4.3.3 Керування проєктами та процесами

Модуль керування проєктами охоплює ключову частину логіки системи, яка відповідає за створення, структурування й підтримку життєвого циклу робочих процесів. Незважаючи на те, що загальна концепція реалізована згідно зі стандартними підходами, саме її масштабність, взаємодія з іншими модулями та роль у координації доступу до даних визначають її як одну з найбільш значущих частин системи.

Кожен проєкт є контейнером, що об'єднує завдання, користувачів і шаблони. При створенні проєкту адміністратор або керівник вказує його назву, опис, дедлайн, а також визначає учасників та їхні ролі. Учасники призначаються з-поміж зареєстрованих користувачів системи, при цьому кожному з них відповідно до його ролі визначається рівень доступу до ресурсів проєкту. Модель Project є базовою сутністю, пов’язаною з іншими сутностями системи через відношення, зокрема з шаблонами, завданнями (Task) і прив’язками користувачів (ProjectMember).

Завдання — основна структурна одиниця робочого процесу. Вони можуть створюватися вручну або базуватись на шаблонах, описаних у попередніх розділах. Завдання включають основні метадані (назва, опис, дедлайн, статус), посилання на виконавця та шаблон, а також поточний стан заповнення. Реалізація завдань передбачає окрему логіку для відображення прав доступу. Наприклад, виконавець бачить лише ті задачі, до яких він призначений, і лише поля, дозволені йому відповідно до ролі. Створення завдань доступне керівникам проєкту, які також можуть змінювати статус завдання та призначати користувачів.

Процеси роботи над проєктом реалізовано таким чином, щоб забезпечити прозору й передбачувану взаємодію між користувачами з різними рівнями доступу. Кожен тип ролі — від адміністратора до виконавця — отримує лише ті повноваження, які закладені в межах його дозволів. Система реалізує централізовану логіку авторизації через проміжне програмне забезпечення (middleware), яке контролює запити на основі збережених у базі ролей і дозволів.

З технічної точки зору, цей модуль представлений окремим NestJS-модулем ProjectsModule, який реалізує контролери для маршрутизації запитів, сервіси для

бізнес-логіки та інтеграцію з базою даних MongoDB. Дані проєкту зберігаються в колекції, структура якої дозволяє зберігати вкладені об'єкти, що містять інформацію про завдання, ролі, учасників і стан процесу виконання. Завдяки можливостям MongoDB, забезпечується висока гнучкість при фільтрації, агрегації та оновленні стану завдань без суттєвого навантаження на систему.

Таким чином, модуль керування проєктами не лише виконує адміністративну функцію структурування даних, а й координує логіку взаємодії між користувачами, завданнями та шаблонами, забезпечуючи цілісність і контрольованість процесів у системі.

4.3.4 Аудит дій

Система аудиту дій у проєкті реалізована як засіб фіксації ключових змін і взаємодій користувачів із застосунком. Хоча вона не включає повноцінного інтерфейсу для перегляду або аналізу збережених подій, проте її наявність відіграє важливу роль у забезпеченні прозорості, зворотного відстеження дій та можливості подальшого розширення в напрямку аудиту безпеки чи моніторингу продуктивності.

Усі дії, що мають значення для цілісності даних або можуть викликати зміну стану системи (наприклад, створення, оновлення або видалення ресурсів, призначення ролей, змінення статусів задач), логуються автоматично. Для цього використовується спеціалізований модуль логування, побудований на основі вбудованого LoggerService у NestJS. Завдяки централізованій реалізації, логіка аудиту не дублюється в окремих модулях, а підтримується на рівні middleware — проміжного шару, який перехоплює вхідні HTTP-запити й обробляє їх перед передачею у відповідні контролери.

Middleware виконує перевірку контексту запиту — включно з методом, шляхом, ідентифікатором користувача, який його ініціював, і результатом виконання. Ця інформація форматується у стандартизований запис і зберігається у файл логів на сервері, що дозволяє уникнути надмірного навантаження на базу даних. Формат файлу визначений таким чином, щоб забезпечити подальше

парсування, навіть якщо воно здійснюється зовнішніми інструментами або вручну. Залежно від критичності події, лог може позначатися відповідним рівнем (інформаційний, попереджувальний, критичний), що полегшує фільтрацію при аналізі.

Таке рішення дозволяє зберігати гнучкість у розробці — модуль логування може легко масштабуватися або доповнюватися, зокрема шляхом перенаправлення повідомлень до централізованої системи моніторингу або вивантаження до зовнішнього лог-сховища. Крім того, модуль має на меті забезпечити певний рівень відповідності вимогам нормативно регульованих проєктів, де наявність історії змін і контроль доступу до критичних дій є обов'язковою вимогою.

Отже, реалізація аудиту дій хоч і не передбачає інтерактивного інтерфейсу чи глибокого аналітичного інструментарію, однак забезпечує надійну основу для відстеження ключових операцій у системі та може бути розширена у разі зростання вимог до безпеки або регламенту.

4.4 Розгортання

Процес розгортання програмного забезпечення реалізований з використанням Docker та Docker Compose — інструментів, які забезпечують ефективне розгортання та масштабування застосунку в ізольованому середовищі. Таке рішення дозволяє створювати стабільні, передбачувані середовища для запуску як фронтенд-частини, так і бекенду разом із базою даних та іншими допоміжними сервісами.

Docker Compose дозволяє розгорнути усю систему за допомогою однієї команди, що значно спрощує розгортання у тестовому або продуктивному середовищі. Зовнішні параметри (такі як URL до бази даних, секретні ключі, токени тощо) зберігаються у .env файлі, що дозволяє розділити конфігурацію й код застосунку та покращує безпеку.

На Рисунку 4.5 представлене налаштування Docker на проєкті.

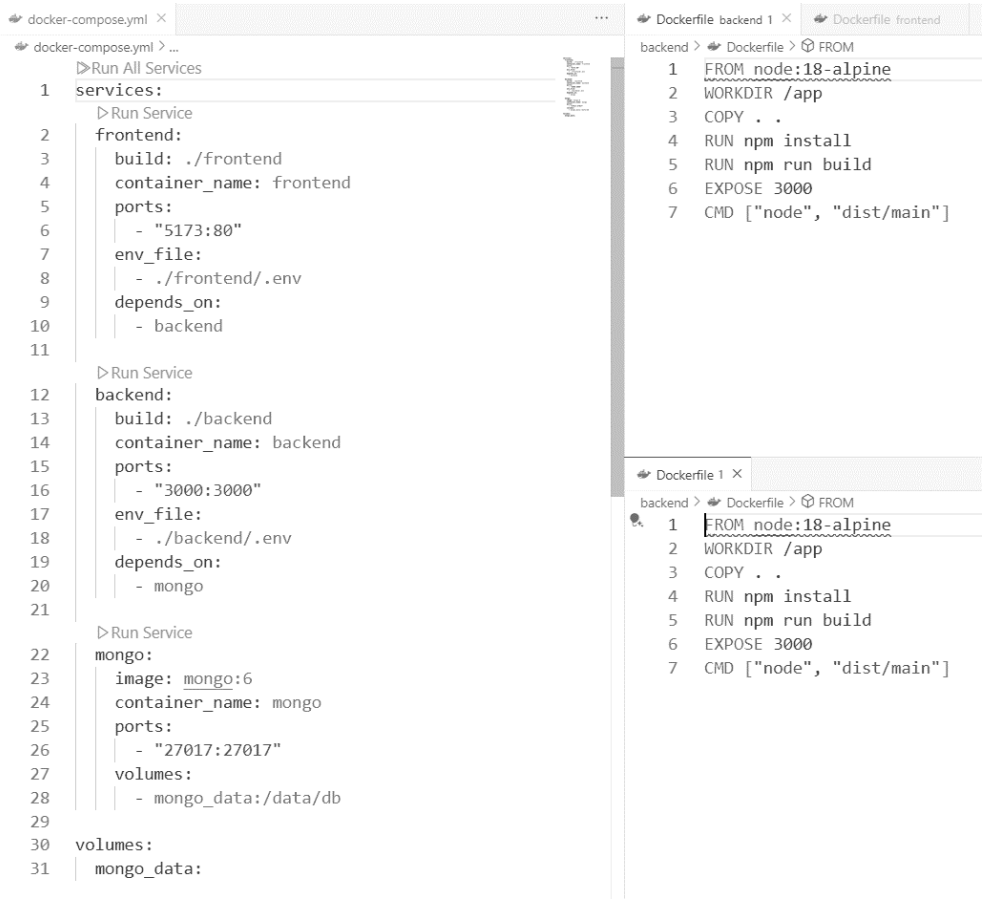


Рис. 4.5 Налаштування Docker

Крім того, використання Docker забезпечує повторюваність розгортання незалежно від середовища, що особливо важливо при роботі в команді або під час розгортання на сервері, де важлива стабільність та контроль над версіями компонентів. Також створено базовий сценарій ініціалізації даних (наприклад, попереднє створення адміністративного користувача або фіксованих ролей), що автоматизує первинне налаштування.

Загалом, обрана стратегія розгортання є гнучкою, масштабованою і зручною як для локальної розробки, так і для публічного хостингу або розміщення в хмарних середовищах.

ВИСНОВКИ

У ході виконання кваліфікаційної бакалаврської роботи було спроєктовано та реалізовано програмне забезпечення для керування проєктами, орієнтоване на внутрішнє використання в невеликих командах. Основною метою роботи було створення зручного, гнучкого та розширюваного інструменту для організації робочого процесу, що включає шаблонування завдань, систему ролей, відстеження результатів і базовий аудит.

Основні результати роботи такі:

Проведено порівняльний аналіз сучасних інструментів керування проєктами (Notion, Redmine, OpenProject, Asana), а також підходів до розробки подібного програмного забезпечення. Визначено ключові вимоги до функціональності, ролей та надійності збереження даних. Обґрунтовано доцільність створення окремого рішення, адаптованого під специфіку внутрішньої командної роботи.

Розроблено архітектуру програмного забезпечення, що складається з клієнтської частини (React + TailwindCSS) та серверної частини (NestJS + MongoDB), які взаємодіють через REST API. Архітектура забезпечує гнучкість, масштабованість і легкість підтримки.

На основі зібраних користувацьких історій сформульовано вимоги до ключових функцій системи. Це дозволило виділити мінімально життєздатну версію продукту (MVP), яка задовольняє основні потреби користувачів і водночас дає змогу гнучко розширювати функціональність у майбутньому.

Реалізовано модуль шаблонування завдань — ключову особливість програмного забезпечення. Завдяки шаблонам користувачі працюють з уніфікованими структурами завдань. Адміністори створюють шаблони, керівники заповнюють загальні поля, а виконавці — результативні. Це забезпечує контроль, узгодженість та ефективність роботи.

Впроваджено систему ролей із трирівневою ієрархією доступу (адміністратор організації, керівник, виконавець/спостерігач), з можливістю

керування доступами на рівні організації, проєкту та окремого шаблону. Це дозволяє адаптувати застосунок до різних сценаріїв використання.

Реалізовано базовий аудит дій користувачів, що забезпечує прозорість і відповідальність у командній роботі. Історія змін дає змогу відстежувати, хто й коли редагував дані, що особливо важливо в контексті нормативно регульованих проєктів.

Створено мінімально життєздатну версію програмного забезпечення (MVP), що включає: створення організацій та проєктів, роботу з шаблонами, заповнення завдань, призначення ролей, базову навігацію та авторизацію. Інтерфейс орієнтований на зручність і простоту взаємодії.

У результаті виконання кваліфікаційної роботи було досягнуто поставлених цілей — реалізовано повноцінний прототип інструменту керування проєктами, який може бути основою для подальшого розвитку, інтеграції нових функцій та адаптації до конкретних потреб команди. Отриманий досвід дозволив поглибити знання в галузі веброзробки, архітектури ПЗ, організації роботи та практичного застосування підходів до побудови MVP.

Результати дослідження апробовано та опубліковано у наступних тезах доповіді на конференціях:

1. Максименко Д.С., Залива В.В. Забезпечення цілісності та безпеки внутрішніх процесів у системі управління нормативно регульованими проєктами. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в ІКТ», 24.04.25, ДУІКТ, Київ, Україна, с.434-436.[1]
2. Максименко Д.С., Залива В.В. Порівняння засобів захисту інформації в системах управління проєктами при хмарному та локальному розгортанні. Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, с.295-297.[2]

ПЕРЕЛІК ПОСИЛАНЬ

1. Максименко Д.С., Залива В.В. Забезпечення цілісності та безпеки внутрішніх процесів у системі управління нормативно регульованими проєктами. // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ», 24.04.25, ДУІКТ, Київ, Україна, с.434-436.
2. Максименко Д.С., Залива В.В. Порівняння засобів захисту інформації в системах управління проєктами при хмарному та локальному розгортанні. // XII Всеукраїнська науково-практична конференція молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, с.295-297.
3. Fielding P.J. How to manage projects: essential project management skills to deliver on-time, on-budget result. Kogan Page, Limited, 2019.
4. Герило В. М. Ощадливе виробництво як організаційно-економічний механізм підвищення ефективності діяльності підприємства. Економіка і організація управління. 2022. № 3. С. 41–46. URL: https://doi.org/10.31558/2307-2318.2022.3.4.
5. Moving an existing waterfall project to prince2 agile. PRINCE2 agile an implementation pocket guide. 2016. P. 106–109. URL: https://doi.org/10.2307/j.ctt1bj4t44.14.
6. Notion [Електронний ресурс] URL: https://www.notion.com/
7. Redmine [Електронний ресурс] URL: https://www.redmine.org/
8. OpenProject [Електронний ресурс] URL: https://www.openproject.org/
9. Asana [Електронний ресурс] URL: https://asana.com/
10. Rehkopf M. Epics, stories, themes, and initiatives | atlassian. Atlassian. URL: https://www.atlassian.com/agile/project-management/epics-stories-themes.
11. MVP development process for software startups / L. Pompermaier et al. Lecture notes in business information processing. Cham, 2019. P. 409–412. URL: https://doi.org/10.1007/978-3-030-33742-1_33.
12. Mead A. Advanced Node.js Development: Master Node.js by building real-world applications. Packt Publishing, 2018. 592 p.
13. Leonard A. Spring boot persistence best practices: optimize java persistence performance in spring boot applications. Apress, 2020. 1057 p.
14. Zemskov M. Reliability of the type system in typescript in software development. Asian journal of research in computer science. 2025. Vol. 18, no. 2. P. 50–59. URL: https://doi.org/10.9734/ajrcos/2025/v18i2561.
15. Basumatary B., Agnihotri N. Benefits and challenges of using nodejs. International journal of innovative research in computer science & technology. 2022. P. 67–70. URL: https://doi.org/10.55524/ijircst.2022.10.3.13.

16. NodeJS and postman for serverless computing / R. K. Kannan et al. Advances in systems analysis, software engineering, and high performance computing. 2024. P. 195–204. URL: https://doi.org/10.4018/979-8-3693-1682-5.ch012.
17. Vossen G. ACID properties. Encyclopedia of database systems. New York, NY, 2018. P. 23–25. URL: https://doi.org/10.1007/978-1-4614-8265-9_831.
18. Conway J., Berkus J. Power postgresql: maximizing postgresql performance, reliability, and utility. Pearson Education, Limited, 2025. 600 p.
19. Practical mongodb: architecting, developing, and administering mongodb. Apress, 2015. 272 p.
20. React.js: a premier frontend solution. International research journal of modernization in engineering technology and science. 2024. URL: https://doi.org/10.56726/irjmets60634.
21. Deo Eka Putra R., Lestari A., Kristianti N. Analisis perbandingan CSS framework tailwind CSS dan bootstrap dalam pengembangan landing page website POS digitaliz. Journal of information technology and computer science. 2025. Vol. 5, no. 1. P. 63–75. URL: https://doi.org/10.47111/jointecoms.v5i1.19807.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка програмного забезпечення для управління нормативно регульованими проєктами

Виконав студент 4 курсу
групи ПД-41
Максименко Дмитро Сергійович
Керівник роботи
доктор філософії (PhD), старший викладач кафедри Залива Віталій Вікторович

Київ – 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи** - автоматизація процесу виконання нормативно регульованих проєктів.
- **Об'єкт дослідження** - процес керування та виконання нормативно регульованих процесів.
- **Предмет дослідження** - програмне забезпечення для управління нормативно регульованими проєктами.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз вже існуючих рішень, порівнявши їх сильні та слабкі сторони.
2. На основі аналізу, розробити вимоги до програмного забезпечення.
3. Дослідити методи, які задовольняють вимоги, порівняти їх сильні та слабкі сторони, обрати ті які відповідають вимогам до програмного забезпечення.
4. Відповідно до вимог спроєктувати архітектуру та дизайн програмного забезпечення.
5. Розробити програмне забезпечення для керування нормативно регульованими проєктами.

3

АНАЛІЗ АНАЛОГІВ

Показник	Notion	Redmine	OpenProject	Asana	NormalProject
Шаблонізація процесів	+, шаблони сторінок, автоматизація створення	-, лише через сторонні плагіни	+, шаблони сторінок	+, шаблони задач	+, шаблони проєктів і сторінок
Однорідність	-, високий рівень доступної кастомізації створює неоднорідність	+, повна однорідність всіх проєктів	+, повна однорідність всіх проєктів	+, повна однорідність всіх проєктів	+, повна однорідність в межах кожного окремого проєкту
Ролі та права доступу	-, базова система ролей	+, гнучка система ролей	+, гнучка система ролей	+, гнучка система ролей	+, гнучка система ролей
Контроль змін	+, але лише в межах кожної окремої сторінки	+, повний лог	+, повний лог	-, частковий аудит	+, логування змін, аудит доступу
Простота освоєння	+, простий інтерфейс, низький поріг входу, але хаотична вкладеність	-, застарілий інтерфейс, висока вкладеність	-, висока вкладеність	+, сучасний інтерфейс, низька вкладеність	+, сучасний інтерфейс, низька вкладеність

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

- **Шаблонізація**: можливість створення та використання шаблонів як для всього проєкту, так і для окремих сторінок;
- **Ролі та права доступу**: розділення відповідальностей та можливостей щодо створення та редагування шаблонів, редагування розміщення та заповнення окремих даних користувачами;
- **Управління процесами**: можливість встановлювати відповідальних за задачу, встановлювати дедлайни та інших обмежень, відповідно до нормативних документів;
- **Відслідковування змін**: логування всіх дій користувача, перегляд історії змін.

Нефункціональні вимоги:

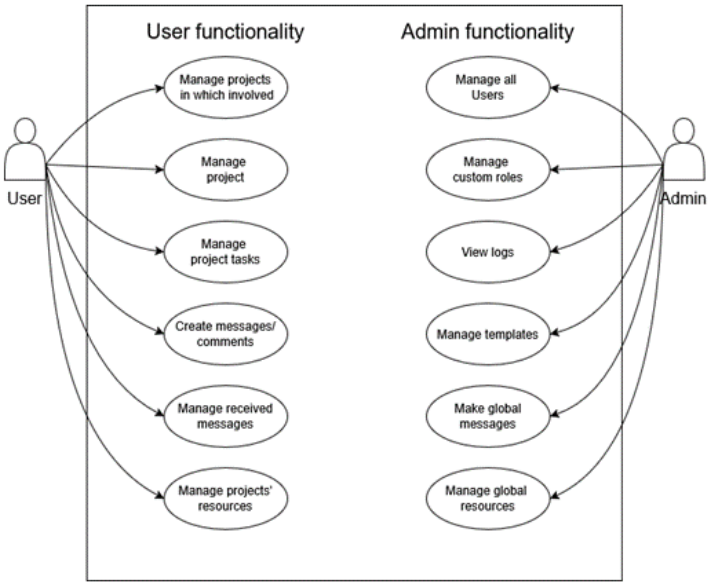
- Зручність використання (UI/UX): адаптивний інтерфейс під різні розміри екрану, мінімальна вкладеність для підвищення інтуїтивності.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

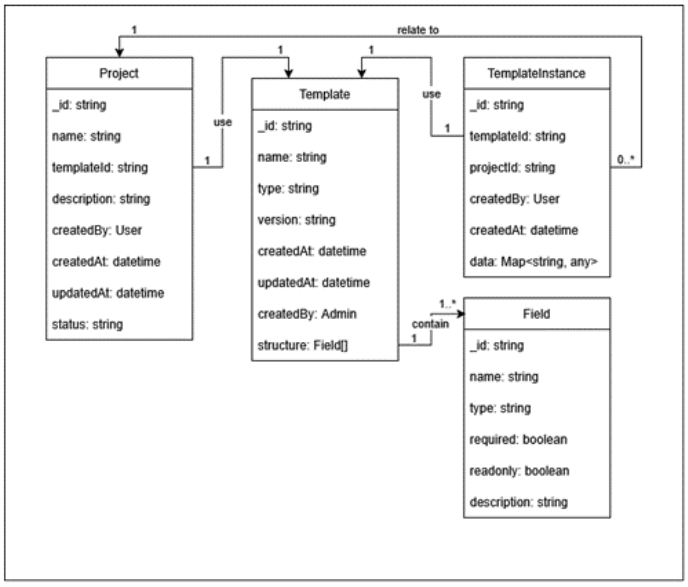
6

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



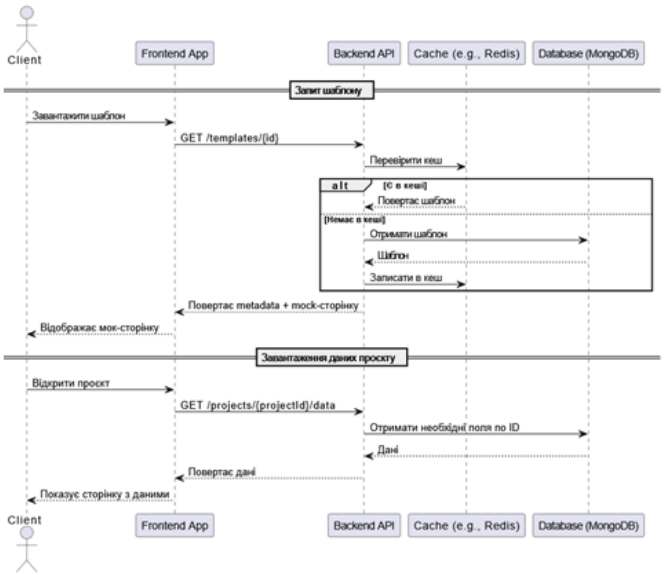
7

ОПИС КЛЮЧОВИХ СТРУКТУР ДАНИХ ДЛЯ РОБОТИ З ШАБЛОНАМИ



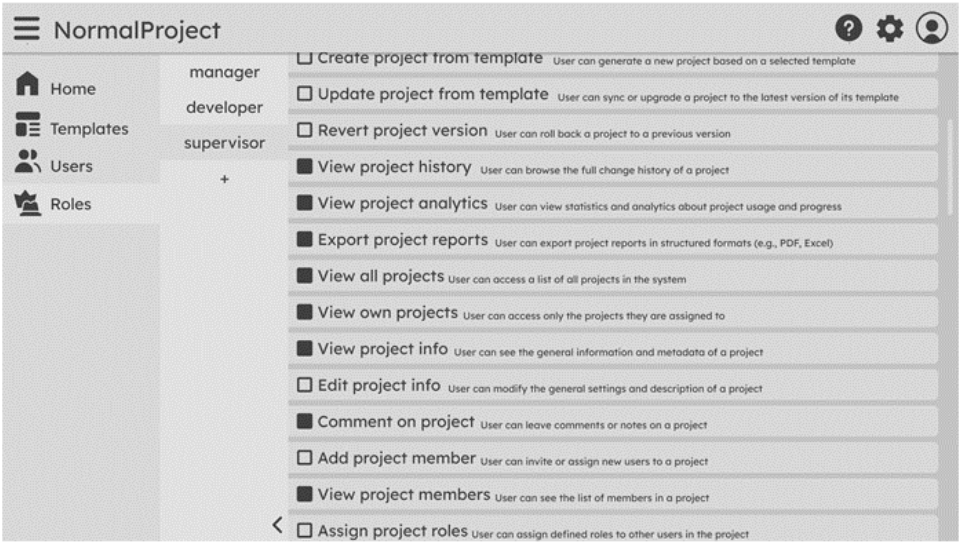
8

ДІАГРАМА ПОСЛІДОВНОСТЕЙ ЗАВАНТАЖЕННЯ СТОРІНКИ ПРОЄКТУ



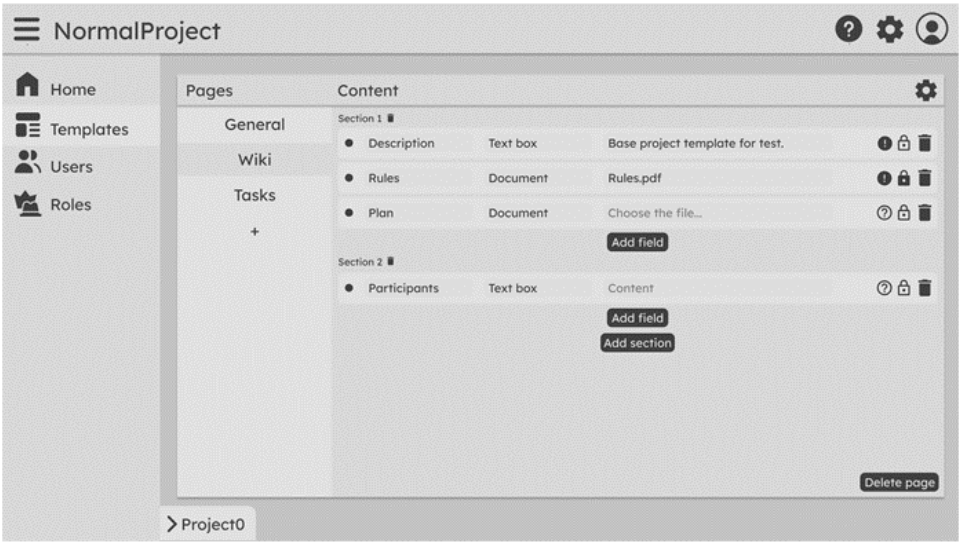
9

ЕКРАННІ ФОРМИ



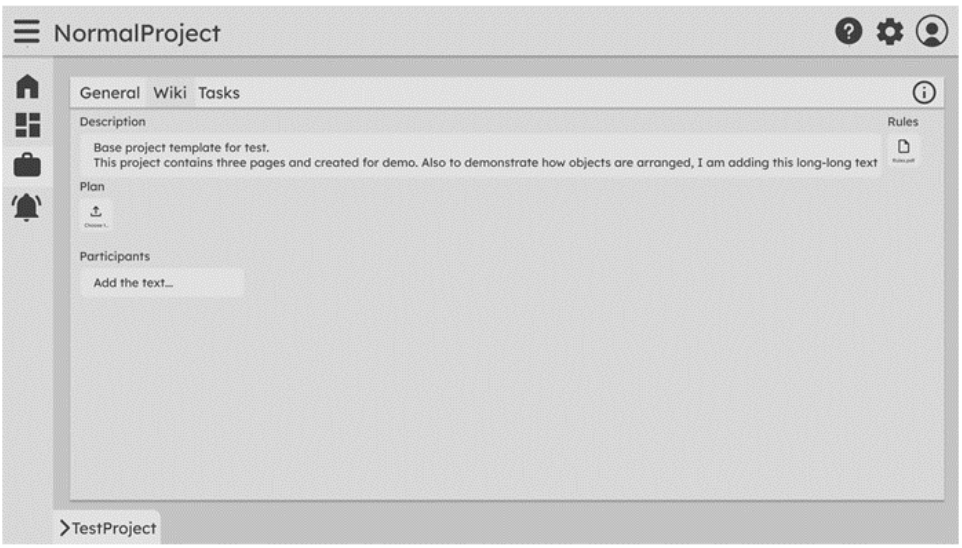
10

ЕКРАННІ ФОРМИ



11

ЕКРАННІ ФОРМИ



12

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Тези:

1. Максименко Д.С., Залива В.В. Забезпечення цілісності та безпеки внутрішніх процесів у системі управління нормативно регульованими проєктами. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в ІКТ», 24.04.25, ДУІКТ, Київ, Україна, с.434-436.
2. Максименко Д.С., Залива В.В. Порівняння засобів захисту інформації в системах управління проєктами при хмарному та локальному розгортанні. Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, с.295-297.

13

ВИСНОВКИ

1. Проведено аналіз вже існуючих рішень для керування проєктами, порівнявши їх сильні та слабкі сторони стосовно регульованих проєктів.
2. На основі аналізу було розроблено вимоги до програмного продукту, які забезпечать адекватний функціонал для застосунку для нормативно регульованих проєктів.
3. Спроєктовано архітектура та дизайн програмного забезпечення.
4. Розроблено програмне забезпечення для керування нормативно регульованими проєктами.
5. Застосунок було успішно розгорнуто за допомогою Docker.

14

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
// main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { LoggerMiddleware } from './logger/logger.middleware';
import { ValidationPipe } from '@nestjs/common';
import * as cookieParser from 'cookie-parser';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.use(cookieParser());
  app.use(LoggerMiddleware);
  app.useGlobalPipes(new ValidationPipe({ whitelist: true, transform: true }));

  require('dotenv').config();
  await app.listen(process.env.PORT ?? 3000);}
bootstrap();

// app.module.ts
import { Module } from '@nestjs/common';
import { TemplateModule } from './templates/template.module';
import { RolesModule } from './roles/roles.module';
import { ProjectsModule } from './projects/projects.module';
import { LoggerModule } from './logger/logger.module';
import { MongooseModule } from '@nestjs/mongoose';
import { ConfigModule } from '@nestjs/config';

@Module({
  imports: [
    ConfigModule.forRoot({ isGlobal: true }),
    MongooseModule.forRootAsync({
      useFactory: async () => ({
        uri: process.env.MONGO_URI,
        dbName: process.env.MONGO_DB_NAME,
      }),
    }),
  ],
  controllers: [AppController],
  providers: [AppService, AppLogger],
})
export class AppModule implements NestModule{
  configure(consumer: MiddlewareConsumer) {

consumer.apply(LoggingMiddleware).forRoutes('*');
  }
}
// logger/logger.module.ts
import { Module, MiddlewareConsumer, NestModule } from '@nestjs/common';
import { LoggerMiddleware } from './logger.middleware';

@Module({})
export class LoggerModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {

consumer.apply(LoggerMiddleware).forRoutes('*');
  }
}

// logger/logger.middleware.ts
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  private logFilePath: string;

  constructor() {
    const logsDir = path.resolve(__dirname, '../../../logs');
    if (!fs.existsSync(logsDir)) {
```

```
            fs.mkdirSync(logsDir, { recursive: true });
          }
          this.logFilePath = path.join(logsDir, 'app.log');
        }

        private writeToFile(level: string, message: string, trace?: string) {
          const timestamp = new Date().toISOString();
          const logMsg = `[${level}] ${timestamp} ${message}${trace ? ' ' + trace : ''}\n`;
          fs.appendFile(this.logFilePath, logMsg, err => {
            if (err) {
              console.error('Failed to write log to file:', err);
            }
          });
        }

        log(message: string) {
          console.log(`[LOG-b] ${message}`);
          this.writeToFile('LOG-b', message);
        }

        error(message: string, trace?: string) {
          console.error(`[ERROR-b] ${message}`, trace);
          this.writeToFile('ERROR-b', message, trace);
        }

        warn(message: string) {
          console.warn(`[WARN-b] ${message}`);
          this.writeToFile('WARN-b', message);
        }

        debug(message: string) {
          console.debug(`[DEBUG-b] ${message}`);
          this.writeToFile('DEBUG-b', message);
        }}

      // templates/template.module.ts
      import { Module, Admin, Field } from '@nestjs/common';
      import { TemplateController } from './template.controller';
      import { TemplateService } from './template.service';
      import { MongooseModule } from '@nestjs/mongoose';
      import { Template, TemplateSchema } from './template.schema';

      @Module({
        imports: [
          MongooseModule.forFeature([{ name: Template.name, schema: TemplateSchema }]),
        ],
        controllers: [TemplateController],
        providers: [TemplateService],
      })
      export class TemplateModule {}

      // templates/template.schema.ts
      import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
      import { Document } from 'mongoose';

      @Schema()
      export class Template extends Document {
        @Prop({ required: true })
        name: string;

        @Prop({ required: true })
        type: string[];

        @Prop({ required: true })
        version: string[];

        @Prop({ required: true })
        version: string[];

        @Prop({ default: Date.now })
        createdAt: Date;
      }

        @Prop({ default: Date.now })
        updatedAt: Date;
      }

        @Prop({ required: false })
        createdBy: Admin;
      }
```

```
  @Prop({ required: false })
  structure: Field[];
}

export const TemplateSchema = SchemaFactory.createForClass(Template);

// templates/template.controller.ts
import { Controller, Get, Post, Body, Param } from '@nestjs/common';
import { TemplateService } from './template.service';
import { CreateTemplateDto } from './dto/create-template.dto';

@Controller('templates')
export class TemplateController {
  constructor(private readonly templateService: TemplateService) {}

  @Post()
  create(@Body() createTemplateDto: CreateTemplateDto) {
    return this.templateService.create(createTemplateDto);
  }

  @Get()
  findAll() {
    return this.templateService.findAll();
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.templateService.findOne(id);
  }
}

// templates/template.service.ts
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Model } from 'mongoose';
import { Template } from './template.schema';
import { CreateTemplateDto } from './dto/create-template.dto';

@Injectable()
export class TemplateService {

constructor(@InjectModel(Template.name) private model: Model<Template>) {}

  create(dto: CreateTemplateDto) {
    const template = new this.model(dto);
    return template.save();
  }

  findAll() {
    return this.model.find().exec();
  }

  findOne(id: string) {
    return this.model.findById(id).exec();
  }
}

// templates/dto/create-template.dto.ts
import { IsString, IsArray } from 'class-validator';

export class CreateTemplateDto {
  @IsString()
  name: string;

  @IsArray()
  fields: string[];
}
// src/roles/roles.controller.ts
import { Controller, Get, Post, Body, Param, UseGuards } from '@nestjs/common';
import { RolesService } from './roles.service';
import { CreateRoleDto } from './dto/create-role.dto';
import { AuthGuard } from '../auth/auth.guard';

@Controller('roles')
@UseGuards(AuthGuard)
export class RolesController {
  constructor(private readonly rolesService: RolesService) {}

  @Post()
  create(@Body() createRoleDto: CreateRoleDto) {
    return this.rolesService.create(createRoleDto);
  }
```

```
  @Get()
  findAll() {
   return this.rolesService.findAll();
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
   return this.rolesService.findOne(id);
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() updateRoleDto: CreateRoleDto) {
   return this.rolesService.update(id, updateRoleDto);
  }

  @Delete(':id')
  remove(@Param('id') id: string) {
   return this.rolesService.remove(id);
  }
 }

 // src/roles/roles.service.ts
 import { Injectable } from '@nestjs/common';
 import { InjectModel } from '@nestjs/mongoose';
 import { Model } from 'mongoose';
 import { Role, RoleDocument } from './schemas/role.schema';
 import { CreateRoleDto } from './dto/create-role.dto';

 @Injectable()
 export class RolesService {
  constructor(@InjectModel(Role.name) private roleModel: Model<RoleDocument>) {}

  async create(createRoleDto: CreateRoleDto): Promise<Role> {
   const newRole = new this.roleModel(createRoleDto);
   return newRole.save();
  }

  async findAll(): Promise<Role[]> {
   return this.roleModel.find().exec();
  }

  async findOne(id: string): Promise<Role> {
   return this.roleModel.findById(id).exec();
  }

  async update(id: string, updateRoleDto: CreateRoleDto): Promise<Role> {
   return this.roleModel.findByIdAndUpdate(id, updateRoleDto, { new: true }).exec();
  }

  async remove(id: string): Promise<Role> {
   return this.roleModel.findByIdAndDelete(id).exec();
  }
 }

 // src/roles/schemas/role.schema.ts
 import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
 import { Document } from 'mongoose';

 export type RoleDocument = Role & Document;

 @Schema()
 export class Role {
  @Prop({ required: true })
  name: string;

  @Prop({ default: [] })
  permissions: string[];
 }

 export const RoleSchema = SchemaFactory.createForClass(Role);


 // src/roles/dto/create-role.dto.ts
 export class CreateRoleDto {
  readonly name: string;
  readonly permissions?: string[];
 }


 // src/projects/projects.controller.ts
```

```
import { Controller, Get, Post, Body, Param, Put, Delete, UseGuards } from '@nestjs/common';
import { ProjectsService } from './projects.service';
import { CreateProjectDto } from './dto/create-project.dto';
import { UpdateProjectDto } from './dto/update-project.dto';
import { AuthGuard } from '../auth/auth.guard';

@Controller('projects')
@UseGuards(AuthGuard)
export class ProjectsController {
 constructor(private readonly projectsService: ProjectsService) {}

 @Post()
 create(@Body() createProjectDto: CreateProjectDto) {
  return this.projectsService.create(createProjectDto);
 }

 @Post('from-template/:templateId')
 createFromTemplate(@Param('templateId') templateId: string, @Body() overrideDto: Partial<CreateProjectDto>) {
  return this.projectsService.createFromTemplate(templateId, overrideDto);
 }

 @Get()
 findAll() {
  return this.projectsService.findAll();
 }

 @Get(':id')
 findOne(@Param('id') id: string) {
  return this.projectsService.findOne(id);
 }

 @Put(':id')
 update(@Param('id') id: string, @Body() updateProjectDto: UpdateProjectDto) {
  return this.projectsService.update(id, updateProjectDto);
 }

 @Delete(':id')
 remove(@Param('id') id: string) {
  return this.projectsService.remove(id);
 }
}

// src/projects/projects.service.ts
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Model } from 'mongoose';
import { Project, ProjectDocument } from './schemas/project.schema';
import { CreateProjectDto } from './dto/create-project.dto';
import { UpdateProjectDto } from './dto/update-project.dto';

@Injectable()
export class ProjectsService {

constructor(@InjectModel(Project.name) private projectModel: Model<ProjectDocument>) {}

 async create(createProjectDto: CreateProjectDto): Promise<Project> {
  const project = new this.projectModel(createProjectDto);
  return project.save();
 }

 async createFromTemplate(templateId: string, overrideDto: Partial<CreateProjectDto>): Promise<Project> {
  const template = await this.projectModel.findById(templateId).exec();
  if (!template) throw new NotFoundException('Template not found');

  const mergedData: CreateProjectDto = {
   name: overrideDto.name || `${template.name} Copy`,
   description: overrideDto.description || template.description,
```

```
        members: overrideDto.members || [...template.members],
      };

      const newProject = new this.projectModel(mergedData);
      return newProject.save();
    }

    async findAll(): Promise<Project[]> {
      return this.projectModel.find().exec();
    }

    async findOne(id: string): Promise<Project> {
      return this.projectModel.findById(id).exec();
    }

    async update(id: string, updateProjectDto: UpdateProjectDto): Promise<Project> {
      return this.projectModel.findByIdAndUpdate(id, updateProjectDto, { new: true }).exec();
    }

    async remove(id: string): Promise<Project> {
      return this.projectModel.findByIdAndDelete(id).exec();
    }
  }

  // src/projects/schemas/project.schema.ts
  import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
  import { Document } from 'mongoose';

  export type ProjectDocument = Project & Document;

  @Schema({ timestamps: true })
  export class Project {
    @Prop({ required: true })
    name: string;

    @Prop()
    description: string;

    @Prop()
    templateId: string;

    @Prop({ type: [String], default: [] })
    members: string[];

    @Prop({ default: 'draft' })
    status: string;
  }

    @Prop({ default: Date.now })
    createdAt: Date;
  }

    @Prop({ default: Date.now })
    updatedAt: Date;
  }

    @Prop({ required: false })
    createdBy: Admin;
  }

  export const ProjectSchema = SchemaFactory.createForClass(Project);


  // src/projects/dto/create-project.dto.ts
  export class CreateProjectDto {
    readonly name: string;
    readonly description?: string;
    readonly members?: string[];
  }


  // src/projects/dto/update-project.dto.ts
  export class UpdateProjectDto {
    readonly name?: string;
    readonly description?: string;
    readonly members?: string[];
    readonly status?: string;
  }
  // index.tsx
  import React from "react";
  import ReactDOM from "react-dom/client";
  import App from "./App";
  import "./index.css";

  const root = ReactDOM.createRoot(document.getElementById("root")!);
  root.render(
```

```
  <React.StrictMode>
    <App />
  </React.StrictMode>
);


// App.tsx
import React from "react";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import { Toaster } from "react-hot-toast";
import Home from "./pages/Home";
import Login from "./pages/Login";
import Dashboard from "./pages/Dashboard";
import TemplatePage from "./pages/TemplatePage";
import ProjectPage from "./pages/ProjectPage";
import ProtectedRoute from "./components/ProtectedRoute";

export default function App() {
  return (
    <Router>
      <Toaster position="top-right" />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/login" element={<Login />} />
        <Route
          path="/dashboard"
          element={
            <ProtectedRoute>
              <Dashboard />
            </ProtectedRoute>
          }
        />
        <Route
          path="/templates/:id"
          element={
            <ProtectedRoute>
              <TemplatePage />
            </ProtectedRoute>
          }
        />
        <Route
          path="/projects/:id"
          element={
            <ProtectedRoute>
              <ProjectPage />
            </ProtectedRoute>
          }
        />
      </Routes>
    </Router>
  );
}


// components/ProtectedRoute.tsx
import React from "react";
import { Navigate } from "react-router-dom";
import { useAuth } from "../hooks/useAuth";

export default function ProtectedRoute({ children }: { children: JSX.Element }) {
  const { isAuthenticated } = useAuth();

  if (!isAuthenticated) return <Navigate to="/login" replace />;
  return children;
}


// hooks/useAuth.ts
import { useEffect, useState } from "react";

export function useAuth() {
  const [isAuthenticated, setIsAuthenticated] = useState(false);

  useEffect(() => {
    const token = localStorage.getItem("token");
    setIsAuthenticated(!!token);
  }, []);

  return { isAuthenticated };
}


// pages/Home.tsx
import React from "react";
import { Link } from "react-router-dom";

export default function Home() {
  return (
```

```
        <div className="p-10">
          <h1 className="text-4xl font-bold mb-4">Welcome to Project Manager</h1>
          <Link
            to="/login"
            className="px-4 py-2 bg-blue-600 text-white rounded hover:bg-blue-700"
          >
            Login
          </Link>
        </div>
      );
    }


    // pages/Login.tsx
    import React, { useState } from "react";
    import { useNavigate } from "react-router-dom";
    import toast from "react-hot-toast";

    export default function Login() {
      const [email, setEmail] = useState("");
      const [password, setPassword] = useState("");
      const navigate = useNavigate();

      const handleLogin = async () => {
        try {
          const response = await fetch("http://localhost:3000/auth/login", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({ email, password }),
          });

          const data = await response.json();
          if (!response.ok) throw new Error(data.message);
          localStorage.setItem("token", data.access_token);
          toast.success("Logged in successfully!");
          navigate("/dashboard");
        } catch (err: any) {
          toast.error(err.message);
        }
      };

      return (
        <div className="flex flex-col items-center justify-center min-h-screen">
          <h2 className="text-2xl font-bold mb-6">Login</h2>
          <input
            type="email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
            placeholder="Email"
            className="mb-4 px-4 py-2 border rounded"
          />
          <input
            type="password"
            value={password}
            onChange={(e) => setPassword(e.target.value)}
            placeholder="Password"
            className="mb-4 px-4 py-2 border rounded"
          />
          <button
            onClick={handleLogin}
            className="px-4 py-2 bg-blue-600 text-white rounded hover:bg-blue-700"
          >
            Login
          </button>
        </div>
      );
    }


// models.ts

export interface Field {
  _id: string;
  name: string;
  type: string;
  templateId?: string;
  required: boolean;
  readonly: boolean;
  description: string;
  fields?: Field[];
}

export interface Template {
  _id: string;
  name: string;
  type: string;
  version: string;
  createdAt: Date;
```

```
    updatedAt: Date;
    createdBy: Admin;
    structure: Field[];
}

export interface Project {
    _id: string;
    name: string;
    templateId: string;
    description: string;
    createdBy: User;
    createdAt: Date;
    updatedAt: Date;
    status: string;
}

export interface TemplateInstance {
    _id: string;
    templateId: string;
    projectId: string;
    createdBy: User;
    createdAt: Date;
    data: Map<string, any>;
}

// backend/Dockerfile
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build
EXPOSE 3000
CMD ["node", "dist/main"]

// fronend/Dockerfile
FROM node:18-alpine AS builder
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build

FROM nginx:alpine
COPY --from=builder /app/dist
/usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]

//docker-compose.yml
services:
  frontend:
    build: ./frontend
    container_name: frontend
    ports:
      - "5173:80"
    env_file:
      - ./frontend/.env
    depends_on:
      - backend

  backend:
    build: ./backend
    container_name: backend
    ports:
      - "3000:3000"
    env_file:
      - ./backend/.env
    depends_on:
      - mongo

  mongo:
    image: mongo:6
    container_name: mongo
    ports:
      - "27017:27017"
    volumes:
      - mongo_data:/data/db

volumes:
  mongo_data:
```