МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ ТА БІОТЕХНОЛОГІЙ ІМ. С.З.ҐЖИЦЬКОГО

ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

другого (магістерського) рівня вищої освіти

на тему: **“ Методи та інструменти управління командною розробкою IT-проєктів на основі Agile ”**

Виконав: ст. гр. Іт-62_____

Спеціальності 126 – «Інформаційні системи та технології»_____
(шифр і назва)

_____Смуток Василь Васильович_____
(прізвище та ініціали)

Керівник: __к.е.н., доц. Смолінський В.Б._____
(прізвище та ініціали)

Рецензент: _____
(прізвище та ініціали)

ДУБЛЯНИ-2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВЕТЕРИНАРНОЇ МЕДИЦИНИ ТА БІОТЕХНОЛОГІЙ ІМ. С.З.ГЖИЦЬКОГО

ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ
ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

другий (магістерський) рівень вищої освіти
126 – «Інформаційні системи та технології»

"ЗАТВЕРДЖУЮ"
Завідувач кафедри_____
д.т.н., проф. А.М. Тригуба
"____"_____2025 р.

ЗАВДАННЯ

на кваліфікаційну роботу студенту

Смутку Василю Васильовичу

1. Тема роботи: «Методи та інструменти управління командною розробкою ІТ-проєктів на основі Agile»

Керівник роботи Смолінський Валентин Броніславович, к.е.н., доцент
Затверджені наказом по університету №140/к-с від 28.02.2025.

2. Строк подання студентом роботи 06.12.2024 р.

3. Початкові дані до роботи: 1. Технічна і довідкова література.
2. Методології координації в команді та планування ІТ-проектів.
3. Функціонал Maven, Java та Spring boot. 4. Методика перевірки продуктивності ПЗ.

4. Зміст розрахунково-пояснювальної записки:

1. Аналіз основ управління розробкою іт-проектів.

2. Методологія та інструментарій командної комунікації в іт-проектах.

3. Реалізація моделі управління командною розробкою іт-проектів.

4. Результати практичного застосування.

5. Охорона праці та безпека в надзвичайних ситуаціях.

Висновки та пропозиції.

Бібліографічний список.

Додатки.

5. Перелік графічного матеріалу: 1 та 2 – Тема, мета, завдання роботи; 3 – Аналіз підходів до розробки програмного забезпечення; 4 – Дошки для планування робіт; 5 – Життєвий цикл проектів; 6 – Особливості поєднання базового інструментарію; 7 – Архітектура системи командної розробки ІТ-продуктів та комунікації; 8 – Структура функціональних кейсів ПЗ; 9 – Розробка програмного засобу для управління командою; 10 – Використання платформи Postman; 11 – Обробка запитів під час клієнт-серверної взаємодії; 12 – Обробка виключень; 13 – Результати дослідження ефективності коду; 14 – Висновки.

6. Консультанти з розділів:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1, 2, 3, 4	*Смолінський В.Б., доцент кафедри інформаційних технологій*		
5	*Городецький І.М., доцент кафедри інженерної механіки*		

Дата видачі завдання 28.02.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту	Строк виконання етапів роботи	Приміт-ка
1	*Написання першого розділу та означення головних завдань роботи*	*01.03-01.05.25*	
2	*Виконання другого розділу та опис інформаційних технологій для виконання завдань роботи*	*01.03-01.05.25*	
3	*Виконання третього розділу, методика вирішення завдань та елементи наукових досліджень*	*01.05-01.07.25*	
4	*Виконання четвертого розділу, представлення результатів та їх узагальнення*	*01.05-01.07.25*	
5	*Написання розділу: «Охорона праці та безпека в надзвичайних ситуаціях»*	*01.07-01.09.25*	
6	*Завершення оформлення розрахунково-пояснювальної записки та презентаційних матеріалів*	*01.09-01.11.25*	
7	*Завершення роботи в цілому*	*01.11-01.12.25*	

Студент _____ Смуток В.В.
(підпис)

Керівник роботи _____ Смолінський В.Б.
(підпис)

УДК: 004.4:005.32:005.8

Кваліфікаційна робота: 73 с. текст. част., 29 рис., 1 табл., 14 слайдів, 20 джерел.

Методи та інструменти управління командною розробкою ІТ-проєктів на основі Agile. Смуток Василь Васильович. Кафедра ІТ. – Дубляни, Львівський НУВМБ, 2025.

Розглянуто теоретичні та практичні засади управління командною розробкою ІТ-проектів у контексті гнучких підходів до створення програмного забезпечення.

На основі аналізу методологій Agile, інструментів підвищення продуктивності команди та життєвого циклу ІТ-проекту обґрунтовано вимоги до системи підтримки командної взаємодії.

Описано технічні аспекти комунікації під час розробки ПЗ, вибір і застосування інструментарію, а також архітектуру системи командної розробки та управління ІТ-проектами.

Практичну частину присвячено реалізації моделі управління командною розробкою на основі реляційної бази даних MySQL, бібліотеки Hibernate, фреймворку Spring Boot та взаємодії з REST API засобами Postman.

Наведено результати перевірки коректності клієнт-серверної взаємодії, особливості обробки виключних ситуацій (<exception>) та представлено результати дослідження ефективності розробленого програмного засобу для підтримки командної роботи над ІТ-проектами.

Ключові слова: гнучка розробка, команда, комунікації, Agile, Scrum управління, програмний засіб, ІТ-проекти.

ЗМІСТ

ВСТУП

Управління командною розробкою ІТ-проектів в умовах динамічного ринку вимагає гнучких підходів, здатних швидко реагувати на зміни вимог, технологій та пріоритетів бізнесу. Класичні каскадні моделі дедалі частіше виявляються недостатньо ефективними, оскільки передбачають жорстке планування та складну адаптацію до змін на пізніх етапах життєвого циклу продукту. На цьому тлі Agile-технологія стала де-факто стандартом організації роботи команд розробників, пропонуючи ітеративність, прозорість процесів, тісну взаємодію з замовником і постійне вдосконалення продукту.

Разом із тим, впровадження Agile – це не лише зміна методології, а й комплексне завдання з управління командою, комунікаціями, плануванням та контролем якості. Від того, наскільки грамотно організовано спринти, розподілено ролі (Product Owner, Scrum Master, Development Team), налагоджено інструменти відстеження задач та зворотного зв’язку, залежить успіх усього ІТ-проекту. Тому дослідження підходів до управління командною розробкою в Agile-середовищі, вибору інструментарію та практик координації роботи команди є актуальним завданням, що має як теоретичне, так і прикладне значення для сучасної індустрії програмної інженерії.

Мета роботи – розробити та дослідити програмний засіб для підтримки командної розробки ІТ-проектів, побудованої за Agile-технологією, який забезпечує ефективну комунікацію між учасниками команди, інтеграцію з інструментами розробки та гнучке управління проектними складовими.

Завдання роботи:

- означити засади управління розробкою ІТ-проектів, підходи гнучкої розробки ПЗ та інструменти підтримки командної взаємодії в Agile-середовищі;
- розробити архітектуру системи командної розробки та управління ІТ-проектами, визначити структуру основних сутностей та модель їх взаємодії;

- реалізувати програмний засіб на базі Java, Spring Boot, Hibernate та MySQL з використанням REST API та інструментів для тестування й документування взаємодії (Postman);
- провести експериментальне дослідження ефективності розробленого програмного забезпечення, оцінивши продуктивність, коректність роботи та зручність застосування.

Об'єктом роботи є процес командної розробки та управління IT-проектами в умовах застосування гнучких методологій (Agile).

Предметом роботи є модель та програмний засіб управління командною розробкою IT-проектів, реалізований на базі Java, Spring Boot, Hibernate, MySQL та REST API, а також підходи до підтримки командної комунікації й інтеграції з інструментами розробки.

Новизна роботи:

- розроблено систему управління IT-проектами у вигляді окремого API, орієнтованого на потреби команди розробників, без прив’язки до складного графічного інтерфейсу;

- забезпечено модульну структуру (проекти, спринти, епіки, тікети, користувачі) з гнучким підключенням функціональних компонентів та взаємодією через REST-сервіси;

- експериментально досліджено ефективність розробленого програмного забезпечення, оцінено продуктивність, коректність роботи та зручність застосування.

РОЗДІЛ 1
АНАЛІЗ ОСНОВ УПРАВЛІННЯ РОЗРОБКОЮ ІТ-ПРОЕКТІВ

1.1. Аналіз підходів до гнучкої розробки програмного забезпечення

Гнучка розробка програмного забезпечення (Agile) належить до підходів, що базуються на ітеративному циклі створення продукту, коли вимоги та технічні рішення формуються й уточнюються у процесі постійної взаємодії між самоорганізованими кросфункціональними командами. Ключова ідея полягає в тому, що продукт не намагаються спроектувати “раз і назавжди” на старті, а розвивають його крок за кроком, отримуючи зворотний зв’язок і адаптуючись до змін.

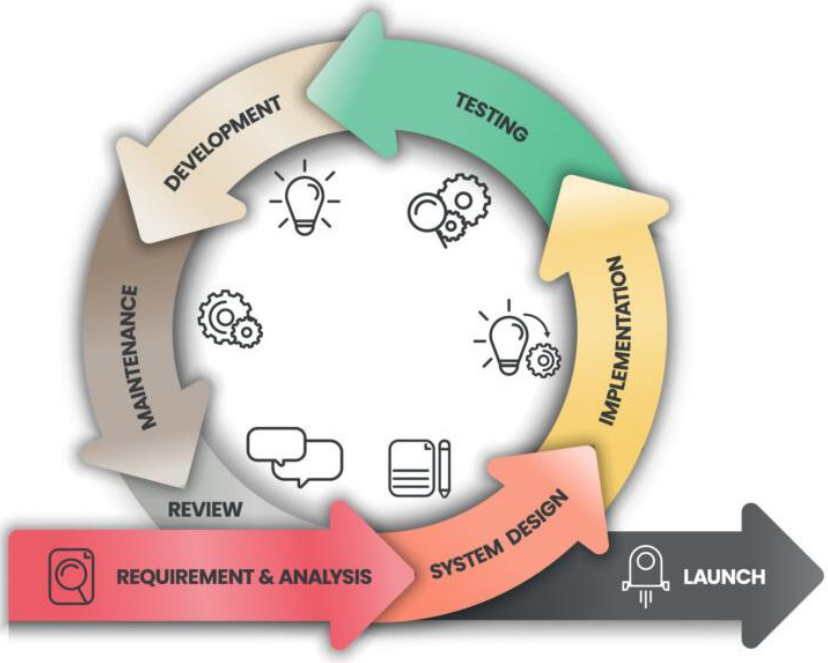

Рисунок 1.1 – Модель Agile [6, 7, 9]

Головна цінність Agile-підходу полягає в можливості швидше доставляти користувачеві прикладну цінність, підтримуючи при цьому належну якість, передбачуваність результатів та високу гнучкість у реагуванні на нові вимоги чи зміну пріоритетів. Серед найпоширеніших методологій сімейства Agile зазвичай виділяють Scrum та Kanban [6, 7, 9].

Scrum розглядають як одну з реалізацій *Agile* – це легковаговий процесний фреймворк для гнучкої розробки, який отримав найбільше поширення у практиці командної роботи. Під “process framework” розуміють набір обов’язкових практик та артефактів, дотримання яких забезпечує відповідність конкретного процесу заданій структурі. Наприклад, у Scrum передбачено використання фіксованих ітерацій розробки, що називаються спринтами, тоді як у XP (Extreme Programming) вимагається застосування парного програмування та низки інших інженерних практик.

Визначення Scrum як “легкого” фреймворку означає, що накладні витрати на процес (кількість формальних процедур, зустрічей, документації) залишаються мінімальними, щоб максимально збільшити частку реального продуктивного часу розробників, спрямованого на створення корисного функціоналу [6, 7, 9].

Процес Scrum відрізняється від інших agile-підходів тим, що спирається на набір специфічних понять і практик, згрупованих у три основні категорії: ролі, артефакти та часові «контейнери» (timeboxes). Кожна з цих груп описує, хто залучений до процесу, які робочі сутності використовуються (беклог, інкремент, тощо) та в яких часових рамках усе це відбувається (спринт, дейлі мітинги, рев’ю). Саме поєднання цих елементів робить Scrum чітко структурованим фреймворком, а не просто загальним підходом до гнучкої розробки. Найчастіше Scrum застосовують для управління розробкою складних програмних продуктів, де використовується ітераційний та інкрементальний підхід – система розвивається невеликими порціями функціоналу, які постачаються замовнику.

У порівнянні з класичними каскадними («waterfall») моделями, Scrum дає змогу значно скоротити час до появи перших відчутних результатів та підвищити загальну ефективність роботи команди.

Завдяки своїй гнучкій природі Scrum допомагає організаціям легше адаптуватися до динамічних вимог бізнесу, швидко реагувати на зміну

пріоритетів і випускати продукт, який краще відповідає актуальним цілям компанії.

Використання цього фреймворку позитивно впливає на декілька ключових аспектів: підвищується якість кінцевого результату, команда вчиться не лише справлятися зі змінами, а й очікувати їх як нормальну частину процесу, зменшуються витрати часу на оцінювання задач, водночас оцінки стають більш реалістичними, а прозорість стану проекту та контроль над графіком виконання робіт зростають.

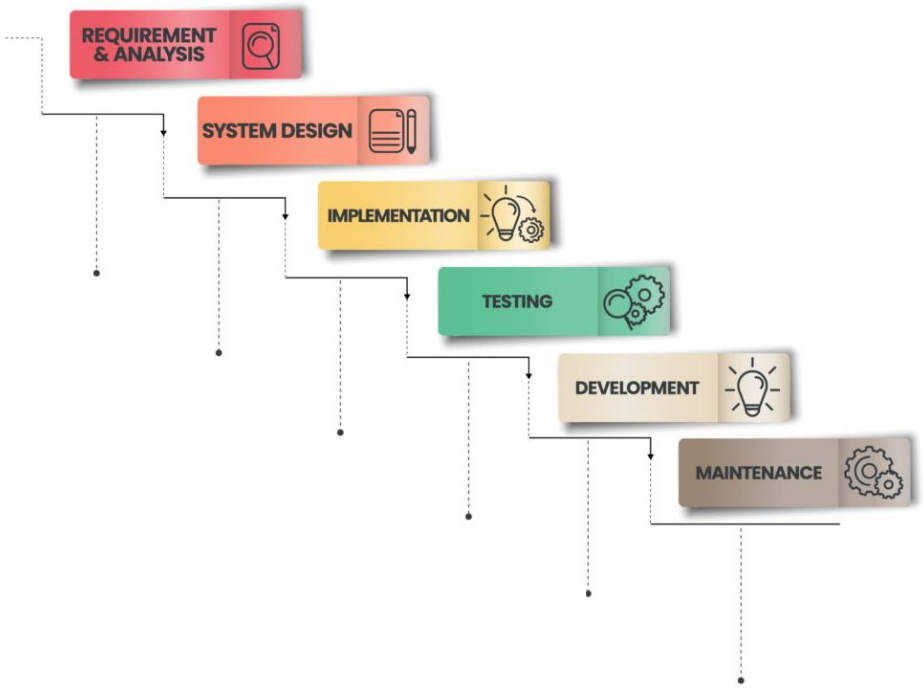

Рисунок 1.2 – Модель Waterfall [6]

Робота організовується у форматі невеликих кросфункціональних команд, які включають усіх необхідних спеціалістів – від аналітиків до розробників і тестувальників. Окрему роль відіграє Scrum-майстер – людина, яка стежить за дотриманням процесу, прибирає перешкоди для команди та підтримує здорову, конструктивну атмосферу співпраці.

Важливо, що Scrum не обмежується лише сферою розробки ПЗ. Він добре масштабується на інші типи продуктів – від стартап-проектів до маркетингових кампаній – і виступає передусім як управлінський фреймворк, а не повна методологія розробки. Це означає, що в реальних проектах Scrum зазвичай

доповнюють технічними та аналітичними практиками з інших гнучких підходів: наприклад, інженерними практиками з Extreme Programming (XP) або методами аналізу та моделювання вимог з підходу ICONIX. Функціональні вимоги при цьому розбиваються на невеликі, орієнтовані на користувача частини – user stories, які максимально слабо пов’язані між собою. Сукупність таких історій формує беклог продукту. Далі елементи беклогу впорядковуються за пріоритетом, а для кожної історії виконується відносна оцінка трудомісткості (наприклад, у story points), що дозволяє планувати обсяг робіт на спринт і візуалізувати ці оцінки у вигляді діаграм чи спеціальних артефактів (рис. 1.3).

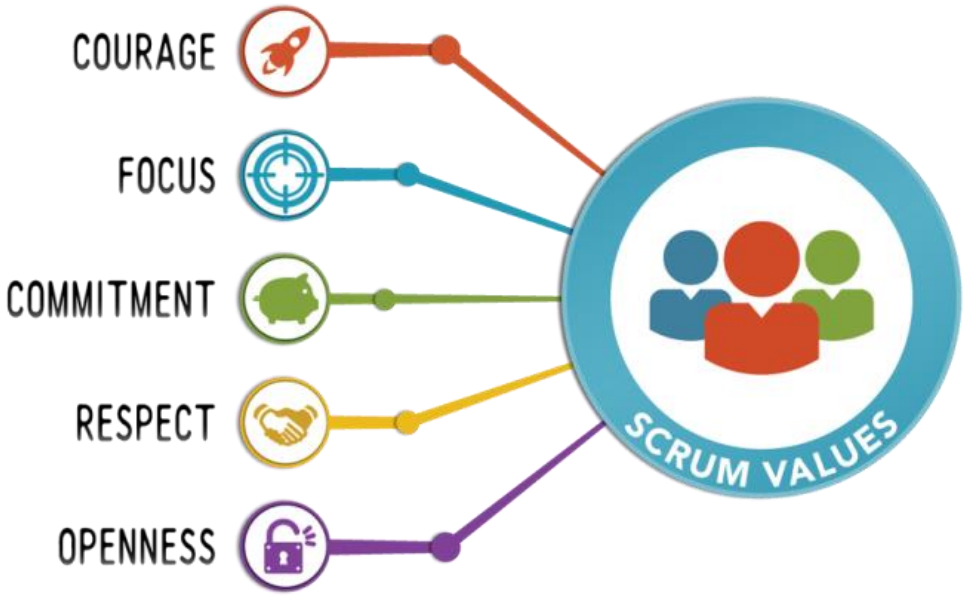

Рисунок 1.3 – Елементи Scrum [6, 7, 9]

Наприкінці кожного спринту команда проводить огляд результатів (Sprint Review), під час якого отримує зворотний зв’язок від власника продукту щодо реалізованого функціоналу, а також ретроспективу спринту (Sprint Retrospective), де аналізуються процеси всередині команди та визначаються шляхи їх оптимізації.

На основі побаченого власник продукту може скоригувати вимоги, переглянути пріоритети у беклозі та, за потреби, частково змінити напрямок розвитку продукту перед стартом наступного спринту.

У Scrum окремо виділяють роль власника продукту (Product Owner) – саме він відповідає за формування та актуальність вимог, їхню пріоритизацію та комунікацію з усіма стейкхолдерами. Робота команди організована у форматі коротких, фіксованих за тривалістю ітерацій – спринтів (від 1 до 4 тижнів), наприкінці кожної з яких створюється завершений інкремент продукту – готовий до потенційного релізу функціонал, який за потреби можна вивести на ринок.

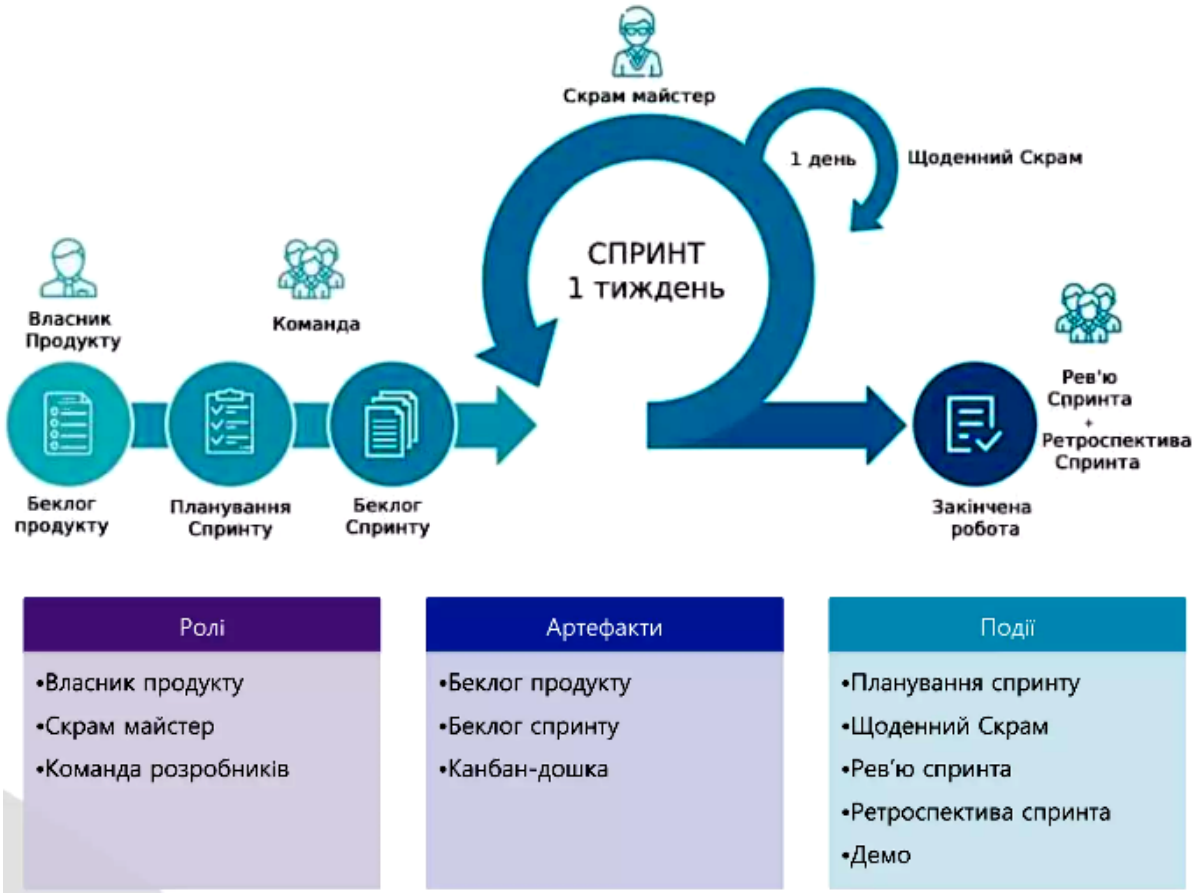

Рисунок 1.4 – Як працює Scrum [7]

Команда, орієнтуючись на свою середню швидкість (velocity), набирає у спринт стільки задач із беклогу, скільки реально здатна виконати в межах одного циклу. Щодня проводиться короткий Scrum-мітинг (daily stand-up), на якому учасники синхронізують поточний статус, окреслюють перешкоди та узгоджують подальші кроки. Упродовж спринту члени команди послідовно беруть у роботу елементи беклогу відповідно до визначених пріоритетів, що

забезпечує прозорість процесу та фокус на найбільш цінному функціоналі (рис.).

Scrum не є єдиним підходом у сімействі гнучких методологій, хоча зараз він і залишається найпоширенішим серед інших Agile-фреймворків. Водночас для ІТ-фахівця важливо розуміти не лише Scrum, а й інші підходи – їх сильні сторони, обмеження та типові сценарії застосування. Це дає змогу свідомо обирати інструменти під конкретний продукт або команду, а не намагатися “прикрутити” Scrum до будь-якого процесу. Крім того, на зрілих етапах впровадження Scrum-команди нерідко інтегрують окремі практики з інших методологій, наприклад, інженерні техніки з XP чи підходи до планування з Kanban, що дозволяє адаптувати процес під конкретні потреби проекту [2].

Рисунок 1.5 – Діаграма популярності гнучких методологій [6]

Окремої уваги заслуговують результати дослідження Agile Survey, у якому аналізується популярність різних гнучких підходів у реальних командах розробки (рис. 1.5). Такі опитування дають уявлення про те, які фреймворки найчастіше застосовуються на практиці, як змінюються тренди використання Scrum, Kanban, Lean, XP та інших методологій, а також допомагають оцінити, наскільки вибраний підхід відповідає сучасним галузевим тенденціям.

1.2. Аналіз інструментів для підвищення продуктивності команди IT-проектів

На сьогодні існує цілий набір програмних продуктів, створених спеціально для того, щоб спростити управління робочими процесами в командах різних типів. Спочатку ці системи виступали як інструменти для трекінгу задач і помилок, однак з часом еволюціонували у потужні платформи для управління роботою, які можна застосовувати в найрізноманітніших сценаріях – від менеджменту вимог та тестових сценаріїв до підтримки повного циклу agile-розробки програмного забезпечення.

Проаналізуємо найпопулярніші.

ClickUp – це один із найбільш відомих у світі інструментів підвищення продуктивності команд, який активно використовують компанії різного масштабу. Сервіс пропонує безкоштовний тариф із функціоналом, що включає призначені коментарі до задач, списки справ, чеклісти в рамках окремих завдань, розширене редагування тексту, багатозадачну панель керування, підтримку простих і користувацьких статусів, роботу зі спринтами, постановку та трекінг цілей тощо (рис. 1.6). Окремо відзначають якісний сервіс підтримки користувачів. Механізм ClickApps дозволяє глибоко налаштовувати логіку роботи команди в кожному окремому робочому просторі: за допомогою цієї можливості можна вмикати пріоритети, теги, кастомні поля та підлаштовувати конфігурацію простору під конкретний робочий процес.

FlexiProject – це комплексне та високотехнологічне рішення для управління проектами та портфелями, що забезпечує широкий спектр інструментів для планування та реалізації проектів будь-якої складності. Для команд проектного менеджменту платформа пропонує можливості організації роботи, тоді як для PMO вона стає інструментом впровадження стандартів управління процесами.

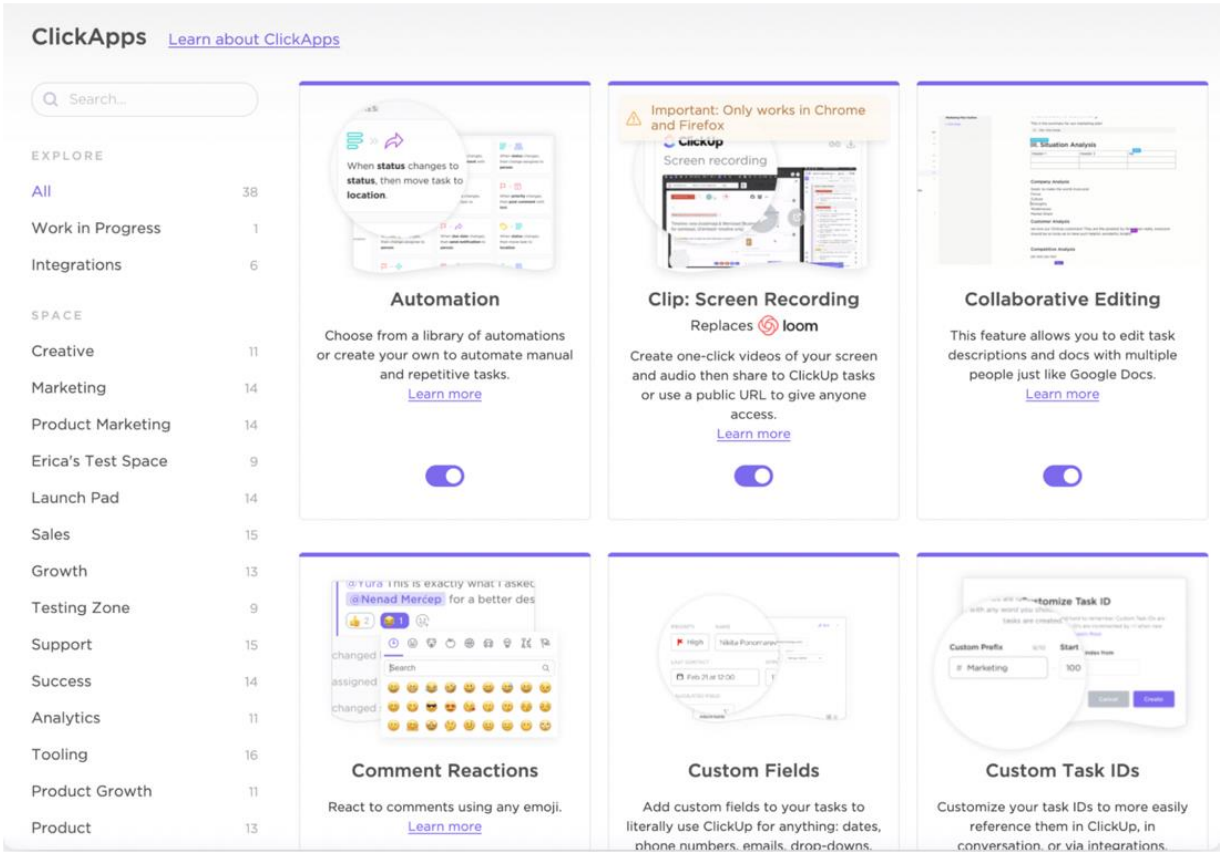

Рисунок 1.6 – Головна сторінка дошки ClickUp

Керівництво компанії отримує наочну та актуальну інформацію про стан поточних проектів. Центральним елементом системи є зручний та інтуїтивно зрозумілий графік, що включає діаграму Ганта та канбан-дошки.

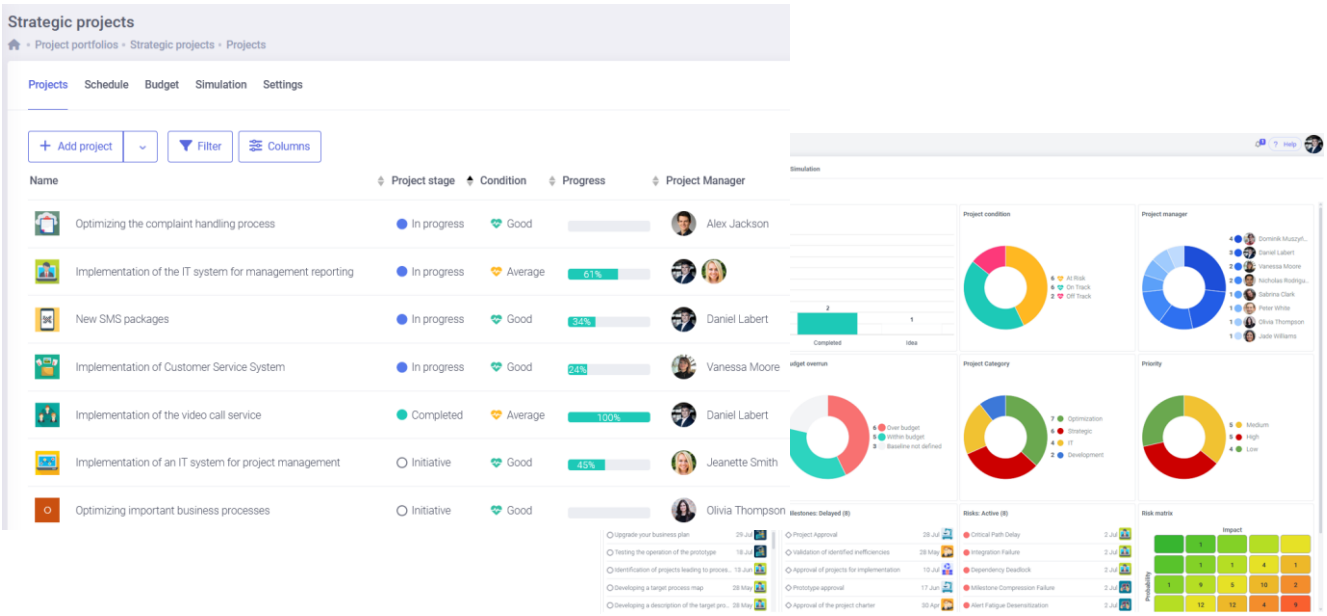

Рисунок 1.7 – Загальний вигляд дошки FlexiProject

Кожен проект у FlexiProject можна доповнити не лише календарним планом, а й комплексним фінансовим бюджетом, реєстром ризиків, набором продуктів та ресурсів, а також детальним планом управління проектом. Організаційний відділ управління проектами може впроваджувати стандарти, використовуючи гнучкі можливості налаштувань та конфігурації платформи.

Wrike є багатофункціональною платформою для управління проектами, яка дозволяє організовувати командні робочі простори та необмежену кількість проектів, завдань і підзадач. Інтерактивні форми запитів автоматизують введення даних і спрощують керування роботою, а структура платформи адаптується під потреби команди, дозволяючи розподіляти завдання з урахуванням продуктивності та можливостей співробітників.

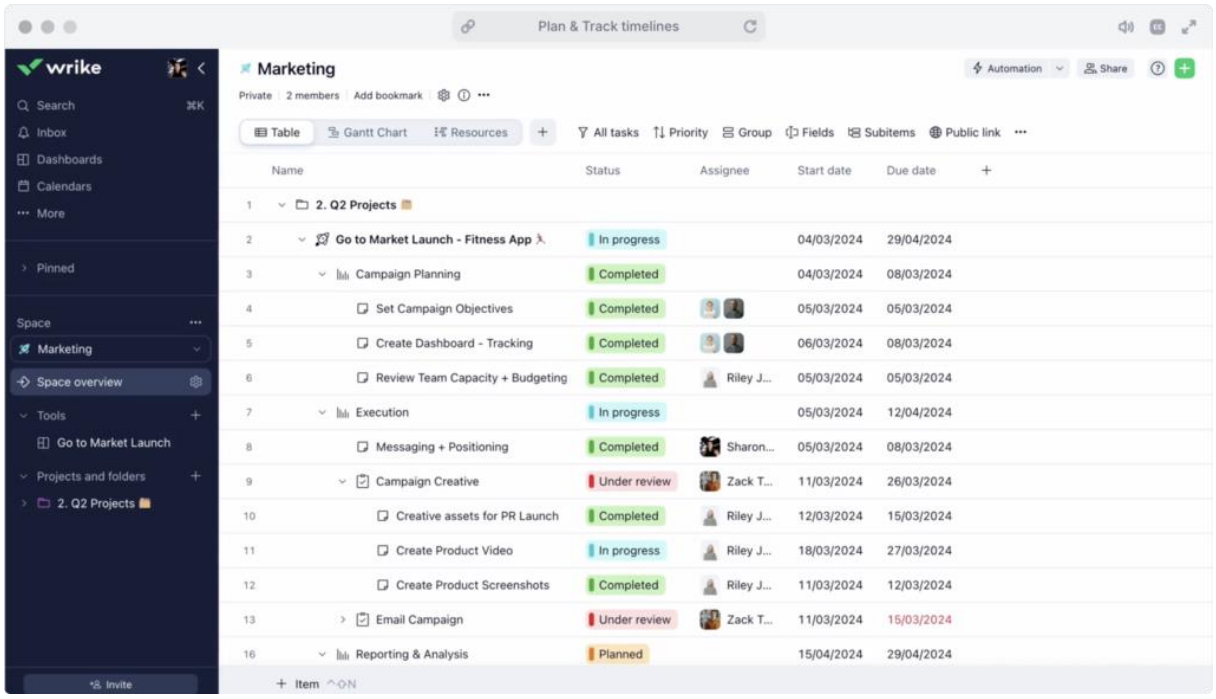

Рисунок 1.8 – Загальний вигляд дошки Wrike

Wrike підтримує міжкомандну співпрацю, керування проектами, просування стратегічних ініціатив та досягнення поставлених цілей. Платформа надає миттєвий доступ до актуальної інформації через дашборди в режимі реального часу, що сприяє прийняттю обґрунтованих рішень на основі даних. Wrike інтегрується з популярними сервісами, такими як Email, Zoom, Slack та MS Teams.

Planview дозволяє визначати пріоритетність проектів і пов'язувати їх зі стратегією компанії. Перевагою інструменту є можливість групувати проекти в портфелі та керувати ними, щоб швидше реалізовувати стратегію компанії. Це цінний додаток для онлайн-управління проектами. Крім того, це одне з найстаріших програм для управління проектами SaaS – перша версія з'явилася на ринку в 1997 році. Сьогодні це програмне забезпечення користується незмінною популярністю і в першу чергу орієнтоване на підтримку проектних команд, що працюють за гнучкими методологіями.

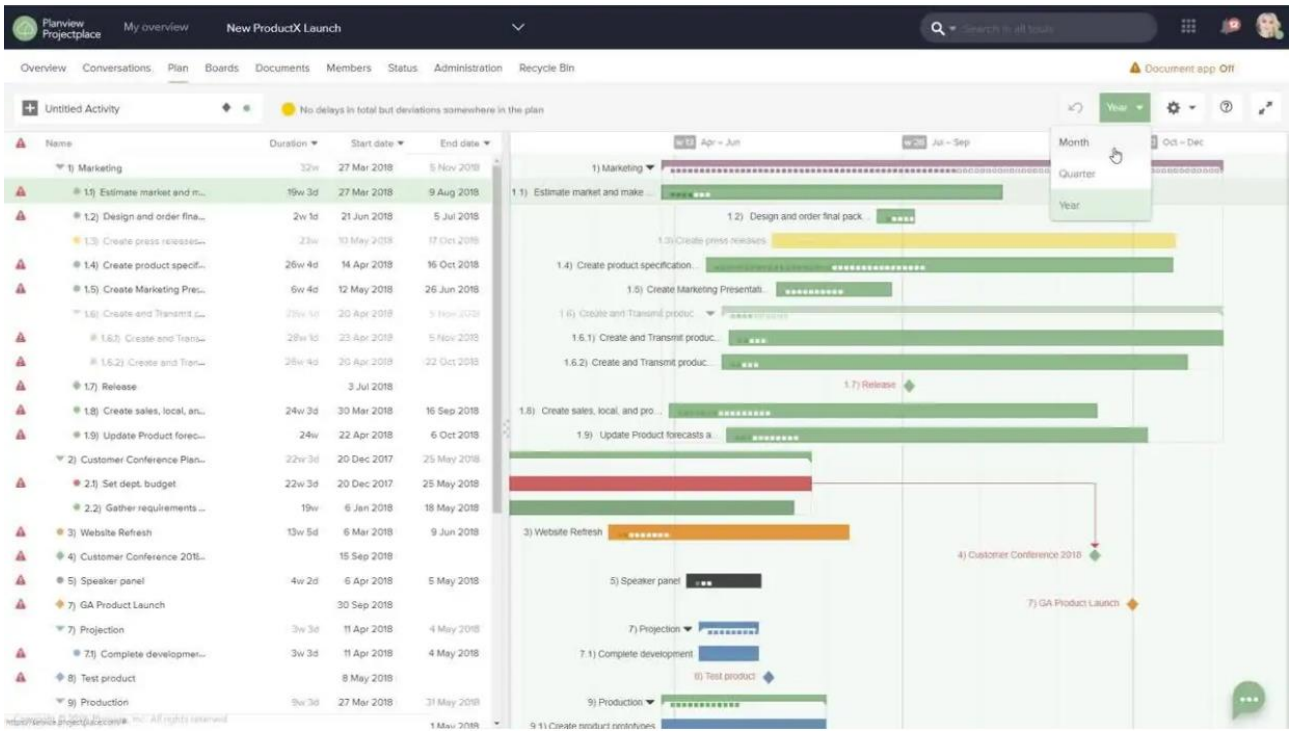

Рисунок 1.9 – Загальний вигляд дошки Planview

Asana здобула популярність завдяки зручності у керуванні проектами та оптимізації командної взаємодії. Платформа надає можливість працювати у декількох робочих просторах, додавати відповідальних і вкладення до завдань, відстежувати прогрес і спільно редагувати завдання в реальному часі. Користувачі також можуть бачити пріоритети колег, що підвищує прозорість роботи та сприяє ефективному управлінню проектами у гнучкому середовищі.

Clubhouse привертає увагу простим і зрозумілим інтерфейсом, який забезпечує корисний функціонал без надмірного навантаження. Основою

роботи є історія, у яку додаються квитки, помилки та завдання, формуючи логічну структуру проекту. Платформа також пропонує наочні діаграми, що допомагають відстежувати прогрес, оцінки та ризики, полегшуючи аналіз роботи команди.

ProofHub надає розширені можливості для управління проектами та командами, забезпечуючи централізоване джерело правди для завдань, комунікацій та процесів. Користувачі отримують гнучкість у роботі з завданнями завдяки простим спискам справ, адаптивним робочим процесам та канбан-дошкам. Додатково платформа пропонує діаграми Ганта для планування проектів і кращої візуалізації термінів.

Kanbanize дозволяє командам ефективно візуалізувати основні ініціативи та розбивати їх на конкретні робочі елементи через інтуїтивні канбан-дошки. Інструмент підтримує обмеження робочих елементів, фільтри, доступ на основі ролей і налаштовувані поля, що забезпечує зручне представлення та контроль прогресу роботи у будь-якому форматі, відповідно до потреб команди.

1.3. Життєвий цикл IT-проекту

Для того, щоб ідея перетворилася на життєздатний IT-продукт, першочергово необхідно чітко **окреслити її *концепцію***. Важливу роль у цьому процесі відіграє визначення реальних потреб замовника: досвідчені розробники не поспішають занурюватися у технічні деталі, доки не розуміють ключові проблеми бізнесу клієнта. Ідеї можуть надходити як для створення стартапу, так і як запити на «клонування» існуючих рішень конкурентів. Команда розробників активно взаємодіє із замовником через інтерв’ю, опитування та дослідження, щоб достовірно визначити бізнес-потреби та запропонувати оптимальні технічні рішення.

Після первинного формулювання концепції настає етап **оцінки доцільності та аналізу ринку**. Оцінка доцільності (*feasibility study*) охоплює

технічні, економічні та операційні аспекти і має на меті визначити реальну життєздатність ідеї. Одночасно проводиться ринкове дослідження, яке дозволяє зрозуміти цільову аудиторію, конкурентне середовище та актуальні тренди галузі.

Етап ініціалізації передбачає планування та визначення стратегії розвитку продукту. На цьому етапі встановлюються ключові вимоги проекту, його часові рамки та бюджет. Формується первинна структура проекту: розподіляються ролі в команді, обираються технології та платформи, створюється команда розробників, а також формується дорожня карта продукту.

На основі зібраної аналітики визначаються конкретні та вимірювані **бізнес-цілі проекту**, які часто структурують за методологією *SMART*. Вони мають бути конкретними (*Specific*), вимірюваними (*Measurable*), досяжними (*Achievable*), актуальними (*Relevant*) та обмеженими в часі (*Time-bound*). Хоча цей крок може здатися формальним, саме чітко сформульовані цілі визначають подальший напрямок управління IT-проектом і впливають на всі наступні етапи його реалізації.

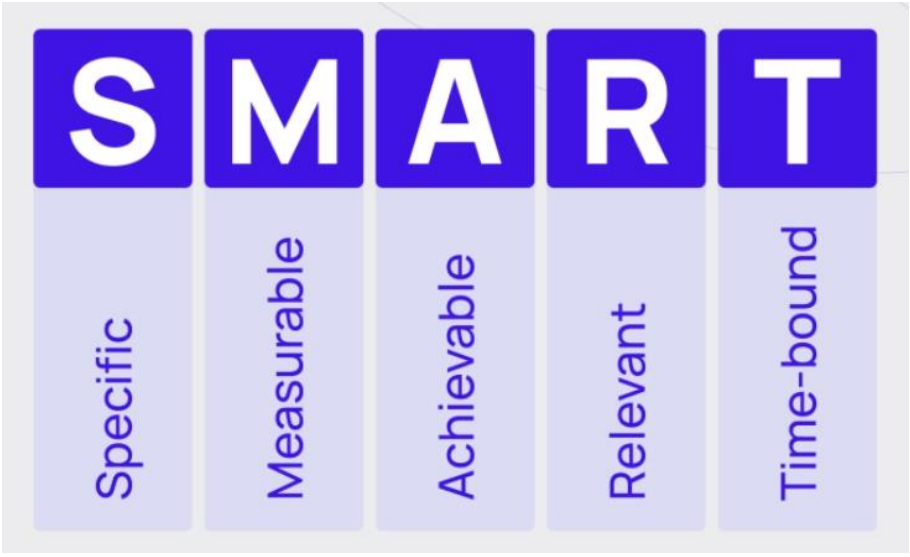

Рисунок 1.10 – Методологія SMART

Відповідно до поставлених цілей визначаються реалістичні терміни виконання кожного етапу розробки, а також всього проекту в цілому. На основі

оцінки робочого часу формується бюджет, який охоплює всі необхідні витрати: оплату праці команди, ліцензії на програмне забезпечення, придбання обладнання, маркетингові активності та інші ресурси, необхідні для успішного виконання проекту.

Етап розробки передбачає безпосереднє створення програмного забезпечення – той момент, коли ідеї та концепції втілюються у працюючий продукт за допомогою коду. Технічна реалізація IT-проекту має власний життєвий цикл, який організовується розробниками відповідно до принципів *SDLC* (*Software Development Life Cycle*), забезпечуючи структурований та контрольований процес створення програмного продукту.

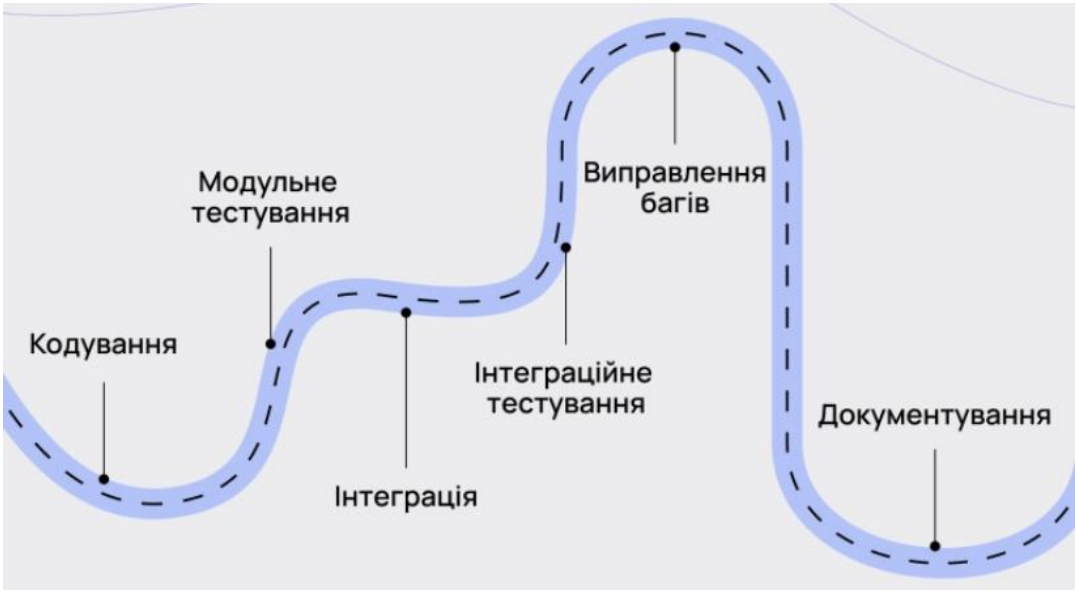

Рисунок 1.11 – Життєвий цикл розробки ПЗ

Цей підхід ґрунтується на принципах гнучкості та ітеративності, і в рамках нього можна виділити кілька ключових стадій. Спочатку відбувається кодування, коли програмісти реалізують функціонал відповідно до технічного завдання. Потім проводиться модульне тестування, що дозволяє перевірити правильність роботи окремих компонентів коду. Далі відбувається інтеграція, коли всі модулі та компоненти об’єднуються в єдину систему, після чого виконується інтеграційне тестування для перевірки взаємодії між ними. Виявлені помилки (баґи) усуваються на етапі виправлення дефектів, а паралельно створюється технічна та користувацька документація.

Реліз – є підсумком тривалого процесу планування та розробки. Якщо попередні етапи були успішними, продукт виходить на ринок у готовому та стабільному стані, відповідає вимогам і не містить критичних помилок, що можуть знизити його цінність для бізнесу.

Щоб продукт потрапив до кінцевих користувачів, його розгортають у **продакшн-середовищі** (*production environment*), яке включає сервери, бази даних, мережеве обладнання та іншу інфраструктуру.

Підготовка до розгортання передбачає налаштування серверів і баз даних, оптимізацію продуктивності системи під реальним навантаженням, забезпечення безпеки через шифрування, брандмауери та системи виявлення вторгнень, а також налаштування моніторингу та логування для відстеження стану продукту та швидкого реагування на проблеми.

Підтримка та масштабування. Після запуску робота над продуктом не припиняється: життєвий цикл переходить у фазу підтримки та масштабування, спрямовану на довгострокову стабільність, актуальність і ефективність ПЗ. Підтримка передбачає оцінку продуктивності, оперативне реагування на інциденти та надання користувачам необхідної допомоги. Для цього команда працює за кількома напрямками, забезпечуючи безперервну роботу, оновлення та вдосконалення продукту:

- **Моніторинг продуктивності:** відстеження ключових показників роботи системи, таких як швидкодія, час відгуку, використання обчислювальних ресурсів та інші критично важливі метрики. Це дозволяє своєчасно виявляти «вузькі місця» та оперативно усувати їх, забезпечуючи стабільну роботу продукту.

- **Аналіз поведінки користувачів:** застосування аналітичних інструментів для дослідження сценаріїв взаємодії користувачів із продуктом – які функції вони використовують найчастіше, де виникають труднощі та як покращити робочий процес.
- **Збір відгуків щодо UX:** опитування, UX-тестування та аналіз звернень до служби підтримки. Ці дані допомагають виявляти проблемні аспекти взаємодії користувачів із продуктом і знаходити ефективні рішення для його вдосконалення.

Документація та аналіз результатів є невід'ємною складовою збереження та передачі досвіду в IT-проектах. Внутрішня технічна документація охоплює архітектуру системи, бізнес-логіку, код та API, забезпечуючи зрозуміле відображення роботи продукту. За підсумками проекту формуються загальні та проектні звіти, технічні документи, користувацькі гайди та довідники. Якісне ведення документації гарантує безперервність розробки та значно спрощує підтримку, оновлення та адаптацію нових членів команди в майбутньому.

Рисунок 1.12 – Етапи після-проектної аналітики

Після-проектна аналітика (*Post-Mortem Analysis*) є ключовим інструментом для вдосконалення процесів та продуктів. Вона включає оцінку відповідності проекту початковим бізнес-цілям і вимогам, аналіз дотримання термінів і бюджету, а також причин можливих відхилень. Оцінюється якість продукту щодо продуктивності, надійності та користувацького досвіду, а також ефективність роботи команди, її сильні та слабкі сторони у взаємодії та комунікації. Вимірюється економічна ефективність проекту через ROI (*Return on Investment*), порівнюючи витрати з отриманими результатами.

На основі отриманих даних формуються висновки та рекомендації для майбутніх проектів: успішні практики зберігаються і масштабуються, а для процесів, інструментів, комунікації та управління ризиками розробляються конкретні пропозиції щодо вдосконалення.

Цей етап завершує життєвий цикл управління проектом, перетворюючи накопичений досвід у цінні уроки, що сприяють постійному розвитку команди та підвищенню якості майбутніх IT-продуктів.

РОЗДІЛ 2
МЕТОДОЛОГІЯ ТА ІНСТРУМЕНТАРІЙ КОМАНДНОЇ КОМУНІКАЦІЇ В ІТ-ПРОЕКТАХ

2.1. Технічні аспекти командної комунікації під час розробки програмного забезпечення

З технічної точки зору програмне забезпечення (ПЗ) для підтримки командної комунікації в процесі розробки ПЗ розглядається не як звичайний месенджер, а як інтегрований елемент інженерної екосистеми, тісно пов’язаний з інструментами розробки, деплойменту та підтримки продукту. Архітектура таких систем має забезпечувати подієву модель обміну даними, роботу в режимі реального часу, керування доступом до артефактів розробки, підтримку масштабованості та надійності, а також високий рівень інформаційної безпеки.

Окреме значення мають можливості розширення через API та бот-платформи, вбудований пошук і механізми збереження знань, а також інтерфейс, оптимізований під потреби розробників. Сукупність цих технічних характеристик визначає здатність комунікаційного інструмента органічно вбудовуватися в життєвий цикл програмного продукту та підтримувати сучасні підходи до управління розробкою, зокрема Agile і DevOps.

По-перше, **інтеграції**: сучасний інструмент для комунікації розробників зазвичай зав’язаний із системами контролю версій (GitHub, GitLab, Bitbucket), таск-трекерами (Jira, Trello, YouTrack, Azure DevOps), CI/CD (Jenkins, GitHub Actions, GitLab CI). Технічно це API-інтеграції та вебхуки: коміт, pull request або фейл білда автоматично породжує повідомлення в канал команди. Завдяки цьому комунікація стає подієвою: не людина бігає за статусами, а система сама «штовхає» релевантну інформацію туди, де її побачать.

По-друге, **структура даних і модель прав доступу**: у будь-якому командному месенджері/хабі є сутності на кшталт робочих просторів, каналів, тредів, згадок, реакцій. З точки зору архітектури це ієрархія об’єктів з ролями

та політиками доступу (адмін, учасник, гість), плюс окреме управління доступом до інтегрованих ресурсів (репозиторії, борди завдань, документація). Для розробників важливо, щоб система дозволяла прив'язувати обговорення до конкретних артефактів – таска, гілки, коміту, релізу.

По-третє, **реальний час і масштабованість**: WebSocket або інші механізми постійних підключень (long polling / Server-Sent Events), черги повідомлень, механізми fan-out, кеші та шардінг. Для розподілених команд із десятками/сотнями учасників критичні: низька затримка доставки повідомлень, коректна синхронізація статусів (прочитано/непрочитано), стійкість до відмов і горизонтальне масштабування.

По-четверте, **пошук і зберігання знань**: повнотекстовий пошук (індексація повідомлень, вкладень, посилань на таски), іноді з елементами семантичного пошуку. По суті, хороший інструмент комунікації перетворюється на внутрішню «базу знань», де рішення збережені у вигляді тредів, а не «загубилися в усній розмові». Тут же можна згадати інтеграцію з wiki/документацією (Confluence, Notion, GitHub Wiki).

По-п'яте, **безпека та відповідність політикам**: це шифрування трафіку (TLS), шифрування даних у сховищі, політики retention (скільки зберігаються логи/повідомлення), аудит дій користувачів, SSO/SAML, інтеграція з системами управління обліковими записами (LDAP, Azure AD). У командній розробці часто обговорюються конфіденційні речі – архітектурні рішення, вразливості, ключі доступу, тому технічні вимоги до безпеки тут не формальність.

По-шосте, **підтримка DevOps-культури**: інструмент комунікації стає «нервовою системою» DevOps-процесів: нотифікації про викат релізу, деплой, алерти з моніторингу, інциденти, статуси CI/CD. Технічно це купа інтеграцій з monitoring & alerting (Prometheus, Grafana, New Relic, Sentry тощо), які шлють структуровані повідомлення в конкретні канали. Звідси народжується chatops – коли дії з інфраструктурою виконуються через чат-ботів (наприклад, запустити деплой або перезапустити сервіс командою в чаті).

По-сьоме, **розширюваність**: бот-платформа та API. Добре, якщо ПЗ для комунікації надає SDK і REST/GraphQL API для створення власних ботів, плагінів, інтеграцій. Тоді команда може «прикрутити» автоматичне створення тасків із повідомлень, автогенерацію проектних звітів, нагадування про дедлайни спринту, стендапи тощо.

І нарешті – **UX для розробника**: підтримка клавіатурних шорткатів, форматування коду (markdown, підсвічування синтаксису), зручна робота з посиланнями на репозиторії, таски та білди, нормальний пошук по історії – усе це теж технічні вимоги, просто вони лежать на межі інженерії та дизайну.

2.2. Особливості використаного інструментарію для розробки

Для виконання завдань кваліфікаційної роботи, щодо розробки відповідного програмного засобу нами поєднано відповідний інструментарій, а саме – Maven та Java у рамках Spring Boot. Таке технічне рішення є надзвичайно практичним і доцільним з точки зору ефективності розробки, управління залежностями та масштабованості проекту.

За такої конфігурації Maven виступає як система автоматизації збірки та управління залежностями, що дозволяє розробнику легко інтегрувати всі необхідні бібліотеки та модулі Spring Boot через starter-пакети, забезпечуючи стандартизовану структуру проекту та відтворюваність збірок у різних середовищах.

Всі програми Spring Boot конфігуруються від **spring-boot-starter-parent**, тому перед подальшим визначенням залежностей додається **starter-parent** наступним чином:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.1.RELEASE</version>
</parent>
```

Завдяки декларативній конфігурації у файлі pom.xml автоматично завантажуються потрібні версії бібліотек, плагіни для компіляції, тестування та розгортання, що значно скорочує ручну роботу та знижує ймовірність конфліктів залежностей.

Java як мова програмування забезпечує стабільну і портативну платформу для реалізації бізнес-логіки та компонентів Spring Boot. Використання Java дозволяє ефективно застосовувати об’єктно-орієнтовані принципи, розробляти модульний та легко тестований код, а також інтегрувати різні сервіси і фреймворки (JPA, Hibernate, Spring Security). У поєднанні з Maven це забезпечує повністю автоматизований життєвий цикл розробки: від компіляції та виконання юніт-тестів до пакування і розгортання готового програмного засобу. Така інтеграція робить процес розробки більш передбачуваним, контрольованим і масштабованим, що особливо важливо при створенні складних програмних систем або мікросервісів у командних проектах.

Maven – це інструмент автоматизації збірки та управління проектами для Java та сумісних мов програмування [9, 15]. Він дозволяє централізовано описувати структуру проекту, керувати залежностями, процесом компіляції, тестуванням, пакуванням і розгортанням.

Основні характеристики Maven: п*о-перше,* він використовує декларативний підхід: конфігурація проекту задається у файлі pom.xml (Project Object Model), де визначаються залежності, плагіни та цілі збірки.

По-друге, Maven автоматично завантажує бібліотеки та плагіни з центрального репозиторію, спрощуючи керування зовнішніми залежностями.

По-третє, він підтримує життєвий цикл збірки, який включає етапи компіляції, тестування, пакування та деплойменту, а також інтеграцію з CI/CD-системами.

Maven широко використовується в корпоративній розробці, оскільки

стандартизує процеси збірки, спрощує керування складними проектами з великою кількістю модулів і залежностей, а також забезпечує відтворюваність збірок у різних середовищах.

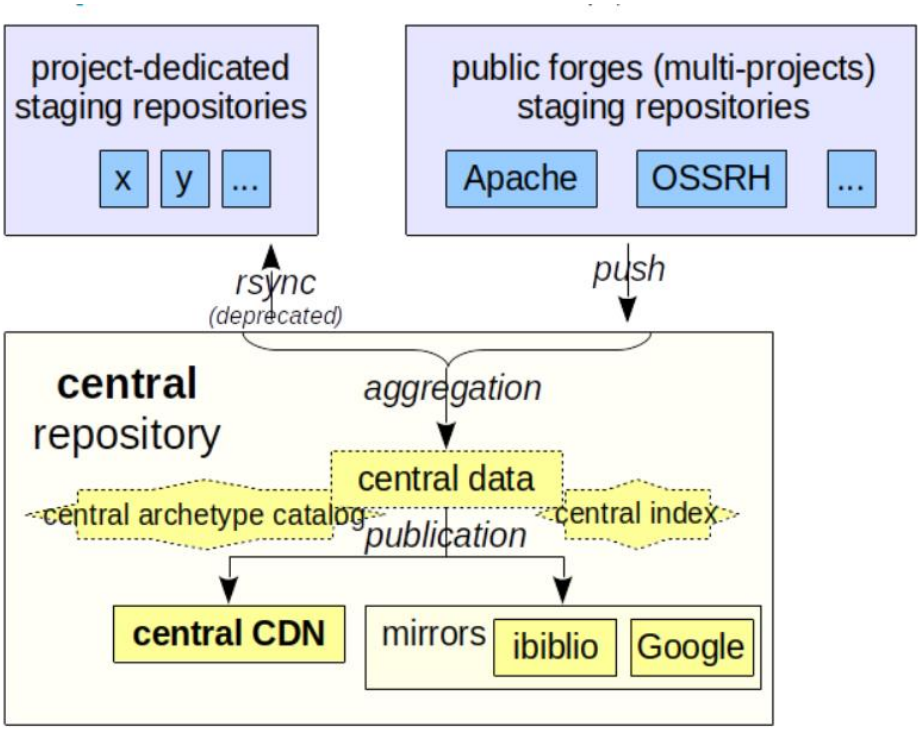

Рисунок 2.1 – Репозиторії та залежності Maven

Java – це високорівнева, об’єктно-орієнтована мова програмування, розроблена компанією Sun Microsystems (тепер Oracle) на початку 1990-х років. Вона призначена для створення переносимих, надійних і масштабованих програм, які можуть виконуватися на різних платформах без змін коду завдяки принципу «write once, run anywhere» (WORA).

Технічно Java працює через Java Virtual Machine (JVM), яка інтерпретує байт-код – проміжну форму програми, згенеровану компілятором (рис. 2.2). Це дозволяє виконувати однаковий код на будь-якій операційній системі, де встановлена JVM. Java підтримує об’єктно-орієнтовані концепції: класи, об’єкти, наслідування, поліморфізм і інкапсуляцію, а також має багатий стандартний API для роботи з колекціями, потоками, мережею, базами даних, графічними інтерфейсами та іншим [10].

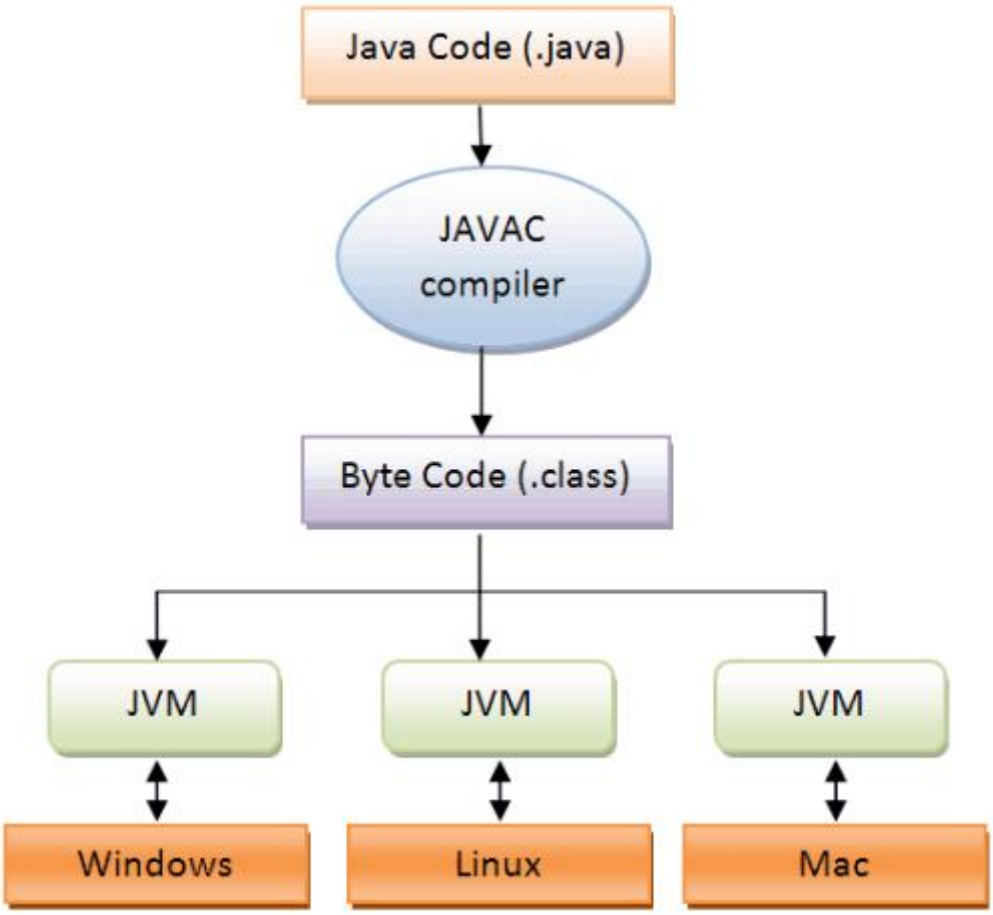

Рисунок 2.2 – Архітектура віртуальної машини Java [10]

Java активно використовується у розробці серверних додатків, веб-сервісів, мобільних додатків (Android), корпоративних систем, а також у хмарних і розподілених рішеннях, завдяки стабільності, великій спільноті та наявності численних фреймворків (Spring, Hibernate, JavaFX тощо).

Spring Boot — це фреймворк для розробки Java-додатків, який спрощує створення автономних, готових до продакшну сервісів і мікросервісів [17]. Він базується на екосистемі Spring і надає конфігурацію за замовчуванням, що дозволяє розробникам зосередитися на бізнес-логіці, а не на налаштуванні інфраструктури.

Технічно Spring Boot автоматично підключає необхідні залежності, налаштовує сервери (наприклад, вбудований Tomcat), бази даних, безпеку та інші компоненти через принцип «starter dependencies». Він підтримує інверсію керування (IoC), впровадження залежностей (Dependency Injection) і

модульність, що спрощує тестування і масштабування.

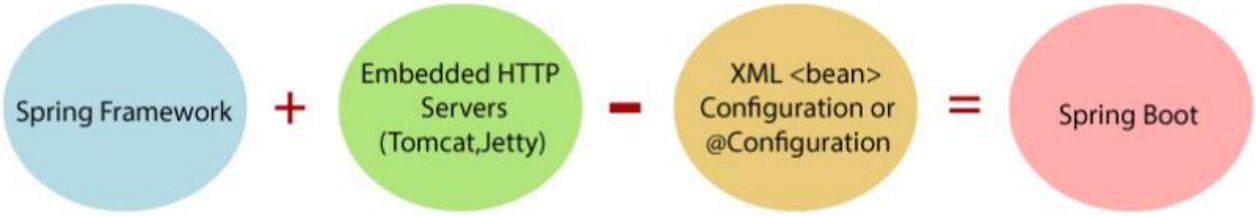

Рисунок 2.3 – Екосистема Spring Boot

Крім того, Spring Boot забезпечує інтеграцію з різними фреймворками і сервісами, такими як JPA/Hibernate для роботи з базами даних, Spring Security для аутентифікації та авторизації, а також RESTful API для побудови веб-сервісів.

Цей фреймворк широко використовується для розробки мікросервісів, корпоративних веб-додатків та хмарних сервісів, оскільки скорочує час на конфігурацію і дозволяє швидко запускати стабільні і масштабовані рішення.

Отже, Maven є інструментом для автоматизації збирання проектів Java, а Spring Boot — це фреймворк, що використовує Java для спрощення розробки веб-застосунків. Maven керує залежностями (бібліотеками) та процесом збірки проекту, тоді як Spring Boot надає автоматичну конфігурацію та вбудовані сервери, що робить його зручним для розробки на Java. В одному проекті Spring Boot вони використовуються разом, де Maven керує проектом, а Java є мовою розробки.

Методологія їх взаємодії та спільної роботи полягає в наступному [10, 15, 17]:

1. **Створення проекту**: створюється новий проект Spring Boot за допомогою IDE, використовуючи Maven для керування його структурою.
2. **Конфігурація**: налаштовується файл *pom.xml*, додаючи батьківську залежність (*spring-boot-starter-parent*) та специфічні стартер-залежності для проектованого додатка.
3. **Розробка**: пишеться Java-код, використовуючи анотації Spring Boot, такі як @SpringBootApplication та @RestController, для створення логіки

додатка.

4. **Збірка**: використовуються команди Maven (наприклад, *mvn clean install*) для компіляції коду та створення виконуваного JAR-файлу.

5. **Запуск**: запускається створений JAR-файл, який запускає вбудований сервер Spring Boot.

2.3. Архітектура системи командної розробки та управління IT-проектами

Унікальність запропонованої системи командної розробки та управління проектами полягає в тому, що вона реалізована у вигляді окремого програмного інтерфейсу (API). Такий підхід надає розробникам зручний спосіб доступу до всього функціоналу системи без потреби взаємодіяти зі складним графічним інтерфейсом. Це дає змогу швидко та ефективно інтегрувати засоби управління проектами у власні програмні рішення. При цьому розробники отримують високу гнучкість у виборі необхідних інструментів: вони можуть використовувати лише ті методи API, які відповідають конкретним вимогам їхнього застосунку, і не задіювати зайвий функціонал. Це сприяє раціональному використанню ресурсів та підвищує ефективність роботи із системою командного управління IT-проектами.

Реалізація системи управління проектами як окремого API також позитивно впливає на швидкість розробки програмного забезпечення. Розробники мають можливість спиратися на вже готові модулі та функції системи управління проектами, уникаючи дублювання реалізації типових механізмів і зосереджуючись на специфічних задачах свого продукту. Це скорочує тривалість циклу розробки та пришвидшує досягнення цілей проекту. У межах запропонованої системи управління передбачено подальше розширення функціональних можливостей, що в перспективі дозволить ще якісніше підтримувати процес планування, контролю та координації робіт у

межах програмних проектів.

Spring Security – це складова екосистеми Spring, що забезпечує конфігурацію доступу та процесів автентифікації й авторизації, відіграючи ключову роль у побудові безпечних програмних рішень.

Підсистема звітності в системах управління проектами є важливим інструментом контролю та оцінювання прогресу. Вона передбачає збір, обробку та подання даних про ключові параметри проекту, що дає змогу керівникам і стейкхолдерам своєчасно отримувати необхідну інформацію для ухвалення управлінських рішень і коригування планів.

Механізм ticket linking (зв'язок між тікетами) використовується для встановлення відношень між основним тікетом і підтікетом (sub ticket). Головний тікет відображає базовий запит або задачу, тоді як підтікет представляє окреме, пов'язане підзавдання, яке потребує самостійного опрацювання, але логічно належить до основної задачі.

Підтримка коментарів для різних об'єктів системи – задач, спринтів, проектної документації, змін, помилок тощо – дає змогу формувати контекстні обговорення та фіксувати комунікацію безпосередньо в межах конкретного елемента проекту.

Можливість відмічати користувачів у коментарях спрощує взаємодію між учасниками команди, особливо в ситуаціях, коли повідомлення містять значний обсяг інформації або стосуються вузькоспеціалізованих питань, що потребують реакції конкретних фахівців.

Функція фільтрації за періодом для спринтів, епіків і тікетів дозволяє швидко віднаходити й переглядати записи, які були створені, змінені або закриті в межах визначеного проміжку часу, тим самим полегшуючи аналіз динаміки роботи над проектом.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ МОДЕЛІ УПРАВЛІННЯ КОМАНДНОЮ РОЗРОБКОЮ ІТ-ПРОЕКТІВ

3.1. Опис структури функціональних кейсів системи

Архітектура програмного забезпечення є фундаментом будь-якої успішної програмної системи, оскільки визначає її зручність підтримки, можливості масштабування, стабільність та рівень захищеності протягом усього життєвого циклу. Одним із перших кроків під час упровадження нової системи є побудова архітектурної діаграми [2, 5, 15].

Базова функціональність розроблюваного програмного забезпечення передбачає наявність таких основних модулів.

Клас Project (Проект) надає користувачам змогу створювати нові проекти та налаштовувати їхні параметри, вводячи ключову інформацію про проект на етапі ініціалізації.

Клас User (Користувач) відповідає за реєстрацію, ідентифікацію та керування правами доступу користувачів до ресурсів і можливостей системи, автоматизуючи призначення ролей.

Клас Epic (Епік) описує великі функціональні блоки або області системи, які за потреби можуть деталізуватися на менші завдання.

Клас Sprint (Спринт) використовується для подання ітерацій розробки тривалістю, як правило, від одного до чотирьох тижнів; новий спринт ініціюється після завершення попереднього або на старті проекту.

Клас Ticket (Тікет) призначений для фіксації окремих робіт: завдань, дефектів, побажань чи інших активностей, що потребують уваги команди.

Клас Comments (Коментарі) забезпечує ведення текстових обговорень у межах проекту, дозволяючи учасникам команди залишати зауваження, ставити запитання та документувати робочі процеси.

Для відображення архітектури розробленої системи використано *UML*

(Unified Modeling Language) діаграму класів як уніфікований засіб моделювання [5, 18]. Діаграми класів є ключовим елементом об’єктноорієнтованого проектування: вони демонструють класи системи, їх атрибути, операції та взаємозв’язки. У типових засобах моделювання клас подається у вигляді трисекційної структури – з назвою, списком атрибутів та переліком методів.

Розроблена діаграма класів (рис. 3.1) описує систему керування проектами, спринтами, епіками, тікетами, користувачами та коментарями.

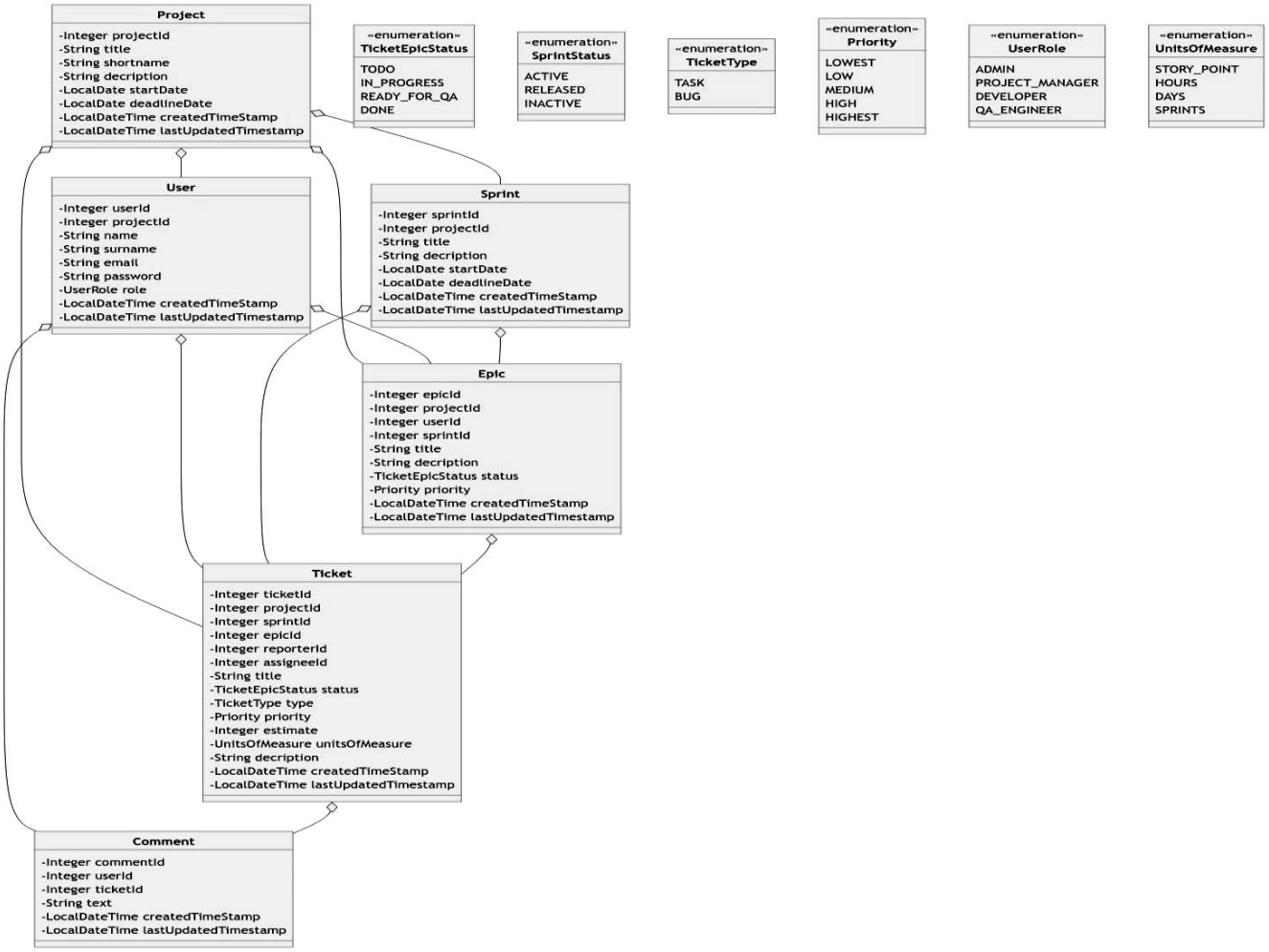

Рисунок 3.1 – Зв’язки UML-діаграми класів

Клас Project (Проект) має зв’язки типу «один-до-багатьох» із класами *User* (Користувач), *Sprint* (Спринт), *Epic* (Епік) та *Ticket* (Тікет) і містить такі атрибути, як *projectId* (ідентифікатор проекту), *title* (назва), *shortname*

(скорочене позначення), *description* (опис), *startDate* (дата початку), *deadlineDate* (кінцева дата), *createdTimeStamp* (час створення) та *lastUpdatedTimestamp* (час останнього оновлення).

Клас User (Користувач) пов'язаний відношенням «один-до-багатьох» з класами *Epic* (Епік), *Ticket* (Тікет) та *Comment* (Коментар). До його атрибутів належать *userId* (ідентифікатор користувача), *projectId* (ідентифікатор пов'язаного проекту), *name* (ім'я), *surname* (прізвище), email (адреса електронної пошти), *password* (пароль), role (роль користувача), *createdTimeStamp* (час створення запису) та *lastUpdatedTimestamp* (час останнього оновлення). Перерахування *UserRole* (Роль користувача) визначає можливі ролі, які може мати користувач у системі.

Клас Sprint (Спринт) має зв'язок «один-до-багатьох» з класами *Epic* (Епік) та *Ticket* (Тікет). Він містить атрибути *sprintId* (ідентифікатор спринту), *projectId* (ідентифікатор пов'язаного проекту), *title* (назва), *description* (опис), startDate (дата початку), *deadlineDate* (дата завершення), *createdTimeStamp* (час створення) та *lastUpdatedTimestamp* (час останньої модифікації). Перерахування *SprintStatus* (Статус спринту) задає можливі стани спринту в процесі його виконання.

Клас Epic (Епік) перебуває у відношенні «багато-до-одного» з класами *Project* (Проект), *User* (Користувач) та *Sprint* (Спринт). Він включає такі основні поля: *epicId* (ідентифікатор епіка), *projectId* (ідентифікатор пов'язаного проекту), *userId* (ідентифікатор користувача), *sprintId* (ідентифікатор спринту), *title* (назва), *description* (опис), *status* (статус), *priority* (пріоритет), *createdTimeStamp* (час створення) та *lastUpdatedTimestamp* (час останнього оновлення). Перерахування *TicketEpicStatus* (Статус епіка та завдання) визначає допустимі стани епіка, а перерахування Priority (Пріоритет) задає можливі рівні пріоритетності.

Клас Ticket (Тікет/Завдання) має зв'язки типу «багато-до-одного» з класами Project (Проект), *Sprint* (Спринт), Epic (Епік) та *User* (Користувач). Серед його атрибутів: *ticketId* (ідентифікатор тікета), *projectId* (ідентифікатор

проекту), sprintId (ідентифікатор спринту), *epicId* (ідентифікатор епіка), *reporterId* (ідентифікатор користувача, який створив тікет), *assigneeId* (ідентифікатор виконавця), *title* (назва), *status* (статус), *type* (тип), *priority* (пріоритет), *estimate* (оцінка обсягу робіт), *unitsOfMeasure* (одиниці виміру оцінки), *description* (опис), *createdTimeStamp* (час створення) та *lastUpdatedTimestamp* (остання зміна).

Перерахування TicketType (Тип заявки) задає можливі типи тікетів, а *UnitsOfMeasure* (Одиниці виміру) – формат оцінки (наприклад, у годинах чи інших одиницях).

Клас Comment (Коментар) пов'язаний відношенням «багато-до-одного» із класами *User* (Користувач) та *Ticket* (Тікет). Він містить такі поля: *commentId* (ідентифікатор коментаря), *userId* (ідентифікатор автора), *ticketId* (ідентифікатор пов'язаного тікета), *text* (текст коментаря), *createdTimeStamp* (час створення) та *lastUpdatedTimestamp* (час останнього редагування).

Діаграма класів додатково включає низку перерахувань, які фіксують замкнені набори значень для окремих атрибутів. Зокрема, *TicketEpicStatus* (Статус епіка та завдання) передбачає чотири можливі стани: *TODO* (До виконання), *IN_PROGRESS* (У процесі), *READY_FOR_QA* (Готово до тестування) та *DONE* (Виконано). SprintStatus (Статус спринту) описує життєвий цикл спринту та має три значення: *ACTIVE* (Активний), *RELEASED* (Завершений/випущений) та *INACTIVE* (Неактивний).

Перерахування TicketType (Тип заявки) розрізняє основні види тікетів: *TASK* (Завдання) і *BUG* (Помилка). Перерахування *Priority* (Пріоритет) задає градацію важливості або терміновості епіка чи тікета за п'ятьма рівнями: *LOWEST* (Найнижчий), LOW (Низький), *MEDIUM* (Середній), *HIGH* (Високий) та *HIGHEST* (Найвищий). Перерахування *UserRole* (Роль користувача) фіксує основні ролі учасників системи управління проектами: *ADMIN* (Адміністратор), *PROJECT_MANAGER* (Менеджер проекту), *DEVELOPER* (Розробник) та *QA_ENGINEER* (Тестувальник), що визначають їхні повноваження та доступ до функцій системи.

Перерахування UnitsOfMeasure (Одиниці виміру) встановлює формати оцінки трудомісткості завдань: *HOURS* (Години), *DAYS* (Дні), *SPRINTS* (Спринти) та *STORY_POINT* (Оціночні бали). Використання *story point* ґрунтується на узгодженому в команді уявленні про складність завдання з урахуванням технічних вимог, ризиків, необхідних ресурсів і рівня компетенцій, потрібних для його виконання.

Реалізація системи управління проектами здійснена з використанням засобів автоматизації Maven та мови Java у фреймворку Spring Boot. Для швидкого створення каркасу Spring Boot-застосунку використовується сервіс *spring initializr* (рис. 3.2). Структура коду визначається, зокрема, через конфігураційний файл pom.xml, у якому описуються всі необхідні залежності й плагіни для коректної роботи застосунку.

Фрагмент *pom.xml* файлу наведено в додатку А.

Створюємо *Aplication* клас, для запуску програми:

```
@Configuration
@SpringBootApplication
public class PMSoftwareApplication {
	public static void main(String[] args) {
	SpringApplication.run(PMSoftwareApplication.class, args);
	}
}
```

Контролери Spring Boot є однією з ключових складових фреймворку, оскільки відповідають за обробку HTTP-запитів і формування відповідей відповідно до логіки роботи застосунку. Їх основне призначення полягає в тому, щоб інтерпретувати вхідні запити від клієнтів (веббраузерів, мобільних застосунків тощо), виконувати необхідні дії на стороні сервера та повертати коректні результати у вигляді HTTP-відповідей.

Таким чином контролери виступають проміжною ланкою між зовнішніми клієнтами та серверним застосунком, реалізуючи прикладний інтерфейс взаємодії. Нижче наведено приклад оголошення класу контролера:

У Spring Boot існують чотири основні типи HTTP-запитів: *GET, POST,*

PATCH та *DELETE*, які можна обробляти за допомогою контролерів.

```
@RestController
@RequestMapping("/users")
public class UserController {
// Методи для обробки HTTP-запитів
}
```

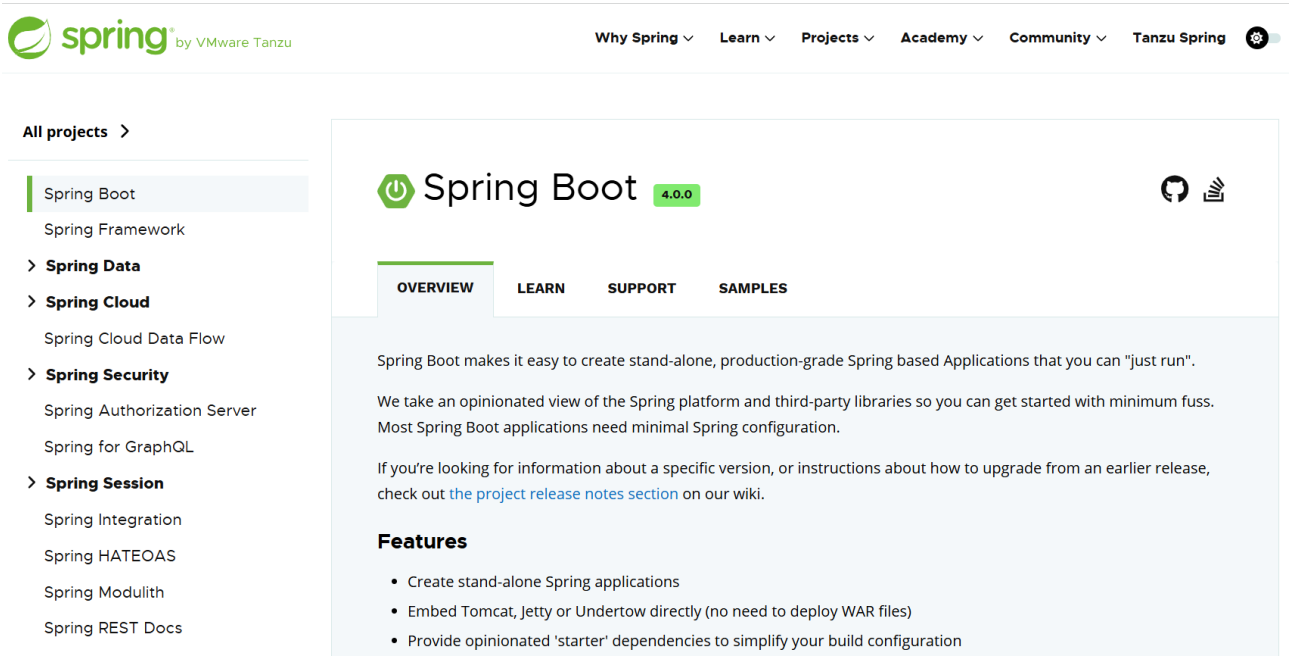

Рисунок 3.2 – Інструменти DEVELOPMENT TOOLS на сайті Spring Boot [18]

Опис кожного з цих запитів:

1. *GET*-запит використовується для отримання даних з сервера. У Spring Boot для обробки GET-запитів використовується анотація `@GetMapping`. Зазвичай цей метод повертає дані клієнту у форматі JSON, або HTML сторінку:

```
@GetMapping("/get/{id}")
public ResponseEntity<UserDTO> getUser(@PathVariable(name = "id") Integer userId) {
return ResponseEntity.ok(userService.getUser(userId));
}
```

2. *POST*-запит використовується для створення нових ресурсів на сервері, або відправки даних для обробки. У Spring Boot для обробки POST-запитів використовується анотація `@PostMapping`. Ця анотація дозволяє визначити

метод контролера, який оброблятиме POST запити. Зазвичай цей метод отримує дані від клієнта через параметри запиту, або тіло запиту і виконує відповідні дії, наприклад, зберігає дані в базі даних:

```
@PostMapping("/create")
public ResponseEntity<UserDTO> createUser(@Valid @RequestBody UserDTO user) {
return ResponseEntity.ok(userService.createUser(user));

}
```

3. *PATCH*-запит використовується для часткового оновлення ресурсу на сервері. Він дозволяє вносити зміни лише до певних полів, або властивостей ресурсу, не торкаючись інших полів. Для обробки PATCH-запитів у Spring Boot використовує анотацію *@PatchMapping*:

```
@PatchMapping("/update")
public ResponseEntity<UserDTO> updateUser(@RequestBody UserDTO user){
return ResponseEntity.ok(userService.updateUser(user));

}
```

4. *DELETE*-запит використовується для видалення ресурсів на сервері. У Spring Boot для обробки DELETE-запитів використовується анотація *`@DeleteMapping`*. Цей метод приймає параметри, наприклад, ідентифікатор ресурсу, який потрібно видалити, і виконує відповідні дії для видалення ресурсу з сервера:

```
@DeleteMapping("/delete")
public void deleteUser(@RequestBody UserDTO user) {
userService.deleteUser(user);
}
```

Контролери служать як проміжний рівень між вхідними запитами і бізнес-логікою додатку. Вони розділяють логіку, пов'язану з обробкою запитів, від логіки, пов'язаної з бізнес-правилами та обробкою даних.

Контролери дозволяють встановлювати правила валідації для вхідних даних та обробляти помилки. Вони дають можливість контролювати

інформацію, що надходить від клієнта, і реагувати на некоректні запити.

В прикладі вище в методі createUser(@Valid @RequestBody UserDTOuser анотація @Valid вказує на те, що деякі поля можуть мати обмеження. Обмеження теж вказуються анотаціями, але вже в класі UserDTO.

DTO (Data Transfer Object) — шаблон проектування, який виконується для передачі даних між підсистемами програми.

Також над класом контролера та класом DTO потрібно вказати анотацію @Validated. Класи DTO та контролери наведені в EpicComtroller.java (Додаток А).

3.2. Поєднання реляційної бази даних MySQL Workbench та бібліотеки Hibernate в системі управління IT-проектами

Для реалізації запропонованої системи керування проектами використано реляційну систему керування базами даних MySQL, у якій інформація зберігається не в одному монолітному сховищі, а в окремих взаємопов'язаних таблицях. Фізична структура бази даних представлена набором файлів, оптимізованих для швидкого обміну даними, тоді як логічна модель включає таблиці, подання, рядки та стовпці, формуючи гнучке середовище для програмування. Користувач задає правила, що визначають типи зв'язків між полями й сутностями – «один до одного», «один до багатьох», унікальні, обов'язкові чи необов'язкові значення, а також зовнішні ключі, які формують «вказівники» між таблицями [2].

Створення бази даних у *MySQL Workbench* передбачає послідовне виконання кількох кроків: спочатку необхідно встановити з'єднання з сервером баз даних, потім створити нову схему (базу) та виконати SQL-скрипт для формування таблиць. Лістинг відповідного скрипту наведено в *EpicController.java* (Додаток А).

Робота безпосередньо з реляційними БД у поєднанні з об’єктно-орієнтованим програмним кодом є доволі громіздкою та трудомісткою, тому в межах розробки застосовано *Hibernate* – засіб об’єктно-реляційного відображення (ORM – Object Relational Mapping) у середовищі Java. Використання ORM спрощує створення, модифікацію та доступ до даних, оскільки дозволяє пов’язувати об’єкти програми з відповідними записами в таблицях бази даних.

Фактично Hibernate виступає як бібліотека для взаємодії з БД: вона не лише дає змогу відображати Java-класи у таблиці, а типи даних Java – у відповідні SQL-типи, але й надає механізми для виконання запитів і пошуку. Основне призначення Hibernate – мінімізувати кількість рутинного та повторюваного коду, пов’язаного зі збереженням і вибіркою даних.

Для інтеграції Hibernate та MySQL з Maven-проектом у конфігураційному файлі pom.xml додаються залежності hibernate-core та mysql-connector-java. Перша забезпечує підтримку ORM-функціональності, друга – містить драйвер для встановлення з’єднання зі СКБД MySQL. Після цього необхідно налаштувати файл application.properties у Spring Boot-проекті, де вказуються параметри підключення до бази даних MySQL та інші супутні налаштування.

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
datasource.hostname=localhost
spring.datasource.url=jdbc:mysql://${datasource.hostname}/pmsoftwar
e?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
server.port=8081
```

У наведених параметрах pmsoftware виступає як назва бази даних, username – це ім’я користувача, під яким здійснюється підключення до СКБД, а password – відповідний пароль цього користувача.

Наступним кроком є конфігурація Hibernate-анотацій. Для цього створюються класи сутностей (Entity-класи), екземпляри яких зберігатимуться в базі даних за посередництва Hibernate.

До цих класів необхідно додати відповідні анотації, зокрема @Entity, @Table, @Id та інші, які описують відповідність між полями класу та стовпцями таблиць, а також задають особливості відображення та ключі. Загальна структура оформлення Entity-класу має такий вигляд:

```
@Entity
@Table(name = "users")
public class UserModel {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(nullable = false, name = "user_id")
private int userId;
…
}
```

Для взаємодії програмного коду з базою даних використовується інтерфейс JpaRepository з бібліотеки Spring Data JPA, який забезпечує зручну роботу з даними на основі підходу ORM (Object-Relational Mapping). Він є однією з реалізацій шаблону «репозиторій» і слугує проміжною ланкою між об’єктами Java та записами в реляційних таблицях, автоматизуючи процес їх прив’язки.

Інтерфейс JpaRepository надає вбудований набір методів для виконання типових операцій з базою даних: збереження (save), оновлення, видалення (delete) та отримання (find/get) записів. Це дає змогу реалізовувати базові CRUD-операції (створення, читання, оновлення, видалення) над сутностями без необхідності вручну формувати SQL-запити.

Окрім базового функціоналу, JpaRepository підтримує побудову більш складних запитів, включаючи використання іменованих методів, специфікацій або кастомних JPQL/SQL-запитів. Нижче може бути наведено приклад оголошення репозиторію для роботи з певною сутністю.

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
// Можна додати власні методи пошуку даних в базі
}
```

Після того як база даних і необхідні бібліотеки для роботи з нею підключені, наступним кроком є створення класу сервісу (Service), який інкапсулює логіку взаємодії з репозиторієм. У цьому класі викликаються методи JpaRepository для виконання операцій з базою даних, таких як отримання, збереження або оновлення даних:

```
@Service
public class UserServiceImpl {
private final UserRepository userRepository;
public UserServiceImpl(UserRepository userRepository) {
this.userRepository = userRepository;
}
public User getUserById(Integer userId) {
return userRepository.findById(userId).orElse(null);
}
// Можна додати інші методи та бізнес-логіку
}
```

Наприклад, у простому сервісі можна використовувати репозиторій для отримання об’єкта User за його унікальним ідентифікатором. Такий підхід дозволяє централізовано реалізувати бізнес-логіку та відокремити її від контролерів, забезпечуючи чисту архітектуру і підвищуючи підтримуваність системи.

3.3. Створення способу взаємодії з API завдяки платформі Postman

Розгортання програмних продуктів для командної роботи вимагає інструментів, які забезпечують просту взаємодію із серверами та сервісами, дозволяють інтерпретувати їх відповіді, виконувати тестування і документувати API. Для цих цілей широко застосовується Postman.

Postman – це комплексна платформа для роботи з API, що підтримує всі етапи розробки: від створення і тестування до документування та інтеграції. Основна функція інструменту – забезпечити розробникам зручний і наочний

спосіб взаємодії з API. Завдяки цьому Postman став одним із найпопулярніших інструментів у сфері розробки.

Серед основних можливостей платформи виділяють: тестування API із використанням JavaScript для перевірки статус-кодів і вмісту відповідей, відправку запитів різних типів (GET, POST, PUT, DELETE тощо), аналіз серверних відповідей у форматах JSON, XML та інших, а також спрощення розробки і тестування API завдяки інтегрованому інтерфейсу для роботи з запитами та тестами.

У рамках нашого проекту Postman використовується для перевірки функціональності системи та демонстрації результатів. Для створення нового запиту необхідно обрати тип HTTP-запиту (GET, POST, PUT, DELETE), вказати URL API у полі "Enter request URL" та, за потреби, передати дані у тілі запиту через вкладку "Body", наприклад у форматі JSON або XML.

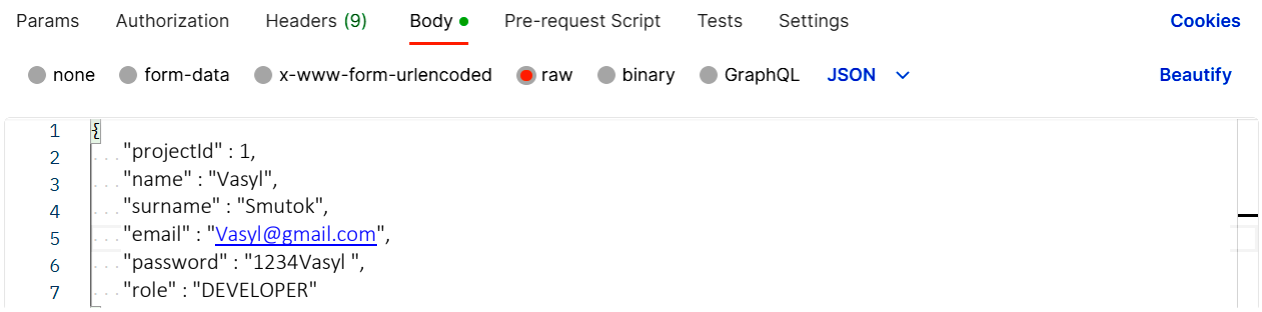

Рисунок 3.3 – Дані сформовані у запиті через вкладку Body

Після відправки HTTP-запиту в Postman платформа відображає отриману відповідь, включно зі статусом запиту, заголовками, тілом відповіді та іншою супровідною інформацією. У результаті можна бачити, наприклад, перелік створених користувачів у базі даних, що дозволяє перевірити коректність роботи API.

user_id	project_id	user_name	surname	email	user_password	user_role	created_time_
1	1	Julia	Voloshina	juliavoloshina02@gmail.com	1234Juli	ADMIN	2023-05-28 10:
2	1	Helen	Hrebeniuk	helen_kyuneberg@gmail.com	123Helen	DEVELOPER	2023-05-28 10:
3	1	Angrew	Motruk	andrewmmmm@gmail.com	1234Andr	DEVELOPER	2023-05-29 19:

Рисунок 3.4 – Переліку створених користувачів

Для оновлення інформації на сервері застосовується запит типу PATCH, при цьому змінюється URL та тіло запиту, де зазначаються поля для оновлення. Після виконання запиту API повертає оновлені дані, що демонструє успішне внесення змін.

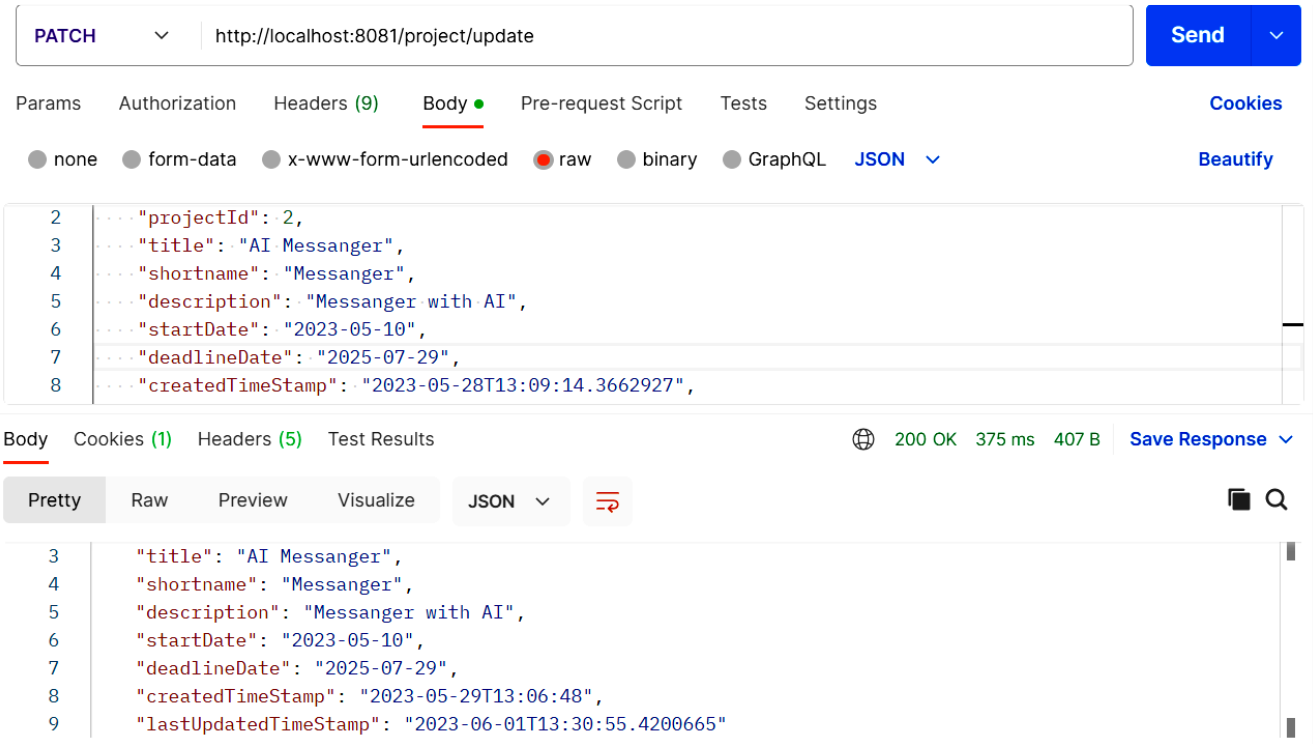

Рисунок 3.5 – HTTP-запит для оновлення даних

Аналогічно, для видалення даних використовується запит DELETE: у тілі передається об’єкт, який необхідно видалити, а у відповідь приходить порожнє тіло і статус «200 OK», що свідчить про успішне видалення.

Таким чином, робота з Postman дозволяє автоматизувати перевірку функціональності API, роблячи процес тестування більш ефективним і допомагаючи виявляти помилки на ранніх стадіях розробки.

Платформа використовує JavaScript для написання тестів, надаючи гнучкі можливості для створення різноманітних перевірок за допомогою вбудованих функцій, таких як *tests*, *pm.response*, *pm.expect* та інших.

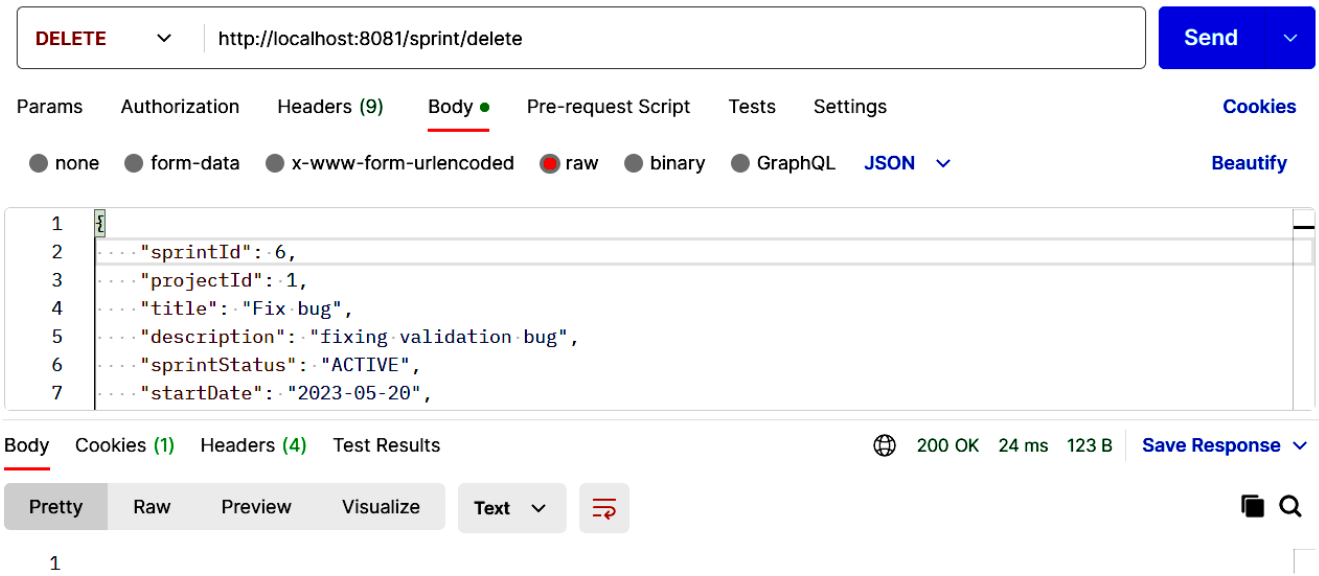

Рисунок 3.6 – HTTP-запит на видалення даних Спрінта

Postman сприяє підвищенню якості API на всіх етапах розробки: під час проектування та прототипування – створюються колекції запитів для швидкої перевірки концепції; під час розробки та налагодження – здійснюється надсилання запитів і швидке усунення помилок; під час тестування та автоматизації – виконуються автоматизовані тести та інтеграція в CI/CD-пайплайни через Newman; для документації – автоматично генерується актуальна документація API на основі колекцій запитів.

Загалом, використання Postman дозволяє розробникам забезпечити стабільність, надійність і зрозумілість API, пришвидшити перевірку змін та зменшити ризик появи помилок у процесі командної розробки.

РОЗДІЛ 4
РЕЗУЛЬТАТИ ПРАКТИЧНОГО ЗАСТОСУВАННЯ СИСТЕМИ

4.1. Перевірка типів запитів під час клієнт-серверної взаємодії

Архітектура клієнт-сервер є фундаментальним шаблоном побудови програмного забезпечення, який широко використовується у розробці розподілених мережних застосунків. В її основі лежить розподіл компонентів на сервери, клієнти та мережу, що забезпечує обмін даними між ними. Сервери надають інформацію або різні сервіси, а клієнти – використовують ці сервіси для виконання завдань. При цьому сервери та клієнти функціонують незалежно один від одного: один сервер здатен одночасно обслуговувати кілька клієнтів, а клієнт може звертатися до різних серверів без знання про інших користувачів системи.

Клієнт-серверна модель визначає розподіл обов’язків між клієнтською та серверною частинами. Традиційно виділяють три рівні: рівень представлення даних, що відповідає за інтерфейс користувача і введення команд; прикладний рівень, на якому реалізується логіка обробки інформації; та рівень управління даними, що забезпечує зберігання та доступ до інформації.

У дворівневій архітектурі функції можуть розподілятися по-різному: модель тонкого клієнта передбачає, що вся обробка та управління даними зосереджені на сервері, а клієнт реалізує лише інтерфейс користувача; модель товстого клієнта – сервер керує лише даними, а обробка та інтерфейс виконуються на стороні клієнта.

Для отримання даних із серверу у рамках запропонованої системи управління користувачами застосовуються GET-запити. На їх основі формується URL із зазначенням необхідного фільтра, наприклад ролі користувачів, що дозволяє швидко отримати інформацію про відповідних учасників системи. Такий підхід забезпечує гнучкий і ефективний механізм доступу до даних у клієнт-серверній архітектурі.

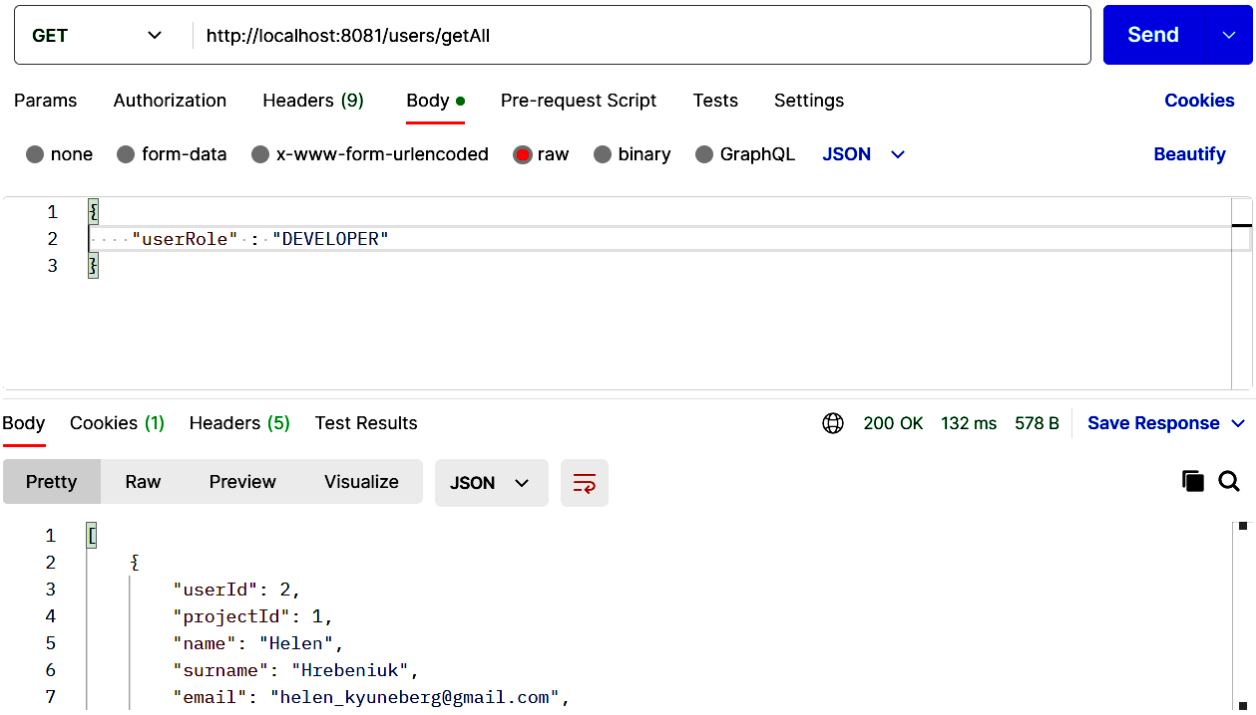

Рисунок 4.1 – HTTP-запит із фільтром за даними користувача

Повне тіло відповіді на GET-запит для отримання всіх користувачів із роллю “DEVELOPER” у форматі JSON може виглядати так:


```
{
  "userId": 0,
  "projectId": 1,
  "name": "Vasyl",
  "surname": "Smutok",
  "email": "Vasyl@gmail.com",
  "password": "1234Vasyl",
  "role": "DEVELOPER",
  "createdTimeStamp": "2025-11-23T10:00:00",
  "lastUpdatedTimestamp": "2025-11-23T10:00:00"
},
{
  "userId": 1,
  "projectId": 101,
  "name": "Ivan",
  "surname": "Petrenko",
  "email": "ivan.petrenko@example.com",
  "role": "DEVELOPER",
  "createdTimeStamp": "2025-11-23T10:15:30",
  "lastUpdatedTimestamp": "2025-11-23T12:20:45"
},
{
  "userId": 2,
  "projectId": 1,
  "name": "Helen",
```

```
    "surname": "Hrebeniuk",
    "email": "helen_kyuneberg@gmail.com",
    "password": "123Helen",
    "role": "DEVELOPER",
    "createdTimeStamp": "2025-11-21T14:30:50",
    "lastUpdatedTimestamp": "2025-11-21T15:45:15"
  },
  {
    "userId": 3,
    "projectId": 101,
    "name": "Andriy",
    "surname": "Koval",
    "email": "andriy.koval@example.com",
    "role": "DEVELOPER",
    "createdTimeStamp": "2025-11-21T14:30:50",
    "lastUpdatedTimestamp": "2025-11-21T15:45:15"
  }
```


У цьому запиті кожен об’єкт представляє окремого користувача, а поля відповідають структурі класу User у системі. Поле role визначає роль користувача, тут усі користувачі мають роль “DEVELOPER”.

Поля createdTimeStamp та lastUpdatedTimestamp відображають час створення запису та останнього оновлення даних.

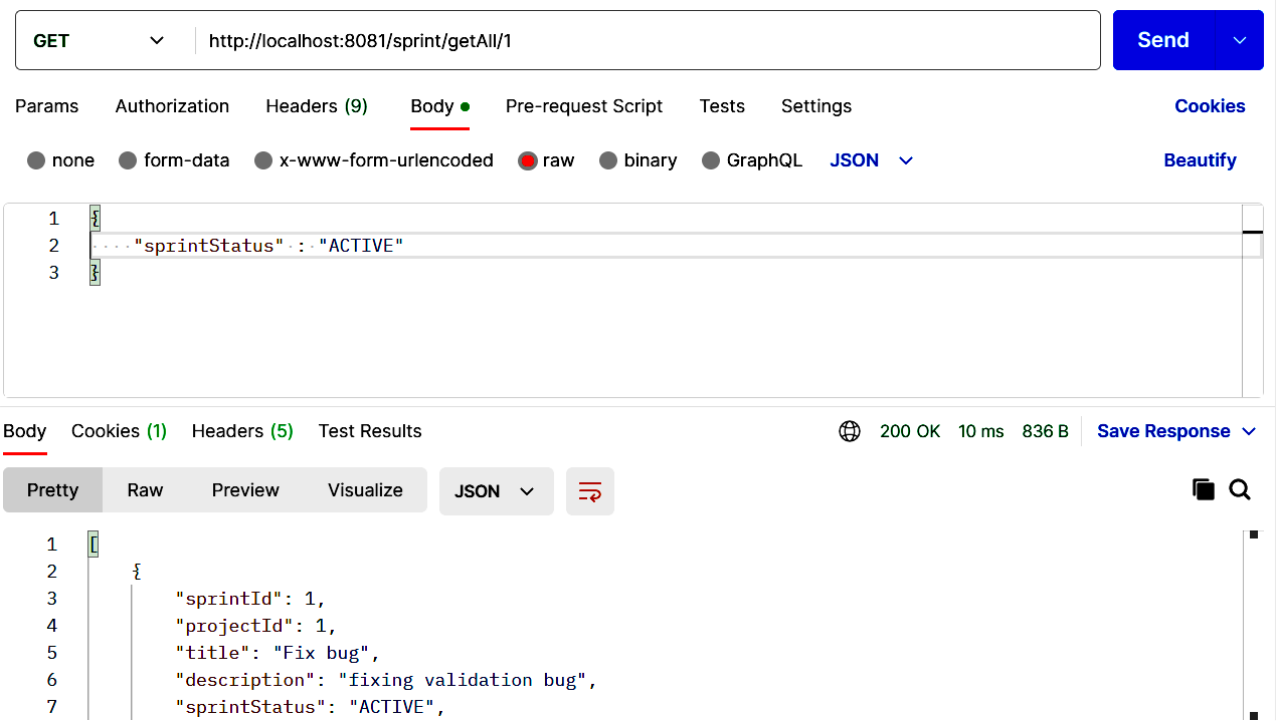

Рисунок 4.2 – HTTP-запит за фільтром та ID користувача

Для отримання інформації зі сервера з урахуванням фільтрів та конкретного унікального ідентифікатора потрібно внести певні зміни у запит (рис. 4.2). У випадку окремого об’єкта Спринта вибираються записи з бази даних, які відповідають проекту з ідентифікатором «1» (значення ID додається в кінці URL-адреси запиту) та заданому статусу спрінту через фільтр sprint status.

З повного тіла відповіді сервера видно, що всі отримані Спринти мають статус “ACTIVE” і належать проекту з ID «1»:


```
{
"sprintId": 1,
"projectId": 1,
"title": "Fix bug",
"description": "fixing validation bug",
"sprintStatus": "ACTIVE",
"startDate": "2023-05-20",
"deadlineDate": "2023-05-29",
"createdTimeStamp": "2023-05-28T13:43:57",
"lastUpdatedTimeStamp": null
},
{
"sprintId": 3,
"projectId": 1,
"title": "Handler creation",
"description": null,
"sprintStatus": "ACTIVE",
"startDate": "2023-05-20",
"deadlineDate": "2023-06-02",
"createdTimeStamp": "2023-05-28T13:59:44",
"lastUpdatedTimeStamp": null
},
{
"sprintId": 7,
"projectId": 1,
"title": "Validation Bug",
"description": null,
"sprintStatus": "ACTIVE",
"startDate": "2023-06-20",
"deadlineDate": "2023-06-29",
"createdTimeStamp": "2023-06-01T14:26:20",
"lastUpdatedTimeStamp": null
}
```


Таким чином, функціонування запропонованої системи підтверджує її практичну придатність: клієнт-серверна взаємодія реалізується через стандартні

HTTP-запити з використанням фільтрів та унікальних ідентифікаторів для точного отримання необхідних даних.

4.2. Обробка <*exception*> під час виконання програми

Під час розробки та експлуатації програмного забезпечення виникнення помилок або непередбачуваних ситуацій у роботі програми називають винятками (exceptions). Вони можуть з'являтися через некоректні дії користувача, відсутність необхідних ресурсів на диску, або втрату мережевого з'єднання з сервером. Причини винятків часто пов'язані з помилками програмування або неправильним використанням API, тому програма повинна передбачати механізм їх обробки. У Java для цього існує стандартний механізм винятків, який дозволяє керувати такими ситуаціями ефективно та передбачувано.

Окрім *RESTful* веб-служб, для обробки динамічного HTML-контенту можна використовувати Spring MVC, який підтримує різні шаблонізатори, включаючи *Thymeleaf*, *FreeMarker* та *JSP*. *Spring* Boot надає автоконфігурацію для шаблонізаторів *FreeMarker*, *Groovy*, *Thymeleaf* і *Mustache*, а шаблони за замовчуванням підбираються з папки src/main/resources/templates. У разі помилок Spring Boot забезпечує відображення стандартної сторінки /error, яка для браузерних клієнтів показує HTML-представлення (*whitelabel*), а для машинних клієнтів повертає JSON із кодом стану HTTP та деталями помилки.

Для побудови клієнт-серверної архітектури використовуються сервлети Java, які отримують запити від клієнта і формують відповідь. Для персоналізації обробки помилок у Spring Boot можна налаштувати властивості server.error, створити власний *ErrorController*, або додати бін типу *ErrorAttributes*. Глобальна обробка виключень реалізується через клас, анотований *@RestControllerAdvice*, у якому методи обробки винятків позначаються анотаціями *@ExceptionHandler* і *@ResponseStatus*. Перша

визначає тип HTTP-відповіді, який буде повернений клієнту, а друга вказує метод, що обробляє конкретний тип винятку. Такий підхід дозволяє централізовано керувати помилками і надавати клієнтам структуровану інформацію про проблеми, що виникли під час виконання програми:

```
@RestControllerAdvice
public class GlobalExceptionHandler{
@ExceptionHandler(UserNotFoundException.class)
public ResponseEntity<Object> showUserAbsence() {
ExceptionResponse response = new ExceptionResponse();
response.setMessage("User is not found.");
return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
}
// Інші обробники виключень
}
```

Якщо під час виконання програми виникає виняток UserNotFoundException, сервер поверне відповідь із HTTP-статусом 404 Not Found і повідомленням "User is not found".

Повний перелік обробників всіх можливих винятків у розробленій системі наведено в Додатку В.

Нижче наведено приклади того, як система реагує на некоректні або неповні дані при виконанні запитів.

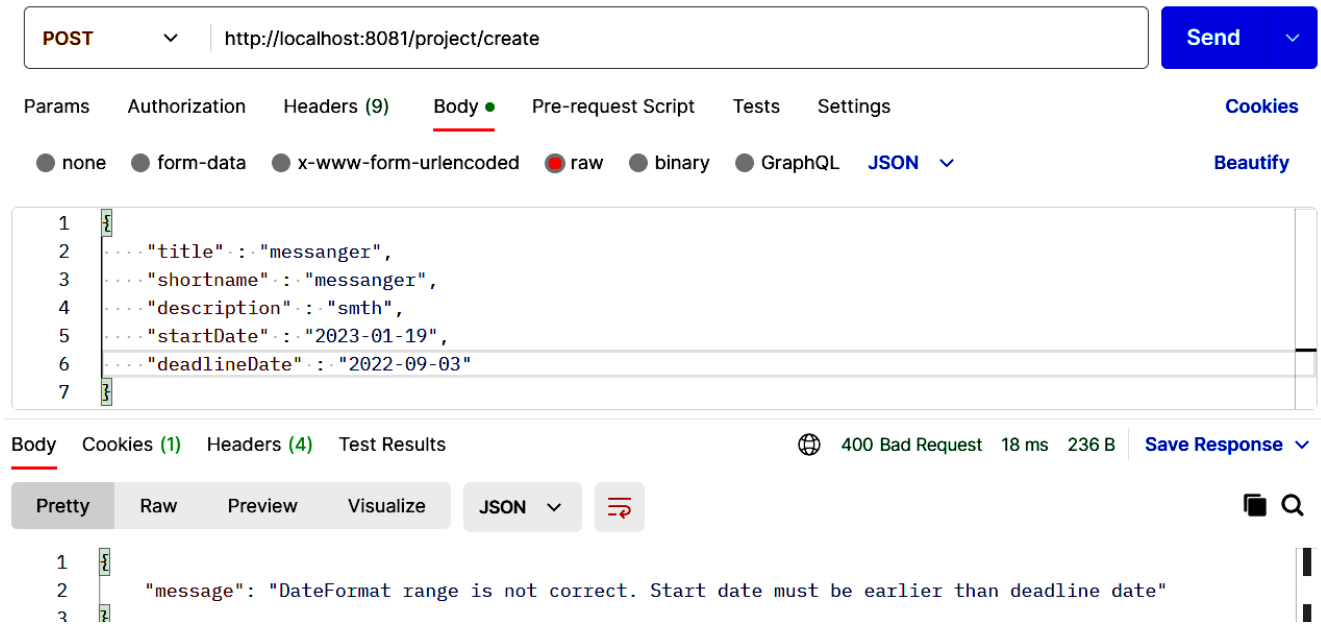

Рисунок 4.3 – Відповідь сервера про неправильний проміжку часу розробки програми

Кожен випадок демонструє, що обробка винятків у Spring Boot дозволяє програмі надійно реагувати на помилки, повертати структуровані повідомлення клієнту та уникати аварійного завершення роботи застосунку (рис. 4.3)

У наведених прикладах демонструються типові помилки під час роботи з HTTP-запитами в системі. *Перший випадок* показує запит, де дата початку проекту (startDate) вказана пізніше за дату завершення (deadlineDate), що є некоректним і викликає виняток.

Другий приклад ілюструє помилку при спробі створити користувача з електронною поштою, яка вже існує в системі (рис. 4.4).

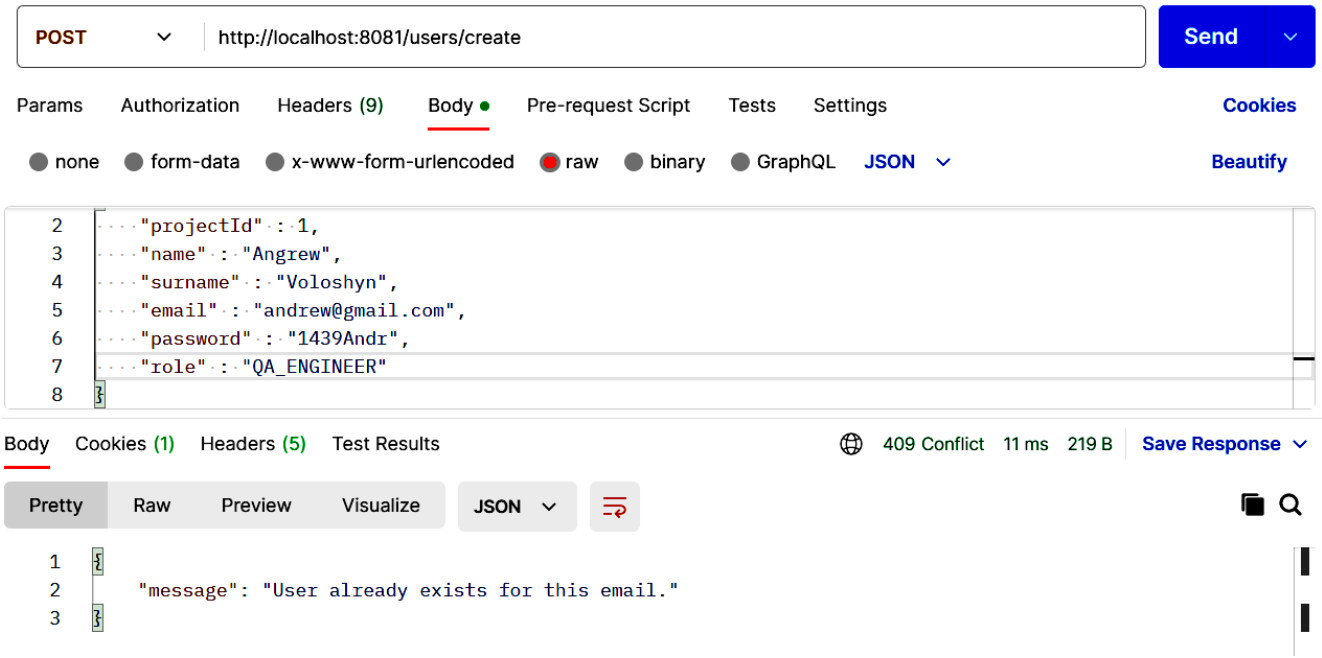

Рисунок 4.4 – Відповідь сервера про помилки дублювання електронної пошти

Третій випадок демонструє запит на отримання Спринта за унікальним ідентифікатором проекту, якого немає в базі даних сервера, що також призводить до виникнення винятку (рис. 4.5). Усі ці сценарії підтверджують необхідність обробки помилок і гарантують стабільність роботи застосунку.

Запропоноване програмне забезпечення може бути додатково розширене функціями аутентифікації користувачів, можливістю залишати коментарі в усіх

компонентах для покращення командної взаємодії, а також впровадженням звітності для підтримки прийняття ефективних управлінських рішень.

Рисунок 4.5 – Відповідь сервера щодо неіснуючого проекту

Реалізація системи колективного управління IT-проектами у вигляді окремого API забезпечує розробникам зручний доступ до всіх необхідних функцій, полегшує інтеграцію з існуючими проектами, надає гнучкість у використанні та прискорює процес розробки. Використання Spring Boot дозволяє скоротити час розгортання, забезпечити гнучке налаштування системи, інтеграцію з іншими технологіями та спрощує процес розробки. Завдяки організації коду через контролери, сервіси та репозиторії досягається розподілена архітектура з чітким розділенням відповідальності між компонентами.

Основна функціональність системи включає створення та відстеження Тікетів, Спринтів і Епіків, а також управління користувачами. Завдяки цьому компоненти системи забезпечують зручне та ефективне використання її можливостей у реальних IT-проектах, підтримуючи комплексне управління та контроль над робочими процесами команди.

4.3. Результати дослідження ефективності розробленого програмного засобу

Для оцінки ефективності розробленого ПЗ застосовано наступну методику дослідження трьох ключових показників:

1) продуктивність (Performance);
2) навантажувальна здатність (Scalability / Load Handling);
3) коректність функціонування (Functional Accuracy).

Методика **дослідження продуктивності** (Performance) системи проводилася за наступними кроками: 1) встановлювали тестові дані для об’єктів Project, Sprint та Epic у базі даних MySQL; 2) виконували серію запитів через REST-контролери (GET, POST, PATCH, DELETE) із різною кількістю об’єктів; 3) фіксували час виконання запитів та час відповіді сервера; 4) аналізували залежність часу обробки від обсягу даних, щоб оцінити ефективність роботи системи при зростанні навантаження.

Методика **дослідження навантажувальної здатності** (Scalability / Load Handling) системи проводилася за наступними кроками: 1) використовуємо інструменти тестування навантаження, такі як JMeter або Postman Collection Runner, для імітації реальних запитів до системи; 2) встановлюємо кількість паралельних запитів до контролерів EpicController та інших сервісів, щоб оцінити поведінку системи під різними рівнями одночасного навантаження; 3) вимірюємо час відповіді та фіксуємо кількість помилок при кожному рівні навантаження для визначення стабільності та надійності системи; 4) аналізуємо отримані результати, щоб визначити поріг навантаження, після якого продуктивність системи істотно знижується.

Методика **дослідження коректності функціонування** (Functional Accuracy): 1) виконуємо створення, оновлення, видалення та фільтрацію об’єктів через REST API; 2) порівнюємо отримані результати з очікуваними (дані в базі та у відповіді контролерів); 3) перевіряємо обробку помилок і виняткових ситуацій, наприклад, відсутність запису або некоректні дані.

Для виконання досліджень *використано недороге обладнання* для тестового сервера – **Supermicro 2-socket сервер:** практичний і простий сервер з двома процесорами Opteron, 16 ядер; підходить для тестів з багато багатопотокових навантажень. Використання такого серверного заліза (як Supermicro) значно дешевше, ніж нові сервери, але дає хорошу обчислювальну потужність для навантажувального тестування [18].

Наведена методика дозволяє оцінити три ключові аспекти: швидкість обробки запитів, здатність системи витримувати навантаження та коректність функціонування під час взаємодії з базою даних і клієнтськими запитами.

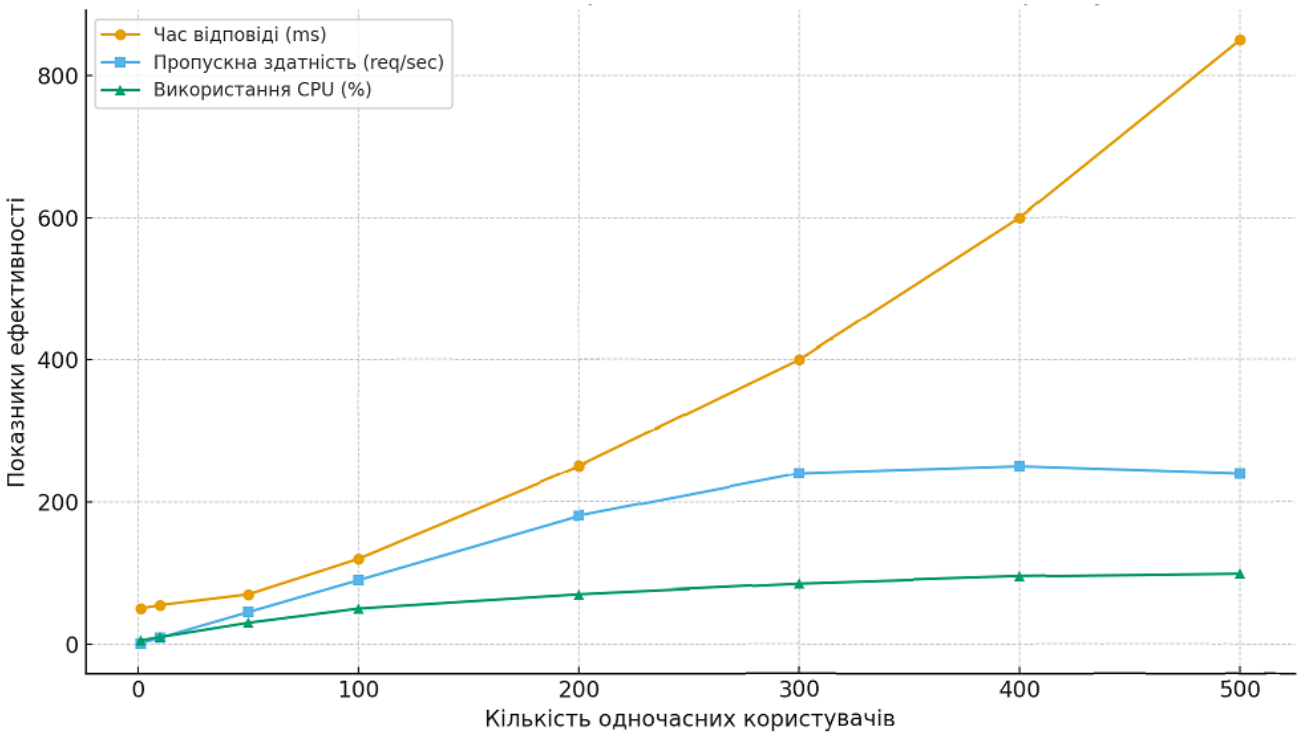

Рисунок 4.6 – Залежність показників ефективності системи від кількості одночасних користувачів: часу відповіді (ms), пропускної здатності (запити/сек) та використання CPU (%).

Отримані дані переконують в тому, що з ростом навантаження час відповіді та завантаження CPU зростають, а пропускна здатність спершу зростає, але потім досягає максимуму і стабілізується.

РОЗДІЛ 5
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

5.1. Розробка логіко-імітаційної моделі виникнення травм і аварій

Методикою оцінки рівня небезпеки робочих місць, машин, виробничих процесів та окремих виробництв передбачено пошук об'єктивного критерію рівня небезпеки для конкретного об'єкта [7]. Таким показником вибрана ймовірність виникнення аварії, травми залежно від явища, що досліджується.

Для побудови логіко-імітаційної моделі процесу, формування і виникнення аварії та травми в процесі створення мікрокліматичних умов у приміщенні оцінюють відповідні небезпечні події. Кожній із них присвоїмо ймовірність виникнення:

Шифр	Назва події	Ймовірність
P_1	Відсутність захисного заземлення	0,02
P_2	Пошкодження захисного заземлення	0,04
P_3	Спрацювання складових захисту	0,1
P_4	Неправильна експлуатація захисту	0,02
P_5	Відсутність профілактичних заходів	0,2
P_6	Відсутність захисного щита	0,12
P_7	Недотримання правил вибору взуття	0,15
P_8	Незнання правил техніки безпеки	0,1
P_9	Відсутність засобів індивідуального захисту	0,2
P_{10}	Легковажність	0,08

На основі наведених подій будуємо матрицю логічних взаємозв'язків між окремими пунктами, графічна інтерпретація якої зображено на рис. 5.1.

Розрахуємо ймовірності виникнення подій, що формують логіко-імітаційну модель процесів створення мікрокліматичних умов. Розглянемо травмонебезпечну ситуацію, що виникає за умови роботи працівників із

електронебезпекою.

Підставивши дані ймовірностей базових подій у формулу, отримаємо ймовірність події 13: $P_{13} = 0{,}2 + 0{,}4 - 0{,}2 \cdot 0{,}4 = 0{,}0592$.

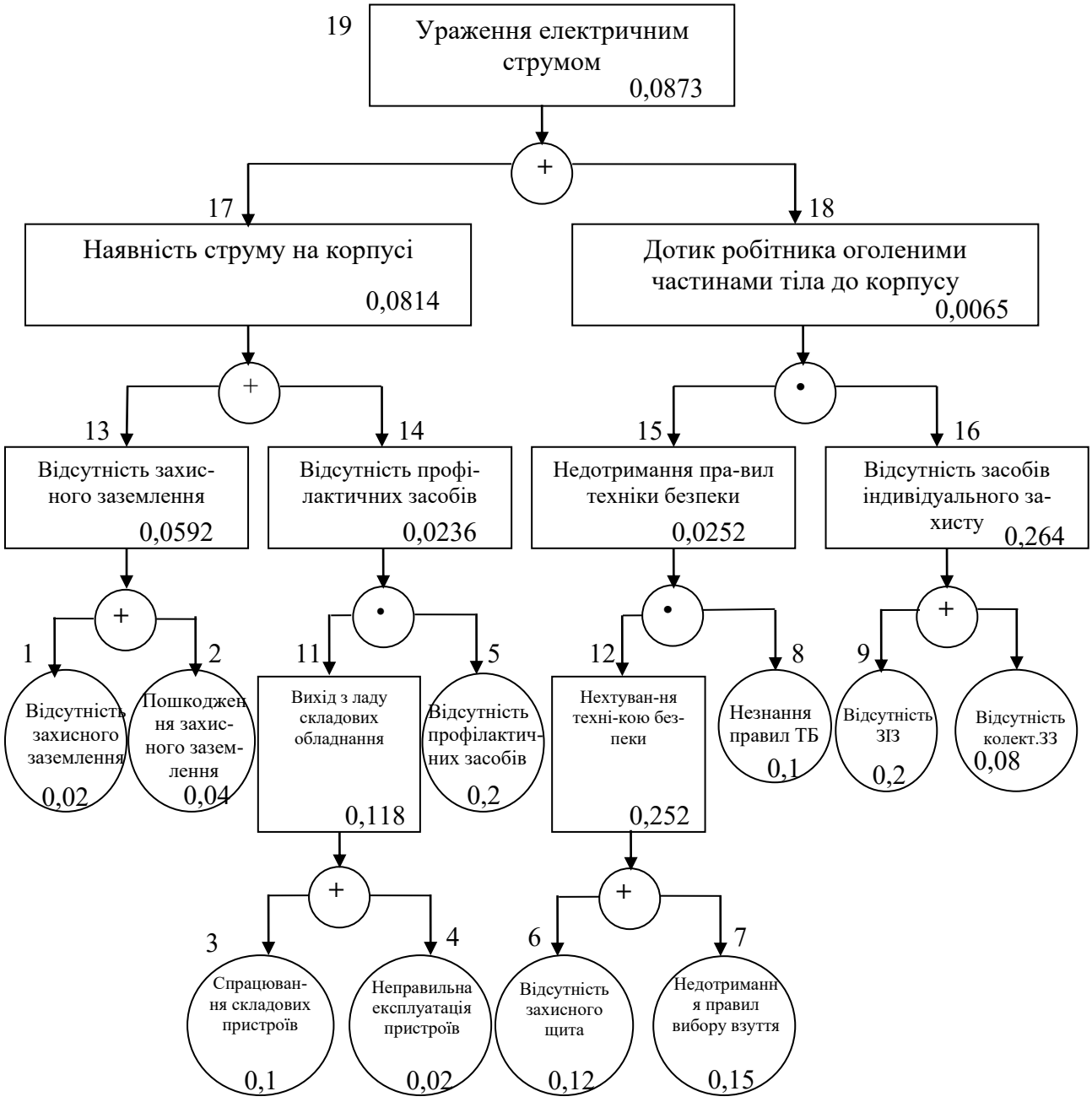

Рисунок 5.1. Матриця логічних взаємозв'язків між окремими подіями травмонебезпечної ситуації [7]

Аналогічно визначаємо ймовірність інших подій:

$$P_{11} = P_4 + P_5 - P_4P_5 = 0{,}3 + 0{,}4 - 0{,}3 \cdot 0{,}4 = 0{,}118.$$
$$P_{12} = P_6 + P_7 - P_6P_7 = 0{,}3 + 0{,}5 - 0{,}3 \cdot 0{,}5 = 0{,}252.$$
$$P_{16} = P_9 + P_{10} - P_9P_{10} = 0{,}2 + 0{,}15 - 0{,}2 \cdot 0{,}15 = 0{,}264.$$

$$P_{14} = P_{11} \cdot P_5 = 0{,}118 \cdot 0{,}2 = 0{,}0236.$$
$$P_{15} = P_{12} \cdot P_8 = 0{,}252 \cdot 0{,}1 = 0{,}0252.$$
$$P_{17} = P_{13} + P_{14} - P_{13} \cdot P_{14} = 0{,}592 + 0{,}0236 - 0{,}0592 \cdot 0{,}0236 = 0{,}0814.$$
$$P_{18} = P_{15} \cdot P_{16} = 0{,}264 \cdot 0{,}0252 = 0{,}0065.$$
$$P_{19} = P_{17} + P_{18} - P_{17} \cdot P_{18} = 0{,}0065 + 0{,}0814 - 0{,}0065 \cdot 0{,}0814 = 0{,}0873.$$

Таким чином, ймовірність перекидання машини та наслідкового виникнення травми працівника є досить мала і становить – $P_{19} = 0,0873$.

5.2. Планування заходів із покращення умов праці

До заходів щодо покращення умов праці належать всі види діяльності, спрямовані на попередження, нейтралізацію або зменшення негативної дії шкідливих і небезпечних виробничих факторів на працівників.

Рівень умов праці оцінюють порівнянням за фактичними і нормативними значеннями узагальнених (групових) показників.

Заходи щодо поліпшення умов праці здійснюють з метою створення безпечних умов праці шляхом:

- доведення до нормативного рівня показників виробничого середовища за елементами умов праці;
- захисту працівників від дії небезпечних і шкідливих виробничих факторів.

До показників ефективності заходів щодо поліпшення умов праці належать:

а) зміни стану умов праці:

- зміна кількості засобів виробництва, приведених у відповідність до вимог стандартів безпеки праці;
- покращання санітарно-гігієнічних показників;
- покращання психофізичних показників, зменшення фізичних і нервово-психічних навантажень, в т.ч. монотонних умов праці;

б) соціальні результати заходів:

- збільшення кількості робочих місць, що відповідають нормативним вимогам;
- зниження рівня виробничого травматизму;
- престиж та задоволення працею.

Отже, на покращення охорони праці потрібно виділити кошти на відновлення вентиляційних систем у ремонтних майстернях, естетично оформити приміщення офісу, відновити кабінет з охорони праці, поновити протипожежний інвентар.

5.3. Безпека в надзвичайних ситуаціях

Актуальність проблеми природно-техногенної безпеки для населення і території, зумовлена зростанням втрат людей, що спричиняється небезпечними природними явищами, промисловими аваріями та катастрофами [7]. У системі цивільної оборони окремого господарства необхідно забезпечити захист населення таким чином:

Укриття в захисних спорудах, якому підлягає усе населення відповідно до приналежності, досягається створенням фонду захисних споруд.

Евакуаційні заходи, які проводяться в містах та інших населених пунктах, які мають об’єкти підвищеної небезпеки, а також у воєнний час, основним способом захисту населення є евакуація і розміщення його у позаміській зоні.

Медичний захист проводиться для зменшення ступеня ураження людей, своєчасного надання допомоги постраждалим та їх лікування, забезпечення епідеміологічного благополуччя в районах надзвичайних ситуацій.

Радіаційний і хімічний захист включає заходи щодо виявлення і оцінки радіаційної та хімічної обстановки, організацію і здійснення дозиметричного та хімічного контролю, розроблення типових режимів радіаційного захисту, забезпечення засобами індивідуального захисту, організацію і проведення спеціальної обробки.

ВИСНОВКИ

Аналіз наявних систем управління IT-проектами показує, що багато з них характеризуються складним інтерфейсом користувача та обмеженою функціональністю, що ускладнює роботу розробників. Тому було прийнято рішення створити систему управління IT-проектами у вигляді API, яке дозволяє розробникам працювати з функціоналом системи просто і зручно, без необхідності взаємодіяти зі складним інтерфейсом та обмеженнями традиційних систем.

Розширення функціоналу до API надає розробникам можливість легко інтегрувати та використовувати необхідні компоненти системи у власних проектах. Це спрощує процес розробки, оскільки можна обирати лише потрібні елементи API, уникаючи зайвого навантаження та непотрібного коду. Такий підхід також забезпечує ефективний обмін даними між різними системами управління проектами та гарантує їхню сумісну роботу.

Запропонована система має інноваційний характер – вона дозволяє розробникам взаємодіяти з функціоналом управління IT-проектами без зайвих складнощів і втрати часу. Система забезпечує ефективне управління проектами за методологією Agile, надаючи командам можливість працювати швидше, гнучкіше та результативніше.

Розроблена UML-діаграма класів відображає архітектурну структуру системи. Її впровадження передує створенню Spring Boot проекту та реалізації контролерів, сервісів, репозиторіїв і обробників помилок, які взаємодіють між собою для забезпечення високої якості та ефективності командного управління проектами.

В цілому, розробка системи командного управління IT-проектами на Java з використанням Agile підходів є значущим внеском у сферу програмного забезпечення. Вона демонструє, що процес ефективного управління проектами можна спростити та оптимізувати завдяки сучасним технологіям і інструментам.

Для оцінки ефективності розробленого ПЗ застосовано наступну методику дослідження трьох ключових показників: 1) продуктивність (Performance); 2) навантажувальна здатність (Scalability / Load Handling); 3) коректність функціонування (Functional Accuracy);

Для виконання досліджень *використано недороге обладнання* для тестового сервера – Supermicro 2-socket сервер: практичний і простий сервер з двома процесорами Opteron, 16 ядер; підходить для тестів з багато багатопотокових навантажень. Використання такого серверного заліза (як Supermicro) значно дешевше, ніж нові сервери, але дає хорошу обчислювальну потужність для навантажувального тестування.

Отримані дані переконують в тому, що з ростом навантаження час відповіді та завантаження CPU зростають, а пропускна здатність спершу зростає, але потім досягає максимуму і стабілізується.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Застосування архітектурного проектування в гнучких методах розробки програмних продуктів / Харченко О.Г., Боднарчук І.О., Галай І.О., Лісовий В. // Інженерія програмного забезпечення. 2012. № 3-4 (11-12). С. 5-11.

2. Лавріщева К.М. Програмна інженерія. / К.М. Лавріщева. К.: Видавничий дім "Академперіодика", 2018. 319 с.

3. Лехман С.Д. та ін. Запобігання аварійності і травматизму у сільському господарстві / С.Д. Лехман, В.І. Рубльов, Б.І. Рябцев. К.: Урожай, 1993. 272 с.

4. Основи управління IT проектами: Навч. посіб. для студ. спеціальності 122 «Комп'ютерні науки» / В. О. Кузьміних, Р. А. Тараненко. Київ: КПІ ім. Ігоря Сікорського, 2019. 75 с.

5. Підходи до управління програмними проектами у SWEBOK V3. URL: https://www.researchgate.net/publication/316493834 (дата звернення: 10.11.2025).

6. Agile Development at Scale: The Next Frontier. URL: https://www.researchgate.net/Agile_Development_at_Scale (дата звернення 01.12.2025)

7. Agile manifesto. URL: http://agilemanifesto.org (дата звернення 10.10.2025)

8. Classical Waterfall Model. URL: https://www.geeksforgeeks.org/software-engineering*classical-waterfall-model (дата звернення 29.10.2025)

9. Collaboration Software. 2024. URL:https://www.computerworld.com/article/3226447/what-is-trello-a-guide-toatlassians-collaboration-and-work-management-tool.html (дата звернення 29.10.2025)

10. Head First Java. Легкий для сприйняття довідник. Фабула. 2022. 720.

11. Henrik Kniberg. Scrum and XP from the trenches. C4Media, 2007. С. 140. ISBN 978-1-4303-2264-1.

12. Iterative Model: What Is It And When Should You Use It? URL: https://airbrake.io/iterative-model (дата звернення 25.10.2025)

13. Kanban. URL: https://www.agilealliance.org/kanban (дата звернення 18.10.2025)

14. Managing Successful Projects with PRINCE2 (2019 Edition). The Stationery Office/Tso; 2019th edition, 2019. 327.

15. Maven: The Definitive Guide: The Definitive Guide. "O'Reilly Media, Inc.". 2008. 470.

16. Postman – що це за інструмент і які його основні функції? 2024. URL: https://foxminded.ua/postman/ (дата звернення 25.11.2025)

17. Spring Boot. URL: https://spring.io/projects/spring-boot (дата звернення 28.11.2025)

18. Spring Initializr – веб-інструмент для генерування базової структури проекту в Spring Boot. URL: https://start.spring.io/. (дата звернення 30.10.2025)

19. V-Model. URL: https://www.tutorialspoint.com/sdlc/v_model (дата звернення 30.10.2025)

20. What is Agile Kanban Methodology? URL: https://www.inflectra.com/methodologies/kanban (дата звернення 26.10.2025)

ДОДАТКИ

Додаток А

Фрагмент коду головних файлів програми - pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <parent>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-parent</artifactId>
           <version>2.7.8</version>
           <relativePath/>
      </parent>
      <groupId>com.diploma</groupId>
      <artifactId>pmsoftware</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <name>pmsoftware</name>
      <properties>
            <java.version>11</java.version>
      </properties>
      <dependencies>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-web</artifactId>
            </dependency>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-data-jpa</artifactId>
            </dependency>
            <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
            <exclusion>
                  <groupId>org.junit.vintage</groupId>
                  <artifactId>junit-vintage-engine</artifactId>
            </exclusion>
            </exclusions>
      </dependency>
      <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
      </dependency>
      <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.33</version>
      </dependency>
      <dependency>
            <groupId>javax.validation</groupId>
            <artifactId>validation-api</artifactId>
            <version>2.0.1.Final</version>
      </dependency>
      <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator</artifactId>
```

```
            <version>6.0.10.Final</version>
      </dependency>
      <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>12.0</version>
      </dependency>
      <dependency>
            <groupId>org.modelmapper</groupId>
            <artifactId>modelmapper</artifactId>
            <version>3.1.1</version>
      </dependency>
      </dependencies>
      <build>
      <plugins>
      <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
      </plugins>
      </build>
</project>
```

Лістинг файлу ProjectModel.java

```
@AllArgsConstructor
@RequiredArgsConstructor
@Getter
@Setter
@Entity
@Table(name = "project")
public class ProjectModel {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(nullable = false, name = "project_id")
private int projectId;
@Column(nullable = false, name = "title")
private String title;
@Column(name = "shortname")
private String shortname;
@Column(name = "project_description")
private String description;
@Column(name = "start_date")
private LocalDate startDate;
@Column(name = "deadline_date")
private LocalDate deadlineDate;
@Column(nullable = false, name = "created_time_stamp")
private LocalDateTime createdTimeStamp;
@Column(name = "last_updated_time_stamp")
private LocalDateTime lastUpdatedTimeStamp;
}
```

Лістинг файлу SprintModel.java

```
@Getter
@Setter
@Entity
@Table(name = "sprint")
public class SprintModel {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(nullable = false, name = "sprint_id")
private int sprintId;
@Column(nullable = false, name = "project_id")
private int projectId;
@Column(nullable = false, name = "title")
```

```
private String title;
@Column(name = "sprint_description")
private String description;
@Column(nullable = false, name = "sprint_status")
private SprintStatus sprintStatus;
@Column(name = "start_date")
private LocalDate startDate;
@Column(name = "deadline_date")
private LocalDate deadlineDate;
@Column(nullable = false, name = "created_time_stamp")
private LocalDateTime createdTimeStamp;
@Column(name = "last_updated_time_stamp")
private LocalDateTime lastUpdatedTimeStamp;
}
```

Лістинг файлу EpicComtroller.java

```
@RestController
@RequestMapping("/epic")
public class EpicController {
private final EpicServiceImpl epicService;
public EpicController(EpicServiceImpl epicService) {
this.epicService = epicService;
}
@PostMapping("/create")
public ResponseEntity<EpicDTO> createEpic(@Valid @RequestBody EpicDTO epic) {
return ResponseEntity.ok(epicService.create(epic));
}
@GetMapping("/getByFilter")
public ResponseEntity<List<EpicDTO>> getByFilter(@RequestBody Filter filter) {
return ResponseEntity.ok(epicService.getAllByFilter(filter));
}
@GetMapping("/getAllByProject/{project-id}")
public ResponseEntity<List<EpicDTO>> getAllByProjectId(@PathVariable(name =
"project-id") Integer projectId, @RequestBody Filter filter) {
return ResponseEntity.ok(epicService.getAllByProject(projectId, filter));
}
@GetMapping("/getAllBySprint/{sprint-id}")
public ResponseEntity<List<EpicDTO>> getAllBySprintId(@PathVariable(name =
"sprint-id") Integer sprintId, @RequestBody Filter filter) {
return ResponseEntity.ok(epicService.getAllBySprint(sprintId, filter));
}
@GetMapping("/get/{epic-id}") 65
```

Продовження додатку Г

```
public ResponseEntity<EpicDTO> getEpic(@PathVariable(name = "epic-id") Integer
epicId) {
return ResponseEntity.ok(epicService.getEpic(epicId));
}
@DeleteMapping("/delete")
public void deleteEpic(@RequestBody EpicDTO epic) {
epicService.delete(epic);
}
@PatchMapping("/update")
public ResponseEntity<EpicDTO> updateEpic(@RequestBody EpicDTO epic){
return ResponseEntity.ok(epicService.update(epic));
}
}
```

Додаток Б.
SQL запит на створення таблиць в БД

```
CREATE TABLE IF NOT EXISTS project(
project_id int NOT NULL auto_increment PRIMARY KEY,
title varchar(255) NOT NULL,
shortname varchar(255),
project_description varchar(255),
start_date datetime,
deadline_date datetime,
created_time_stamp datetime NOT NULL,
last_updated_time_stamp datetime
);

CREATE TABLE IF NOT EXISTS users(
user_id int NOT NULL auto_increment PRIMARY KEY,
project_id int,
user_name varchar(255) NOT NULL,
surname varchar(255) NOT NULL,
email varchar(255) NOT NULL,
user_password varchar(255) NOT NULL,
user_role varchar(255),
created_time_stamp datetime NOT NULL,
last_updated_time_stamp datetime,
FOREIGN KEY (project_id) REFERENCES project(project_id)
);

CREATE TABLE IF NOT EXISTS sprint(
sprint_id int NOT NULL auto_increment PRIMARY KEY,
project_id int NOT NULL,
title varchar(255) NOT NULL,
sprint_description varchar(400),
sprint_status varchar(255) NOT NULL,
start_date datetime,
deadline_date datetime,
created_time_stamp datetime NOT NULL,
last_updated_time_stamp datetime,
FOREIGN KEY (project_id) REFERENCES project(project_id)
);

CREATE TABLE IF NOT EXISTS epic(
epic_id int NOT NULL auto_increment PRIMARY KEY,
project_id int NOT NULL,
user_id int NOT NULL,
sprint_id int NOT NULL,
title varchar(255) NOT NULL,
epic_description varchar(400),
epic_status varchar(255) NOT NULL,
priority varchar(255) NOT NULL,
created_time_stamp datetime NOT NULL,
last_updated_time_stamp datetime,
FOREIGN KEY (project_id) REFERENCES project(project_id),
FOREIGN KEY (user_id) REFERENCES users(user_id),
FOREIGN KEY (sprint_id) REFERENCES sprint(sprint_id)
);

CREATE TABLE IF NOT EXISTS ticket(
ticket_id int NOT NULL auto_increment PRIMARY KEY,
project_id int NOT NULL,
sprint_id int NOT NULL,
epic_id int NOT NULL,
reporter_id int NOT NULL,
assignee_id int,
```

```
title varchar(255) NOT NULL,
ticket_description varchar(400),
ticket_status varchar(255) NOT NULL,
ticket_type varchar(255) NOT NULL,
priority varchar(255) NOT NULL,
estimate int,
units_of_measure varchar(255),
created_time_stamp datetime NOT NULL,
last_updated_time_stamp datetime,
FOREIGN KEY (project_id) REFERENCES project(project_id),
FOREIGN KEY (assignee_id) REFERENCES users(user_id),
FOREIGN KEY (reporter_id) REFERENCES users(user_id),
FOREIGN KEY (sprint_id) REFERENCES sprint(sprint_id),
FOREIGN KEY (epic_id) REFERENCES epic(epic_id)
);

CREATE TABLE IF NOT EXISTS comments(
comment_id int NOT NULL auto_increment PRIMARY KEY,
user_id int NOT NULL,
ticket_id int NOT NULL,
comment_text varchar(400) NOT NULL,
created_time_stamp datetime NOT NULL,
FOREIGN KEY (user_id) REFERENCES users(user_id),
FOREIGN KEY (ticket_id) REFERENCES ticket(ticket_id)
)
```

Додаток В.
Обробка виключень *Exception*

Лістинг файлу GlobalExceptionHandler.java

```
@RestControllerAdvice
public class GlobalExceptionHandler {
@ResponseStatus(HttpStatus.BAD_REQUEST)

@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<Object>
handleValidatorException(MethodArgumentNotValidException exception) {
Map<String, String> errors = new HashMap<>();
exception.getBindingResult().getAllErrors().forEach((error -> {
String fieldName = ((FieldError) error).getField();
String errorMessage = error.getDefaultMessage();
errors.put(fieldName, errorMessage);
}));
return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
}

@ExceptionHandler(UserAlreadyExistsException.class)
public ResponseEntity<Object> showUserExistence() {
ExceptionResponse response = new ExceptionResponse();
response.setMessage("User already exists for this email.");
return new ResponseEntity<>(response, HttpStatus.CONFLICT);
}

@ExceptionHandler(UserNotFoundException.class)
public ResponseEntity<Object> showUserAbsence() {
ExceptionResponse response = new ExceptionResponse();
response.setMessage("User is not found.");
return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
}

@ExceptionHandler(ProjectNotFoundException.class)
public ResponseEntity<Object> showProjectAbsence() {
ExceptionResponse response = new ExceptionResponse();
response.setMessage("Project is not found.");
return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
}

@ExceptionHandler(SprintNotFoundException.class)
public ResponseEntity<Object> showSprintAbsence() {
ExceptionResponse response = new ExceptionResponse();
response.setMessage("Sprint is not found.");
return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
}

@ExceptionHandler(EpicNotFoundException.class)
public ResponseEntity<Object> showEpicAbsence() {
ExceptionResponse response = new ExceptionResponse();
response.setMessage("Epic is not found.");
return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
}

@ExceptionHandler(TicketNotFoundException.class)
public ResponseEntity<Object> showTicketAbsence() {
ExceptionResponse response = new ExceptionResponse();
response.setMessage("Ticket is not found.");
return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
}
```

```
@ExceptionHandler(CommentNotFoundException.class)
public ResponseEntity<Object> showCommentAbsence() {
ExceptionResponse response = new ExceptionResponse();
response.setMessage("Comment is not found.");
return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
}

@ExceptionHandler(DateRangeNotCorrectException.class)
public ResponseEntity<Object> showBadRange() {
ExceptionResponse response = new ExceptionResponse();
response.setMessage("DateFormat range is not correct. Start date must be earlier
than deadline date");
return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
}

@ExceptionHandler(UserRoleNotFoundException.class)
public ResponseEntity<Object> showBadUserRole() {
ExceptionResponse response = new ExceptionResponse();
response.setMessage("User role is not correct. There are such roles as
'DEVELOPER', 'QA_ENGINEER', 'ADMIN', 'PROJECT_MANAGER'");
return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
}

@ExceptionHandler(SprintPeriodIsNotCorrectException.class)
public ResponseEntity<Object> showBadSprintPeriod() {
ExceptionResponse response = new ExceptionResponse();
response.setMessage("Sprint period is not correct. It has to last from one to
four weeks.");
return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
}
}
```