МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____Литвиненко О.Є.
«_____» _____2021 р.

ДИПЛОМНИЙ ПРОЄКТ

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ЗДОБУВАЧА ОСВІТНЬОГО СТУПЕНЯ
"БАКАЛАВР"

Тема: Програмна система управління ІТ-проєктами

Виконавець: Слобожан І.А.

Керівник: Марченко Н.Б.

Нормоконтролер: Тупота Є.В.

Київ 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет Кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп’ютеризованих систем управління
Спеціальність 123 «Комп'ютерна інженерія»
(шифр, найменування)

Освітньо-професійна програма «Системне програмування»
Форма навчання денна

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Литвиненко О.Є.
«_____»_____2021 р.

ЗАВДАННЯ
на виконання дипломної роботи (проєкту)

Слобожан Інни Анатоліївни
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема дипломної роботи (проєкту): Програмна система управління ІТ-проєктами затверджена наказом ректора від «4» лютого 2021 р. № №135/ст
2. Термін виконання роботи (проєкту): з 17.05.2021 по 20.06.2021
3. Вихідні дані до роботи (проєкту): технічне завдання для розробки програми, державні стандарти України
4. Зміст пояснювальної записки:
   1) аналіз предметної області і постановка задачі;
   2) опис обраних технологій для розробки;
   3) проєктування бази даних та функціональної схеми системи;
   4) опис процесу розробки та головних модулів системи.
5. Перелік обов'язкового графічного (ілюстративного) матеріалу:
   1) *UML*-діаграма бази даних для системи управління ІТ-проєктами;
   2) діаграма послідовності системи;
   3) *Use-Case* діаграма функцій системи.

6. Календарний план-графік:

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1	Проаналізувати літературу за темою дипломної роботи	17.05.2021 – 19.05.2021	
2	Проаналізувати існуючі рішення та зформулювати постановку задачі	20.05.2021 – 21.05.2021	
3	Спроєктувати базу даних та функціональну схему системи	22.05.2021 – 23.05.2021	
4	Розробити систему за функціональною схемою	24.05.2021 – 15.06.2021	
5	Оформити пояснювальну записку	16.06.2021 – 18.06.2021	
6	Оформити графічний та ілюстративний матеріал	19.06.2021 – 20.06.2021	

7. Дата видачі завдання: «17» травня 2021 р.

Керівник дипломної роботи (проєкту) _____ Марченко Н.Б.
(підпис керівника) (П.І.Б.)

Завдання прийняла до виконання _____ Слобожан І.А.
(підпис випускника) (П.І.Б

РЕФЕРАТ

Пояснювальна записка до дипломного проєкту «Програмна система управління IT-проєктами»: 53 с., 9 рис., 1 табл., 15 літературних джерел, 1 додаток.

Ключові слова: *WEB*-ДОДАТОК, ПРОЄКТ, УПРАВЛІННЯ, *SCRUM*, МЕТОДОЛОГІЯ, СПРИНТ.

Об’єкт дослідження: процес створення програмної системи управління IT-проєктами.

Предмет дослідження: системи управління IT-проєктами.

Мета дипломної проєкту: проєктування та розробка програмної системи управління IT-проєктами.

Методи дослідження: *Microsoft Visual Studio 2019 IDE, C#, ASP.NET Core, ASP.NET Core MVC, Entity Framework, JavaScript, HTML, CSS, PostgreSQL.*

В даному дипломному проєкті був здійснений огляд та аналіз існуючих засобів управління IT-проєктами і була спроєктована та реалізована програмна система управління такими проєктами. Програмна система оптимізує планування та управління проєктами в сфері розробки програмного забезпечення, а також забезпечує прозору комунікацію серед учасників проєкту.

Матеріли дипломного проєкту рекомендується використовувати під час проєктування та розробки *WEB*-додатків.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ІТ – Інформаційні Технології

ПЗ – Програмне Забезпечення

WEB – Всесвітня павутина

LINQ (Language Integrated Query) – Інтегрована мова запитів

SQL (Structured Query Language) – Структурована мова запитів

HTTP (Hypertext Transfer Protol) – Протокол передачі гіпертексту

ВСТУП

Сьогодні проблема чіткого планування та управління роботами, що пов’язані з розробкою програмного забезпечення, стає все більш актуальною. Великі команди розробників, учасники яких можуть працювати з найрізноманітніших куточків світу, та складні проєкти мають величезні бази знань, які повинні бути структурованими для забезпечення якості виконуваної роботи. Не менш важливим елементом організації роботи є забезпечення прозорої комунікації між замовниками, бізнес-партнерами та розробниками. Всі вимоги, документація, рішення, статус виконання тієї чи іншої задачі повинні бути доступними всім учасникам процесу розробки програмного забезпечення.

Для полегшення управління вимогами та ресурсами, пріорітизації задач, планування робіт існують різні підходи та інструменти управління проєктами. Слід виділити наступні найпопулярніші методології: каскадна модель або водоспад та методологія гнучкого управління *Scrum*.

Використання методів управління проєктами значно підвищує продуктивність команди та якість розроблюваного ПЗ, але зі збільшенням проєкту та його учасників збільшується і кількість документів та знань, які повинні бути зафіксовані. Також погіршується комунікація між різними сторонами проєкту. Для того, щоб уникнути такі негативні чинники, необхідно використовувати системи управління IT-проєктами. Серед існуючих нині рішень найбільш популярними є наступні програмні засоби: *Jira* та *Trello*.

Найбільш поширеними системами управління проєктами є системи у вигляді *WEB*-додатків, а саме за використанням клієнт-серверної архітектури.

Актуальність роботи. Існує висока затребуваність в системах управління проєктами, так як вони є щоденним інструментом управління та контролю командам розробників. Особливо такі застосунки необхідні, коли відсутня можливість фізичного контрою над виконавцями проєкту, тобто коли учасники працюють віддалено,але потреба в відслідковуванні їх робочого часу залишається.

Об’єкт дослідження: процес створення програмної системи управління ІТ-проєктами.

Предмет дослідження: системи управління ІТ-проєктами.

Мета дипломного проєкту: проєктування та створення програмної системи управління ІТ-проєктами для визначення та успішного досягнення цілей проєктів з розробки програмного забезпечення шляхом керування обсягом робіт, ресурсами та часом.

Галузь застосування – компанії-розробники програмного забезпечення.

Структура та зміст теоретичної та практичної частини дипломної роботи. Дипломна робота складається зі вступу, трьох розділів та висновків до них.

У першому розділі проведено аналіз предметної області і існуючих засобів, підняті питання теоретичних основ управління проєктами, тобто загальні характеристики систем управління та методології управління в сфері ІТ. Також були сформульвані вимоги до розробки програмної системи управління. Серед функціональних вимог можна перерахувати наступні:

– реєстрація та авторизація користувачів, які мають доступ до певних проєктів;

– відстеження проєктів та їх задач і проблем;

– завантаження та видалення файлів в проєкті;

– створення та конфігурація проєкту;

– створення спринтів та пов’язаних задач;

– створення та редагування документації до проєкту;

– планування випусків продукту;

– фільтрація задач.

Серед технічних вимог до системи виділяються такі:

– відкритість;

– масштабованість;

– кросплатформенність;

– захищеність та безпечність системи;

– висока швидкодія запитів до бази даних.

Другий розділ представляє собою опис архітектури розроблюваного *WEB*-додатку та перелік і опис використовуваних технологій. Також в ньому наводиться схема бази даних та функціональна схема системи.

Третій розділ присвячений опису процесу створення *WEB*-додатку.

РОЗДІЛ 1
ОГЛЯД ІСНУЮЧИХ РІШЕНЬ У СФЕРІ УПРАВЛІННЯ ІТ-ПРОЄКТАМИ І ПОСТАНОВКА ЗАДАЧІ ДЛЯ ДИПЛОМНОГО ПРОЄКТУВАННЯ

1.1. Аналіз предметної області

Успіх проєкту полягає у використанні підходящого ПЗ для командної продуктивності та управління проєктами. Основа ціль таких програмних систем – це дати можливість всім зацікавленим сторонам бути в курсі проєкту. Менеджер та власники повинні бачити загальну картину проєкту, і мати змогу швидко змінювати вимоги та оцінки завдань. Так як проєкти бувають досить великими та складними, без використання допоміжного ПЗ можлива ситуація виходу проєкту з-під контролю.

Щоб уникнути весь всі негативні чинники та їх наслідки, під час управління та планування, необхідно використовувати спеціалізоване ПЗ, яке дасть змогу в повній мірі використовувати обрану методологію управління.

Управління проєктами включає в себе планування та координацію. Будь то малий бізнес, децентралізована команда чи світовий бренд, для управління проєктами повинен бути наявний спеціалізований інструмент.

Розпочинаючи проєкт, кожна компанія розвиває великі надії на корисні результати і стає досить складно контролювати всі процеси, управляти ризиками, коли їх проєкт розростається від самого початку та до повністю готового продукту. Потреба в успішному управлінні проєктами стає необхідністю. Це стає важливою опорою для виконання роботи в чітко визначені терміни.

1.1.1. Характеристика систем управління IT-проєктами

Програмне забезпечення для управління проєктами повинно володіти наступними основними характеристиками:

- хмарні рішення;
- концентрація усіх робочих інструментів в одному місці;
- співпраця двох та більше команд;
- чітке визначення ролі кожного учасника;
- простота документування;
- організація зустрічей;
- відстеження прогресу;
- планування випусків.

Хмарні рішення вже давно є новим фаворитом у світі сфері розробки програмного забезпечення. Найкращим варіантом ПЗ для управління проєктами є хмарні сервіси. Оскільки команди проєктів можуть бути віддаленими та великими, потрібна запезбечити просту комунікацію та співпрацю в таких кмандах, а хмарні технології полегшують цю безперебійну співпрацю. Більше того, хмарні рішення досить економічні порівняно з розгортанням власних серверів та додатків. Їх легше впровадити та забезпечити гнучкість та масштабованість.

Часто виходить так, що на проєкті використовуються різні інструменти для різних функціональних можливостей. Один інструмент для спілкування, інший для управління завданнями, інший для обміну файлами, тощо. Використання програмного забезпечення для управління проєктами дає кращий спосіб управління такими речами. З однією системою команда отримує мжливість поєднати більше своїх інструментів, щоб заощадити час, ніж використовуючи купу інструментів. Отже в подальшому, кожен учасник матимете більше часу, щоб витратити його на свої задачі, замість того, щоб керувати декількома інструментами.

Керуючи великим проєктом, менеджер та його власники мають необхідність постійного спілкування, щоб переконатися, що всі усвідомлені на правильно розуміють деталі проєкту. Спілкування представників різних команд допомагає спростити їх роботу та взаємодію. Важливо інформувати усі команди про певні зміни в якійсь одній частині проєкту та взагалі оптимізувати обмін документами та важливою інформацією, щоб контролювати, скільки роботи зроблено та скільки залишилось.

Важливі функції, якими повинне володіти ПЗ для управління проєктами:

– спільний доступ до файлів, для швидкого обміну та спільного редагування;

– забезпечення швидкої комунікації;

– обмін даними клієнтів, їх контактною інформацією, відгуками та звітами про помилки.

Продуктивність команди залежить від того, хто за що відповідає. Якщо члени команди працюють разом, використовуючи ПЗ для управління проєктом, вони завжди будуть проінформовані про свої завдання та їх зміни без зайвих зусиль. Такі системи дають змогу учасникам знати свої терміни виконання робіть, що від них очікують та прогрес для досягнення цілей. Менеджери можуть доручити їм завдання і встановити кінцевий термін, зробити їх відповідальними за якість виконаної справи.

Немає сумнівів, що документація є важливою частиною управління проєктами. Управління проєктними документами є важливою частиною для кожного розроблюваного продукту. Оскільки документи варіюються від звітів до планів проєкту та файлів, що стосуються конкретних завдань, необхідно мати можливість зв’язувати усю вхідну документацію з конкретними завданнями. Часто для таких цілей використовують зберігання файлів в *Dropbox* і *Google Drive*. Використання програмного забезпечення для управління проєктами гарантує легке читання, впорядковану та точну документацію.

Важливою складовою управління проєктами є зустрічі. За допомогою програмного забезпечення для управління проєктами, менеджер отримує наступні можливості:

– розсилати порядок засідання заздалегідь;
– встановлювати час для початку та закінчення зустрічі;
– створювати нотатки для різних тем, які будуть обговорюватися;
– обговорювати протоколи засідань.

Для різних команд відстеження та продуктивність виглядають по-різному. Обов'язок керівництва - забезпечити фільтрування вузьких місць, щоб це не загрожувало цілям проєкту. Отже, за допомогою інструменту управління проєктами можна централізувати свої завдання, щоб побачити, наскільки продуктивно працює ваша насправді. Таким чином, менеджер може легко відстежувати прогрес кожного члена команди та мати огляд термінів проєкту.

1.1.2. Методології управління IT-проєктами

Незалежно від сфери розробки, будь-який проєкт повинен дотримуватися послідовності дій, які підлягають контролю и управлінню. Згідно з Інститутом управління проєктами (*PMI*), типовий процес управління проєктами включає такі етапи:

– ініціація;
– планування;
– виконання;
– моніторинг;
– закриття проєкту.

Використовані як дорожня карта для виконання конкретных завдань, ці етапи формують життєвий цикл управління проєктами.

На основі вищеописаної класичної основи традиційних методологій, зформувалась ітераційна модель, або водоспад. Таким чином, проєкт

проходить ініціацію, планування, виконання, моніторинг проєкту до його закриття на наступних етапах. Модель водоспаду робить сильний акцент на плануванні та розробці специфікацій, що займає до 40% часу та бюджетного проєкту. Іншим основним принципом цього підходу є чіткий порядок етапів проєкту. Новий етап проєкту починається лише після завершення попереднього.

Метод добре працює для чітко визначених проєктів з фіксованими термінами. Цей підхід вимагає ретельного планування, великої проєктної документації та жорсткого контролю за процесом розробки. Теоретично це повинно стимулювати команду розробників до якісного виконання задач, дотримання бюджету, низьких проєктних ризиків та передбачених кінцевих результатів.

Однак, використовуючи на реальних проєктах, ітераційний метод має тенденцію уповільнювати проєкт, робити його дорогим і негнучками через часті обмеження. У багатьох випадках його нездатність до швидких змін мінімальних вимог часто призводить до величезної втрати ресурсів та можливого занепаду проєкту.

На відміну від традиційного підходу, філософія управління проєктами *Agile* була введена, як спроба зробити програму інженерію більш гнучкою. За даними на 2016 рік, гнучкий підхід став галузевим стандартом в управлінні проєктами.

На відміну від прямолінійної моделі водоспаду, гнучкі проєкти складаються з ряду менших циклів - спринтів. Кожен з них представляє собою швидкий цикл проєкту: він має певні терміни та складається з етапів проєктування, впровадження, тестування та розгортання в рамках заздалегідь визначеного обсягу роботи. У кінці кожного спринту команда готує потенційно можливий новий випуск продукту. Таким чином, з кожною ітерацією до продукта додаються нові функції, що призводять до поступового розвитку проєкту. За допомогою тестування функцій по мірі їх імплементації, шанси випустити потенційно невдалий продукт стає значно нижчим. Етапи життєвого циклу проєкту, який використовує гнучку методологію, наведено на рис. 1.1.

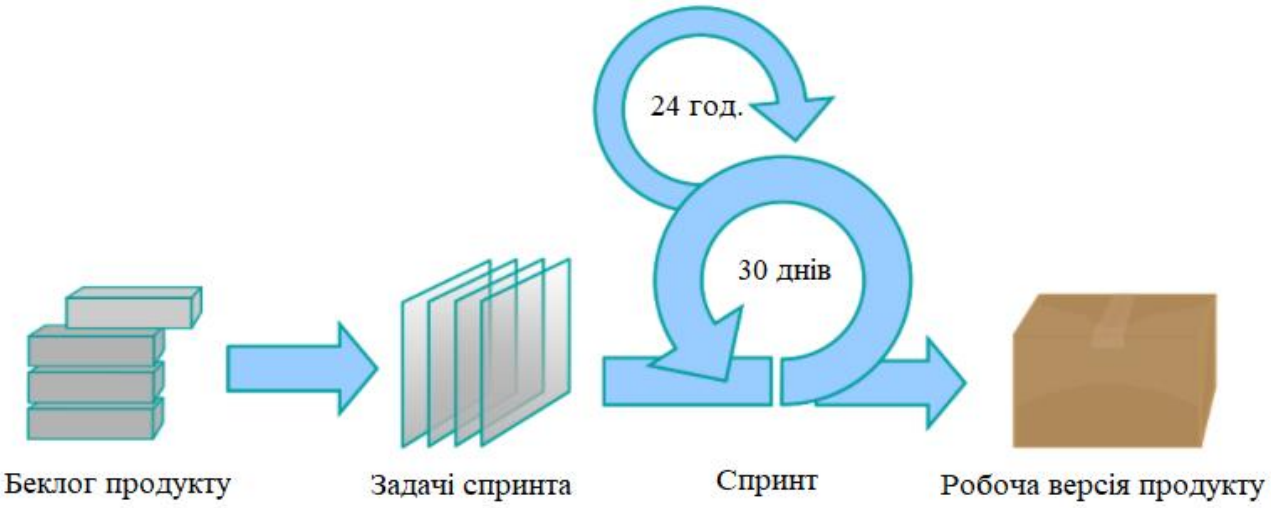

Рис. 1.1. Етапи життєвого циклу проєкту з гнучкою методологією управління

Основні *Agile*-аспекти:

– гнучкість, тобто пріоритети завдань можуть змінюватися з надходженням нових вимог;

– розбиття всього проєкту на невеликі цикли;

– цінність командної роботи, адже саме учасники команд тісно мають чітке бачення своїх потреб та оцінок на задачі;

– проведення оцінювання процесу розробки продукту;

– співпраця з клієнтом, який бере участь у розробці і може змінити вимоги або прийняти пропозиції команди.

Scrum – домінуюча гнучка методологія. Вона спрямована на підтримку міцної співпраці між людьми, які працюють над складними продуктами. Базується на систематичній взаємодії між трьома основними ролями: *scrum*-мастер, власник продукту та команда.

Scrum-мастер – це центральна фігура проєкту. Його головна відповідальність – контролювати процес розробки програмного продукту.

Власник продукту, як правило, клієнт або інша зацікавлена сторона, активно бере участь у проєкті, передаючи глобальні вимоги та забезпечуючи команду зворотнім зв’язком по виконаній роботі після кожного спринту. Команда – це багатофункціональна та самоорганізована група людей, яка відповідає за

впровадження продукту. Команда повинна вміщати в себе до 7 учасників, щоб залишатися гнучкими та продуктивними.

Основна одиниця роботи в *Scrum* – це спринт, тобто короткий цикл розробки, необхідний для забезпечення випуску готового продукту. Спринт може тривати від 1 до 4 тижнів: більша тривалість ітерації знижує передбачуваність та гнучкість, які є основними перевагами такої моделі розробки.

Scrum спирається на три основні артефакти, які використовуються для управління вимогами та відстеження прогресу - беклог продукту, спринт, діаграма спринту. Процес розробки також доповнюється низкою повторюваних зустрічей, таких як щоденні зустрічі (*Stand-up*), планування спринту, огляд та ретроспективні наради.

Беклог продукту – це впорядкований список задач, які можуть знадобитися в кінцевому продукті проєкту. Беклог продукту оновлюється, коли нові вимоги, виправлення, функції та деталі змінюються або додаються.

Спринт – це список завдань, які команда повинна виконати, щоб забезпечити збільшення функціонального програмного забезпечення в кінці кожного спринта. Іншими словами, члени команди домовляються про те, які нові функціональності поставляти, та визначають план, як це зробити.

Діаграма спринту – це ілюстрація роботи, що залишилася у спринті. Це допомагає як команді, так і *Scrum*-мастеру, оскільки вона демонструє повсякденний прогрес і може передбачити, чи буде досягнута мета спринту за розкладом.

Завдяки *Scrum*-зустрічам процес розробки стає більш зрозумілим, серед них щоденні зустрічі (*Standup*), планування спринту, огляд та ретроспективні зустрічі (ретроспектива спринту).

Щоденні зустрічі – це збори обмежені по часу, протягом яких команда розробників координує свою роботу і встановлює план на найближчі 24 години.

Роботи, які потрібно завершити, плануються на планування спринту. Усі учасники спринту беруть участь у цій події. Вони відповідають на два ключові питання: яку роботу можна виконати і як ця робота буде виконана. Планування

спринту триває не більше восьми годин протягом місяця спринту. При коротших спринтах зустріч зазвичай займає менше часу.

Наприкінці кожного спринта команда та власник продукту зустрічаються на огляді спринту. Під час цієї неформальної зустрічі, команда показує завершену роботу та відповідає на питання щодо виконаної роботи над продуктом. Усі учасники обговорють, що робити далі, щоб збільшити якість продукту.

Вся команда вирушає на ретроспективні зустрічі, щоб обміркувати свою роботу під час спринту. Учасники обговорюють, що пішло добре чи не так, знаходять шляхи вдосконалення та планують, як здійснити ці позитивні зміни. Ретроспектива спринту проводиться після огляду та до наступного планування спринту.

Так як *Scrum*-методологія є найбільш популярною на сьогодні, тому розроблена в даній дипломній роботі програмна система управління ІТ-проєктами базується саме на такому методі управління розробкою програмного забезпечення.

1.2. Аналіз наявних засобів управління ІТ-проєктами

На сьогоднішній день існує досить багато систем управління проєктами. В цьому підрозділі наведено опис та аналіз найбільш популярних програмних продуктів у цій сфері.

1.2.1. *Jira*

Jira – це продукт для відстеження задач, що належить компанії *Atlassian*. *Jira* є частиною сімейства продуктів, призначених для допомоги командам усіх типів в управлінні роботою. Спочатку *Jira* була розроблена як трекер помилок та проблем. Але сьогодні компанія *Jira* перетворилася на потужний інструмент управління роботою для будь-яких випадків використання, починаючи від вимог

та управління тестовими кейсами і закінчуючи гнучкою розробкою програмного забезпечення. Приклад інтерфейсу *Jira* наведено на рис. 1.2.

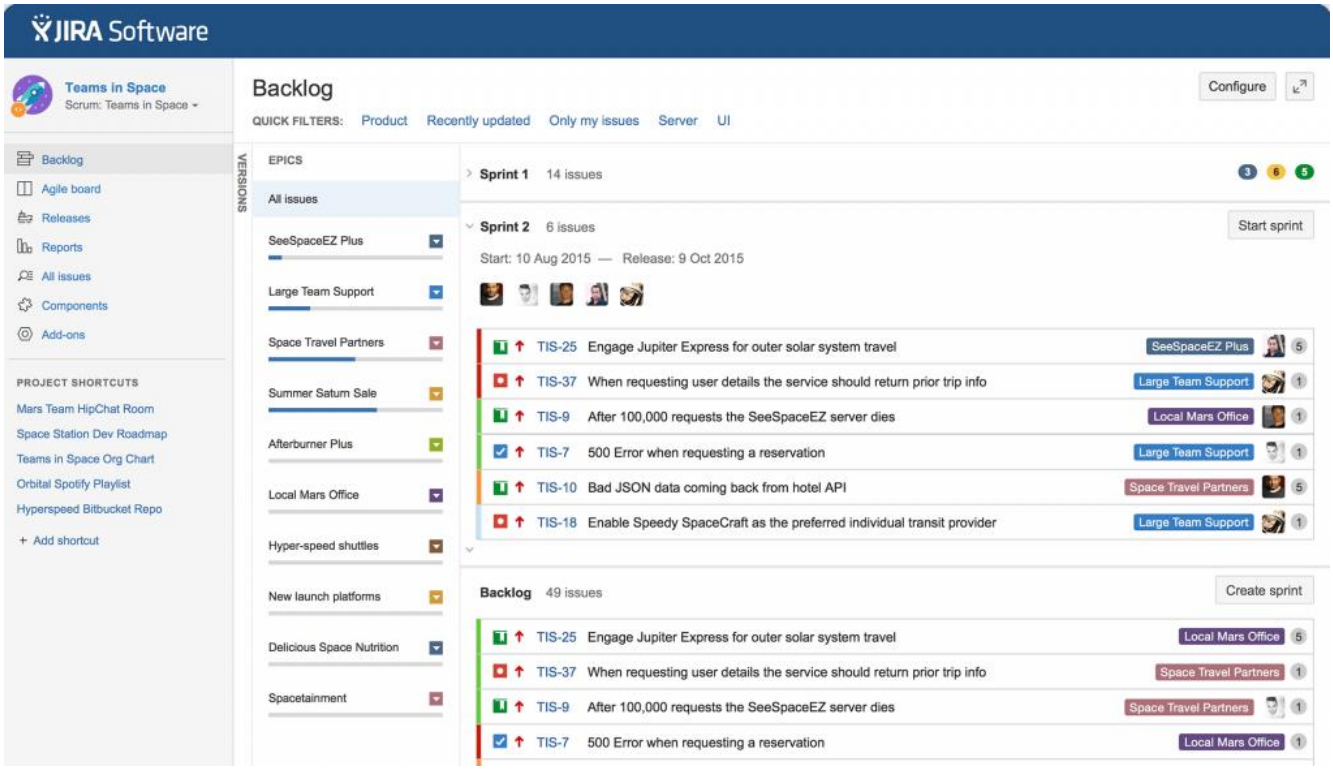

Рис. 1.2. Інтрефейс *Jira*

Для команд, які практикують гнучкі методології, *Jira* пропонує готові дошки *Scrum*. Дошки – це центри управління завданнями, де завдання зіставляються з настроюваними робочими процесами. Дошки забезпечують прозорість у роботі в команді та наочність стану кожного робочого предмета. Можливості відстеження часу та звіти про ефективність роботи в режимі реального часу (діаграми вигорання, звіти про спринт, графіки швидкості) дозволяють командам уважно стежити за своєю продуктивністю з часом.

Jira підтримує будь-яку гнучку методологію розробки програмного забезпечення.

Програмне забезпечення *Jira* можна налаштувати на будь-який тип проєкту. Команди можуть розпочати з шаблону проєкту або створити власний робочий процес. Випуски *Jira*, також відомі як завдання, відстежують кожну роботу, яку потрібно пройти через етапи робочого процесу до завершення. Настроювані параметри дозволяють адміністраторам визначати, хто

може бачити та виконувати які дії. З усією інформацією про проєкт можна створювати звіти для відстеження прогресу і продуктивності.

Jira надає інструменти планування та дорожньої карти, щоб команди могли управляти зацікавленими сторонами, бюджетами та вимогами до функцій з першого дня. *Jira* інтегрується з різноманітними інструментами *CI/CD* (неперервна доставка/неперервне розгортування), щоб полегшити прозорість протягом усього життєвого циклу розробки програмного забезпечення. Коли продукт готовий до розгортання, у випуску *Jira* з'являється інформація про стан виробничого коду. Інтегровані засоби дозволяють командам поступово та безпечно впроваджувати нові функції.

Помилки – це лише назва завдань, що виникають через проблеми в програмному забезпеченні, яке створює команда. Командам важливо переглянути всі завдання та помилки у відставанні, щоб вони могли визначити пріоритетні цілі загальної картини. Потужний механізм робочого процесу *Jira* гарантує, що помилки автоматично призначаються та визначаються за пріоритетами після їх вилучення. Потім команди можуть відстежувати помилку до завершення проєкту.

1.2.2. *Trello*

Trello – це інструмент, розроблений однойменною компанією, який організовує проєкти у вигляді дошки. *Trello* – це найбільш зручне рішення для управління невеликими проєктами. Має інтуїтивно зрозумілий та звичний інтерфейс управління завданнями по гнучким методологіям розробки ПЗ та допомагає всім членам команди візуалізувати хід виконання завдань протягом роботи над проєктом. Картки *Trello* представляють роботу або завдання, які користувачі можуть сортувати, змінювати та переміщувати на різних етапах роботи. Картка може містити опис завдань, коментарі та активність. Користувачі можуть виконувати різні функції, такі як додавання учасників, зміна терміни виконання та вкладення файлів. Також є можливість пов'язувати картки,

вставивши *URL*-адресу як назву картки. Приклад інтерфейсу *Trello* наведено на рис. 1.3.

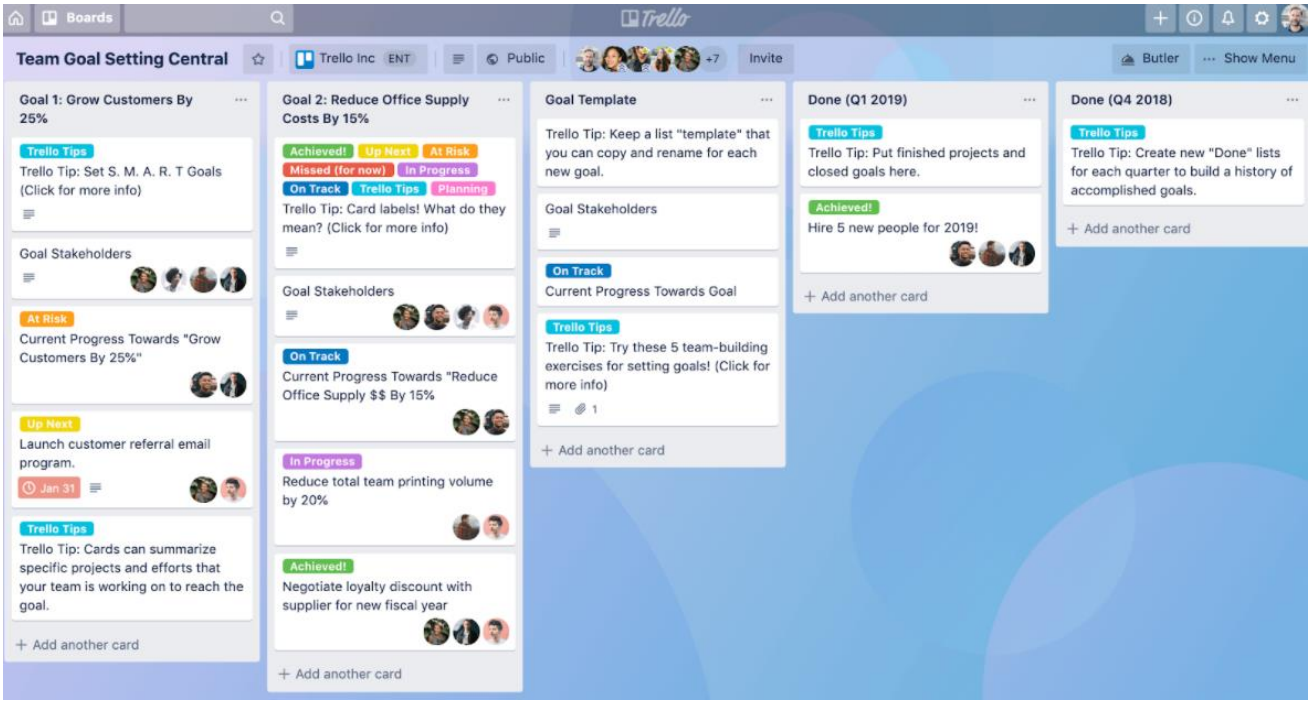

Рис. 1.3. Інтерфейс *Trello*

Trello синхронізує всю інформацію на різних пристроях, будь то настільні комп'ютери, планшети чи мобільні пристрої. Тож, де б не знаходились користувачі, вони можуть співпрацювати з рештою команди з будь-якого місця. Також ця система забезпечує вбудовану автоматизацію для всіх користувачів. Автоматизація повторюваних завдань допомагає командам економити час і сили. Інтеграція електронної пошти дозволяє використовувати електронні листи для створення карток та коментарів. Також доступний безкоштовний публічний *API (Application Programming Interface)* від розробника цієї системи.

1.2.3. Недоліки існуючих рішень

Серед великої кількості переваг та можливостей описаних систем, існують також і недоліки. Серед них можна виділити наступні:

– обмежений вигляд проєкту;

– складність застосування у великих проєктах;

– відсутність вбудованих інструментів створення та редагування документів.

Так вся платформа *Trello* базується на дошках. *Jira* хоч має можливість представляти задачі у вигляді беклогу або дошки, ці системи не мають функції представлення задач у вигляді діаграми Ганта. Дошки *Trello* чудово підходить для невеликих проєктів, але команди програмного забезпечення та організації, що використовують передові гнучкі методології управління проєктами, можуть виявити *Trello* трохи обмеженим в функціоналі.

1.3. Постановка задачі

Програмна система управління IT-проєктами повинна бути розроблена як *WEB*-базована система з клієнт-серверною архітектурою (рис. 1.4) для команд, які використовують методологію *Scrum* при розробці *ПЗ*.

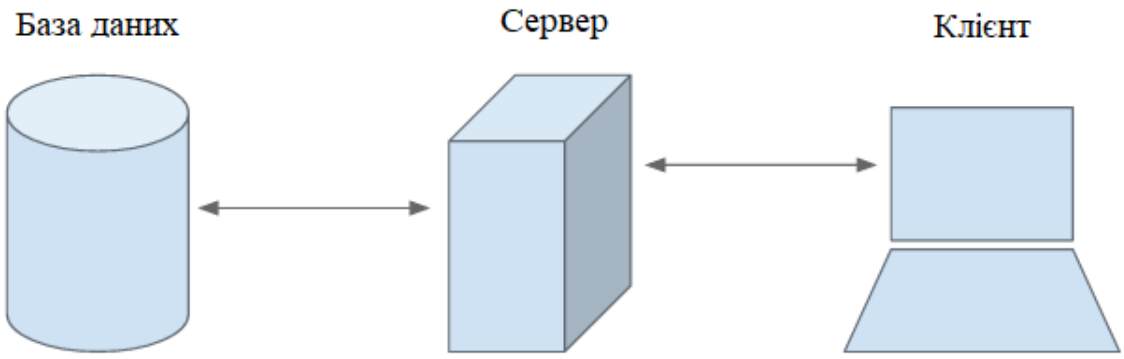

Рис. 1.4. Схема клієнт-серверної архітектури

Можна виділити наступні функціональні можливості такої системи на клієнтській стороні:

– реєстрація та авторизація користувачів, які мають доступ до певних проєктів;

– відстеження проєктів та їх задач і проблем;

– завантаження та видалення файлів в проєкті;

– створення та конфігурація проєкту;
– створення спринтів та пов’язаних задач;
– створення та редагування документації до проєкту;
– планування випусків продукту;
– фільтрація задач.

Також при розробці програмної системи слід приділити особливу увагу до того, щоб інтерфейс такого застосунку був максимально простим та інтуїтивним у використанні.

Серверна частина повинна відповідати наступним вимогам:
– відкритість;
– масштабованість;
– кросплатформенність;
– захищеність та безпечність системи.
– Вимоги до бази даних:
– незалежність даних;
– безпека даних;
– висока швидкодія запитів до бази даних;
– масштабованість.

1.4. Висновки до розділу

В даному розділі було проведено аналіз предметної області, а саме аналіз існуючих методів та інструментів управління IT-проєктами. Були розглянуті основні характеристики, якими повинна володіти така система та її взаємозв’язок з методологіями розробки програмного забезпечення, а саме ітераційна та гнучка (*Scrum*) моделі.

В процесі порівняльного аналізу були розглянуті найбільш популярні на сьогодні системи управління проєктами: *Jira* і *Trello*. Можна виділити наступні спільні характеристики цих систем:

– ці системи є *WEB*-додатками;

– простий та зрозумілий інтерфейс;

– можливість перегляду завдань у вигляді дошки;

– налаштування завдань по різним ознакам (тип задачі, пріоритетність).

Але також були виділені певні недоліки, а саме проблемність використання таких систем у великих проєктах, обмежений вигляд проєкту та відсутність вбудованих інструментів для створення та редагування проєктів.

При порівнянні описаних вище *WEB*-застосунків були враховані всі переваги та недоліки та сформульовано функціональні і технічні вимоги до кожної складової системи, а саме до клієнтського додатку (сайт), серверу та бази даних.

РОЗДІЛ 2
ПРОЄКТУВАННЯ СИСТЕМИ ТА ВИКОРИСТАНІ ТЕХНОЛОГІЇ

2.1. Вибір технологій розробки

Розроблювана програмна система управління IT-проєктами складається з трьох наступних частин: сервер, клієнтський додаток у вигляді *WEB*-сайту та база даних. Для кожної з цих компонент були обрані наступні інструменти розробки:

– сервер – *C#*, фреймворк *ASP.NET Core MVC* на базі платформи *ASP.NET Core, Entity Framework*;

– клієнтський додаток – *JavaScript, HTML, CSS*;

– база даних – *PostgreSQL*.

2.1.1. Мова програмування *C#*

C# – це строго типізована мова, яка дозволяє розробникам створювати різноманітні безпечні та надійні програми на платформі *.NET*. Дає можливості створювати клієнтські програми під *Windows*, *MacOS* та *Unix*-подібних систем, веб-служби, розподілені компоненти та клієнт-серверні програми.

Це об’єктно-орієнтована мова, і вона дає можливість використовувати наступні методи програмування:

– інкапсуляція – моживість приховання внутрішніх даних класу;

– наслідування – можливість створити дочірній клас, який має пряме відношення з батьківським класом;

– поліморфізм – можливість одному методу мати різні реалізації.

Щодо можливостей роботи з пам’яттю, *C#* підтримує вказівники та небезпечний код для прямого доступу до пам’яті.

2.1.2. Платформа *ASP.NET Core*

ASP.NET Core – це сучасна платформа, для кросплатформенної *WEB*-розробки. Ця платформа має ряд наступних особливостей:

– підтримка кількох платформ: програми *ASP.NET Core* можуть працювати на *Windows, Linux* та *MacOS*. Це звільняє розробників від необхідності створювати різні програми для кількох платформ, використовуючи різні фреймворки;

– швидкість роботи, а саме *ASP.NET Core* дозволяє включати лише ті пакети, які потрібні для певної програми, тим самим зменшуючи конвеєр запитів та покращує продуктивність та масштабованість системи;

– інтеграція з сучасними фреймворками для розробки інтерфейсів, такими як *AngularJS, ReactJS, Umber, Bootstrap*, використовуючи *Bower* (веб-менеджер пакетів);

– хостинг, тобто *WEB*-додаток *ASP.NET Core* може розміщуватися на декількох платформах з будь-яким *WEB*-сервером, такими як *IIS, Apache* та іншими.

2.1.3. Фреймворк *ASP.NET Core MVC*

ASP.NET Core MVC – це фреймворк, призначений для розробки *WEB*-додатків на базі платформи *ASP.NET Core*. Його особливість полягає у використанні шаблону проєктування *MVC*. Шаблон *MVC* (модель-представлення-контролер) пропонує три основні компоненти або об'єкти, які будуть використані при розробці програмного забезпечення:

– модель, що представляє собою логічну структуру даних додатку та класів високого рівня, пов'язану з ними та не містить жодної інформації про користувальницький інтерфейс;

– представлення, що складається з набору класів , що представляють елементи, призначені для користувача інтерфейсу (все те, що користувач може бачити і з чим може взаємодіяти);

– контролер, який представляє собою класи, що зв'язують модель і представлення, і використовується для обміну даними між ними.

2.1.4. *Entity Framework*

Entity Framework – це об’єктно-реляційна система відображення об’єктів з відкритим кодом, що підтримується корпорацією *Microsoft*. Ця система підвищує продуктивність розробки, оскільки дозволяє працювати з даними, використовуючи моделі об'єктів класів, не фокусуючись на таблицях бази даних, де ці дані самі зберігаються. Це позбавляє від необхідності розробки додаткового коду для доступу до даних, які зазвичай можуть стрімко змінюватись вродовж роботи над проєктом.

Entity Framework забезпечує абстрактний рівень для розробників для роботи з реляційною таблицею та стовпцями за допомогою доменного об'єкта. Це в свою чергу також збільшує читабельність коду.

Особливості *Entity Framework*:

– кросплатформеність;

– використовування запитів *LINQ* для обробки даних у базі даних замість запитів *SQL*;

– ведення обліку значень, які були змінені у властивостях об’єктів;

– збереження змін в проміжні таблиці, які виконуються операціями вставки, видалення або оновлення;

– паралельний доступ до об’єктів даних;

– автоматичне управління транзакціями;

– кешування результатів часто виконуваних запитів;

– підтримка параметризованих запитів.

2.1.5. Мова програмування *JavaScript*

JavaScript – це мова програмування, яку можна запускати локально у браузері. *WEB*-браузер, незалежно від того, чи є це *Internet Explorer, Safari, Firefox, Opera* чи *Chrome*, вони всі мають механізми *JavaScript*. Операційна система запускає *WEB*-браузер, браузер відкриває сторінку, а сторінка містить *JavaScript*.

Крім того, *JavaScript* може також працювати на сервері за допомогою *Node JS*. *JavaScript* обмежений таким чином, що він не має доступу до файлів операційної системи, на якій він працює. З цієї причини в *JavaScript* немає операторів для відкриття чи збереження файлів локально з міркувань безпеки.

Переваги використання *JavaScript*:

– менше взаємодії з сервером, наприклад перевірка введених користувачем даних перед відправкою їх на сервер, що економить серверний трафік, а це означає менше навантаження на нього;

– негайний зворотний зв’язок до користувачів – їм не потрібно чекати перезавантаження сторінки, щоб побачити, чи не забули вони щось ввести;

– підвищена інтерактивність – надається змога створювати елементи інтерфейсів, які реагують, коли користувач наводить на них курсор миші або активує їх за допомогою клавіатури.

2.1.6. *HTML* і *CSS*

HTML – це мова розмітки гіпертексту, де ключовим словом є саме розмітка. Більше того, *HTML* – це тип мови розмітки. Розмітка – це додаткова інформація,

яка додається до вмісту, до даних чи інших видів інформації, щоб дати вказівки щодо того, як ці дані повинні бути представлений або як їх слід використовувати.

Документ *HTML* – це лише текстовий файл, який можна переглядати в будь-якому браузері таким самим чином, як можна переглядати документ *Word*, наприклад, у *Microsoft Word*.

CSS – це каскадні таблиці стилів, які є елементом дизайну *WEB*-сайтів. *CSS* використовується для додавання стилю на сторінку, адже інакше *WEB*-додаток мав би простий зміст.

CSS допомагає розробникам розмістити всі свої стилі у самостійному документі, а в розмітці *HTML* лише посилатися на них. Таблиця стилів відокремлена від *HTML*, що означає, що може бути створена нова таблиця стилів, що застосовуються до одного і того ж файлу *HTML* і мають зовсім інший стиль для *WEB*-сайту.

2.1.7. *PostgreSQL*

PostgreSQL є однією з найбільш популярних систем управління базами даних. Як повнофункціональна система управління реляційними базами даних з відкритим кодом, *PostgreSQL* надає безліч функцій для підтримки критично важливих транзакцій додатків.

Найбільш конкурентноспроможною особливістю є безпека захисту даних. Це здійснюється через механізми аутентифікації різних підприємств. *PostgreSQL* змушує користувачів використовувати визначені обмеження при додаванні або зміні даних, щоб забезпечити їх якість та безпеку.

PostgreSQL має простий інструмент резервного копіювання – механізм відновлення в точці часу, тобто адміністратори можуть швидко відновити або повернути дані.

На додаток до того, що *PostgreSQL* є безкоштовною та з відкритим кодом, для цієї системи існує багато розширень. Існує дві області, які *PostgreSQL*

підкреслює, коли користувачам потрібно налаштовувати та контролювати свою базу даних. По-перше, ця система у високій мірі відповідає стандартам *SQL*, що збільшує її взаємодію з іншими програмами. По-друге, *PostgreSQL* надає користувачам контроль над метаданими. Однією з ключових відмінностей *PostgreSQL* від стандартних реляційних систем баз даних є те, що *PostgreSQL* зберігає набагато більше інформації в своїх каталогах: не тільки інформацію про таблиці та стовпці, але й інформацію про типи даних, функції, методи доступу, тощо. Ці таблиці можуть бути змінені користувачем, і оскільки *PostgreSQL* базує свою роботу на цих таблицях, це означає, що *PostgreSQL* може бути розширений користувачами. Для порівняння, звичайні системи баз даних можуть бути розширені лише шляхом зміни жорстко закодованих процедур.

2.2. Проєктування схеми бази даних

База даних представляє собою головний ресурс всієї необхідної інформації для розробки та функціонування програмного забезпечення. База даних для розроблюваної програмної системи управління IT-проєктами представлена у вигляді реляційної моделі, тобто основною сутністю такої моделі є відношення між таблицями.

Основними етапами при проєктуванні схеми бази даних слід виділити:

– визначення цілі та призначення системи;
– визначення усіх таблиць (об’єктів) створюваної системи;
– визначення необхідних полів для кожної з таблиць;
– визначення типу кожного з полів та які з них повинні бути унікальними;
– встановлення взаємозв’язків між таблицями.

Для того, щоб визначити ціль системи, необхідно побудувати детальний сценарій її використання та зрозуміти, з якими об’єктами повинен стикатися користувач. Для системи управління проєктами можна виділити наступні цілі:

– управління задачами та ресурсами проєкту;
– управління документацією.

Визначення таблиць є найскладнішим кроком у процесі проєктування бази даних. Це пов’язано з тим, що не завжди дані, які, наприклад, відображаються на сайті, мають таку ж саму структуру і в базі даних.

При проєктуванні таблиць необхідно дотримуватись таких основних принципів:

– таблиця не повинна містити дублікатів;

– кожна таблиця повинна містити інформацію про одну тему.

Таким чином, виникає необхідність у дотримуванні правил нормалізації баз даних.

Перша нормальна форма:

– кожен набір даних не повинен містити дублікатів;

– жоден з атрибутів не можна розділити на декілька;

– кожен набір даних повинен мати первинний ключ.

Друга нормальна форма:

– створення окремих таблиць для наборів значень, які використовуються для кількох сутностей;

– зв'язування таблиць по зовнішнім ключам.

Третя нормальна форма проголошує те, що поля таблиці повинні знаходитись у нетранзитивному відношенні від первинного ключа.

Після пройдення усіх кроків по проєктуванню бази даних, була сформована *UML*-діаграма, яка включає в себе всі об’єкти системи та відношення між ними (рис. 2.1).

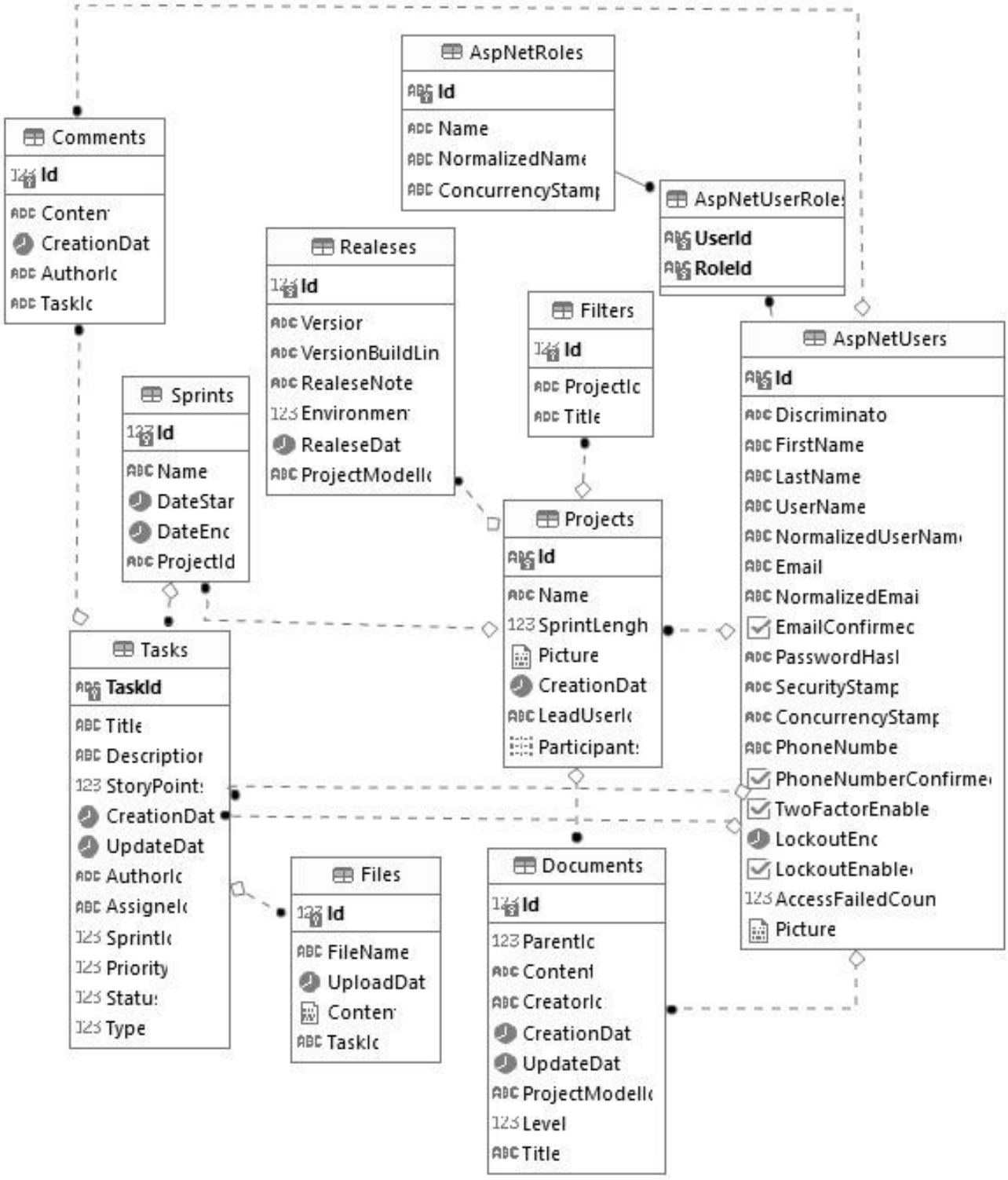

Рис. 2.1. *UML*-діграма бази даних для системи управління IT-проєктами

Вся інформація в базі даних розділена по таблицям. Кожна таблиця являє собою окрему сутність, де її стовпці – це властивості, а рядки – окремі об’єкти. Опис усіх таблиць зі схеми бази даних наведено в таблиці 2.1.

Таблиця 2.1

Описання таблиць бази даних

Назва таблиці	Опис таблиці
AspNetUsers	Список користувачів системи
Projects	Список всіх доступних проєктів для корстувача
Sprints	Список спринтів для кожного проєкту
Filters	Список фільтрів для кожного проєкту
Tasks	Список задач для кожного спринту
Comments	Список коментарів до кожної задачі
Files	Список файлів, що можуть бути прикріплені до задачі
Documents	Список документів, що має проєкт
AspNetRoles	Список доступних ролей на проєкті
AspNetUserRoles	Сисок ролей, якими володіє користувач
Releases	Список усіх випусків продукту

2.3. Функціональна схема

В ході проєктування системи, було виділено наступні основні частини:

– адміністративна, тобто створення та конфігурація проєкту, додавання до нього користувачів та надання їм ролей (адміністратор, менеджер, виконавець) за допомогою окремої сторінки, доступ до якої буде надаватись лише тому, хто створив проєкт;

– управлінська, а саме створення спринтів та задач у рамках одного проєкту, назначення виконавців для задач та виставлення оцінок, планування випусків продукту;

– модуль управління документацією, тобто створення та редагування документів в рамках одного проєкту.

2.3.1. Основні об’єкти системи

Основними об’єктами розроблюваної системи управління проєктами є:

– користувач;

– проєкт;

– спринт;

– задача;

– документ.

Кожен користувач повинен мати роль, а саме адміністратор, менеджер та виконавець. Різні ролі будуть мати доступ до різного функціоналу системи. Так, наприклад, виконавець не зможе створювати спринт, а лише задачі та документи для проєкту, на відміну від менеджера, який може створювати як задачі і документацію, так і спринти, а також планувати випуски продукту. Щодо адміністратора, то цьому користувачу надається доступ до всього функціоналу, який також доступний менеджеру, але також він може додавати нових користувачів до проєкту, або видаляти останній.

Проєкт – це об’єкт системи, який повинен обов’язково мати назву, на основі якої буде також формуватись його *ID*. *ID,* в цьому контексті, представляє собою ключ проєкту, за допомогою якого будуть формуватись посилання для всіх сутностей проєкту, наприклад задачі.

Проєкт складається зі спринтів, документації, плану випусків, та, звичайо ж, користувачів.

Спринт – це об’єкт системи, який представляє собою контейнер для задач. Спринт може бути прив’язаний лише до одного проєкту, складатись з назви та довжини. Так, довжина спринту визначається при створенні проєкту, наприклад 14 днів. Це означає, що при запуску спринта, його задачі будуть відображатись на дошці проєкту, а по завершенню 14 днів, автоматично закриватись.

Задачі в системі управління проєктами представляють собою об’єкти, які повинні обов’язково складатись з наступних частин:

– назва задачі, тобто короткий опис того, що потрібно зробити;

– вимоги до задачі, які можуть містити в собі посилання на документацію;

– назначені виконавці, а саме розробник і тестувальник;

– оцінки часу на розробку і тестування;

– тип задачі (історія, задача, помилка);

– пріоритет задачі (високий, середній, низький);

– позначки приналежності до певної частин проєкту, наприклад, якщо це задача, в якій йдеться мова про зміни стилів на сторінках проєкту, то необхідно позначити цю задачу міткою *UI/UX* (це дозволить користувачам системи застосовувати фільтрацію задач по позначкам та швидше орієнтуватись на дошці).

Також задачі можуть містити в собі різні вкладення, наприклад файли або зображення. Важливою складовою задачі також є коментарі. Кожен учасник проєкту має змогу створювати та редагувати свій коментар.

Документ – це об‘єкт, який складається з заголовку, тексту та зображень. Система повинна надавати користувачеві можливість створення і редагування документу. Також кожен документ можна форматувати, застосовувати різні стилі до його частин, створювати заголовки, змінювати позицію тексту. Вся документація до кожного проєкту представлена у вигляді дерева. Це дозволить спростити навігацію по всій документації та якісно структурувати усі вимоги.

На основі головних об’єктів системи було побудовано діаграму взаємодії (рис. 2.2), в якій наведено основні приклади взаємодії між трьома рівнями системи, тобто між клієнтським додатком, сервером та базою даних.

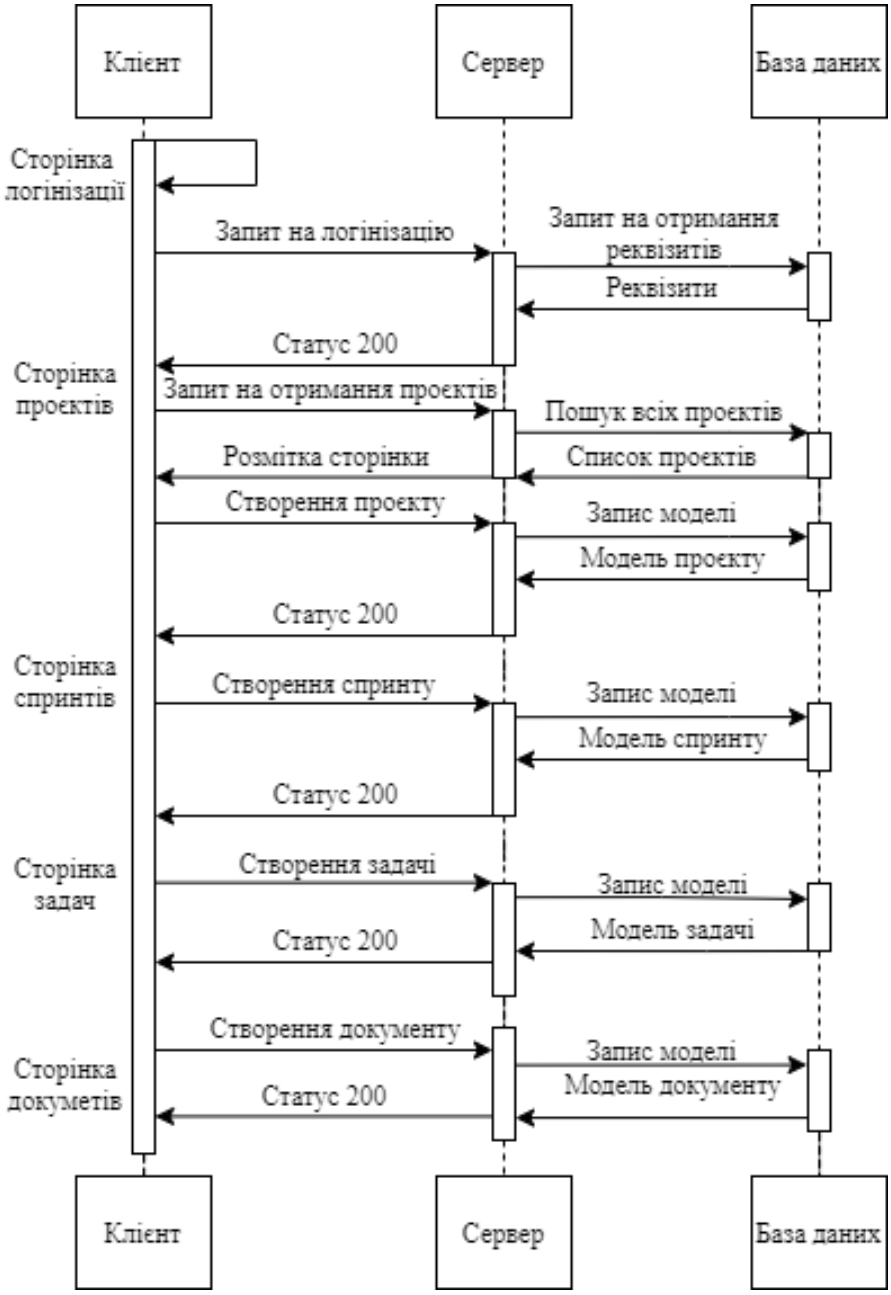

Рис. 2.2. Діаграма послідовності системи

2.3.2. Основні функції системи

Як вже було зазначено попередньому пункті, доступ до різного функціоналу обмежується по ролі користувачів. На рис. 2.3 наведено *Use-Case* діаграму використання системи користувачами з різними ролями.

Рис. 2.3. *Use-Case* діаграма функцій системи

Окрему увагу слід приділити алгоритму логінізації та створення проєкту, адже ці поняття є досить пов’язаними. Особливість полягає в тому, що при реєстрації та логінізації користувача, який ще не є частиною жодного проєкту, він автоматично наділяється правом створювати проєкт. Схему такого алгоритму наведено на рис. 2.4.

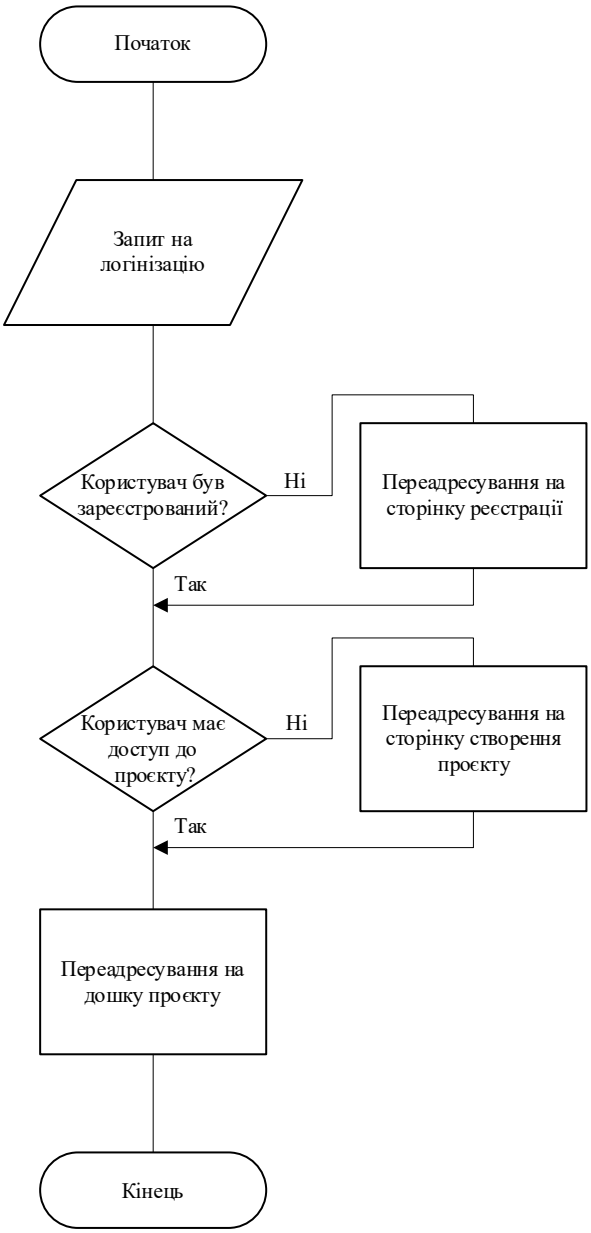

Рис. 2.4. Схема алгоритму логінізації та створення проєкту

Основною сторінкою будь-якого проєкту в системі управління є дошка. Дошка представляє собою сторінку зі списком задач поточного спринта та швидкими фільтрами і розділена на 4 основні колонки, які дають змогу відслідковувати стан задач, а саме:

– не розпочато;
– в прогресі;
– в тестуванні;
– завершено.

Ці стани задач є загальною практикою в розробці програмного забезпечення. Для того, щоб завершити задачу, вона повинна пройти через всі попередні колонки на дошці. Діаграма станів задачі в системі управління проєктами, що базується на методології розробки *Scrum*, зображена на рис. 2.5.

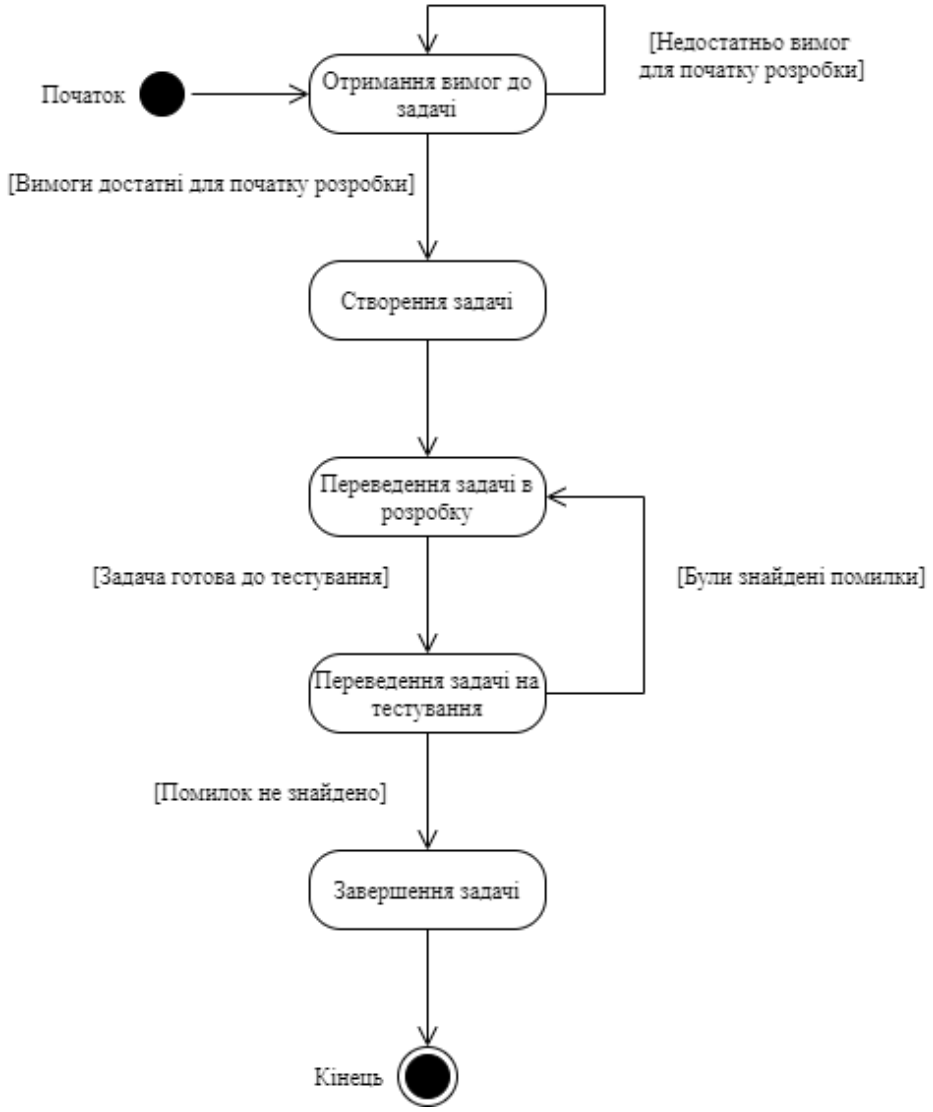

Рис. 2.5. Діаграма станів задачі в системі управління проєктами

2.4. Висновки до розділу

У даному розділі було проаналізовано та обрано інструменти для розробки програмної системи управління ІТ-проєктами.

Так як розроблювана система складається з трьох частин, а саме з серверу, клієнтського додатку та бази даних, був визначений наступний інструментарій для кожної з них:

– сервер – *C#*, фреймворк *ASP.NET Core MVC* на базі платформи *ASP.NET Core, Entity Framework*;

– клієнтський додаток – *JavaScript, HTML, CSS*;

– база даних – *PostgreSQL*.

Була спроєктована та описана схема бази даних, а також наведена діаграма відношень між різними сутностями системи.

Для проєктування потрібного функціоналу системи були визначенні 3 її основні складові :

– адміністративна;

– управлінська;

– модуль створення та редагування документації.

На основі цих складових було виділено головні об’єкти системи (користувач, проєкт, спринт, задача, документ) та навколо них визначено основний функціонал, яким повинна володіти система управління проєктами.

РОЗДІЛ 3
РОЗРОБКА СИСТЕМИ УПРАВЛІННЯ ПРОЄКТАМИ

3.1. Організація клієнт-серверної архітектури

Основною концепцією клієнт-серверної архітектури є розподіл завдань між постачальники ресурсу (які є серверами) і тими, хто запитує цей ресурс (клієнти). Таким чином, саме клієнти визначають, коли сеанс починається і закінчується, так як вони є тими, хто розсилає запити на сервери. Сервери зазвичай працюють у режимі очікування, чекаючи нових пакетів даних від клієнтів, потім оброблюють ці дані та надсилають відповіді.

Якщо дві основні частини цієї структури розділити, то клієнтська сторона не зможе працювати як окремий додаток самостійно, адже клієнт собою представляє додаток, що, використовуючи власний функціонал, викликає методи сервера, відправляє або отримує пакети даних від нього.

Хоча сторона клієнта завжди однозначна і представлена єдиним цілим, серверна сторона може бути дуже складною.

Для роботи з відправкою та отримання даних використовується *HTTP*. Ця технологія забезпечує предачу всіх запитів да відповідей між клієнтом і сервером, використовуючи *HTTP*-методи. В реалізації серверу системи управління проєктами використовується всього 3 типи методів: *GET, POST, DELETE.*

3.2. Маршрутизація адрес

Основним об’єктом в системі, базованій на *ASP.NET Core MVC,* є контролер. Контролер представляє собою клас, який відповідає за наступні дії:

– обробка запитів браузера;

– отримання даних з моделі;

– передача отриманих з моделі даних до шаблону представлення.

Контролери на платформі *ASP.NET Core* використовують проміжне програмне забезпечення, підпрограми, для маршрутизації, узгоджуючи *URL*-адреси вхідних запитів та зіставляють їх з необхідними методами.

Для маршрутизації адрес, а саме отримання доступу до головної сторінки кожного проєкту, використовується *Startup.cs* файл, що створено, на сервері. Використовуючи метод *UseEndpoints,* визначається шлях до строніки проєкту.

```
app.UseEndpoints(endpoints =>
        {
          endpoints.MapControllerRoute(
             name: "default",
             pattern: "{controller=Project}/{action=Index}");
          endpoints.MapRazorPages();
        });
```

В наведеному відривку коду створюється маршрут для сторінки за замовчування, контролер якої було передано, тобто *Project*. За допомогою метода в параметрі *action* відмальовується представлення сторінки. В цьому випадку, це метод за замовчуванням – *Index.*

Також маршрутизація інших сторінок задається в класах-контролерах за допомогою атрибуту *Route*.

Наприклад, нижче приведено реалізацію маршруту до сторінки задачі:

```
[Route("{sprintId}/{projectId}/{taskId}")]
public async Task<IActionResult> Index(string projectId, int sprintId, string taskId)
{
    if (string.IsNullOrEmpty(projectId) || sprintId < 0 || string.IsNullOrEmpty(taskId))
    {
        return BadRequest();
    }
    var project = await this._projectProvider.GetProject(projectId);
    if (project == null)
    {
        return NotFound();
    }
    var sprint = project.Sprints.Find(x => x.Id == sprintId);
    if (sprint == null)
    {
        return NotFound();
    }
    var task = sprint.Tasks.Find(x => x.TaskId == taskId);
    if (task == null)
    {
        return NotFound();
    }
    ViewData["Project"] = project.Name;
    ViewData["ProjectImage"] = project.Picture;
    return View(task);
}
```

Атрибут *Route* визначає, які атрибути будуть прийматись для отримання посилання на задачу, а саме проєкт, спринт, та саму задачу. Також в цьому методі

реалізоване відловлювання таких помилок: не знайдено потрібний проєкт, сприт або задачу. У разі виникнення помилок надсилаються певні коди *HTTP*-статусу. Якщо потрібна задача була знайдена, то повертаться її представленняя.

3.3. Основні частини *WEB*-додатку

WEB-додаток складається з таких основних сторінок:

– сторінка логінізації
– сторінка доступних проєктів для користувача;
– сторінка спринтів;
– сторінка задачі;
– стронка поточного спринта;
– строінка випусків продукту;
– сторінка документації;
– сторініка зі списком всіх доступних користувачів.

3.3.1. Аутентифікація в системі за допомогою *ASP.NET Core Identity*

ASP.NET Core Identity представляє собою *API*, що підтримує користувальницький фнтерфейс всіх дій, пов’язаних за аутентифікацією та дозволя управляти користувачами, їх ролями та клеймами, відправляти та отримувати токени і листи підтвердження. Так як цей модуль є повністю інетгрованим з платформою *ASP.NET Core*, тому при створенні проєкту були підключені всі бібліотеки та файли, що пов’язані для роботи з ним.

Цей модуль дозволив автоматично підлкючити весь необхідний функціонал для роботи з логінізацією та управлінням користувачами.

3.3.2. Головна сторінка

Усі сторінки системи мають єдиний кістяк в своїй основі, в яку вони згодом влаштовуються за допомогою механізів *ASP.NET Core MVC*. Розмітка головної сторінки наведена на фрагменті коду нижче:

```
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-light bg-white border-bottom box-shadow">
            <a class="navbar-brand">Scrum board</a>
            <div class="nav-container">
                <div class="navbar-collapse collapse d-sm-inline-flex">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-controller="Project" asp-action="Index">Проєкти</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-controller="Project" asp-action="Filters">Фільтри</a>
                        </li>
                        <li>
                            <button data-project="@ViewContext.RouteData.Values["projectId"]" id="#create-task-global" class="btn btn-primary">Створити</button>
                        </li>
                    </ul>
                    <ul class="navbar-nav flex-grow-1 flex-end">
                        <li class="nav-item">
```

```
                    <div class="wrap">
                        <div class="search">
                            <input type="text" class="searchTerm"
placeholder="Search...">
                            <button type="submit" class="searchButton">
                                <i class="fa fa-search"></i>
                            </button>
                        </div>
                    </li>
                    <li class="nav-item">
                        <a href="/Identity/Account/Manage"><img
style="height:32px" src="/img/user.svg" /></a>
                    </li>
                </ul>
            </div>
        </div>
    </nav>
    <div class="sidebar">
        <div class="head-container">
            @{
                byte[] projectImage = (byte[])ViewData["ProjectImage"];
            }
            @await Html.PartialAsync("_Picture", new PictureModel { Picture =
projectImage, CssClass = "project-pic", DefaultImage = "/img/user.svg" })
            <text class="project-name">@ViewData["Project"]</text>
        </div>
        <div class="list">
```

```
        <a id="dashboard-item"
href="/Project/@ViewContext.RouteData.Values["projectId"]"><span
class="material-icons">dashboard</span> Панель задач</a>
                <a id="backlog-item"
href="/Backlog/@ViewContext.RouteData.Values["projectId"]"> <span
class="material-icons">reorder</span> Усі задачі</a>
                <a id="realese-item"
href="/Realese/@ViewContext.RouteData.Values["projectId"]"><span
class="material-icons">verified</span> Випуски</a>
                <a id="wiki-item"
href="/Wiki/@ViewContext.RouteData.Values["projectId"]"><span class="material-
icons">description</span> Вікі</a>
                <a id="people-item"
href="/Users/@ViewContext.RouteData.Values["projectId"]"><span class="material-
icons">groups</span> Люди</a>
            </div>
        </div>
    </header>
    <div class="main-container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>
</body>
```

Ця розмітка представляє собою заголовок та бокову панель сайту, які містять елементи-кнопки, за допомогою яких користувач може взаїмодіяти зі системою та пересуватись по ній. Використовуючи *@RenderBody()* відмальовуються усі представлення конртролерів, тобто в це місце на розмітці буде підставлятись саме розмітка представлень необхідних контролерів.

3.3.3. Реалізація шаблону *MVC* на прикладі об’єкту задачі

Для того, щоб забезпечити роботу з таким об’єктом системи як задача, спочатку було створено її модель:

```
public class TaskModel
  {
  [Required, Key, DatabaseGenerated(DatabaseGeneratedOption.None)]
public string TaskId { get; set; }
            [Required]
             public string Title { get; set; }
             public string Description { get; set; }
             public int StoryPoints { get; set; }
             public DateTime CreationDate { get; set; }
             public DateTime UpdateDate { get; set; }
             public TaskStatus Status { get; set; }
             public TaskPriority Priority { get; set; }
             public TaskType Type { get; set; }
             public ApplicationUser Author { get; set; }
             public ApplicationUser Assigne { get; set; }
            public List<CommentModel> Comments { get; set; }
             public List<LogWorkModel> LogWorks { get; set; }
             public List<TaskFileModel> Files { get; set; }
             public SprintModel Sprint { get; set; }
           [NotMapped]
             public int IdSprint { get; set; }
            [NotMapped]
            public string IdProject { get; set; }
   }
```

Далі було реалізовано контролер задачі, який містить в собі опис маршруту до кожної з них та його методи, наприклад створення задачі *Create*:

```
[HttpPost]
[Route("Create")]
public async Task<IActionResult> Create([FromForm] TaskModel model)
    {
        await this._sprintProvider.CreateTask(model);
        return
Redirect($"/Task/{model.IdSprint}/{model.IdProject}/{model.TaskId}");
    }
```

Метод *Create* представляє собою *POST HTTP*-запит, тіло якого повинно передавати серверу модель задачі. Коли запит був переданий, серверний метод *CreateTask* валідує його. Після валідації та запису до бази даних нової задачі, метод виконує переадресування користувача по наступній адресі: */Task/{model.IdSprint}/{model.IdProject}/{model.TaskId}*.

Після цього була реалізована розмітка сторінки задачі.

3.4.3. Логіка роботи з базою даних

Так як при розробці системи використовувався *Entity Framework*, тому для реалізації усіх сутностей бази даних було використано підхід *Code First*, логіка роботи якого полягає у наступному: спочатку описується модель об’єкту, потім функціонал *Entity Framework* створює конструктор цієї моделі, а потім надається можливість визначити усі відношення цього класу до інших і як цей об’єкт повинен відображатись у базі даних.

На наступному фрагменті коду наведено приклад опису відношень класу *TaskModel*:

```
 modelBuilder.Entity<TaskModel>()
.HasOne(x => x.Sprint)
```

```
.WithMany(y => y.Tasks)
.OnDelete(DeleteBehavior.Cascade);
```

Наведена реалізація проголошує наступні відношення: задача може бути прив’язана лише до одного спринту, але у спринта може бути безліч задач. Також проголошується те, що при видаленні спринту з бази даних, послідовно видаляться всі прив’язані до нього задачі.

Після створення усіх моделей та опису їх відношень, були реалізовані методи роботи з базою даних, що поміщують у собі усю необхідно логіку для обробки запитів від користувача. При імплементації методів, був використаний шаблон проєктування *Provider*. *Provider* представляє собою інтерфейс для обробки даних.

На фрагменті коду нижче наведено приклад однієї з можливих взаємодій з базою даних, а саме створення нової задачі:

```
public async Task CreateTask(TaskModel model)
{
  var currentUser = await this._userProvider.GetCurrentUser();
  var project = await this._projectProvider.GetProject(model.IdProject);
  if (project is null)
  {
        throw new ArgumentNullException(nameof(project));
  }
  var sprint = project.Sprints.Find(x => x.Id == model.IdSprint);
  if (sprint is null)
  {
        throw new ArgumentNullException(nameof(sprint));
  }
  model.Author = currentUser;
  model.CreationDate = DateTime.Now;
  sprint.Tasks.Add(model);
  await this._dbContext.Tasks.AddAsync(model);
```

```
    this._dbContext.Sprints.Update(sprint);
    await this._dbContext.SaveChangesAsync();
}
```

Для того, щоб створити нову задачу в базі даних, необхідно передати її модель з *POST*-запиту, а також заповнити деякі поля, інформацію про які можна отримати лише з бази даних, а саме: хто створив задачу, час її створення. Далі до спринту додається задача і викликається повне збереження обновленої бази даних.

3.4. Висновки до розділу

В даному розділі було описано процес розробки системи управління IT-проєктами.

Була описана клієнт-серверна архітектура та як саме вона застосовувалась при розробці даної системи. Також були зазначені загальні підходи та принципи, що застосовувались під час розробки, а саме *MVC, Code First* та *Provider*.

У розділі наведені приклади та пояснено принцип роботи налаштування маршрутизації адрес, як організована аутентифікація та деякі методи, а саме методи створення нових задач, розмітка головної сторінки системи. Описано принцип роботи шаблону *MVC* на прикладі одного об’єкту системи, а саме задачі. Були викладені приклади взаємодії між додатком та сервером, використовуючи *HTTP*-запити.

ВИСНОВКИ

В ході виконнаня допломного проєкту була розроблена програмна система управління IT-проєктами, що складається з бази даних, серверу та клінтського додатку у вигляді *WEB*-сайту.

Дана система була розроблена за допомогою наступного стеку технологій:

– сервер – *C#*, фреймворк *ASP.NET Core MVC* на базі платформи *ASP.NET Core, Entity Framework*;

– клієнтський додаток – *JavaScript, HTML, CSS*;

– база даних – *PostgreSQL*.

Система відповідає поставленим до неї вимогам, а саме володіє наступним функціоналом:

– реєстрація та авторизація користувачів, які мають доступ до певних проєктів;

– відстеження проєктів та їх задач і проблем;

– завантаження та видалення файлів в проєкті;

– створення та конфігурація проєкту;

– створення спринтів та пов’язаних задач;

– створення та редагування документації до проєкту;

– планування випусків продукту;

– фільтрація задач.

У першому розділі був проведений аналіз предметної області, існуючих методів та інструментів управління IT-проєктами, визначено основні характеристики таки систем та виділено їх недоліки на прикладі *Jira* і *Trello*.

За результатами порівняльного аналізу була сформульовані вимоги до кожної з частин системи та постановка задачі, яка складається з двух основних компонент: функціональної та технічної.

У другому розділі було проаналізовано та обрано інструменти для розробки програмної системи управління IT-проєктами.

Була спроєктована та описана схема бази даних і сформована діаграма відношень між різними сутностями системи. Також була визначена функціональна схема системи та основні можливості, якими повинна володіти система управління проєктами.

У третьому розділі було описано процес розробки системи управління IT-проєктами. Був описаний процес організації клієнт-серверної архітектури. Також були зазначені загальні підходи та принципи, що застосовувались під час розробки, а саме *MVC, Code First* та *Provider*. Були привдені фрагменти коду різних частин системи, таких як: методи створення нових задач, розмітка головної сторінки системи.

Матеріли дипломного проєкту рекомендується використовувати під час проєктування та розробки *WEB*-додатків, а розроблену систему управління під час розробки IT-проєктів, як інструмент управління задачами та іншими ресурсами.

СПИСОК БІЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Janoff N.S., Rising L. The Scrum Software Development Process for Small Teams. – IEEE*, 2000. – 32 *p*.

2. *Conway K. Software project management: from concept to deployment. – Coriolis Group,* 2000. – 856 *p*.

3. *Hughes B., Cotterell M. Software project. – McGraw-Hill,* 2002. – 373 *p*.

4. Богданов В. Управление проектами. Корпоративная система шаг за шагом. – Инфра -М, 2010. – 248 с.

5. Рихтер Д. *CLR via C#*. Программирование на платформе *Microsoft .NET Framework* 4.5 на языке *C#*. – Питер, 2017. – 896 с.

6. Фримен А. *ASP.NET MVC5* с примерами на *C#* 5.0 для профессионалов. – Вильямс, 2015. – 736 с.

7. Троелсен Э. Язык программирования *C#* 5.0 и платформа *.NET* 4.5. – Вильямс, 2015. – 1312 с.

8. Резиг Д., Фергюсон Р. *JavaScript* для профессионалов. – Вильямс, 2016. – 240 с.

9. Макфарланд Д. Большая книга *CSS3*. – Питер, 2014. – 608 с.

10. Дронов В. *HTML5, CSS3* и *Web2.0*. Разработка современных *Web*-сайтов. – БХВ-Петербург, 2011. – 414 с.

11. Пасічник В., Пасічник О. Веб-технології. – Магнолія, 2013. – 336 с.

12. Фримен А. *jQuery* для профессионалов. – Вильямс, 2012 – 960 с.

13. Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проєкти) випускників Національного авіаційного університету. – К.: НАУ, 2017. – 63.

14. ДСТУ 3008–95 “Документація. Звіти у сфері науки і техніки. Структура і правила оформлення”.

15. ГОСТ 2.106-96 ЕСКД “Текстовые документы”.

ДОДАТОК А

ВИХІДНИЙ КОД МОДУЛЮ ОБРОБКИ ДАНИХ ПРОЄКТУ

```
public ProjectProvider(ApplicationDbContext dbContext, IUserProvider
userProvider)
{
    this._dbContext = dbContext;
    this._userProvider = userProvider;
}
public async Task<ProjectModel> GetProject(string id)
{
    if (string.IsNullOrEmpty(id))
    {
        throw new ArgumentNullException(nameof(id));
    }
    var projects = await GetProjects();
    return projects?.FirstOrDefault(x => x.Id == id);
}
public async Task AddUser(string id, string userEmail)
{
    if (string.IsNullOrEmpty(id))
    {
        throw new ArgumentNullException(nameof(id));
    }
    var project = await GetProject(id);
    project.Participants.Add(userEmail);
    this._dbContext.Projects.Update(project);
    await this._dbContext.SaveChangesAsync();
}
```

```
public async Task DeleteUser(string id, string userId)
        {
            if (string.IsNullOrEmpty(id))
            {
                throw new ArgumentNullException(nameof(id));
            }
            if (string.IsNullOrEmpty(userId))
            {
                throw new ArgumentNullException(nameof(userId));
            }
            var project = await GetProject(id);
            var user = await GetProjectUsers(id);
            var userToDelete = user.FirstOrDefault(x => x.Id == userId);
            project.Participants.RemoveAll(x => x == userToDelete.Email);
            this._dbContext.Projects.Update(project);
            await this._dbContext.SaveChangesAsync();
        }
public async Task<IEnumerable<ProjectModel>> GetProjects()
        {
            var result = new List<ProjectModel>();
            var user = await this._userProvider.GetCurrentUser();
            var userEmail = user?.Email ?? string.Empty;
            var project = await this._dbContext.Projects
                .Include(x => x.Sprints)
                    .ThenInclude(x => x.Tasks)
                        .ThenInclude(x => x.Assigne)
                .Include(x => x.Sprints)
                    .ThenInclude(x => x.Tasks)
                        .ThenInclude(x => x.Author)
                 .Include(x => x.Sprints)
```

```
                .ThenInclude(x => x.Tasks)
                    .ThenInclude(x => x.LogWorks)
                .Include(x => x.Sprints)
                    .ThenInclude(x => x.Tasks)
                        .ThenInclude(x => x.Comments)
                .Include(x => x.Sprints)
                    .ThenInclude(x => x.Tasks)
                        .ThenInclude(x => x.Files)
                .Include(x => x.LeadUser)
                .Include(x => x.Filters)
                .Include(x => x.Realeses)
                .Include(x => x.Documents)
                .ToListAsync();
            return project.Where(x => x.Participants.Contains(userEmail));
        }
    public async Task CreateProject(ProjectModel model)
        {
            if (model is null)
            {
                throw new ArgumentNullException(nameof(model));
            }
            model.CreationDate = DateTime.Now;
            model.LeadUser = await this._userProvider.GetCurrentUser();
            if (model.PictureFile is not null && model.PictureFile.Length > 0)
            {
                using (var stream = new MemoryStream())
                {
                    model.PictureFile.CopyTo(stream);
                    model.Picture = stream.ToArray();
                }
```

```
            }
            if (model.Participants is null)
            {
                model.Participants = new List<string>();
            }
            model.Participants.Add(model.LeadUser.Email);

            await this._dbContext.Projects.AddAsync(model);
            await this._dbContext.SaveChangesAsync();
        }
    public async Task DeleteProject(string id)
        {
            if (string.IsNullOrEmpty(id))
            {
                throw new ArgumentNullException(nameof(id));
            }
            var project = await this.GetProject(id);
            this._dbContext.Projects.Remove(project);
            await this._dbContext.SaveChangesAsync();
        }
    public SprintModel GetActiveSprint(ProjectModel model)
        {
            if(model is null)
            {
                throw new ArgumentNullException(nameof(model));
            }
            return model.Sprints?.FirstOrDefault(x => x.DateStart <=
DateTime.Now && x.DateEnd >= DateTime.Now);
        }
    public async Task<SprintModel> CreateSprint(string id)
```

```
        {
            if (string.IsNullOrEmpty(id))
            {
                throw new ArgumentNullException(nameof(id));
            }
            var project = await GetProject(id);
            var sprints = project.Sprints;
            var sprintNumber = sprints?.Count() ?? default;
            var sprint = new SprintModel
            {
                Name = $"{project.Name.ToUpper()} спринт {sprintNumber}",
                Project = project
            };
            if(project.Sprints is null)
            {
                project.Sprints = new List<SprintModel>();
            }
            project.Sprints.Add(sprint);
            await this._dbContext.Sprints.AddAsync(sprint);
            this._dbContext.Projects.Update(project);
            await this._dbContext.SaveChangesAsync();
            return sprint;
        }
    public async Task ActivateSprint(string projectId, int sprintId)
        {
            if (string.IsNullOrEmpty(projectId))
            {
                throw new ArgumentNullException(nameof(projectId));
            }
            if (sprintId < 0)
```

```
            {
                throw new ArgumentOutOfRangeException(nameof(sprintId));
            }
            var project = await GetProject(projectId);
            var sprint = project.Sprints.Find(x => x.Id == sprintId);
            if(sprint is null)
            {
                throw new ArgumentNullException(nameof(sprint));
            }
            sprint.DateStart = DateTime.Now;
            sprint.DateEnd = DateTime.Now +
TimeSpan.FromDays(project.SprintLenght);
            this._dbContext.Sprints.Update(sprint);
            await this._dbContext.SaveChangesAsync();
        }
    public async Task<IEnumerable<ApplicationUser>> GetProjectUsers(string
projectId)
        {
            var allUsers = await this._userProvider.GetAllUsers();
            var project = await GetProject(projectId);
            if(project is null)
            {
                throw new ArgumentNullException(nameof(project));
            }
            return allUsers.Where(x => project.Participants.Contains(x.Email));
        }
        public async Task CreateRealese(string projectId, RealeseModel model)
        {
            if (string.IsNullOrEmpty(projectId))
            {
```

```
            throw new ArgumentNullException(nameof(projectId));
        }
        var project = await GetProject(projectId);
        if(project is null)
        {
            throw new ArgumentNullException(nameof(project));
        }
        model.RealeseDate = DateTime.Now;
        model.ProjectModel = project;
        project.Realeses.Add(model);
        await this._dbContext.Realeses.AddAsync(model);
        this._dbContext.Update(project);
        await this._dbContext.SaveChangesAsync();
    }
```