МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____Литвиненко О.Є.
«_____» _____2022 р.

КВАЛІФІКАЦІЙНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ЗДОБУВАЧА ОСВІТНЬОГО СТУПЕНЯ
"МАГІСТР"

Тема: Програмний засіб створення та менеджменту тестової документації

Виконавець: Слобожан І.А.

Керівник: Марченко Н.Б.

Нормоконтролер: Тупота Є.В.

Київ 2022

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.
«_____»_____2022 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи (проєкту)
Слобожан Інни Анатоліївни
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема кваліфікаційної роботи (проєкту): Програмний засіб створення та менеджменту тестової документації
затверджена наказом ректора від «16» вересня 2022 р. №1530/ст
2. Термін виконання роботи (проєкту): з 05.09.2022 по 30.11.2021
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) проєктування програмного засобу;
3) розробка програмного засобу;
5. Перелік обов'язкового графічного (ілюстративного) матеріалу:
1) *UML*-діграма бази даних для програмного застосунку;
2) рівні клієнт-серверної архітектури та взаємодія між ними;
3) діаграма послідовності програмного засобу;
4) *Use-Case* діаграма функцій програмного засобу;
5) алгоритм авторизації та створення проєкту;
6) діаграма станів тест-кейсів.

6. Календарний план-графік:

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1	Проаналізувати літературу за темою кваліфікаційної роботи	05.09.2022-10.09.2022	
2	Проаналізувати існуючі рішення та зформулювати постановку задачі	11.09.2022-16.09.2022	
3	Спроєктувати базу даних та функціональні можливості системи	17.09.2022-30.09.2022	
4	Розробити систему за вимогами	01.10.2022-31.10.2022	
5	Оформити пояснювальну записку	01.11.2022-27.11.2022	
6	Оформити графічний та ілюстративний матеріал	28.11.2022-30.11.2022	

7. Дата видачі завдання: «05» вересня 2022 р.

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

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

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Програмний засіб створення та менеджменту тестової документації»: 71 с., 13 рис., 1 табл., 15 літературних джерел, 1 додаток.

Ключові слова: *WEB*-ДОДАТОК, ТЕСТУВАННЯ, ДОКУМЕНТАЦІЯ, ТЕСТ-КЕЙС, ДЕФЕКТ, ТЕСТ-ПЛАН.

Об’єкт дослідження: процес створення програмного засобу створення та менеджменту тестової документації.

Предмет дослідження: системи створення та менеджменту тестової документації.

Мета кваліфікаційної роботи: проєктування та розробка програмного засобу створення та менеджменту тестової документації.

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

В даній кваліфікаційній роботі був здійснений огляд та аналіз існуючих засобів створення та менеджменту тестової документації і було спроєктовано та реалізовано засіб для таких цілей. Програмний засіб оптимізує процеси створення та підтримки актуальності тестової документації, а також робить їх прозорими для всіх зацікавлених сторін.

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

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ПЗ – Програмне Забезпечення

WEB – Всесвітня павутина

QA (Quallity Assuarence) – Забезпечення якості

LINQ (Language Integrated Query) – Інтегрована мова запитів

*SQL (*S*tructured Query Language)* – Структурована мова запитів

HTTP (Hypertext Transfer Protol) – Протокол передачі гіпертексту

ВСТУП

Сьогодні тестування є невідкладною частиною процесу розробки ПЗ, а не лише додатковою опцією. Проєкти стають все складнішими, а також тенденції на ринку розробки ПЗ, наприклад мікросервісна архітектура, інтеграії зі сторонніми сервісами, просто зобов'язують компаній-розробників ПЗ використовувати різні техніки забезпечення якості продукту. Тому зараз тестувальники є важливими учасниками в процесі розробки, адже сами вони виступають гарантами якості розроблюваних систем та можуть приймати рішення чи володіє система достатнім рівнем якості та чи готова вона до випуску на ринок реальних користувачів. Також зараз тестувальники вже не є лише тими, хто тестує та відповідає за якість ПЗ, а також можуть контролювати якість процесів, що перебігають на проєкті в цілому, а саме якість вимог, дизайн-макетів, підтримка методик розробки ПЗ. Це все являє собою більш ширше поняття, ніж тестування, а саме *QA*.

Тестувальники у своїй роботі, задля забезпечення якості продукту, використовують тестову документацію як інструмент для планування, організції та проведення робіт з забезпечення та підтримки якості. Серед них: тест-план, тест-стратегія, тест-шаблон, тест-кейс, баг-репорт і чек-лист.

Використання тестової документації, а також постійна її актуалізація, на проєкті значно підвищує продуктивність команди тестувальників та якість розроблюваного ПЗ, але зі збільшенням проєкту та його учасників збільшується і кількість документів та тестів, які повинні бути зафіксовані. Для того, щоб уникнути можливі негативні наслідки, а саме неактуальність тестових наборів, необхідно використовувати системи створення та менеджменту тестової документації. Серед існуючих нині рішень найбільш популярними є наступні програмні засоби: *qTest* та *TestRail*.

Найбільш поширеними системами створення та менеджменту тестової документації є системи у вигляді *WEB*-додатків, а саме за використанням клієнт-серверної архітектури.

Актуальність роботи. Існує висока затребуваність в системах створення та менеджменту тестової документації, так як вони є щоденним інструментом для тестувальників. Особливо такі застосунки необхідні тоді, коли проєкти є досить складними, з безліччю різних інтеграцій, а також якщо команди розробників розподілені.

Об’єкт дослідження: процес створення програмного засобу створення та менеджменту тестової документації.

Предмет дослідження: системи створення та менеджменту тестової документації.

Мета дипломного проєкту: проєктування та створення програмного засобу створення та менеджменту тестової документації для спрощення забезпечення якості продукту, що тестується.

Галузь застосування – компанії-розробники програмного забезпечення, компанії, що пропонують послуги з тестування.

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 тестування, а також загальні характеристики систем створення та менеджменту тестової документації. Також були сформульвані технічні та функціональні вимоги до такого застосунку. Серед функціональних вимог можна перерахувати наступні:

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

– відстеження проєктів та їх документів і тестів;

– створення та конфігурація проєкту;

– створення тест-планів та пов’язаних тест-кейсів;

– створення та редагування документації, а саме вимог та тест-стратегії, до проєкту;

– планування тестових робіт;

– завантаження та видалення файлів в проєкті;

– фільтрація тестів по призначеній людині.

До технічних вимог можна віднести наступні:

– 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 описано функціональні можливості такого додатку, та як вони взаємодіють один з одним.

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

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАДАЧІ

1.1. Аналіз предметної області

Перш ніж вивести продукт на ринок, потрібно визначити, чи працюватиме він у будь-якому середовищі, у будь-якій мережі, на чиємусь телефоні, у будь-якому домі, і ми повинні мати високий ступінь впевненості, що він справді відповідатиме цим вимогам. Як ми відповімо на ці запитання, щоб визначити, чи продукт готовий до того, щоб бути випущеним? Щоб перевірити, чи фактичний продукт відповідає очікуваним вимогам і чи він бездефектний, нам потрібно проводити ті самі тести багато разів. Отже, є необхідним ці тести формалізувати та фіксувати, щоб кожен раз бути послідовними у виконанні цих тестів та мати змогу запускати їх знову. Також є необхідність бути достатньо гнучкими, щоб мати справу з будь-якими змінами у вимогах, звідки б вони не виникали, тому вже виконані тести та їх результати повинні бути задокументовані, а тестові сценарії актуалізовані.

Зрозуміло, що процес оцінювання та перевірки того, що програмний продукт або програма роблять те, що вони повинні робити, було б набагато простіше виконувати за допомогою інструменту тестування програмного забезпечення, який дозволяє інженерам із забезпечення якості писати тест-кейси, відстежувати їх виконання та інші конфігурації для тестового середовища та можливість відстежувати тенденції щодо того, що було успішним або невдалим протягом певного часу. Крім того, найкращі методи тестування якості програмного забезпечення включають репорти про проблеми розробникам або відстеження їх до системних вимог, щоб тестувальники могли сказати, що певний тест-кейс підтверджує виконання цієї конкретної вимоги. Тобто репорт про проблему, тобто дефект, формується на основі того чи збігся очікуваний результат з реальним.

Написання тестової документації є основною діяльністю та вважається однією з найважливіших частин тестування програмного забезпечення. Вона використовується командою тестування, командою розробників, а також керівництвом. Якщо для програми немає документації, ми можемо використовувати тестову документацію як базовий документ, в більшій мірі саме тест-кейси.

Чому тест-кейси так важливі? Типове визначення тест-кейсу – це набір умов, за яких тестувальник визначатиме, чи програма або система програмного забезпечення чи одна з його функцій працює належним чином. Це означає, що тест-кейси пояснюють, що потрібно зробити для тестування системи. Вони дають нам кроки, які ми виконуємо в системі, значення вхідних даних, які ми вводимо в систему разом із очікуваними результатами під час виконання конкретного тесту.

Тестові випадки об’єднують весь процес тестування. Якщо тестові приклади готові, вони дійсно допоможуть визначити, справдилися чи ні очікування клієнта. Коли ми виконуємо тест-кейси, ми можемо отримати більше дефектів, які можна пропустити під час тестування без них. Ми можемо отримати відповіді на наведені нижче запитання і отримати уявлення про продукт і його стан. Запитання наступні:

- які модулі були перевірені;
- скільки тестів було виконано;
- скільки модулів або функцій є стабільними;
- які модулі потребують додаткової роботи;
- чи достатні комбінації вхідних даних застосовуються;
- чи тестується програма в іншому браузері або на різних платформах;
- чи відповідає інтерфейс користувача специфікаціям;
- чи достатні тест-кейси;
- вони знаходять дефекти;
- чи готове програмне забезпечення до випуску;
- яка якість програми.

1.1.1. Поняття тестування програмного забезпечення

Тестування програмного забезпечення в життєвому циклі розробки програмного забезпечення – це процес виконання та оцінювання програми з метою пошуку дефектів. Загалом, він зосереджується на будь-якій діяльності, спрямованій на оцінку якості програми чи системи та встановлення того, що вона відповідає вимогам специфікації. Насправді програмна система мало чим відрізняється від інших фізичних систем, де присутні певні вхідні та вихідні дані. Проте програмне забезпечення не залежить від фізичних змін. Як правило, воно не зміниться, доки не відбудуться модифікації, оновлення або зміни вимог користувача. Необхідно перевірити та протестувати всі можливі значення, але неможливо провести вичерпне тестування.

Якщо під час попереднього етапу тестування сталася помилка, і код було випущено, програма тепер може працювати для тестового прикладу, для якого вона, можливо, не була призначена для роботи раніше. У відповідь на це верифікація може бути використана для тестування або перевірки елементів, включаючи код, на узгодженість і відповідність шляхом оцінки результатів відповідно до заздалегідь визначених вимог. Налагодження програмного забезпечення має на увазі, що тестування має навмисно націлюватися на те, щоб щось пішло не так, щоб визначити, чи відбуваються речі, коли вони не повинні, чи не відбуваються, коли повинні. Перевірка передбачає коректність системи, наприклад, це процес перевірки того, чи вимога, яку слід вказати, є тим, що дійсно хоче користувач. Це означає, що процес перевірки встановлює, чи розроблювана система є тим, що хоче клієнт. Тому і верифікація, і валідація необхідні як окремі елементи будь-якої діяльності з тестування.

Існує декілька проблем з якими пов’язане тестування. Перша проблема пов'язана з обмеженнями команди тестування. Недостатність може бути результатом обмеженості ресурсів, нестачі досвіду окремих членів команди або проблеми з керівництвом. Крім того, команді можуть бути недоступні відповідні

засоби тестування. Друга проблема – це неактуальність або некоректність вимог. Це пояснюється тим, що зміни різкі специфікації вимог призводять до необхідності перероблювати та перетестовувати продукт, що може забрати досить багато часу. Третя проблема – мануальне тестування. Команда тестувальників програмного забезпечення зайнята мануальним тестуванням замість того, щоб створювати нові тестові специфікації або актуалізувати старі, щоб вони відповідали новим чи зміненим вимогам. Четверта проблема стосується принципу невизначеності. Іноді невизначеність полягає в точних умовах тестування, а також у тому, як були трактовані умови для реплікації. П'ята проблема полягає у виборі правильних тестів.

Тестування програмного забезпечення не може стосуватися деяких істотних аспектів програми або системи тестування програмного забезпечення, коли тестується лише частина необхідних функцій або вибираються лише очікувані взаємодії під час виконання відмовостійкої програми.

Іншим, але не менш важливим, поняттям в тестуванні є забезпечення якості як процес для покращення якості своєї продукції. Яка різниця між забезпеченням якості та контролем якості? Забезпечення якості – це широкий процес запобігання дефектам та іншим проблемам в якості. Команда *QA* бере участь у всіх етапах розробки продукту: створенні, тестуванні та доставці. Навпаки, контроль якості є більш вузьким процесом. Контроль якості зосереджується на виявленні помилок або пропущених вимог у продукті.

Дві ключові відмінності між контролем та забезпеченням якості: забезпечення є проактивним, а контроль реактивним. Команда забезпечення якості працює на випередження. Вони прагнуть виявити та усунути джерела проблем із якістю, наприклад, людські помилки або помилки в вимогах. У той час як команда контролю якості реагує, перевіряючи продукт на наявність помилок або компонентів, які не відповідають специфікаціям. Ось інший спосіб зрозуміти цю відмінність: контроль якості прагне виявити помилки якості, тоді як забезпечення якості прагне виявити та виправити проблеми, які призводять до помилок якості.

1.1.2. Тестові артефакти

Тестові артефакти – це набір понять, які використуються при розробці та тестуванні програмного забезпечення з метою формалізувати та задокументувати різні процеси та результати цих двух подій. Тестові артефакти є фундаментом та базовими інтструментами в плануванні та проведенні робіт з перевірки якості продукту.

Одним з основних артефактів є тестова стратегія – це документ високого рівня, який описує техніку тестування, яка використовується в життєвому циклі розробки програмного забезпечення, і підтверджує типи або рівні тестування, які будуть виконуватися на продукті. Стратегію тестування неможливо змінити після того, як вона написана та прийнята керівником проекту та командою розробників. Крім того, тестова стратегія передбачає наступні деталі, які необхідні під час написання тестового документа:

- яку техніку тестування необхідно застосувати;
- який із модулів буде перевірятися;
- які критерії застосовуються для початку та завершення тестування.

Під час створення тестової стратегії визначається обсяг і огляд, тобто огляд проекту разом із цільовою аудиторією документа. Крім того, визначаються дії з тестування разом із їхніми часовими рамками. Тестовий підхід – на цьому кроці визначається повний процес тестування, а також ролі та обов’язки кожного члена команди. У цьому розділі також визначаються такі речі, як тип тестування, який потрібно виконати, використання засобів автоматизації. Тестове середовище – на цьому кроці ми визначаємо інше тестове середовище, яке використовується в процесі тестування. Наприклад, може бути окреме середовище тестування для різних учасників або зацікавлених сторін. Інструменти – тут ми можемо визначити різні інструменти, які використовуються для різних операцій, наприклад – інструменти для керування тестами, автоматизованого тестування, тестування продуктивності, тестування безпеки.

Керування випусками – у цьому розділі визначається план керування випусками, який допомагає підтримувати відповідну історію версій. Серед ризиків визначаються різні ситуації, пов'язані з тестуванням, а також стратегії пом'якшення. Огляд і схвалення – різні розділи діяльності, що переглядаються та підписуються різними зацікавленими сторонами.

Існують різні типи тестових стратегій. Найбільш популярною є аналітична стратегія. Наприклад, тестування на основі ризиків і тестування на основі вимог – це два типи тестування. Після вивчення умов тестування, таких як ризики або вимоги, команда тестування встановлює обставини тестування, які повинні бути розглянуті. У випадку тестування на основі вимог вимоги перевіряються для визначення обставин тестування. Потім створюються, впроваджуються та запускаються тести, щоб переконатися, що вимоги виконуються. Навіть висновки відстежуються з точки зору вимог.

Стратегія на основі моделі: команда тестування вибирає фактичну або очікувану обставину та будує для неї модель, враховуючи вхідні дані, результати, процеси та можливу поведінку. Моделі також створюються на основі існуючого програмного забезпечення, технологій, швидкості передачі даних, інфраструктури та інших факторів.

Стратегія відповідності стандартам або процесу: цей метод є гарним прикладом медичних систем, які дотримуються вказівок Управління з контролю за продуктами й ліками США (*FDA*). Тестувальники дотримуються методів або рекомендацій, встановлених комітетом зі стандартів або групою спеціалістів підприємства, щоб визначити умови тестування, визначити тестові випадки та зібрати команду тестування. У випадку *Agile*-проєктів тестувальники створять повну стратегію тестування для кожного функціоналу від клієнта, починаючи зі встановлення критеріїв тестування, розробки тестових випадків, проведення тестів, звітування про статус.

Реактивна стратегія: тести розробляються та впроваджуються лише після випуску справжньої програми. Як наслідок, тестування базується на несправностях, виявлених у реальній системі.

Консультаційна стратегія: подібно до того, як у керованому користувачем тестуванні використовуються дані ключових зацікавлених сторін для встановлення обсягу умов тестування, ця техніка тестування також працює. Є сценарій, у якому оцінюється сумісність браузера будь-якої веб-програми. У цьому розділі власник програми надасть список браузерів та їх версій у порядку переваги. Вони також можуть містити список типів з'єднань, операційних систем, програмного забезпечення для захисту від зловмисного програмного забезпечення та інших вимог до програми, на відповідність якій потрібно перевірити. Залежно від пріоритету елементів у наданих списках тестувальники можуть використовувати різні стратегії, такі як попарне тестування або еквівалентний поділ.

Стратегія протидії регресії: у цьому випадку процедури тестування спрямовані на зниження ризику регресії як для функціональних, так і для нефункціональних аспектах продукту. Використовуючи веб-додаток як приклад, якщо програму потрібно перевірити на проблеми регресії, команда тестувальників може розробити автоматизацію тестування як для звичайних, так і для незвичних випадків використання. Вони також можуть використовувати засоби автоматизації на основі графічного інтерфейсу користувача для проведення тестів кожного разу, коли програма оновлюється.

Досить схожим поняттям на тестову стратегію є поняття тест-плану. Але все ж таки існують певні відмінності. Тест-план – це тестовий артефакт, що описує дії, які відбуватимуться під час процесу тестування – від стратегії розробки до критеріїв пошуку помилок. Тест-план дуже важливий, оскільки він підсумовує процес тестування. План розбитий на керовані частини, щоб ми знали, як впоратися з кожним аспектом цього процесу. І це запис наших цілей, тож ми можемо озирнутися назад і побачити, як нам це вдалося. Існує декілька етапів створення тест-плану:

- аналіз продукту, що розробляється;
- розробка стратегії тестування;
- визначення цілей тестування;

– визначення критеріїв старту та завершення тестування;
– планування ресурсів;
– опис тестового середовища;
– складення графіку і оцінка потреби в часі.

Одним з найважливіших понять в тестуванні є тест-кейси. Вони визначають, як тестувати систему. Тест-кейс – це окремий набір дій або інструкцій для виконання тестувальником, який перевіряє певний аспект продукту чи функціональності програми. Якщо тест провалиться, результатом може бути дефект програмного забезпечення. Тестувальник або фахівець із забезпечення якості зазвичай пише тест-кейс, які запускаються після завершення імплементації функції або групи функцій, які складають реліз. Тест-кейси також підтверджують, чи відповідає продукт вимогам програмного забезпечення. Група тест-кейсів завжди організована в набір тестів, який перевіряє логічний сегмент програми, наприклад певну функцію. Існують різні типи тест-кейсів. Ці типи зумовлені насамперед специфікою, яку потрібно застосувати при тестуванні різних аспектів програми. Тестування може бути розділене на дві великі категорії: функціональне та нефункціональне.

Функціональне тестування – це тип тестування програмного забезпечення, при якому базові функції програми перевіряються на відповідність заздалегідь визначеному набору специфікацій. Використовуючи методи тестування чорної скриньки, функціональні тести вимірюють, чи даний вхід повертає бажаний результат, незалежно від будь-яких інших деталей. Результат є завжди двіковим: тести проходять чи не проходять. Функціональне тестування є важливим, оскільки без нього ви можете не точно зрозуміти, чи ваша програма працює належним чином. Програма може пройти нефункціональні тести та в іншому випадку працювати добре, але якщо вона не надає ключових очікуваних результатів кінцевому користувачеві, програма не може вважатися робочою. Функціональні тести перевіряють, чи виконуються визначені функціональні вимоги, де нефункціональні тести можна використовувати для перевірки

нефункціональних речей, таких як продуктивність, безпека, масштабованість або якість програми.

Іншими словами, функціональне тестування стосується того, чи працюють ключові функції, а нефункціональні тести більше стурбовані тим, як відбуваються операції. Існує багато типів функціональних тестів, які можна виконати під час тестування програми.

Модульне тестування. Розбиває бажаний результат на окремі блоки, дозволяючи перевірити, чи невелика кількість вхідних даних дає бажаний результат. Модульні тести, як правило, є одними з найменших тестів для швидкого написання та виконання, оскільки кожен розроблений для охоплення лише однієї частини коду (функції, методу, об’єкта тощо) та перевірки його функціональності.

Димове тестування. Проводиться, щоб переконатися, що найбільш критичні частини програми працюють належним чином. Це перший прохід через процес тестування, і він не має на меті бути вичерпним. Таке тестування гарантує, що програма працює на базовому рівні. Якщо ні, то немає потреби переходити до більш детального тестування, і програму можна відразу повернути до команди розробників на доопрацювання.

Санітарне тестування. Є схожим поняттям до димового тестування, оскільки воно також призначене для перевірки базової функціональності та потенційного уникнення детального тестування зламаного програмного забезпечення. Різниця полягає в тому, що такі тести виконуються пізніше в процесі, щоб перевірити, чи мала нова зміна коду бажаний ефект. Це перевірка працездатності певної зміни, щоб визначити, чи новий код працює приблизно так, як очікувалося.

Інтеграційне тестування. Визначає, чи правильно функціонують комбінації окремих програмних модулів разом. Окремі модулі, можливо, вже пройшли незалежні тести, але коли вони залежать від інших модулів для успішної роботи, такий вид тестування необхідний, щоб переконатися, що всі частини працюють разом, як очікувалося.

Регресійне тестування. Гарантує, що додавання нового коду не порушує роботу існуючих функцій. Регресійні тести націлені на внесені зміни та гарантують, що вся програма продовжує залишатися стабільною та функціонувати належним чином.

Тестування *UI/UX*. Оцінює графічний інтерфейс користувача програми. Продуктивність компонентів інтерфейсу користувача, таких як меню, кнопки, текстові поля тощо, перевіряються, щоб переконатися, що користувацький досвід є ідеальним для користувачів програми. Тестування *UI/UX* також відоме як візуальне тестування та може бути ручним або автоматизованим.

Інші класифікації функціонального тестування включають тестування чорного ящика, тестування білого ящика, тестування компонентів, тестування *API*, тестування системи та тестування виробництва.

Існує багато підходів до оптимізації функціонального тестування, які можна використати для розробки набору тест-кейсів для цього. Перевірки граничних значень оцінюють, що станеться, якщо вхідні дані надходять за межі встановлених обмежень, наприклад користувач вводить занадто велике число (якщо є визначене обмеження). Тест-кейси на основі прийняття рішень перевіряють результати після того, як користувач вирішить виконати дію, наприклад очистити історію. Користувальницькі тести оцінюють, як компоненти працюють разом у програмі. Інші поширені методи функціонального тестування включають тестування на еквівалентність, тестування альтернативного потоку, позитивне тестування та негативне тестування.

Нефункціональне тестування – це тип тестування, який використовується для оцінки продуктивності програмного додатку, зручності використання, надійності та інших нефункціональних характеристик. Він призначений для перевірки готовності системи відповідно до нефункціональних критеріїв, які функціональне тестування ніколи не враховує. Нефункціональне тестування є важливим для підтвердження надійності та функціональності програмного забезпечення. Специфікація вимог до програмного забезпечення (*SRS*) служить основою для цього методу тестування програмного забезпечення, який дозволяє

групам із забезпечення якості перевірити, чи відповідає система вимогам користувача. Підвищення зручності використання, ефективності, та мобільності продукту є метою нефункціонального тестування. Це допомагає знизити виробничий ризик, пов’язаний із нефункціональними компонентами продукту. Нижче наведено параметри нефункціонального тестування.

Безпека. Цей параметр визначає, як система захищена від запланованих і незапланованих вторгнень із внутрішніх і зовнішніх джерел.

Надійність. Послідовність, з якою програмна система виконує необхідні завдання без помилок.

Ефективність. Потужність, кількість ресурсів і час відповіді, якими може керувати програмна система.

Стабільність. Параметр визначає залежність користувача від системи під час роботи.

Масштабованість. Описує, наскільки програмне забезпечення може збільшити свою обчислювальну потужність, щоб впоратися зі зростанням попиту.

Також одним з найважливіших понять в тестуванні є дефект. Дефект – це недолік або помилка в програмі, яка обмежує нормальний через невідповідність очікуваної поведінки програми з фактичною. Дефект виникає, коли розробник припускається будь-якої помилки під час проєктування чи створення програми, і коли цей недолік виявляє тестувальник, це називається дефектом. Обов’язком тестувальника є проведення ретельного тестування програми, щоб знайти якомога більше дефектів, щоб гарантувати, що якісний продукт досягне споживача. Важливо розуміти життєвий цикл дефекту, перш ніж переходити до робочого процесу та різних станів дефекту. Життєвий цикл дефекту – це цикл, з якого дефект проходить, охоплюючи різні стани протягом усього свого життя.

1.2. Аналіз існуючих засобів створення тестової документації

На сьогоднішній день існує багато систем створення та менеджменту тестової документації. В цьому підрозділі наведено опис та аналіз найбільш популярних програмних продуктів у цій сфері.

1.2.1. QTest

QTest – це інструмент керування тест-кейсами, який використовується для керування проєктами і відстеження помилок. Він слідує концепції централізованого управління тест-кейсами, яка допомагає легко спілкуватися і допомагає швидко розробляти завдання для всієї команди *QA* та інших зацікавлених сторін. Використовується для відстеження всіх дій із забезпечення якості безпосередньо з першої фази життєвого циклу тестування програмного забезпечення. Підтримує управління випусками, управління збірками і управління модулями. Корисний при виконанні завдань з тестування, таких як опис тестових випадків, виконання, звіти та ін. Також може бути корисним в управлінні проєктами, відстеженні завдань, управлінні вимогами та управлінні тестами.

QTest дозволяє створювати досить багато видів тестової документації. План тестування – це дозволяє тестувальникам відстежувати розклад збірки. Вимоги – в них можна вводити вимоги або історії користувачів, а також можна створювати тестові випадки безпосередньо з вимог, тому вони автоматично зв’язуються. Дизайн тестів – тут можна створювати свої тест-кейси. Виконання тесту – надається можливість планування свого циклу тестування та структурувати набір тест-кейсів і прогони тестів. Усі результати кожного проведеного тесту записуються. Звіти – тут можна отримувати всілякі корисні дані та налаштувати свої звіти так, щоб вони відображали все, що забажаєте, деталізувати окремі помилки або створити огляд високого рівня, відфільтрований за датою чи полем.

QTest є хмарним рішенням, тому можна помітити деяку затримку залежно від навантаження, яке витримує інтернет-з'єднання. Це також означає, що тестування припиниться, якщо з'єднання перерветься. Що стосується інструментів керування тестуванням, можна досить швидко почати роботу з *QTest*. Якщо є потреба імпортувати багато існуючих даних в різних форматах, то ця система дозволяє досить швидко встановити потрібні налаштування. Коли ви будете готові до роботи, у верхньому правому куті з'явиться зручна піктограма сповіщень, яка нагадує поточний потік оновлень у реальному часі, який інформує вас про будь-які зміни у вашому проєкті. Це дуже корисно з точки зору управління, оскільки це дає змогу бачити проблеми, коли вони виникають, і натискати безпосередньо на звіти про дефекти чи результати тестування. Інструмент керування тестами досить добре виконує свою роботу, автоматично зв'язуючи записи та заповнюючи дані за вас, де це можливо. Такі параметри, як можливість клонувати помилку, значно економлять час. Це робить його швидким і простим у використанні. Кожна дія в системі записується, тому ніколи не виникає сумнівів щодо того, хто що зробив, і є можливість відстежити дефект від вирішення аж до його виявлення. Інтерфейс *QTest* наведено на рис. 1.1. та 1.2.

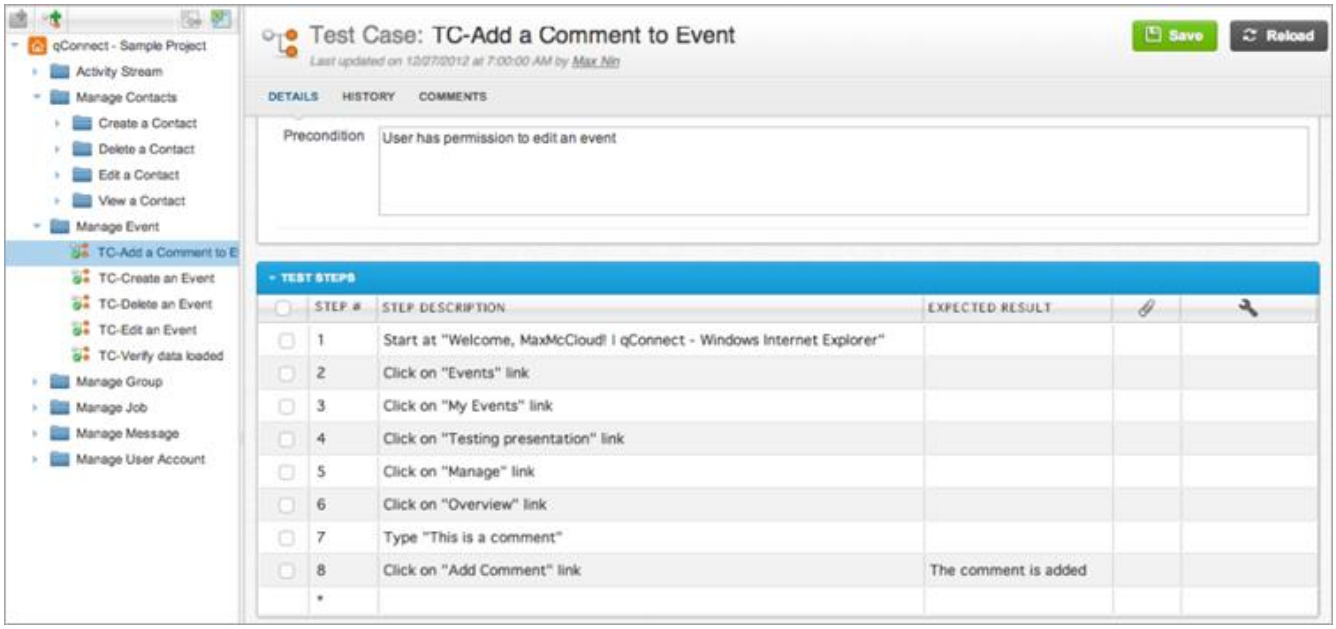

Рис. 1.1. Інтрефейс *QTest.* Вікно тест-дизайну

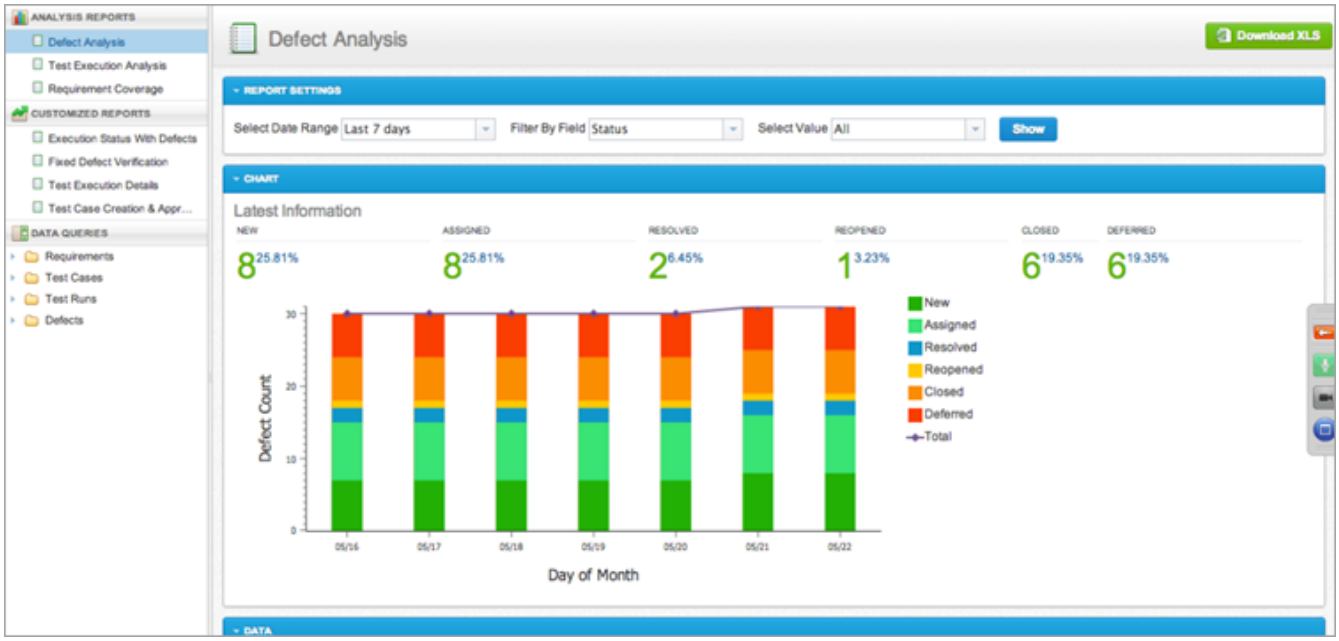

Рис. 1.1. Інтрефейс *QTest.* Вікно звітів

1.2.2. *Jira Zephyr*

Jira Zephyr полегшує можливості керування тестами для будь-якого проєкту *Jira*. Використовуючи *Zephyr*, тести можна створювати всередині *Jira* та виконувати, коли потрібно, або як частину циклу тестування. За допомогою певних налаштувань можна переглядати та відстежувати докладні тестові показники. Серед основних можливостей можна перерахувати наступні:

- створювати, переглядати, редагувати, клонувати та виконувати тести;
- тестування безпосередньо всередині *Jira*;
- пов’язування тестових прикладів з задачами;
- логічне групування тестів в тест-плани;
- планування циклів виконання тестів;
- налаштування та відстежування показників якості.

Zephyr дозволяє користувачеві створювати кілька тестових циклів і переглядати їх у відповідних версіях. Тести можна додавати окремо до наявного циклу тестування або клонувати як групу з іншого циклу тестування (також один

тест-кейс може бути частиною кількох циклів тестування). Після того, як усі тести додано до циклу тестування, фаза виконання буде чітко організована, а відстеження результатів стане легким. Під час виконання панель відстеження прогресу відображає виконані тести та тести, що залишилися, у відсотках. Після завершення виконання для поточного циклу тестування можна наступний цикл тестування (зазвичай пов'язаний із наступною версією) і клонувати потрібні тести в цей цикл.

Опція виконання тестів дозволяє користувачеві виконувати будь-який тест-кейс після доступу до певного циклу тестування; користувач може навіть запустити тест негайно, не включаючи його в будь-який цикл тестування.

Zephyr має просту та базову схему звітності, як показано на рис. 1.3. Для конкретного проєкту загальну кількість виконаних тестів і кількість тестів, що залишилися, можна побачити в підсумку тестів. Тести також можна переглянути на основі конкретної версії або будь-якої конкретної позначки, під якою вони згруповані.

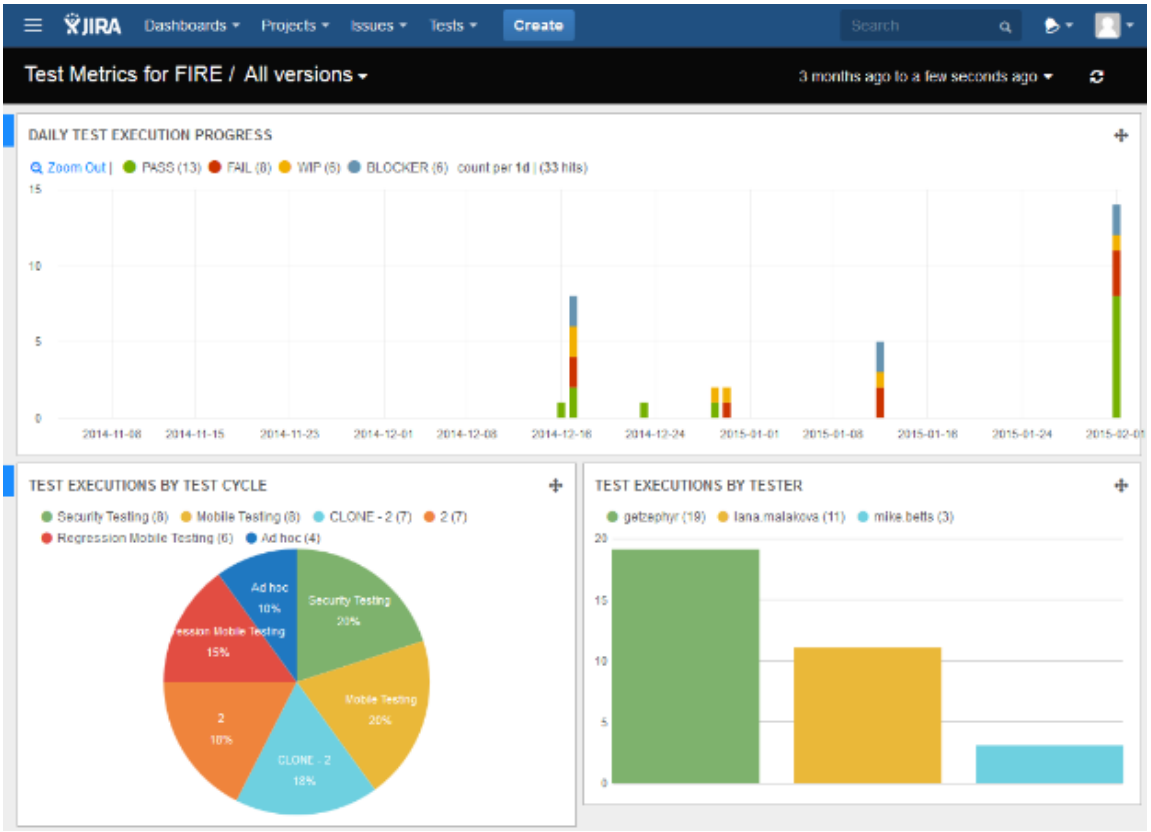

Рис. 1.3. Інтрефейс *Jira Zephyr*. Вікно звітів

Zephyr має досить зрозумілий інтерфейс, що дозволяє швидко розібратись в цій системі. Приклад результатів прогонів тест-кейсів наведено на рис. 1.4.

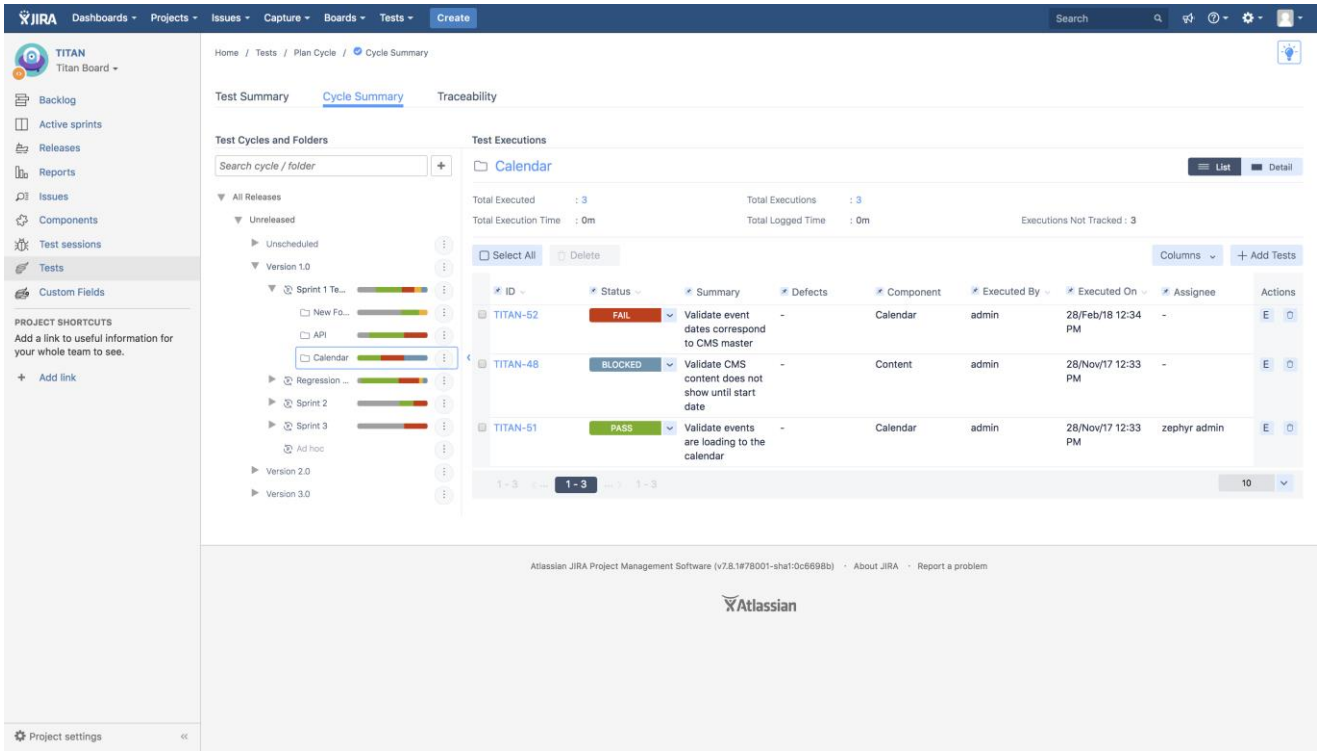

Рис. 1.4. Інтрефейс *Jira Zephyr*. Вікно результатів прогонів тестів

1.2.3. *TestRail*

Основні можливості цього продукту: повна документація тестових кроків зі знімками екрану та доповненими текстовими описами, організація тест-кейсів з можливістю редагування, генерація звітів по найбільш використаним показникам. Дана програма підтримує різні типи тестування. Її можна використовувати для задач ручного та автоматизованого тестування, організації дослідницького тестування, а також для потенційної інтеграції з вибраними інструментами автоматизації перевірок. *TestRail* включає в себе відкритий *API*, а значить, користувачі можуть самостійно створювати відкриті інтеграції. Така гнучкість робить інструмент *TestRail* максимально популярним у своїй ніші.

Організація та координація кількох паралельних тестів, будь то для різних проєктів чи випусків, часто є складною та трудомісткою. Щоб не втратити

відстеження виконаних зусиль з тестування, *TestRail* допомагає керувати важливими даними та структурами програмного забезпечення, такими як етапи проекту, і полегшує інтеграцію з інструментами відстеження помилок.

Користувальницький інтерфейс *TestRail* наведено на рис. 1.5 та 1.6.

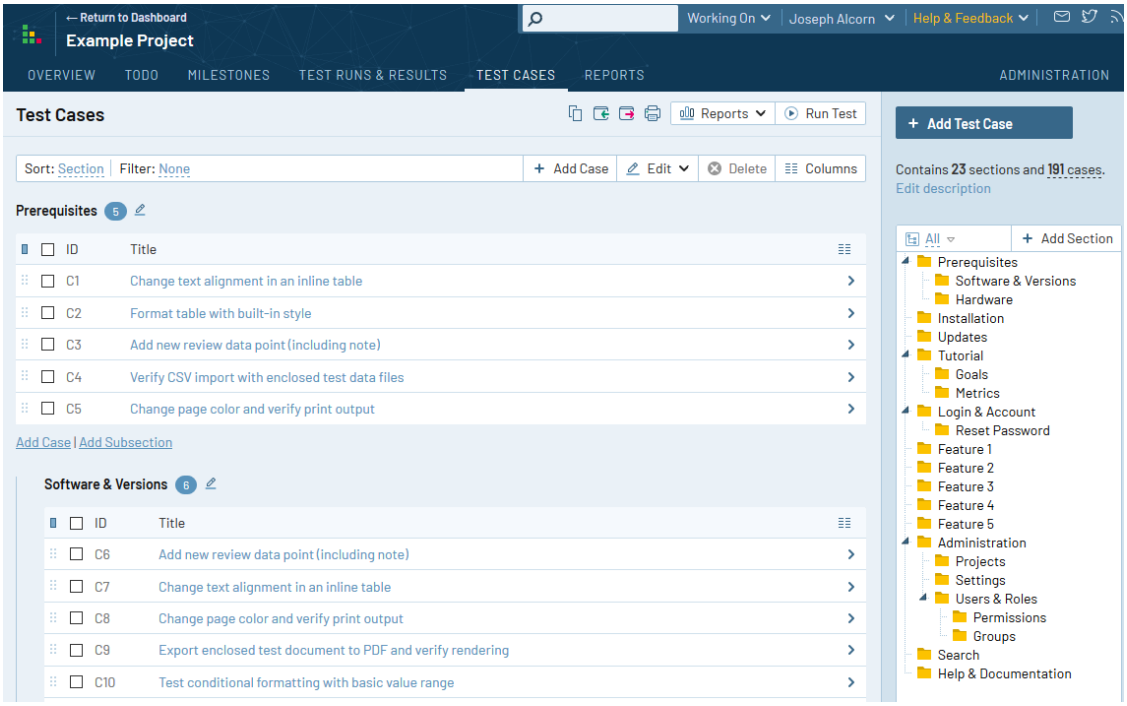

Рис. 1.5. Інтрефейс *TestRail*. Вікно тест-дизайну

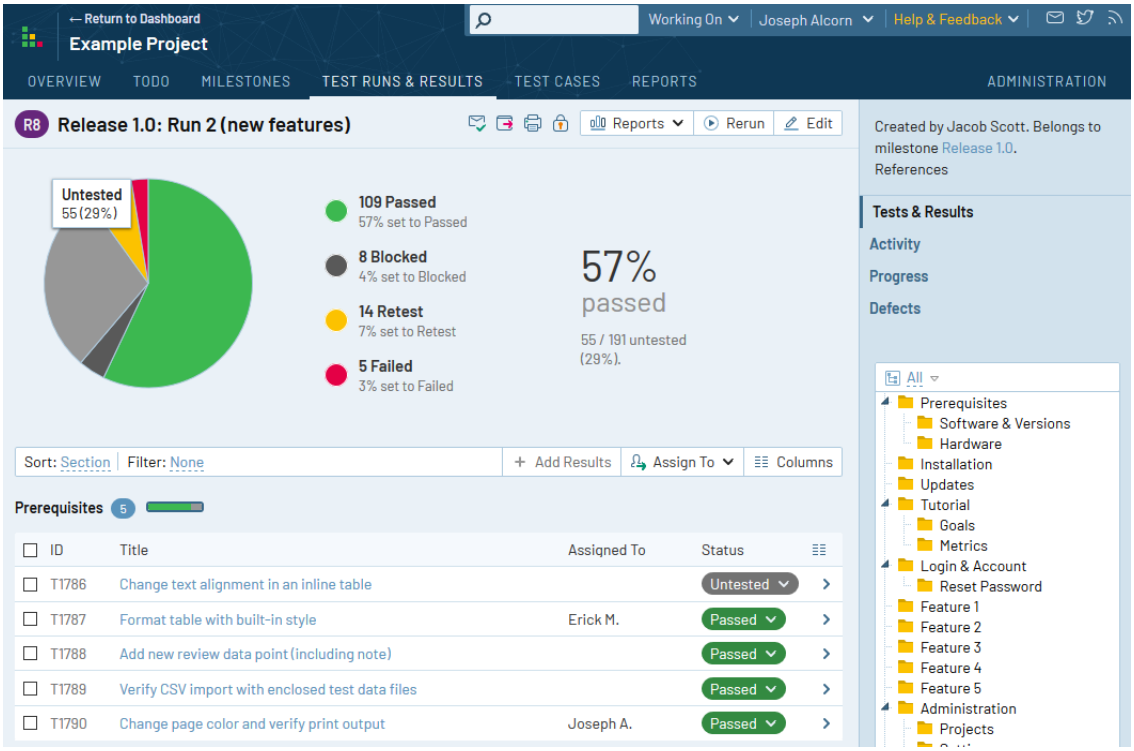

Рис. 1.6. Інтрефейс *TestRail*. Вікно результатів тестів

1.2.4. Недоліки існуючих рішень

Системи, які були проаналізовані, мають велику кількість переваг та є одними з кращих в своїй ніші, але також були виявлені певні недоліки. Серед них можна виділити такі:

– неможливість відстеження стану роботи над тест-кейсом;

– відсутність інструментів створення та редагування документів.

Для того, щоб розуміти в якій стадії роботи знаходиться тест-кейс потрібно мати можливість виставляти чіткий статус та візуалізувати їх на дошках, але жодна з проаналізованих систем не має такої функції.

Стосовно вбудованого інструменту для створення та редагування документів, всі перелічені системи лише мають змогу прив’язувати певні зовнішні посилання на інші документи або вимоги, але набагато зручніше було б тримати всі перераховані документи разом в одному місці.

1.3. Постановка задачі

Програмний засіб створення та менеджменту тестової документації повинен бути розроблений у вигляді *WEB*-системи з клієнт-серверною архітектурою (рис. 1.7) для полегшення та оптимізації роботи тестувальників, а отже покращення якості продукту, що тестується.

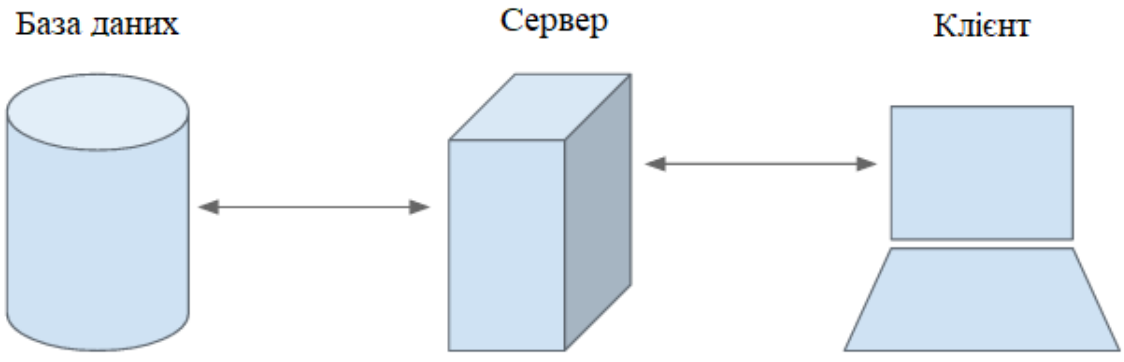

Рис. 1.7. Клієнт-серверна архітектура

Повинні бути розробленні наступні функціональні можливості програмного застосунку:

– реєстрація та авторизація користувачів з наданням або обмеженням доступу до певних тест-репозиторіїв;
– створення тест-репозиторію та надання прав користувачам на перегляд;
– створення та редагування тест-кейсів;
– створення та редагування документів;
– відстеження тест-репозиторіїв та пов’язаних документів і тест-кейсів;
– прикріплення файлів різних форматів до тест-кейсів;
– можливість додавання коментарів до тест-кейсу;
– можливість зміни статусу тест-кейсу;
– фільтрація тест-кейсів.

При розробці програмного засобу необхідно приділити особливу увагу до його інтерфейсу, щоби він відповідав наступним критеріям: інтуїтивність при використанні та простота дизайну.

Сервер продукту повинен відповідати таким критеріям:

– відкритість;
– масштабованість;
– кросплатформенність;
– захищеність та безпечність системи.

Також встановлені певні вимоги до бази даних для оптимізації роботи з системними та користувальницькими ресурсами:

– цілісність, тобто повнота та несуперечність даних;
– багаторазове використання даних з одного і того самого ресурсу;
– швидкодія запитів;
– простота оновлення даних;
– мінімізація надмірності даних;
– захищеність даних.

1.4. Висновки до розділу

В цьому розділі був проведений аналіз предметної області, тобто огляд теорії тестування програмного забезпечення та основі тестові артефакти, задля створення і менеджменту яких і повинен бути реалізований застосунок, а також аналіз існуючих рішень, перелік їх переваг та недоліків.

В процесі порівняльного аналізу були розглянуті найбільш популярні на сьогодні системи створення тестової документації: *QTest*, *Jira Zephyr* і *TestRail*. На основі аналізу було виділено основні спільні риси цих додатків, які також повинні бути застосовані до розроблюваного застосунку:

- *WEB*-додатки з клієнт-серверною архітектурою;
- простий та зрозумілий інтерфейс;
- групування тест-кейсів

Але також були виділені певні недоліки, а саме відсутність інструменту для створення документів і неможливість відстеження конкретного стану тест-кейсу.

Були враховані всі переваги та недоліки проаналізованих застосунків та сформульовано функціональні і технічні вимоги до кожної складової системи, а саме до клієнтського додатку, серверу та бази даних.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАСОБУ

2.1. Вибір технологій для розробки

Програмний засіб створення та менеджменту тестової документації побудований у вигляді додатку з клієнт-серверною архітектурою, так як ця архітектура є набільш гнучкою та дозволяє розробляти декулка клієнських додатків для одного серверу, що може стати досить корисною властивістю.

Програмний засіб складається з наступних частин: сервер, клієнтський додаток у вигляді *WEB*-сайту та база даних.

Для кожного з компонентів було обрано наступні технології та інструментарій:

– сервер – *C#*, фреймворк *ASP.NET Core MVC* на платформі *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 стоврювати 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.

C# 9.0 – це остання версія мови, яка використовується разом із *.NET* 5 і *.NET Core*. Існує екосистема для *C#*, яка робить її незамінним рішенням для загального програмування. Платформа *.NET*: *C#* невіддільний від цієї платформи. Він компілює та запускає програми через загальномовне середовище виконання *CLR*. Остання версія платформи *.NET – .NET 5*. *Visual Studio* – це рідне середовище розробки для *C#*, яке дозволяє розробникам завантажувати та встановлювати мову, писати код, налагоджувати та запускати або компілювати його. *.NET SDK* – це набір інструментів і бібліотек, що створено для розробки додатків *C#* або написання нових бібліотек.

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.

ASP.NET Core – це безкоштовна міжплатформна платформа з відкритим вихідним кодом для створення хмарних додатків, наприклад веб-додатків, додатків Інтернету речей і мобільних серверних програм. Він розроблений для роботи як у хмарі, так і локально. Так само, як і *.NET Core*, він розроблений модульно з мінімальними накладними витратами, а потім інші розширені функції можна додати як пакети *NuGet* відповідно до вимог програми. Це забезпечує високу продуктивність, потребує менше пам'яті, менший розмір розгортання та простоту обслуговування.

ASP.NET Core – це фреймворк із відкритим вихідним кодом, який підтримується корпорацією *Microsoft* і спільнотою, тож є можливість також додати або завантажити вихідний код зі сховища *ASP.NET Core* на *Github*.

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.

ASP.NET Core MVC – це легкий фреймворк із відкритим вихідним кодом, який добре інтегрується з *ASP.NET Core*.

ASP.NET Core MVC надає спосіб створення динамічних *WEB*-сайтів на основі шаблонів, який забезпечує чітке розділення завдань. Це надає повний контроль над розміткою, підтримує тестову розробку та дотримується останніх *WEB*-стандартів.

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 текстова мова програмування, яка використовується як на стороні клієнта, так і на стороні сервера, що дозволяє зробити *WEB* -сторінки інтерактивними. Якщо *HTML* і *CSS* є мовами, які надають структуру та стиль *WEB* -сторінкам, *JavaScript* надає *WEB* -сторінкам інтерактивні елементи, які залучають користувача. Включення JavaScript покращує взаємодію з *WEB*-сторінкою, перетворюючи її зі статичної сторінки на інтерактивну.

Серед можливостей цієї мови слід виділити додавання інтерактивної поведінки до *WEB*-сторінок. *JavaScript* дозволяє користувачам взаємодіяти з *WEB*-сторінками. Майже немає обмежень на те, що ви можете робити за допомогою *JavaScript* на *WEB*-сторінці – це лише кілька прикладів:

– показати або приховати більше інформації одним натисканням;
– змінити колір кнопки, коли на неї наводиться курсор миші;
– гортати карусель зображень на головній сторінці;
– збільшення або зменшення масштабу зображення;
– відображення таймера або зворотного відліку на *WEB*-сайті;
– відтворення аудіо та відео на веб-сторінці;
– відображення анімацій.

Також розробники можуть використовувати різні фреймворки *JavaScript* для розробки та створення *WEB*-додатків. Фреймворки *JavaScript* – це колекції бібліотек коду *JavaScript*, які надають розробникам попередньо написаний код для використання рутинних функцій і завдань програмування – буквально фреймворк для створення *WEB*-сайтів або *WEB*-додатків.

До популярних фреймворків *JavaScript* належать *React*, *React Native*, *Angular* і *Vue*. Багато компаній використовують *Node.js*, середовище виконання *JavaScript*, створене на механізмі *Google Chrome JavaScript V8*.

Крім *WEB*-сайтів і програм, розробники також можуть використовувати *JavaScript* для створення простих веб-серверів і розробки внутрішньої інфраструктури за допомогою *Node.js*.

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

Використовуючи *HTML*, текстовий файл додатково розмічається додатковим текстом, що описує, як має відображатися документ. Щоб зберегти розмітку окремо від фактичного вмісту *HTML*-файлу, використовується спеціальний синтаксис *HTML*. Ці спеціальні компоненти відомі як теги *HTML*. Теги можуть містити пари ім’я-значення, відомі як атрибути, а фрагмент вмісту, укладений у тег, називається елементом *HTML*.

Елементи *HTML* завжди мають відкриваючі теги, вміст посередині та закриваючі теги. Атрибути можуть надавати додаткову інформацію про елемент і включаються у відкриваючий тег. Елементи можна описати одним із двох способів. Перший з них – це елементи рівня блоку починаються з нового рядка документа та займають власне місце. Приклади цих елементів включають заголовки та теги абзаців.

Другий спосіб – це коли будовані елементи не починаються з нового рядка документа і займають лише необхідний простір. Ці елементи зазвичай форматують вміст елементів рівня блоку. Приклади вбудованих елементів включають гіперпосилання та теги текстового формату.

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.

CSS можна використовувати трьома способами – вбудованим, внутрішнім і зовнішнім. Ось різниця між цими трьома.

Зовнішні таблиці стилів дуже корисні, особливо коли ви хочете зберегти один стиль на всьому сайті. Правила стилю потрібно створити лише один раз, і вони застосовуються до всіх необхідних сторінок сайту. Якщо вам потрібно трохи змінити стиль на всіх сторінках, які використовують цей зовнішній аркуш, то його потрібно змінити один раз, і він автоматично застосовуватиметься до всіх сторінок. Загалом, це стандартний спосіб використання *CSS*.

Внутрішнє використання стосується лише сторінки, над якою ви працюєте. Це можна використовувати, якщо для однієї сторінки вашої веб-програми потрібен унікальний стиль.

Вбудований *CSS* стосується конкретного елемента сторінки.

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 бази даних

Проєктування бази даних для застосунку створення та менеджменту тестової документації було розбито на декілька кроків. Перш за все було визначено набір даних, який потрібен для такого застосунку. Насамперед, це користувачі та інформація про них, відомості про проєкт, що тестується, та сама

тестова документація, тому проєктування бази даних додатку було розпочато саме з цих найважливіших об’єктів, тобто приведення їх до вигляду таблиць.

Щоб визначити стовпці в таблиці, було вирішено, яку інформацію про предмет, записаний у таблиці, потрібно відстежувати. Наприклад, для таблиці "Користувач" – це ім'я, електронна адреса, пароль, роль і список доступних проєктів складають хороший початковий список стовпців. Кожен запис у таблиці містить однаковий набір стовпців, тому стає можливим зберігати всю перелічену інформацію про користувача в одному місці. Наприклад, колонка ролей містить перелік доступних ролей для користувача. Кожен запис містить дані про одного користувача, а поле ролей містить роль цього клієнта.

Зібравши цю інформацію, було перейдено до наступного кроку, а саме до визначення первинних ключів. Кожна таблиця повинна містити стовпець або набір стовпців, які однозначно ідентифікують кожен рядок, що зберігається в таблиці. Часто це унікальний ідентифікаційний номер, наприклад унікальна електронна адреса користувача. У термінології баз даних ця інформація називається первинним ключем таблиці.

Первинний ключ завжди повинен мати значення. Якщо в якийсь момент значення стовпця може стати непризначеним або невідомим (відсутнє значення), його не можна використовувати як компонент первинного ключа.

Завжди слід вибирати первинний ключ, значення якого не змінюватиметься. У базі даних, яка використовує більше однієї таблиці, первинний ключ таблиці можна використовувати як посилання в інших таблицях. Якщо первинний ключ змінюється, зміна також має бути застосована всюди, де є посилання на ключ. Використання первинного ключа, який не змінюється, зменшує ймовірність того, що первинний ключ може не синхронізуватися з іншими таблицями, які на нього посилаються.

Часто в якості первинного ключа використовується довільне унікальне число.

Тепер, коли вся інформація була розділена на таблиці, було визначено відношення між таблицями.

Після визначення відношень між таблицями, всі дані були нормалізовані. Нормалізація – це процес мінімізації надмірності зв’язку чи набору зв’язків. Надмірність у відношенні може спричинити аномалії вставки, видалення та оновлення. Отже, це допомагає мінімізувати надмірність у відношеннях. Нормальні форми використовуються для усунення або зменшення надмірності в таблицях бази даних.

Перша нормальна форма відповідає за нормалізацію, якщо відношення містить складений або багатозначний атрибут, воно порушує першу нормальну форму або відношення перебуває в першій нормальній формі, якщо воно не містить жодного складеного або багатозначного атрибута. Відношення знаходиться в першій нормальній формі, якщо кожен атрибут у цьому відношенні є однозначним атрибутом.

Щоб перебувати у другій нормальній формі, відношення має бути у першій нормальній формі, і відношення не повинно містити жодної часткової залежності. Відношення знаходиться в *2NF*, якщо воно не має часткової залежності, тобто жоден непростий атрибут (атрибути, які не є частиною жодного ключа-кандидата) не залежить від будь-якої належної підмножини будь-якого ключа-кандидата в таблиці.

Часткова залежність – якщо належна підмножина ключа-кандидата визначає непростий атрибут, це називається частковою залежністю.

Відношення знаходиться в третій нормальній формі, якщо немає транзитивної залежності для непростих атрибутів, а також у другій нормальній формі.

Після виконання всіх вище описаних кроків була спроєктована база даних для застосунку. *UML*-діаграму бази даних наведено на рис. 2.1.

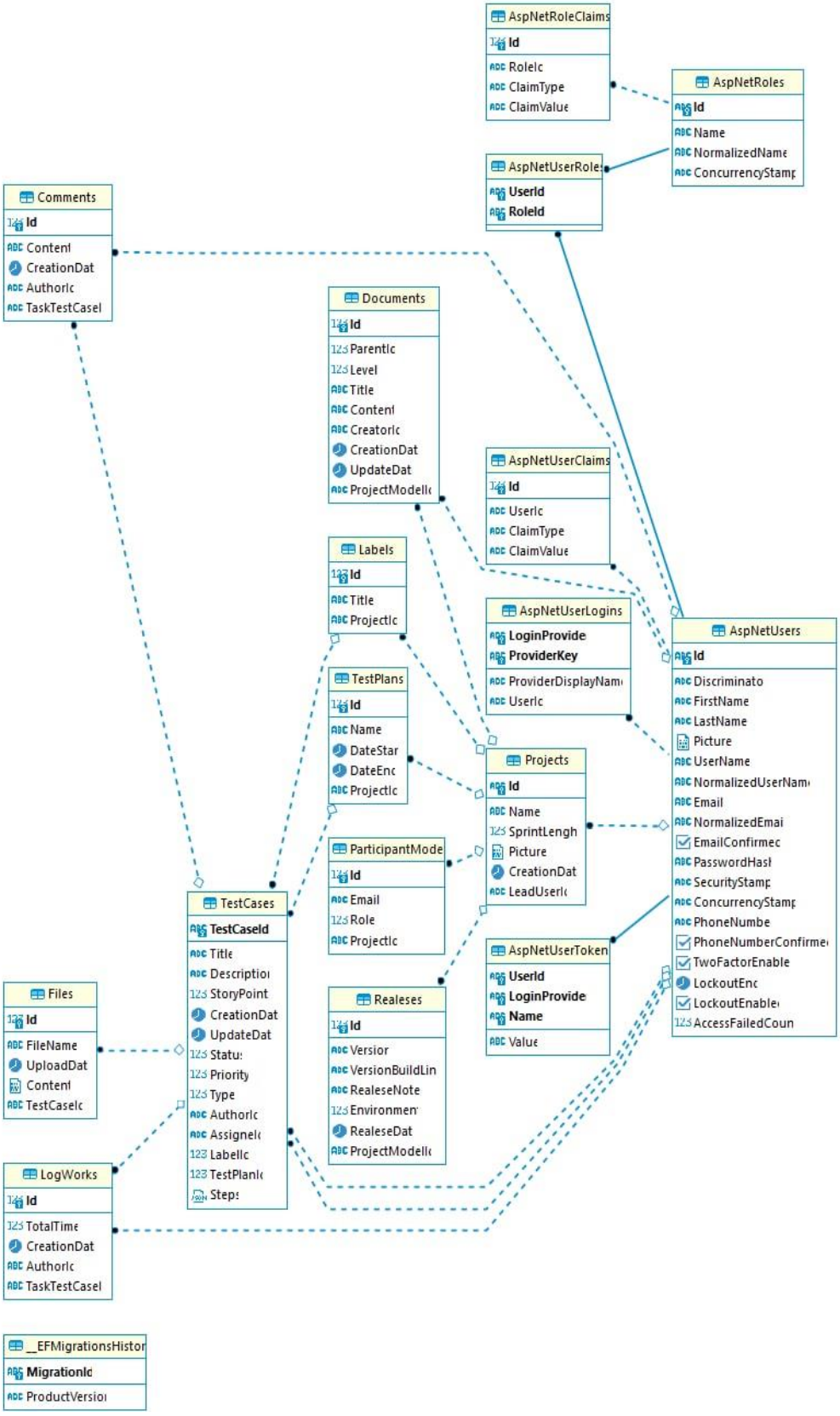

Рис. 2.1. *UML*-діграма бази даних для програмного застосунку

Вся інформація в базі даних розділена по таблицям. Кожна таблиця являє собою окрему сутність, де її стовпці – це властивості, а рядки – окремі об’єкти. Кожен об’єкт, тобто реальна особа класу (наприклад, кожен користувач, який існує в системі), представлений рядком інформації в таблиці бази даних. Рядок визначається в реляційній моделі як кортеж, побудований над заданою схемою. З математичної точки зору кортеж – це функція, яка присвоює постійне значення кожному атрибуту схеми з відповідного домену атрибутів. Зауважте, що оскільки схема є набором атрибутів, ми можемо показати їх у будь-якому порядку, не змінюючи значення даних у рядку (кортежі).

Опис усіх таблиць з бази даних наведено в таблиці 2.1.

Таблиця 2.1

Опис таблиць бази даних

Назва таблиці	Опис таблиці
AspNetUsers	Список користувачів системи
Projects	Список всіх доступних проєктів для корстувача
TestPlans	Список тест-планів для кожного проєкту
Filters	Список фільтрів для кожного проєкту
TestCases	Список тест-кейсів для кожного спринту
Comments	Список коментарів до кожного тест-кейсу
Files	Список файлів, що можуть бути прикріплені до тест-кейсу
Documents	Список документів для проєкту
AspNetRoles	Список доступних ролей на проєкті
AspNetUserRoles	Сисок ролей, якими володіє користувач
Releases	Список усіх випусків продукту

2.3. Архітектура застосунку

Програмний застосунок для створення та менеджменту тестової документації є *WEB*-додатком з клієнт-серверною архітектурою. Архітектура *WEB*-додатків описує зв'язки між базами даних, серверами та додатками в системі. Вона визначає, як функціональні можливості та логіка системи розподіляються між стороною сервера та стороною клієнта. По суті, архітектура відповідає за поєднання всіх елементів програми: те, що бачать ваші користувачі та з чим взаємодіють, і те, як програмне забезпечення виконує операції на внутрішньому рівні. Іншими словами, в клієнт-серверній архітектурі існує три рівні: презентація, логіка та дані.

Рівень презентації – це найбільш високорівневий прошарок додатку, який ще по-іншому називається інтерфейсом користувача. Головна задача інтерфейсу – це транслювати задачі та результати виконання програми у те, що зможе зрозуміти користувач.

Рівень логіки – це рівень, що координує застосунок та команди для нього, виконуючи логічні операції, калькуляції та порівняння. Також він відповідає за передачу даних між двома сусідніми рівнями.

Рівень даних – це рівень, на якому зберігається вся необхідна інформація у базі даних або файлової системи. У випадку з нашим застосунком, це лише база даних. Після запиту будь-яких даних, вони передаються до рівню логіки для обробки, а потів до користувача на рівень презентації для відображення. Рівень логіки та рівень даних взаємодіють один з одним через *SQL*-запити.

На рис. 2.2. наведено приклад взаємодії всіх трьох рівнів в додатках з клієнт-серверною архітектурою та саме в застосунку створення та менеджменту тестової документації.

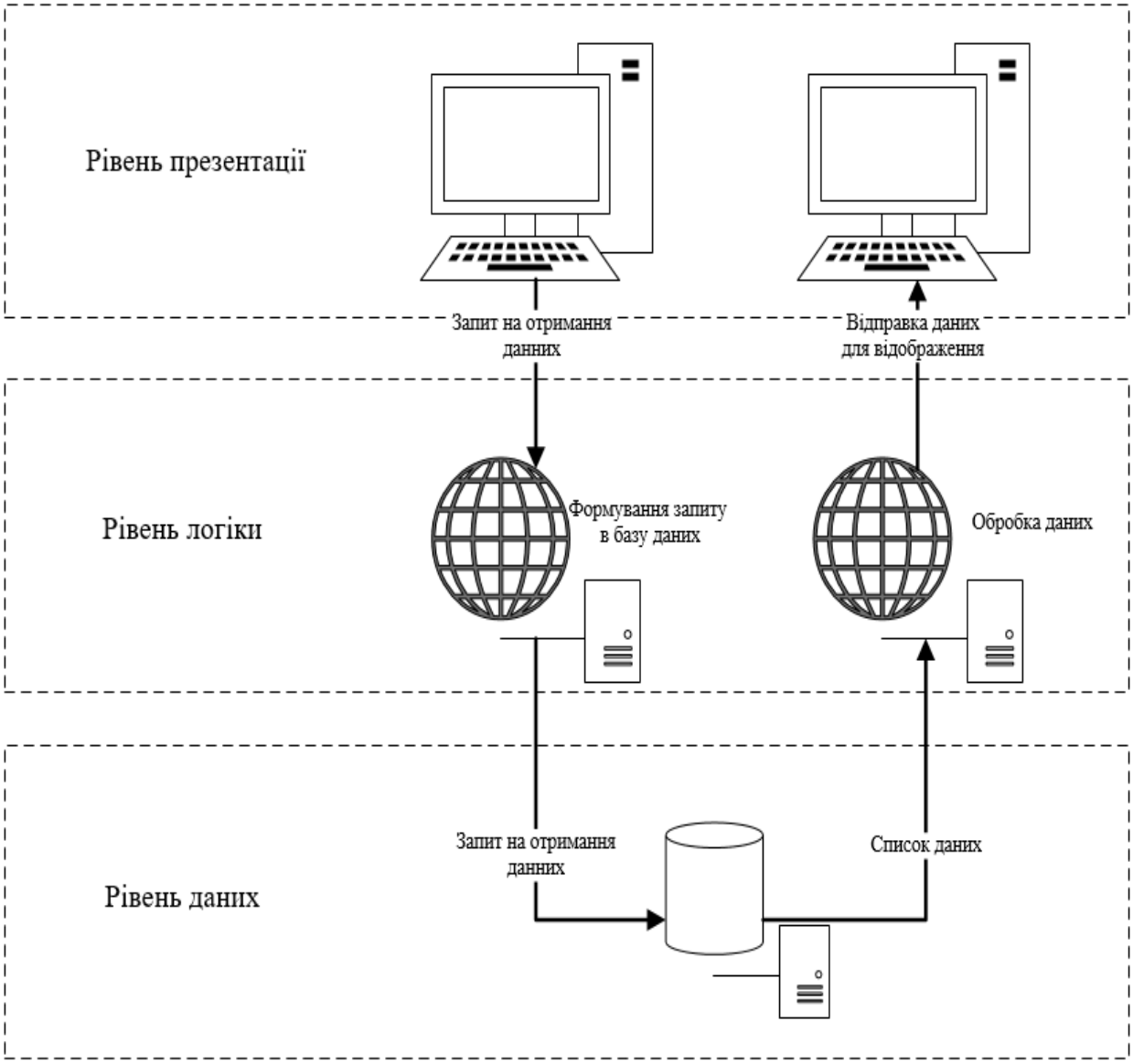

Рис. 2.2. Рівні клієнт-серверної архітектури та взаємодія між ними

2.4. Функціональні можливості

В ході проєктування та розробки застосунку, було визначено такі частини:

– адміністрування, що включає в себе створення проєкту-репозиторію тестів, створення користувачів та надання їм ролей (адміністратор, *QA*-менеджер, тестувальник) за допомогою сторінки адміністрування. Така сторінка буде доступна лише тим користувачам, які створили свій власний проєкт;

– менеджмент, тобто створення тест-планів та випусків продуктів;

– тестування, що включає створення тест-кейсів, їх оцінка та запуски;

– створення документації, тобто створення та редагування документів для певного проєкту.

Серед основних об’єктів застосунку можна виділити наступні:

– користувач;

– проєкт;

– тест-план;

– тест-кейс;

– випуск продукту;

– документ;

– файл.

Користувач – це основна одиниця застосунку, який може створювати будь-які інші об’єкти в ньому. Різні ролі можуть надавати користувачеві доступ до різного функціоналу додатку, а саме до створення різних типів об’єктів.

Найбільш широкими можливостями володіє адміністратор, тобто людина яка створила проєкт. Взагалі всі ролі користувачів краще розглядати з позиції одного проєкту, адже один користувач може мати різні ролі у кожному проєкті, але лише одну роль в поточному. Щодо адміністратора, то ним може бути будь-хто, хто створив проєкт. Також адміністратор володіє правами на створення будь-якого іншого об’єкту в системі.

QA-менеджер – це користувач, якого може призначити таким адміністратор та який має доступ на створення тест-плану, тест-кейсу, випуску продукту, документу та прикріплення файлів.

Тестувальник – це користувач, який може лише створювати тест-кейси, документи та прикріпляти файли.

Проєкт – це репозиторій, який повинен мати назву, для того щоб було можливо згенерувати його унікальний ідентифікатор. Він буде включати в себе тест-плани, випуски продукту та документацію. Проєктів в застосунку може бути безліч.

Тест-план – це набір тест-кейсів, який повинен мати власний ідентифікатор, що буде генеруватись автоматично. Тест-план, загалом, повинен відповідати випуску продукту, якому він повинен бути присвячений.

Випуск продукту – це ідентифікатор, який містить в собі номер випуску та його дату. Випуск продукту повинен містити в собі один тест-план з відповідним номером.

Тест-кейс – це одиниця системи, яка включає в себе кроки виконання, очікуваний результат та результат запуску, який може бути пройденим або не пройденим. Кожен тест-кейс це по-суті окрема задача, яка включає в себе таку інформацію як час створення, хто створим, на кого призначена, оцінка на виконання, назва, опис, коментарі та прикріплені файли. Кількість кроків співпадає з кількістю очікуваних результатів та результатів запусків, тобто кожен крок може мати власний запуск. Кількість таких кроків і їх результатів необмежена, тобто користувач може створювати стільки кроків до тест-кейсу, скільки йому потрібно. Також кожен тест-кейс повинен мати пріоритет, для того щоб тестувальники розуміли, написання та виконання яких тест-кейсів є найбільш важливими в даному тест-плані.

Документ – це об’єкт у вигляді документу, який може в себе включати будь-яку проєктну або тестову документацію. Підтримує різні стилі форматування. Список документів представлений у вигляді дерева з багатьма рівнями.

Файл – це елемент, який може бути прикріплений до будь-якого тест-кейсу. У застосунку підтримуються всі типи файлів.

Також важливо виділити наступні об’єкти системи, а саме коментарі до задач, які повинні також володіти базовими можливостями форматування, оцінка задач, яку може проставляти кожен користувач системи.

На основі головних об’єктів системи було побудовано діаграму послідовності запитів та обробки і отримання даних в програмному засобі створення та менеджменту тестової документації. (рис. 2.3). В ній наведено можливі варіанти використання ресурсів між трьома рівнями системи, а саме авторизація, створення проєктів-репозиторіїв, тест-планів, тест-кейсів та

документів через *HTTP*-запити. При проєктуванні системи використовувались *POST* та *GET* запити. При створенні будь-яких нових об’єктів, як проєкт, тест-план та ін., використовувався *POST*-запит. Для отримання розмітки або списку вже існуючих об’єктів використовується *GET*-запит.

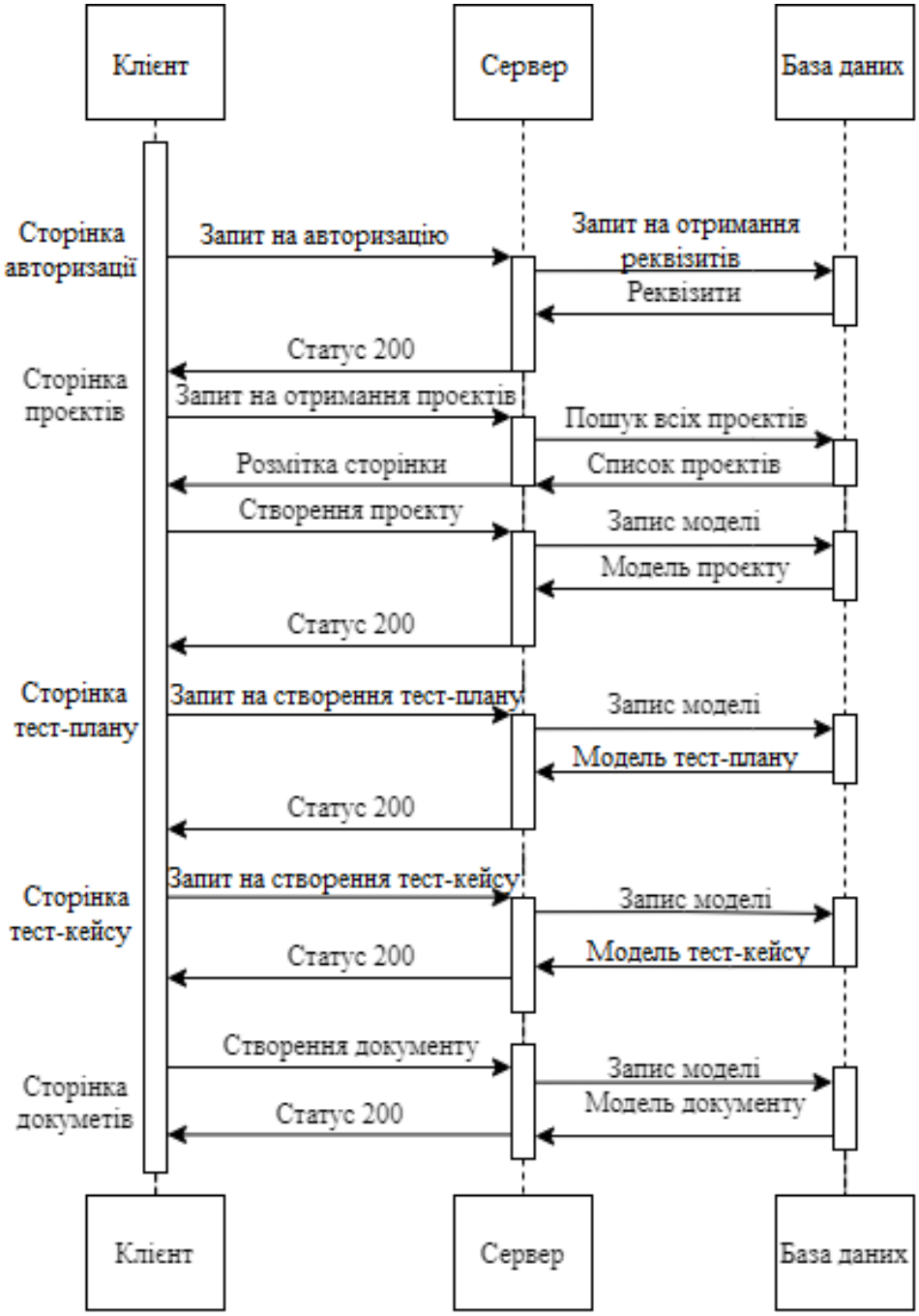

Рис. 2.3. Діаграма послідовності програмного засобу

2.5. Використання функцій програмного засобу

Так як розроблюваний програмний засіб має три рівні доступу до різних функцій системи, а саме адміністратор, *QA*-менеджер та тестувальник, тому для більш простого розуміння як різні користувачі можуть взаємодіяти з функціями системи було побудовано *Use-Case* діаграму (рис. 2.4).

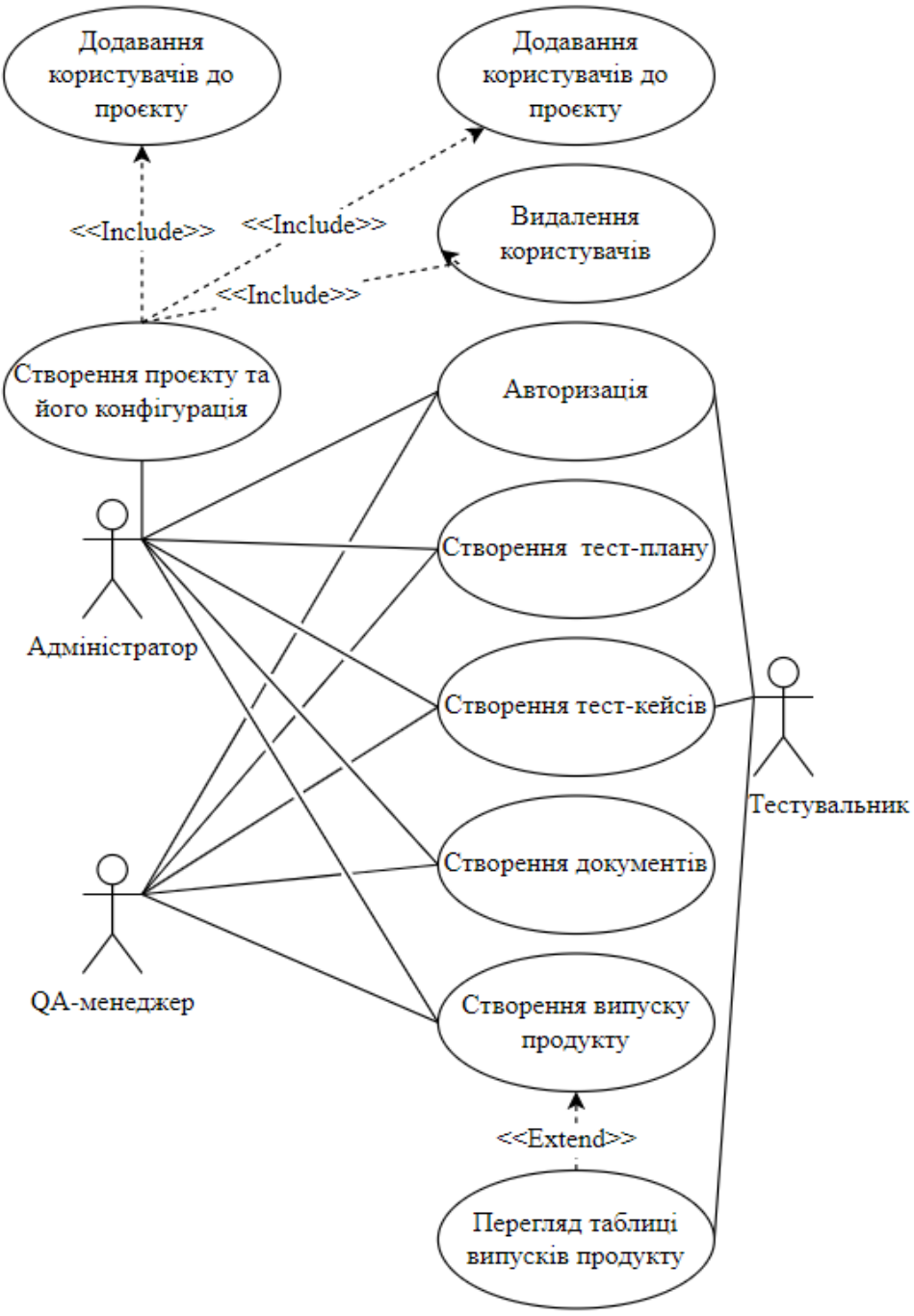

Рис. 2.4. *Use-Case* діаграма функцій програмного засобу

Стосовно авторизації в додаток необхідно виділити, що є два способи додавання користувача до проєкту. Перший, це коли користувач спершу реєструється в системі, тобто він ще не є частиною ніякого проєкту, а вже потім адміністратор існуючого проєкту може його автоматично приєднати. Другий спосіб – це спочатку додавання користувача до проєкту за допомогою електронної адреси. Тобто коли користувач лише зареєструється, він автоматично буде мати доступ мінімум до одного проєкту. В першому випадку користувач буде наділений правом створити власний проєкт одразу ж після реєстрації. Алгоритм реєстрації та створення проєкту наведено на рис. 2.5.

Рис. 2.5. Алгоритм авторизації та створення проєкту

При переадресуванні на сторінку існуючого проєкту, користувач зможе побачити список усіх тест-планів до проєкту. В свою чергу кожен тест-план складається з наборів різної кількості тест-кейсів. При переході на конкретний тест-план, користувачу буде доступний весь список тест-кейсів у вигляді дошки. Дошка складається з станів-колонок, у яких може перебувати тест-кейс, а саме:

– не почато;
– в розробці;
– на перевірці;
– в тестуванні;
– завершено.

Діаграма станів тест-кейсу зображена на рис. 2.6.

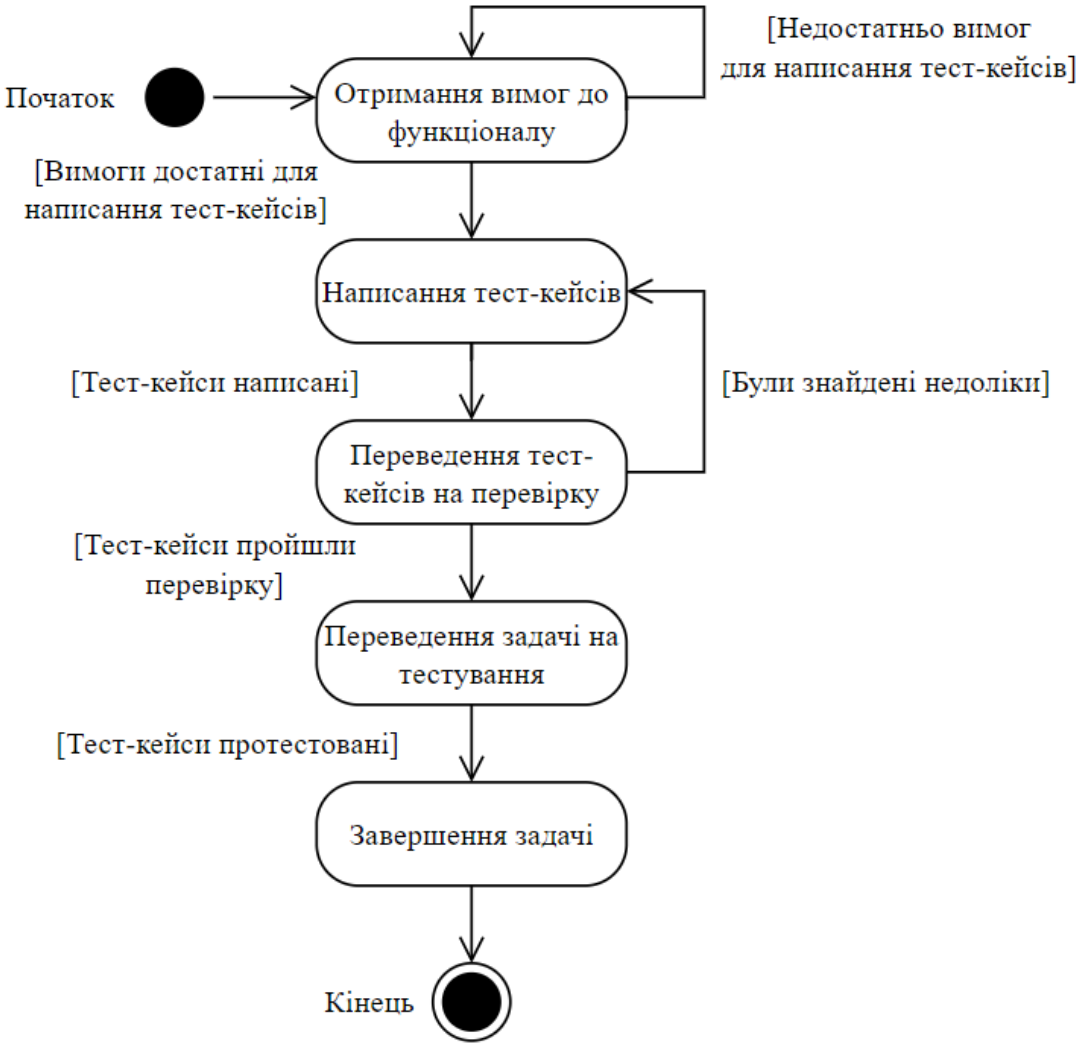

Рис. 2.6. Діаграма станів тест-кейсів

2.6. Висновки до розділу

У цьому розділі було обрано інструменти та фреймворки для розробки програмного засобу створення та менеджменту тестової документації. Було визначено наступні засоби для створення кожної складової програмного засобу:

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

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

– база даних – *PostgreSQL*.

Було спроєктовану базу даних, наведено *UML*-діаграму та описано таблиці системи.

Для проєктування функціональних можливостей системи було визначено головні модулі системи (адміністративний, менеджмент та тестування) та їх головні об’єкти (користувач, проєкт, тест-план, тест-кейс, випуск продукту, документ та файл), а також наведено опис для кожного з них. На основі цих даних були спроєктовано функціонал та наведено діаграми послідовності, використання ті станів тест-кейсів.

Також було спроєктовано та наведено алгоритм реєстрації нового користувача та створення проєкту.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ

3.1. Організація клієнт-серверної архітектури в програмному засобі

Клієнт-серверна архітектура відноситься до виду систем, які розміщують, надають та керують більшістю ресурсів і послуг, які запитує клієнт. У цій моделі всі запити та послуги доставляються через мережу, і її також називають моделлю мережевих обчислень або мережею клієнт-сервер.

Архітектура клієнт-сервер, яку також називають моделлю клієнт-сервер, – це мережева програма, яка розподіляє завдання та навантаження між клієнтами та серверами, які знаходяться в одній системі або об’єднані комп’ютерною мережею.

Клієнт-серверна архітектура зазвичай складається з декількох робочих станцій користувачів, ПК або інших пристроїв, підключених до центрального сервера через Інтернет або іншу мережу.

Клієнт надсилає запит на дані, а сервер приймає та влаштовує запит, надсилаючи пакети даних назад користувачеві, якому вони потрібні.

Коротко підсумовуючи схему роботи таких додатків, можна сказати, що спочатку клієнт надсилає свій запит через мережевий пристрій, потім мережевий сервер приймає та обробляє запит користувача. Нарешті, сервер доставляє відповідь клієнту.

Розглядаючи клієнт та сервер окремо, можна виділити, що клієнтський додаток не може існувати окремо без серверу, але не навпаки. Тобто один сервер може підтримувати безліч клієнтів, в даному випадку реалізовано лише один клієнт у вигляді *WEB*-сайту, але також можна розширити цей список, додавши мобільні застосунки та ін., якщо це буде доцільно. Варто звернути увагу наскільки така можливість є зручною, адже вона дозволяє мати доступ до одного й того самого додатку з різних платформ та девайсів, що значно підвищує працездатність та взагалі можливості таких додатків.

Розроблюваний *WEB*-сайт є тонким клієнтом. Тонкий клієнт – це комп'ютерна система, яка використовується для запуску програм, де більша частина фактичної обробки виконується на віддаленому сервері, підключеному через мережу. Ця концепція є оптимальною, адже заощаджує ресурси на стороні клієнта (браузер), цим самим не навантажуючи систему користувача.

Якщо клієнт є досить простим у своїй реалізації, то сервер є поділеним на багато прошарків.

Клієнт та різні частини серверу спілкуються між собою через *HTTP*-протокол.

HTTP (протокол передачі гіпертексту) – це набір правил для передачі файлів, таких як текст, зображення, звук, відео та інші мультимедійні файли, через Інтернет. Щойно користувач відкриває свій браузер, він використовує *HTTP*. *HTTP* – це прикладний протокол, який працює поверх набору протоколів *TCP/IP*, який є основою Інтернету.

За допомогою протоколу *HTTP* відбувається обмін ресурсами між клієнтськими пристроями та серверами через Інтернет. Клієнтські пристрої надсилають серверам запити на ресурси, необхідні для завантаження веб-сторінки; сервери надсилають відповідь клієнту для виконання запитів. Запити та відповіді спільно використовують піддокументи, такі як дані про зображення, текст, макети тексту тощо, які об'єднуються клієнтським браузером для відображення повного файлу сторінки.

На додаток до файлів сторінок, які він може обслуговувати, сервер містить *HTTP*-демон, програму, яка очікує *HTTP*-запитів і обробляє їх, коли вони надходять.

Браузер – це *HTTP*-клієнт, який надсилає запити на сервери. Коли користувач браузера вводить запити файлів, «відкриваючи» файл, вводячи *URL*-адресу або клацаючи гіпертекстове посилання, браузер створює *HTTP*-запит і надсилає його на адресу Інтернет-протоколу (*IP*-адресу), указану в *URL*-адресі. *HTTP*-демон на сервері призначення отримує запит і надсилає запитуваний файл або файли, пов'язані із запитом.

Кожна взаємодія між клієнтом і сервером називається повідомленням. *HTTP*-повідомлення – це запити або відповіді. Клієнтські пристрої надсилають *HTTP*-запити на сервери, які відповідають, надсилаючи *HTTP*-відповіді клієнтам.

HTTP-запити – це коли клієнтський пристрій, в нашому випадку браузер, запитує у сервера інформацію, необхідну для завантаження сайту. Запит надає серверу необхідну інформацію, щоб пристосувати свою відповідь до клієнтського пристрою.

Кожен запит *HTTP* містить закодовані дані з такою інформацією, як: *URL*, що вказує на ресурс в Інтернеті, метод *HTTP*, що вказує на конкретну дію, яку запит очікує отримати від сервера у своїй відповіді та заголовки запитів *HTTP*. Це включає такі дані, як тип браузера, який використовується, і які дані запит шукає від сервера. Він також може містити файли *cookie*, які показують інформацію, надіслану раніше із сервера, що обробляє запит.

В реалізації програмного засобу було використано три типи методів, а саме *GET, POST* та *DELETE*.

GET – це метод *HTTP* для запиту даних із сервера. Запити з використанням методу *HTTP GET* повинні лише отримувати дані, не можуть включати дані в тіло повідомлення *GET* і не повинні мати будь-якого іншого впливу на дані на сервері. В реалізації системи – це завантаження розмітки сторінок та данних для відображення, наприклад даних про проєкт, тест-план, випуски продукту та тест-кейси.

POST – це метод *HTTP*, призначений для надсилання даних на сервер із клієнта *HTTP*. Метод *HTTP POST* вимагає від сервера прийняти дані, укладені в тілі повідомлення *POST*. В реалізації додатку метод *HTTP POST* використовується під час надсилання даннх для авторизації, створення нових проєктів, тест-планів, тест-кейсів та завантаження файлів і зображень на сервер.

Метод *HTTP DELETE* використовується для видалення ресурсу з сервера. В реалізації системи – це видалення користувачів з проєкту, коментарів та файлів з тест-кейсу.

3.2. Маршрутизація запитів на сервері

Так як розроблюваний програмний засіб, а саме його сервер, реалізовано за допомогою фреймворку *ASP.NET Core MVC,* тому основним поняттям в цій системі є контролер, який в першу чергу відповідає за маршрутизацію запитів на сервері. Загалом, контролер – це спеціальний клас у додатку *ASP.NET Core MVC* з розширенням *.cs* (для мови *C#*).

Контролери в програмі *MVC* логічно групують схожі типи дій разом. Це об'єднання дій або групування подібних типів дій дозволяє визначити набори правил, наприклад кешування, маршрутизація та авторизація, які застосовуватимуться разом. У додатку *ASP.NET Core MVC* клас успадковується від базового класу контролера.

Коли клієнт (браузер) надсилає запит на сервер, цей запит спочатку проходить через конвеєр обробки запитів. Коли запит пройде конвеєр обробки запитів, він потрапляє на контролер. Усередині контролера є багато методів (так звані методи дії), які насправді обробляють вхідний *HTTP*-запит. Метод дії всередині контролера виконує бізнес-логіку та готує відповідь, яка надсилається назад клієнту, який спочатку зробив запит.

Повертаючись до маршрутизації адрес, контролер слугує для отримання доступу до будь-якої сторінки та використовує для цього *Startup.cs* файл, що створено, на сервері.

Розберемо приклад його використання, а саме визначення шляху до головної сторінки проєкту, використовуючи метод *UseEndpoints*:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
       if (env.IsDevelopment())
       {
          app.UseDeveloperExceptionPage();
       }
```

```
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

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

В наведеному коді, маршрутизація відбувається за допомогою патерну побудови *URL*, а саме замість *controller* підставляється ідентифікатор потрібного проєкту, а в *action* будь-яка інша сторінка, що доступна в даному проєкті, де *index* є головною сторінкою за замовчуванням.

Таким чином маршрутизація побудована у всіх інших контролерах, що є в системі, але також можливий варіант використання маршрутизації за допомого *Route*-атрибуту. Слід зазначити, що кожен клас проєкту має свій відповідний контролер та може бути реалізований своїм окремим способом.

На прикладі нижче приведено спосіб побудови маршруту до сторінки тест-кейсу:

```
[Route("{testPlanId}/{projectId}/{testCaseId}")]
public async TestCase<IActionResult> Index(string projectId, int testPlanId, string TestCaseId)
{
    if (string.IsNullOrEmpty(projectId) || testPlanId < 0 || string.IsNullOrEmpty(testCaseId))
    {
        return BadRequest();
    }
    var project = await this._projectProvider.GetProject(projectId);
    if (project == null)
    {
        return NotFound();
    }
    var testPlan = project.TestPlans.Find(x => x.testPlanId == testPlanId);
    if (testPlan == null)
    {
        return NotFound();
    }
    var testCase = sprint.TestCases.Find(x => x.testCaseId == testCaseId);
    if (testCase == null)
    {
        return NotFound();
    }
    ViewData["Project"] = project.Name;
    ViewData["ProjectImage"] = project.Picture;
    return View(testCase);
}
```

Також нижче наведено приклад маршруту до документів проєкту:

```
[Route("{projectId}/{documentId?}")]
    public async Task<IActionResult> Index(string projectId, int? documentId)
    {
        var project = await this._projectProvider.GetProject(projectId);
        if (project == null)
        {
            return NotFound();
        }
        ViewData["Project"] = project.Name;
        ViewData["ProjectImage"] = project.Picture;
        ViewData["ActiveDocument"] = documentId ?? -1;
        return View(project.Documents);
    }
```

У двох випадках *Route*-атрибут визначає *URL* до кожного тест-кейсу та документу. Спільним також є обробка помилок запитів, а саме якщо користувач самостійно змінить *URL* підставивши замість ідентифікатора проєкту ідентифікатор, який не існує, або до проєкту, у якого у користувача немає доступу, то він отримає відповідну помилку у браузері.

3.3. Основні частини програмного засобу

Програмний засіб створення та менеджменту тестової документації складається з таких наступних сторінок:

- сторінка авторизації;
- сторінка доступних проєктів;
- сторінка тест-планів;
- сторінка активного тест-плану
- сторінка тест-кейсу;

– сторінка випусків продукту;

– сторінка документу.

3.3.1. Реалізація сторінок

Уся розмітка до будь-якої сторінки знаходиться в частині представлення *MVC* патерну. Тобто це клас програмного забезпечення, який містить шаблон і форму даних і створює відповідь для браузера. Він отримує дані від контролера *MVC*, пакує їх і представляє браузеру для відображення.

Загалом представлення не дуже автономне, воно схоже на чорну скриньку, куди вкидають деякі дані, і воно відображає користувачеві у зручній для браузера формі. Він збирає вхідні дані з джерел даних, знаходить шаблон після виклику та об’єднує їх у вивід *HTML* під час виконання.

Контролер передає дані в представлення через словник, який називається словником *ViewData*. Цей словник містить запаковані результати, які в кінцевому підсумку перетворюються на вихідний *HTML*-код. Виходом із представлення у розроблюваному програмному засобі є *HTML*, а також *JSON*.

Представлення – це звичайний файл *aspx*, який містить відповідні елементи керування конкретної програми *ASP.NET*. Представлення у даній реалізації є строго типізованим, тобто єдиним для кожного окремого контролера.

На фрагменті коду нижче наведена реалізація представлення для сторінки активного тест-плану:

```
@model ProjectModel
<div class="main-header">
   <div class="page-container">
      <p class="page-name-style">Усі тест-кейси</p>
   </div>
</div>

<div class="project-testPlan-list">
```

```
@if (Model.TestPlans is null || !Model.TestPlans.Any())
{
    <div class="empty-testPlans">
        <div class="empty-testPlans-image"></div>
        <div class="empty-testPlans-message">
            <h5>Сплануйте роботу своєї команди</h5>
            <p>Це список роботи для вашої команди. Створіть тест-кейси та впорядкуйте їх за пріоритетом.</p>
        </div>
    </div>
}
else
{
    @foreach (var sprint in Model.TestPlans)
    {
        @await Html.PartialAsync("_TestPlan", testPlan);
    }
}
<div class="testPlansBacklog-toolbar">
    <button class="btn btn-light create-testPlan-btn">Створити</button>
</div>
</div>
```

Також в додатку А наведена реалізація представлення сторінки тест-кейсу.

3.3.2. Повна реалізація шаблону *MVC*

Для забезпечення роботи з різними об’єктами системи, спочатку необхідно створити їхні моделі, наприклад для тест-кейс:

```
public class TestCaseModel
```

```
    {
    [Required, Key, DatabaseGenerated(DatabaseGeneratedOption.None)]
public string TestCaseId { 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 TestCaseType 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 TestPlanModel TestPlan { get; set; }
               [NotMapped]
                 public int IdTestPlan { get; set; }
                [NotMapped]
                public string IdProject { get; set; }
    }
```

Також на прикладі нижче наведено модель для об’єкту проєкту:

```
    public class ProjectModel
    {
        [Required, Key, DatabaseGenerated(DatabaseGeneratedOption.None)]
       public string Id { get; set; }

        [Required]
```

```
        [Remote("IsProjectExist", "Project", ErrorMessage = "Project already
exists")]
        public string Name { get; set; }
        [Required]
        public int TestPlanLenght { get; set; }
        public byte[] Picture { get; set; }
        [NotMapped]
        public IFormFile PictureFile { get; set; }
        public DateTime CreationDate { get; set; }
        public ApplicationUser LeadUser { get; set; }
        public List<FilterModel> Filters { get; set; }
        public List<SprintModel> Sprints { get; set; }
        public List<RealeseModel> Realeses { get; set; }
        public List<DocumentModel> Documents { get; set; }
        public List<string> Participants { get; set; }
    }
```

Тобто як бачимо, моделі цих двох об’єктів строго повторюють їх представлення у базі даних, іншими словами саме завдяки саме моделям і формуються таблиці в базі даних та відповідні поля з форматом даних, наприклад *int* або будь-який інший об’єкт.

Далі наведено приклад реалізації контролеру тест-кейсу, який містить в собі опис маршруту до кожного з них та його методи, наприклад створення тест-кейсу *Create*:

```
        [HttpPost]
        [Route("Create")]
        public async Task<IActionResult> Create([FromForm] TestCaseModel model)
        {
            await this._testPlanProvider.CreateTestCase(model);
            return
Redirect($"/TestCase/{model.IdTestPlant}/{model.IdProject}/{model.TestCaseId}");
```

```
}
```

Цей метод *Create* є *POST* запитом, тіло якого передає до серверу модель тест-кейсу. Коли запит був доставлений до серверу, серверний метод *CreateTask* почне його обробку. Після такої обробки та запису цих даних до відповідних таблиць для нового тест-кейсу, користувач вже зможе виконати перехід по *URL* за наступним патерном для створеного тест-кейсу з відвідним ідентифікатором: */TestCase/{model.IdTestPlan}/{model.IdProject}/{model.TestCaseId}*.

На фрагменті коду нижче наведено одну з реалізацій *GET* методів, а саме завантаження файлу із системи:

```
[HttpGet("DownloadFile/{testPlanId}/{projectId}/{testCaseId}/{fileId}")]
    public async Task<IActionResult> DownloadFile(int testPlanId, string projectId, string testCaseId, int fileId)
    {
        var files = await this._testPlanProvider.GetTestCaseFiles(testPlanId, projectId, testCaseId);
        var file = files.FirstOrDefault(x => x.Id == fileId);
        if(file is null)
        {
            return NotFound();
        }
        var content = new MemoryStream(file.Content);
        var contentType = "application/octet-stream";
        var fileName = file.FileName;
        return File(content, contentType, fileName);
    }
```

По адресі *"DownloadFile/{testPlanId}/{projectId}/{testCaseId}/{fileId}"* визначається файл, який потрібно завантажити. В методі присутня перевірка на те існує такий файл чи ні, тобто якщо користувач змінить ідентифікатор файлу на неіснуючий в системі, то він отримає помилку. В методі також використовуються такі системні функції як *MemoryStream,* що забезпечують роботу з потоком даних.

На наступному фрагменті коду наведено реалізацію одного з *DELETE* методів у системі, а саме видалення проєкту, яке далі провокує видалення всіх пов'язаних тест-планів, документів та випусків продукту, тобто каскадне видалення:

```
[HttpDelete, Route("Project/{projectId}/Delete")]
    public async Task<IActionResult> Delete(string projectId)
    {
        await this._projectProvider.DeleteProject(projectId);
        return Ok();
    }
```

3.3.3. Логіка взаємодії з базою даних

При розробці програмного засобу створення та менеджменту тестової документації використовувався *Entity Framework*, який пропонує три підходи до створення моделі. Серед них *Code First, Database First, Model First*. В цій реалізації було обрано використовувати *Code First* підхід, що націлений на неіснуючу базу даних, а *Code First* створить її. Він дозволяє визначити модель за допомогою класів *C#*, а додаткова конфігурація може бути виконана за допомогою атрибутів у класах і властивостях або за допомогою сторонніх *API*.

На наступному фрагменті коду наведено опис усіх можливих відношень класів:

```
 modelBuilder.Entity<ProjectModel>()
    .HasOne(x => x.LeadUser)
    .WithMany(y => y.LinkedProjects)
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<TestPlanModel>()
    .HasOne(x => x.Project)
    .WithMany(y => y.TestPlans)
```

```
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<TestCaseModel>()
    .HasOne(x => x.TestPlan)
    .WithMany(y => y.TestCases)
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<CommentModel>()
    .HasOne(x => x.Task)
    .WithMany(y => y.Comments)
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<TestCaseFileModel>()
    .HasOne(x => x.TestCase)
    .WithMany(y => y.Files)
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<LogWorkModel>()
    .HasOne(x => x.Task)
    .WithMany(y => y.LogWorks)
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<CommentModel>()
    .HasOne(x => x.Author)
    .WithMany(y => y.Comments)
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<LogWorkModel>()
    .HasOne(x => x.Author)
    .WithMany(y => y.LogWorks)
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<TestCaseModel>()
    .HasOne(x => x.Assigne)
    .WithMany(y => y.Assignees)
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<RealeseModel>()
```

```
    .HasOne(x => x.ProjectModel)
    .WithMany(y => y.Realeses)
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<DocumentModel>()
    .HasOne(x => x.ProjectModel)
    .WithMany(y => y.Documents)
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<TestCaseModel>()
    .HasOne(x => x.Label)
    .WithMany(x => x.Task);
modelBuilder.Entity<LabelModel>()
    .HasOne(x => x.Project)
    .WithMany(x => x.Labels)
    .OnDelete(DeleteBehavior.Cascade);
modelBuilder.Entity<ParticipantModel>()
    .HasOne(x => x.Project)
    .WithMany(x => x.Participants)
    .OnDelete(DeleteBehavior.Cascade);
```

Після створення моделей даних та опису їх відношень, були реалізовані методи, які описують логіку роботи з базою даних, тобто обробка даних до вигляду, який потребує користувач.

На фрагменті коду нижче наведено два методи, що пов’язані з обробкою даних, а саме створення проєкту та назначення тестувальника на тест-кейс:

```
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<ParticipantModel>();
            }
            model.Participants.Add(new ParticipantModel
            {
                Email = model.LeadUser.Email,
                Project = model,
                Role = Role.Admin
            });
            await this._dbContext.Projects.AddAsync(model);
            await this._dbContext.SaveChangesAsync();
        }
    public async Task UpdateAssigne(int sprintId, string projectId, string taskId, string userId)
        {
            if (string.IsNullOrEmpty(projectId))
            {
                throw new ArgumentNullException(nameof(projectId));
            }
            if (string.IsNullOrEmpty(taskId))
```

```
{
    throw new ArgumentNullException(nameof(taskId));
}
if (sprintId < 0)
{
    throw new ArgumentOutOfRangeException(nameof(sprintId));
}
var project = await this._projectProvider.GetProject(projectId);
if (project is null)
{
    throw new ArgumentNullException(nameof(project));
}
var sprint = project.TestPlans.Find(x => x.Id == sprintId);
if (sprint is null)
{
    throw new ArgumentNullException(nameof(sprint));
}
var task = sprint.TestCases.Find(x => x.TestCaseId == taskId);
if (task is null)
{
    throw new ArgumentNullException(nameof(task));
}
var users = await this._projectProvider.GetProjectUsers(projectId);
var user = users.FirstOrDefault(x => x.Id == userId);
if(user is null)
{
    return;
}
task.Assigne = user;
task.UpdateDate = DateTime.Now;
```

```
            this._dbContext.TestCases.Update(task);
            await this._dbContext.SaveChangesAsync();
        }
```

Ці два методи мають спільну логіку роботи: для того, щоб створити проєкт та назначити тестувальника на задачу та записати ці дані в базу даних в базі, необхідно передати моделі даних з *POST*-запиту, тобто з тіла цього запиту, варто відміти, що певна системна інформація, наприклад час створення або оновлення даних, не передається напряму користувачем, а лише дістається с локального часу сервера.

3.4. Висновки до розділу

В даному розділі було описано процес розробки програмного засобу створення та менеджменту тестової документації.

Була описана клієнт-серверна архітектура та як вона організована у цій конкретній системі. Система складається з трьох ланок, тобто клієнтського додатку, серверу та бази даних, де перші два обмінюються повідомленнями між собою за допомогою *HTTP*-протоколу.

Також були описані підходи та принципи, що застосовувались під час розробки системи, а саме *MVC* та *Code First* патерни.

У цьому розділі також наведені різні фрагменти вихідного коду програми, що пояснюють логіку створення моделей об’єктів, опис відношень між різними об’єктами, приклади налаштування маршрутів до певних ресурсів ситеми, логіку запису та оновлення даних в базі, а також наведено приклади розмітки для одних з головних сторінок додатку.

ВИСНОВКИ

В ході виконнаня кваліфікаційної роботи було розроблено програмний засіб створення та менджменту тестової документації, що складається з бази даних, серверу та клінтського додатку.

У першому розділі був проведений аналіз предметної області, тобто огляд теорії тестування програмного забезпечення та основні тестові артефакти, задля створення і менеджменту яких і повинен бути реалізований застосунок, а також аналіз існуючих рішень, перелік їх переваг та недоліків.

В процесі порівняльного аналізу були розглянуті найбільш популярні на сьогодні системи створення тестової документації: *QTest, Jira Zephyr* і *TestRail*. На основі аналізу було виділено основні спільні риси цих додатків.

Були враховані всі переваги та недоліки проаналізованих застосунків та сформульовано функціональні і технічні вимоги до кожної складової системи, а саме до клієнтського додатку, серверу та бази даних.

У другому розділі було обрано інструменти та фреймворки для розробки програмного засобу створення та менеджменту тестової документації. Було визначено наступні засоби для створення кожної складової програмного засобу:

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

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

– база даних – *PostgreSQL*.

Було спроєктовану базу даних, наведено *UML*-діаграму та описано таблиці системи.

Для проєктування функціональних можливостей системи було визначено головні модулі системи (адміністративний, менеджмент та тестування) та їх головні об’єкти (користувач, проєкт, тест-план, тест-кейс, випуск продукту, документ та файл), а також наведено опис для кожного з них. На основі цих даних

були спроєктовано функціонал та наведено діаграми послідовності, використання ті станів тест-кейсів.

Також було спроєктовано та наведено алгоритм реєстрації нового користувача та створення проєкту.

У третьому розділі було описано процес розробки програмного засобу створення та менеджменту тестової документації.

Була описана клієнт-серверна архітектура та як вона організована у цій конкретній системі. Система складається з трьох ланок, тобто клієнтського додатку, серверу та бази даних, де перші два обмінюються повідомленнями між собою за допомогою *HTTP*-протоколу. Також були описані підходи та принципи, що застосовувались під час розробки системи. У цьому розділі наведені різні фрагменти вихідного коду програми та їх пояснення.

Матеріли кваліфікаційної роботи рекомендується використовувати під час проєктування та розробки *WEB*-додатків, а розроблений програний засіб під час тестування програнмого забезпечення для зробки тестової документаії та її підтримки.

СПИСОК БІЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проєкти) випускників Національного авіаційного університету. – К.: НАУ, 2017. – 63 с.
2. ДСТУ 3008–95 “Документація. Звіти у сфері науки і техніки. Структура і правила оформлення”.
3. *Richter J. CLR via C#. – Microsoft Press,* 2010. – 873 с.
4. *Flanagan D. JavaScript*: *the definitive guide*. *– O'Reilly*, *Beijing*, 2011. – 1086 с.
5. *Freeman A. Pro ASP.NET Core 3: develop cloud-ready web applications using MVC, Blazor, and Razor Pages*. *– Apress L.P*., *Berkeley*, *CA*, 2020. - 736 с.
6. *Troelsen A., Japikse P. Pro C# 10 with .NET 6: foundational principles and practices in programming*. *– Apress*, *New York*, 2022. – 1705 с.
7. *Freeman A. Pro jQuery 2.0. – Apress*, *New York*, 2013. – 1073 с.
8. *Spurlock J. Bootstrap. Responsive Web-Development. – O'Reilly,* 2013. – 128 с.
9. *Boehm B. Software Engineering Economic Prentice-Hall,Inc, N.J. 1981. – 767 c.*
10. *Beizer B. Software testing techniques. (Second edit.) International Thomson Computer Press, 1990. – 550 c.*
11. *Beizer B. Software Testing Techniques. ITP, 1990. – 550 c.*
12. Білас О.Є. Якість программного забезпечення та тестування / О.Є. Білас – Львів: Видавництво Львівської політехніки, 2011. – 216 с.
13. *Hyatt L.E. A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality, 1996. – 467 c.*
14. *ISO/IEC TR 9126-2:2003 Software engineering – Product quality – Part2: External metrics*
15. *ISO/IEC TR 9126-3:2003 Software engineering – Product quality – Part3: Interval metrics*

ДОДАТОК А

ВИХІДНИЙ КОД РОЗМІТКИ СТОРІНКИ ТЕСТ-КЕЙСУ

```
@inject IProjectProvider projectProvider
@model TestCaseModel
@{
    var projectUsers = await
projectProvider.GetProjectUsers(Model.TestPlan.Project.Id);
    var projectLabels = await
projectProvider.GetProjectLabels(Model.TestPlan.Project.Id);
}
<div class="task-header">
    <div class="header-title">
        <div>
            <span class="task-card-id padding-in-
card">@Model.TestCaseId</span>
            <span class="task-card-name padding-in-card">@Model.Title</span>
        </div>
        <div style="padding:15px">
            <select data-task="@Model.TestCaseId" data-
sprint="@Model.TestPlan.Id" data-project="@Model.TestPlan.Project.Id"
class="form-control" id="task-status" asp-for="Status" asp-
items="Html.GetEnumSelectList<TestBoard.Enums.TaskStatus>()">
            </select>
        </div>
        <div class="label-container">
            <span>Мітка:</span>
            <input class="form-control" id="task-label" data-
task="@Model.TestCaseId" data-sprint="@Model.TestPlan.Id" data-
```

```
project="@Model.TestPlan.Project.Id" type="text" list="labels"
value="@Model.Label?.Title"/>
                <datalist id="labels">
                    @foreach(var label in projectLabels)
                    {
                        <option>@label.Title</option>
                    }
                </datalist>
            </div>
        </div>
    </div>
    <div class="task-page-borders">
        <span class="details-header">Деталі</span>
        <div class="task-page-flex">
            <div class="task-page-details" style="width:70%">
                <div data-task="@Model.TestCaseId" class="type">
                    <span>Тип:</span>
                    <div id="@Model.TestCaseId-type" class="enum-descrition">
                        @await Html.PartialAsync("_Picture", new PictureModel {
DefaultImage = ImageHelper.GetTaskTypeImage(Model.Type) })
                        <p>- @EnumHelper.GetEnumDisplayName(Model.Type)</p>
                    </div>
                    <div id="@Model.TestCaseId-dropdown-content" class="dropdown-
content">
                        <div data-type="@TestBoard.Enums.TaskType.Task" data-type-
string="@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskType.Task)"
data-task="@Model.TestCaseId" data-sprint="@Model.TestPlan.Id" data-
project="@Model.TestPlan.Project.Id" class="type">
                            <img src="~/img/task.svg" />
```

```
                    <p> -
@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskType.Task)</p>
                </div>
                <div data-type="@TestBoard.Enums.TaskType.Story" data-type-
string="@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskType.Story)"
data-task="@Model.TestCaseId" data-sprint="@Model.TestPlan.Id" data-
project="@Model.TestPlan.Project.Id" class="type">
                    <img src="~/img/story.svg" />
                    <p> -
@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskType.Story)</p>
                </div>
                <div data-type="@TestBoard.Enums.TaskType.Bug" data-type-
string="@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskType.Bug)"
data-task="@Model.TestCaseId" data-sprint="@Model.TestPlan.Id" data-
project="@Model.TestPlan.Project.Id" class="type">
                    <img src="~/img/bug.svg" />
                    <p> -
@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskType.Bug)</p>
                </div>
            </div>
        </div>
        <div data-task="@Model.TestCaseId" class="priority">
            <span>Пріоритет: </span>
            <div id="@Model.TestCaseId-priority" class="enum-descrition">
                @await Html.PartialAsync("_Picture", new PictureModel {
DefaultImage = ImageHelper.GetPriorityImage(Model.Priority) })
                <p>- @EnumHelper.GetEnumDisplayName(Model.Priority)</p>
            </div>
            <div id="@Model.TestCaseId-dropdown-content" class="dropdown-
content">
```

```
                    <div data-type="@TestBoard.Enums.TaskPriority.High" data-type-
string="@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskPriority.High)"
data-task="@Model.TestCaseId" data-sprint="@Model.TestPlan.Id" data-
project="@Model.TestPlan.Project.Id" class="type">
                        <img src="~/img/high.svg" />
                        <p> -
@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskPriority.High)</p>
                    </div>
                    <div data-type="@TestBoard.Enums.TaskPriority.Medium" data-
type-
string="@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskPriority.Medium
)" data-task="@Model.TestCaseId" data-sprint="@Model.TestPlan.Id" data-
project="@Model.TestPlan.Project.Id" class="type">
                        <img src="~/img/medium.svg" />
                        <p> -
@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskPriority.Medium)</p>
                    </div>
                    <div data-type="@TestBoard.Enums.TaskPriority.Low" data-type-
string="@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskPriority.Low)"
data-task="@Model.TestCaseId" data-sprint="@Model.TestPlan.Id" data-
project="@Model.TestPlan.Project.Id" class="type">
                        <img src="~/img/low.svg" />
                        <p> -
@EnumHelper.GetEnumDisplayName(TestBoard.Enums.TaskPriority.Low)</p>
                    </div>
                </div>
            </div>
            <div class="story-points">
                <span>Загальна оцінка: </span>
                <span id="task-story-points">@Model.StoryPoints</span>
```

```
                </div>
                <div class="story-points">
                    <span contenteditable="true" id="dev-task-story-points" data-type=""
data-task="@Model.TestCaseId" data-sprint="@Model.TestPlan.Id" data-
project="@Model.TestPlan.Project.Id" data-prev-
value="@Model.StoryPoints">@Model.DEVStoryPoints</span>
                </div>
                <div class="story-points">
                </div>
            </div>
            <div class="assignee-list">
                <div class="assigne">
                    <span>Призначений: </span>
                    @if(Model.Assigne is not null)
                    {
                        <div data-current-assinge="@Model.Assigne.Email" data-
task="@Model.TestCaseId" class="person-block">
                            @await Html.PartialAsync("_Picture", new PictureModel {
Picture = Model.Assigne.Picture, DefaultImage = "/img/user.svg" })
                            <span class="name">@($"{Model.Assigne.FirstName}
{Model.Assigne.LastName}")</span>
                        </div>
                    }
                    else
                    {
                        <div class="person-block" data-task="@Model.TestCaseId">
                            <span class="name">Не призначено</span>
                        </div>
                    }
```

```
                <div id="@Model.TestCaseId-dropdown-content" class="dropdown-
content">
                    @foreach (var user in await
projectProvider.GetProjectUsers(Model.TestPlan.Project.Id))
                    {
                        <div data-user="@user.Id" data-task="@Model.TestCaseId"
data-sprint="@Model.TestPlan.Id" data-project="@Model.TestPlan.Project.Id"
class="type person-block">
                            @await Html.PartialAsync("_Picture", new PictureModel {
Picture = user.Picture, DefaultImage = "/img/user.svg" })
                            <span class="name">@($"{user.FirstName}
{user.LastName}")</span>
                        </div>
                    }
                </div>
            </div>
            <div class="author">
                <span>Створив: </span>
                <div class="person-block">
                    @await Html.PartialAsync("_Picture", new PictureModel { Picture
= Model.Author.Picture, DefaultImage = "/img/user.svg" })
                    <span>@($"{Model.Author.FirstName}
{Model.Author.LastName}")</span>
                </div>
            </div>
        </div>
    </div>
    <div class="task-page-flex">
        <div style="width:70%">
            <span class="details-header">Опис</span>
```

```
                <div>
                    <div class="requirements-editor">
                        <form id="update-description-form" method="post"
action="/Task/UpdateDescription/@Model.TestPlan.Id/@Model.TestPlan.Project.Id/@
Model.TestCaseId">
                            <textarea name="update-description-form" id="task-description"
class="rich-html-edit">@Model.Description</textarea>
                        </form>
                    </div>
                </div>
            </div>
            <div class="details-header-data">
                <span class="details-header">Дати</span>
                <div class="assignee-list">
                    <div class="creation-date">
                        <span>Створено: </span>
                        <span>@Model.CreationDate.ToString("dd.MM.yyyy
HH:mm")</span>
                    </div>
                    <div class="update-date">
                        <span>Змінено:  </span>
                        @if(Model.UpdateDate != DateTime.MinValue)
                        {
                            <span>@Model.UpdateDate.ToString("dd.MM.yyyy
HH:mm")</span>
                        }
                    </div>
                </div>
            </div>
        </div>
```

```
<div class="paddings-on-task-page">
    <span class="details-header">Вкладення</span>
    <div class="attachemnts-list">
        <form
action="/Task/UploadFile/@Model.TestPlan.Id/@Model.TestPlan.Project.Id/@Model.
TestCaseId" enctype="multipart/form-data" class="task-dropzone">
            <div class="fallback">
                <input name="file" type="file" />
            </div>
        </form>
    </div>
</div>
<div class="paddings-on-task-page">
    <span class="details-header">Коментарі</span>
    <div class="comments-list">
        @if (Model.Comments is null || !Model.Comments.Any())
        {
        <div class="empty-comments">
            <p>Додайте перший коментар</p>
        </div>
        }
        else
        {
            foreach (var comment in Model.Comments)
            {
                <div class="comment-container">
                    <div class="comments-author">
                        <div class="person-block">
                            @await Html.PartialAsync("_Picture", new PictureModel {
Picture = comment.Author.Picture, DefaultImage = "/img/user.svg" })
```

```
                        <span>@($"{comment.Author.FirstName}
{comment.Author.LastName}")</span>
                    </div>
                    <span>створив коментар -
@comment.CreationDate.ToString("dd.MM.yyyy HH:mm")</span>
                </div>
                <div class="comment-content">
                    @Html.Raw(comment.Content)
                </div>
            </div>
        }
    }
</div>
<form id="add-commnet-form" data-refresh="1" method="post"
action="/Task/AddComment/@Model.TestPlan.Id/@Model.TestPlan.Project.Id/@Mod
el.TestCaseId">
    <textarea name="add-commnet-form" id="task-description" class="rich-
html-edit"></textarea>
</form>
</div>
</div>
<script>
    $(() => {
        new Dropzone('.task-dropzone', {
            init: function () {
                thisDropzone = this;
                $(thisDropzone.element).addClass('dropzone');

$.get('/Task/GetFiles/@Model.TestPlan.Id/@Model.TestPlan.Project.Id/@Model.TestC
aseId', function (data) {
```

```
                    if (data == null) {
                        return;
                    }
                    $.each(data, function (key, value) {
                        var mockFile = { id: value.id, name: value.name, size: value.size
};
                        thisDropzone.emit("addedfile", mockFile);
                        thisDropzone.emit("complete", mockFile);
                    });
                });
                thisDropzone.on("complete", function (file) {
                    var newNode = document.createElement("input");
                    newNode.className = 'task-file-id';
                    newNode.type = "hidden";
                    newNode.value = file.id;
                    newNode.setAttribute('data-action',
'/Task/DownloadFile/@Model.TestPlan.Id/@Model.TestPlan.Project.Id/@Model.TestC
aseId/');
                    file.previewTemplate.appendChild(newNode);
                    $('.task-dropzone .dz-preview').click(e => {
                        let file = $(e.currentTarget);
                        let fileId = file.find('.task-file-id');
                        let url = fileId.data('action') + fileId.val();
                        window.open(url, '_blank');
                    });
                });
            }
        });
    })
</script>
```