Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

РОЗРОБКА МЕТОДУ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ВЕБСЕРВІСУ
(тема)

Виконав:
студент 4 курсу, групи ІТІНФ-19-2

Прокоп’єв С.А.
(прізвище, ініціали)

Спеціальності 122 Комп’ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник доц. Вечірська І.Д.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Прокоп'єву Степану Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка методу тестування програмного забезпечення для вебсервісу

затверджена наказом університету від 15 травня 2023 року № 474 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 27 травня 2023 р.

3. Вихідні дані до роботи науково-методична та науково-технічна література, дані інтернет-мережі, середовище розробки Microsoft Visual Studio Code, мова програмування JavaScript, розроблений вебзастосунок «Conduit», інструмент для e2e тестування Cypress, інструмент для API тестування Postman, інструмент для генерації звітів з автоматизованого тестування Allure, програмне забезпечення для слідкування за життєвим циклом розробки та тестування програмних застосунків JIRA.

4. Перелік питань, що потрібно опрацювати в роботі
1. Ознайомлення з вебзастосунком «Conduit» (UI, API, функціонал та можливості).
2. Побудувати процес тестування впродовж усього STLC.
3. Отримувати певну документацію та метрики після кожного етапу.
4. Провести мануальне, API та e2e тестування, зробити інтеграцію з Allure.
5. Провести тестування безпеки (XSS).
6. Сформувати з отриманої документації та метрики звіт з тестування.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Актуальність проблеми потреби у тестуванні, побудова процесу STLC, переваги автоматизованого тестування, випадки, коли API тестування стає у нагоді, важливість безпеки та її тестування на прикладі XSS вразливості.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Консультант з дотримання діючих стандартів та норм	Доцент Творошенко І.С.		

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	10.04.2023	
2	Аналіз завдання, підбір літератури	11.04.23-17.04.23	
3	Аналіз літератури з досліджуваної проблеми	18.04.23-20.04.23	
4	Аналіз технічних засобів	21.04.23-28.04.23	
5	Розробка методу	29.04.23-14.05.23	
6	Програмна реалізація	15.05.23-25.05.23	
7	Оформлення пояснювальної записки	24.05.23-26.05.23	
8	Перевірка на плагіат	27.05.23	
9	Рецензування	28.05.23	
10	Підготовка презентації та доповіді	29.05.23-30.05.23	
11	Занесення роботи в електронний архів	31.05.23	
12	Попередній захист кваліфікаційної роботи	05.06.23	

Дата видачі завдання 10 квітня 2023 р.

Студент _____
(підпис)

Керівник роботи _____ _____доц Вечірська І.Д._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 64 с., 1 табл., 35 рис., 31 джерело.

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ТЕСТУВАННЯ ВЕБСЕРВІСУ, АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, ТЕСТУВАННЯ API, ТЕСТУВАННЯ БЕЗПЕКИ, ТЕХНІКИ ТЕСТ ДИЗАЙНУ, ТИПИ ТЕСТУВАННЯ, ТЕСТ-КЕЙСИ, БАГ-РЕПОРТИ, ALLURE, POSTMAN, CYPRESS, JIRA.

Об’єктом роботи є процес тестування реального вебсервісу вподовж усього життєвого циклу розробки проєкту з використанням різноманітних типів тестування (динамічне, grey box, позитивне та негативне, автоматизоване та інші), а також технік та підходів до тестування (еквівалентних класів та граничних значень).

Метою роботи є імітація життєвого циклу тестування проєкту з моменту отримання та аналізу вимог до написання звіту по тестуванню та здачі проєкту у експлуатацію.

Очікуваним результатом роботи є виконання усіх етапів STLC та отримання документації наприкінці кожного з етапів, яка сформує метрики та стане основою для написання документу про результат тестування. Таким чином будуть отримані точніші результати щодо стану програмного застосунку, а також буде зекономлено час у пошуку багів.

SOFTWARE TESTING, TESTING OG THE WEBSERVICE, AUTOMATED TESTING, API TESTING, SECURITY TESTING, TEST DESIGN TECHNIQUES, TESTING TYPES, TEST CASES, ALLURE, POSTMAN, CYPRESS, JIRA.

The object of the work is the process of testing a real web service throughout the entire project development life cycle using a variety of testing types (dynamic, grey box, positive and negative, automated and others), as well as testing techniques and approaches (equivalence classes and boundary values).

The purpose of the work is to simulate the life cycle of all STLC stages and the receipt of documentation at the end of each stage, which will form metrics and become the basis for writing a test report document. Thus, more accurate results will be obtained regarding the state of the software application, and time will be saved in search for bugs.

The expected result of the work is the completion of each stage of the project’s testing life cycle, as well as documentation received at the end of each stage.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення

SDLC – Software Development Life Cycle (життєвий цикл розробки ПЗ)

TD – Test Design (тест дизайн)

STLC – Software Testing Life Cycle (життєвий цикл тестування ПЗ

RTM – Requirements Traceability Matrix (таблиця, що вказує покриття тестами певного функціоналу)

API – Application Programming Interface (програмний інтерфейс)

Ендпоінт – URL-адреса, через яку клієнт має змогу отримувати дані та обробляти їх

CRUD – Create Read Update Delete (операції, які можна робити з даними)

ID – Identifier (унікально ідентифікуючий номер, за яким однозначно можна дізнатись про що йде мова)

DOM – Document Object Model (об’єкт документу, який генерується після завантаження сторінки)

HTML – Hyper Text Markup Language (мова верстки та написання структури вебсайтів)

XSS – Cross-Site Scripting (міжсайтовий скріптинг)

OWASP – Open Web Application Security Project (некомерційна група ентузіастів з Інтернету, які цікавляться безпекою)

PoC – Proof of Concept (доказ, який наводиться для того, щоб показати працездатність чогось у програмуванні)

E2E – End to End (тест, який імітує поведінку користувача на вебресурсі)

UI – User Interface

ООП – об’єктно-орієнтоване програмування

ВСТУП

Тестування програмного забезпечення є важливим процесом у SDLC [1]. Хоча й існують компанії, які вважають, що тестування можуть проводити і самі розробники. Це хибна стратегія, яка призведе до вигорання у розробників та незадоволених клієнтів.

Процес тестування неможливо стандартизувати, він дуже відрізняється і причин на це багато [2]:

– замовник може не побажати проводити тестування безпеки, адже це займе додатковий час та фінансування;

– проєкту можуть бути не потрібні певні види тестування (наприклад, сайт-портфоліо Frontend Developer'у не потребує тестування інтернаціоналізації або продуктивності, або безпеки);

– обмеженість у часі;

– і ще багато різноманітних причин.

Тобто процес тестування не можливо віднести до якогось одного шаблону і всюди його використовувати. Тестування це гнучка до змін робота [3]. Звісно у сучасному світі все залежить від замовника та його потреб.

Варто ще зазначити, що провести повне тестування неможливо так саме як і заявити про відсутність багів (проблем у готовому продукті, пов'язаних з його функціональними або нефункціональними частинами). Чому? Ну ви тільки уявіть скільки існує можливих варіантів поєднань різного Environment'у, на якому потенційний користувач може користуватись нашим продуктом? Адже існують різні:

– розширення екранів;

– мова інтерфейсу;

– операційна система (та її версія);

– браузер (та його версія);

– швидкість з'єднання.

І це при тому, що завжди існує обмеження у часі (авжеж якщо не працювати над Waterfall проєктом). Тому баги існують завжди і усюди, навіть у топових гігантів індустрії, таких як Google, Facebook та інші.

Дехто може подумати «Добре, тобто коли ми розробляємо продукт, то ми не хочемо жертвувати часом, тож усі баги знайти неможливо, але ж після релізу майже завжди додають контакти для зв’язку зі службою підтримки на випадок знайдених багів». Це правда, але це не як не гарантує відсутність багів. Адже кожна нова стрічка у коді – це потенційна «бомба». Та до того ж при розробці проєкту та підтримці його після релізу важливо орієнтуватись на пріоритети. Наведу простий приклад, уявіть Укрпошту, вона випускає легендарну марку присвячену тому самому кораблю й кількість бажаючих придбати собі цю продукцію є ну дуже багатою, звісно сервери (або сервер) не витримують такого навантаження та припиняють відповідати на будь-які запити. Це баг? Так, авжеж, адже порушена функціональна складова сайту. Це варто лагодити? Ні, адже в звичайний день попиту на марки немає, тож придбання додаткових серверів буде нерентабельним вирішенням.

Отже, тестування:

- залежить від контексту;
- ґрунтується на вимогах замовника та здоровому глузді;
- не піддається шаблонізації.

Актуальність роботи полягає у освітленні процесу роботи тестувальника на основі обраного вебсервісу, у розробці методу тестування певного продукту, застосування різних TD технік та видів тестування задля демонстрації важливості та значимості праці тестувальника у будь-якій команді.

1 ТЕОРЕТИЧНІ ВІДОМОСТІ. ТЕСТУВАННЯ. ПРИНЦИПИ.

1.1 STLC

Життєвий цикл тестування – суцільний процес тестування ПЗ з моменту отримання вимог до написання звіту з тестування. Зазвичай цей процес складається з шести пунктів, але існують певні фреймворки, які можуть розширяти або навпаки прибирати деякі стадії [4].

Також варто додати, що для того, щоб почати тестування та перейти до наступної стадії потрібно відповідати вхідному критерію (Entry Criteria) та критерію на виході (Exit Criteria). Наприклад, вхідний критерій до виконання тестів є налаштоване тестове середовище, саме тому вихідний критерій попередньої стадії, а саме «Налаштування тестового середовища» є налаштоване тестове середовище.

Отже, стадії.

Крок 1. Аналіз вимог.

Вимоги повинні бути атомарними, тобто розумітися кожним членом команди однаково, з ціллю запобігання розробки неправильного функціоналу та марнування часу [5]. Також вимоги не повинні перечити одна одній. А ще варто уникати вимог від замовника з «пароль повинен бути довгим, надійним». Адже не зрозуміло, що саме є довгий та надійний пароль.

Активності на цій стадії:

- визначаються типи тестів, які необхідно виконати;
- збирається інформація про пріоритети тестування;
- починається підготовка RTM;
- визначаються деталі тестового середовища, де буде проводитись тестування.

Результатом цієї стадії, а отже вихідним критерієм буде розроблений «каркас» RTM.

Крок 2. Планування.

Проводиться оцінка зусиль і витрат на проєкт [6], також складається Тест План. Упродовж цієї стадії визначається стратегія тестування.

Активності на цій стадії:

- підготовлюється стратегія тестування;
- вибираються інструменти для тестування;
- оцінюються необхідні ресурси для тестування;
- плануються ресурси та визначаються ролі та відповідальності.

Результатом цієї стадії є готовий Тест План [7] (в теорії, але зазвичай це доволі громіздкий документ, підтримання якого займає багато часу, тому компанії відмовляються від Тест Плану та документують у тезисному форматі ключові моменти).

Крок 3. Розробка тест-кейсів.

Цей етап включає в себе створення, перевірку та переробку тест кейсів та тестових сценаріїв. На цьому етапі визначається, які тестові дані необхідно підготувати перед початком тестування, та створюються самі тест кейси [8].

Активності на цій стадії:

- створюються тест-кейси, планується автоматизація;
- переглядаються та оптимізуються тест-кейси та авто-тести;
- створюються тестові дані (дані, які необхідні для тестування, наприклад, акаунти з різними привілеями, тож адміністратор, зареєстрований користувач, користувач з преміум акаунтом і таке інше).

Результатом цієї стадії є готові тест-кейси або скрипти (для авто-тестів) та заготовлені тестові дані.

Крок 4. Налаштування тестового середовища.

Цей етап може відбуватися одночасно з попереднім, адже налаштовується розробниками. Тестувальник проводить smoke-тестування (перевірка критичного функціоналу, тільки позитивні кейси) для верифікації того, що середовище налаштовано гарно. Тестове середовище потрібно для тестування функціоналу у місті, де ніхто не заважає тестувальнику (нема

постійних змін коду, нових комітів) та тестувальник у разі чого не заважає користувачам (адже будь які зміни не вплинуть на продукт).

Активності на цій стадії:

– вибирається необхідна архітектура, налаштовується середовище та підготовлюється список вимог до обладнання та ПЗ для тестового середовища;

– налаштовується тестове середовище, імплементуються тестові дані;

– виконуються smoke-тести (health check) – перевірка на адекватну роботу середовища.

Результатом цієї стадії є готове середовище з налаштованими тестовими даними.

Крок 5. Виконання тесту.

На цьому етапі проводиться тестування відповідно до існуючих планів, стратегії тестування та по готовій документації (тест-кейси, чеклісти, use-cases). У разі виникнення помилок ця інформація доноситься до відповідного відділу та проводиться ретест після баг-фіксу.

Активності на цій стадії:

– виконуються заплановані тестові сценарії;

– документуються результати та заводяться баг-репорти;

– заповнюється RTM;

– відбувається ретест після баг-фіксів.

Результатом цієї стадії є повністю завершена RTM, тест кейси з відповідними результатами (Passed, Failed, Blocked) та заведені баг-репорти на тести, що впали та «заблокований функціонал» через ті ж тести, що впали.

Крок 6. Закриття тестового циклу.

Здебільше аналітичний процес, на якому команда тестування обговорює процес тестування та формує якісь поради, бест-практіси для застосування на подальших проєктах.

Активності на цій стадії:

– оцінюються критерії завершення циклу на основі витраченого часу, тестового покриття, вартості, ПЗ, важливих бізнес-цілей та отриманого показника якості;

– підготовлюються тестові показники на основі вищевказаних параметрів;

– створюється звітність якості продукту для замовника;

– аналізуються результати тестів для визначення розподілу багів за типом та серйозністю.

Результатом цієї стадії є тестові показники, наприклад, кількість пройдених тестів, процентне співвідношення, кількість багів розподілена за серйозністю та пріоритетом та інші.

1.2 Принципи тестування

Принципи тестування – це як кодекс порядного тестувальника. Вони описують бест-практіси, яких слід дотримуватись підходячи до тестування будь-якого ПЗ [9].

Принципи:

– тестування – це про наявність дефектів, а не їх відсутність. Тобто відсутність дефектів при першому тестуванні або після ретесту (повторного тестування), баг-фіксів не дає гарантії, що багів взагалі немає. Адже все перевірити просто неможливо, на це піде дуже багато часу і це буде невиправдано. Тестування лише запевняє, що продукт або відповідає або не відповідає явним та неявним вимогам та зменшує кількість ймовірних неточностей;

– вичерпне тестування неможливо. Цей принцип тісно пов’язан з попереднім. Можна додати, що навіть за умови необмеженого часу та ресурсів цей процес все ще є неможливим через зміни в інфраструктурі, появі нових середовищ (браузерів, версій ПЗ, операційної системи, third-party

функціоналу і таке інше). Також варто додати, що іноді баги можуть виникати при дуже нетиповому user-flow (поведінці користувача), наприклад, коли у програмному застосунку для прослуховування музики користувач впродовж тривалого часу переключається між різними вкладками;

– раннє тестування допомагає запобігти більшим витратам на виправлення помилок. Чим раніше буде помічена неточність – тим легше буде її виправлення. Наприклад, якщо при аналізі вимог ми помітимо якусь неточність, то це буде зміна лише у вимогах, ще до початку програмування, побудови структури, архітектури і все таке, що зекономить час та ресурси в подальшому;

– групування дефектів. В даному випадку кластеризація дефектів відбувається за принципом Парето, або ж правило «80-20». Це говорить про те, що 80% багів зазвичай знаходяться у 20% найскладнішого функціоналу [10];

– парадокс пестицидів. Цей принцип каже про необхідність оновлювати та підтримувати тестові дані у актуальному стані. Адже якщо не змінювати тестові дані, то вони просто не будуть знаходити баги в майбутньому;

– тестування є залежним від контексту. Обрані стратегії, типи та техніки тестування будуть відрізнятися, наприклад, для онлайн-магазину та онлайн-банкінгу;

– відсутність дефектів оманлива. Як вже було зазначено, неможливо протестувати проєкт цілком. Але цей принцип ще ґрунтується на тому, що навіть за умови гарно-працюючого продукту ще не факт, що продукт буде популярним серед користувачів. Важливо пам’ятати про верифікацію та валідацію. А також потрібно не забувати про розробку user-friendly інтерфейсу. Інакше існує ймовірність розробляти тривалий час продукт, який нікому не буде потрібен, що призведе до витрачених ні на що грошей, часу та інших ресурсів.

1.3 Методології розробки ПЗ

Упродовж розвитку індустрії створювались, формувались та вдосконалювались нові та вже існуючі методології, які націлені на ефективну розробку ПЗ (рис. 1.1).

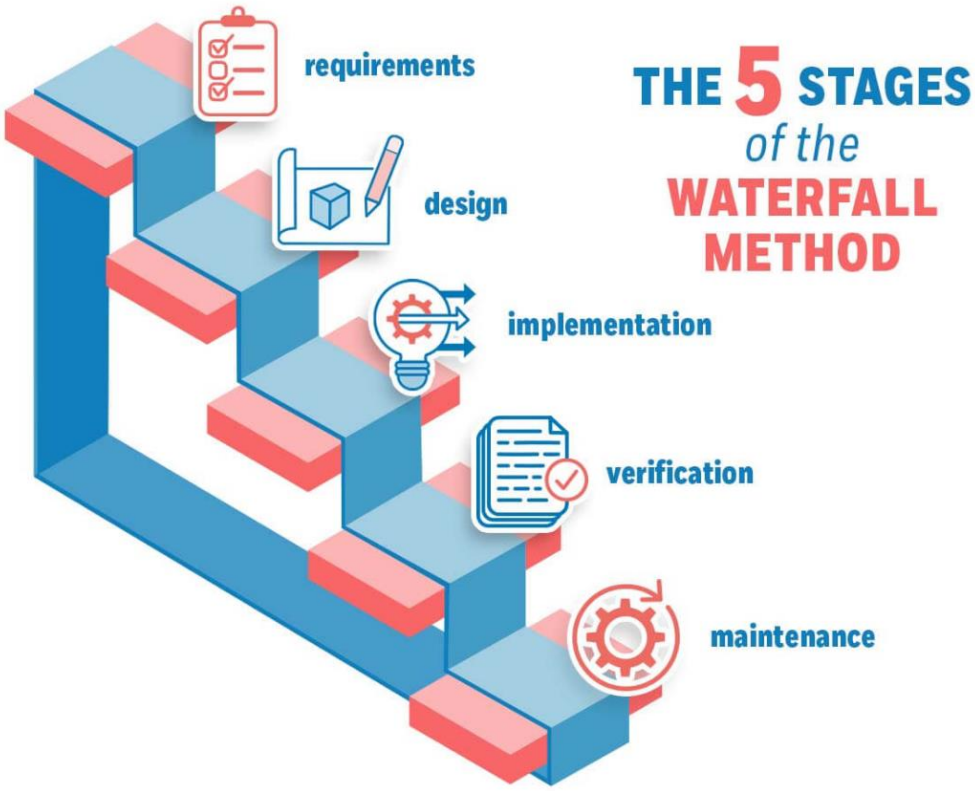

Рисунок 1.1 – Візуальна модель методології Waterfall

Першою такою методологією була модель Waterfall. Вона існує й зараз та має своїх прихільників. Сильною стороною даної методології є повна відповідність вимогам, тож документація буде дуже гарна, перехід до наступної стадії відбувається лише за умови повного виконання попередньої.

З описаних плюсів випливають доволі вразливі недоліки. А саме дуже великі часові витрати та неможливість повернутись на стадію позад щоб виправити щось або додати якийсь функіонал. Тобто в теперішньому світі, де дуже багато продуктових кампаній і потрібно якомога швидше розробляти продукт інакше з часом він може бути вже не актуальним. Тим паче даний

підхід є користним для тих проєктів, де потрібна вичерпна документація і які не поспішають нікуди, адже вони в своєму монополісти. Це зазвичай державні програми, по типу військових та медичних.

На відміну від цінностей Waterfall моделі існує Agile методологія, яка використовується здебільше на сьогоднішний день через свою гнучкість [11]. Agile містить в собі 12 принципів, яких слід дотримуватись, якщо ПЗ розробляється саме за цією методологією та 4 цінностей.

Отже, цінності:

– люди та співпраця важливіші за інструменти та процеси;
– працюючий продукт важливіший за вичерпну длкументацію;
– співпраця із замовником важливіша за обговорення умов контракту;
– готовність до змін важливіша за дотримання плану.

З наведених вище цінностей добре видно, що Agile має в основі своєї філософії інтеракцію та взаємодію з людьми, будь-то члени команди або замовник. Зайва бюрократія, як й відмова йти на зустріч з клієнтом через вже підписаний контракт з конкретними домовленностями – це все про Agile.

Agile є доволі популярним наразі через що люди створюють «фреймворки», тобто свої методології, які базується саме на Agile. Прикладами яких є Kanban та Scrum (рис. 1.2).

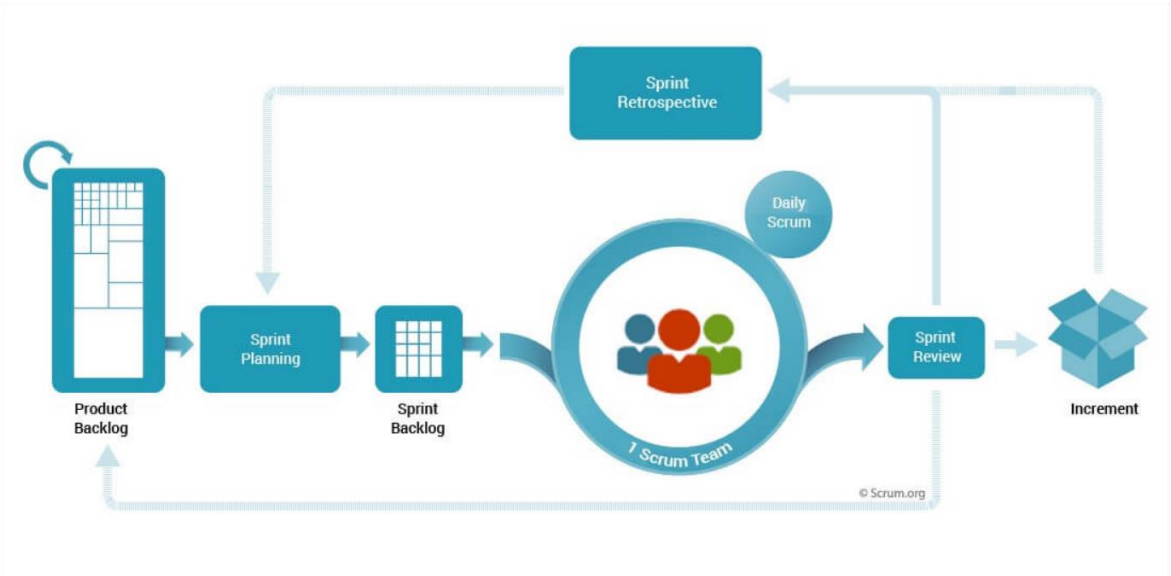

Рисунок 1.2 – Візуальна модель методології Scrum

1.4 Опис предметної області

Предметна область – вебресурс з публікації та перегляду статей на будь-які теми «Conduit». Цей вебпортал побудован з ціллю стати безкоштовним та більш досконалим аніж його попередник «Medium». Адже замовник вірить, що гарні ідеї та лайфхаки до кращої якості життя та підвищення кваліфікації повинні бути у відкритому доступі. Вебсервіс не містить реклами та спонсується різними організаціями, через що й має змогу існувати.

Продукт має user-friendly інтерфейс, зрозумілий у використанні та має швидкий та легкий шлях до розміщення власних ідей на порталі.

Вебресурс нараховує наступні сторінки:

- головна;
- домашня;
- реєстрація;
- аутентифікація;
- створення нової статті;
- редагування статті;
- налаштування профілю;
- сторінка статті;
- сторінка користувача.

Продукт також має гарний функціонал, який дозволяє підписуватися на певних авторів, творчість яких здається вам гарною та розподіляти свою «стрічку» на глобальну, яка містить усі актуальні статті та на вашу, в якій відображається статті ваших улюблених авторів.

Також до статей можна додавати теги (короткі слова, які характеризують цю статтю найкраще). Це важливо, адже у вебресурсі є можливість пошуку та фільтрацію статей саме за тегами. Таким чином це покращує комунікацію у програмному застосунку. Більше користувачів побачать контент, який подобається саме їм та більше гарних ідей від авторів

буде побачено. Тобто обидві сторони й автори й користувачі будуть задоволені. Автор буде почут, а користувач буде мати певні знання та додатковий досвід. Звісно ж це саме те, що потрібно програмному застосунку.

1.5 Постановка задачі

Отже, тестування є актуальним та потрібним процесом на нашому проєкті. Тестування включає у себе багато різних аспектів, від однозначного розуміння вимог до кожного модулю (разом з неявними) до розуміння як працює Інтернет, яку архітектуру повинен мати сервіс, що розробляється. Тому ставиться завдання розробки методу тестування вебсервісу з публікацію різноманітних статей (аналог «Medium») з урахуванням наявних вимог (явних та неявних), а також обмежень (здебільше часових).

Об’єктом роботи є процес тестування реального вебсервісу вподовж усього життєвого циклу розробки проєкту. З використанням різноманітних типів тестування, а також технік та підходів до тестування.

Метою роботи є імітація життєвого циклу тестування проєкту з моменту отримання та аналізу вимог до написання звіту по тестуванню та здачі проєкту у експлуатацію.

Для досягнення мети необхідно вирішити такі завдання:

- провести аналіз вимог [12];
- провести планування;
- розробити тест-кейси;
- налаштування тестового середовища;
- виконання тесту;
- закриття тестового циклу;
- досягти значення показнику пройдених тест-кейсів на рівні не меншим за 92%.

Вище наведені усі етапи STLC, які й потрібно виконати задля досягнення поставленої задачі. Варто зауважити, що пункт з налаштуванням тестового середовища практично буде опущено через те, що моя робота – це лише імітація роботи тестувальника, нажаль в мене нема розробників на зв’язку та інших людей, які зазвичай задіяні у процес розробки продукту. Проте з теоретичної точки уся потрібна інформація буде наведена.

2 HOW THE WEB WORKS. ІНФОРМАЦІЙНА МОДЕЛЬ ТЕСТУВАННЯ

2.1 API

Інтернет це безліч різних мереж, які підключені між собою за допомогою різних пристроїв [13]. Комунікація між пристроями та мережами може відбуватися за допомогою дротового або бездротового з'єднання. Це можуть бути модеми, роутери, Ethernet дроти, Bluetooth та інші.

Для користувачів це може виглядати, як ми пишемо щось у пошуковій стрічці браузеру, натискаємо на «Enter» та перед ними з'являється бажана сторінка, бо так має працювати. Але, насправді, щоб здійснити цей процес відправки запиту до потрібного серверу, підготування відповіді та відправки її до клієнту – це достатньо складний шлях. Сервер – це той самий комп'ютер, лише більш потужний, який знаходиться десь у Інтернеті, у своїй мережі [14]. Тож, для того, щоб запит дістався потрібного серверу, потрібно минути велику кількість «проміжних ланок», таких як модеми, роутери та switch'i.

Тож API фактично організують та підтримують увесь процес обміну даними у мережі [15]. Багато хто полюбляє надавати роботу ресторану як приклад для пояснення концепту API. Отже, інтерпретуємо роботу програмного застосунку на роботу ресторану.

Програмний застосунок має:

– frontend частину, яку можна порівняти з відвідувачами ресторану, які сидять у залі та чекають на свої замовлення;

– backend частину, яка є кухнею у цьому прикладі, тобто тим, що приховано від очей звичайних клієнтів;

– API, що гарантує комунікацію між залом з відвідувачами та кухнею, тобто офіціанти.

Наглядна ілюстрація цього процесу та інтерпретації зображена на рисунку 2.1.

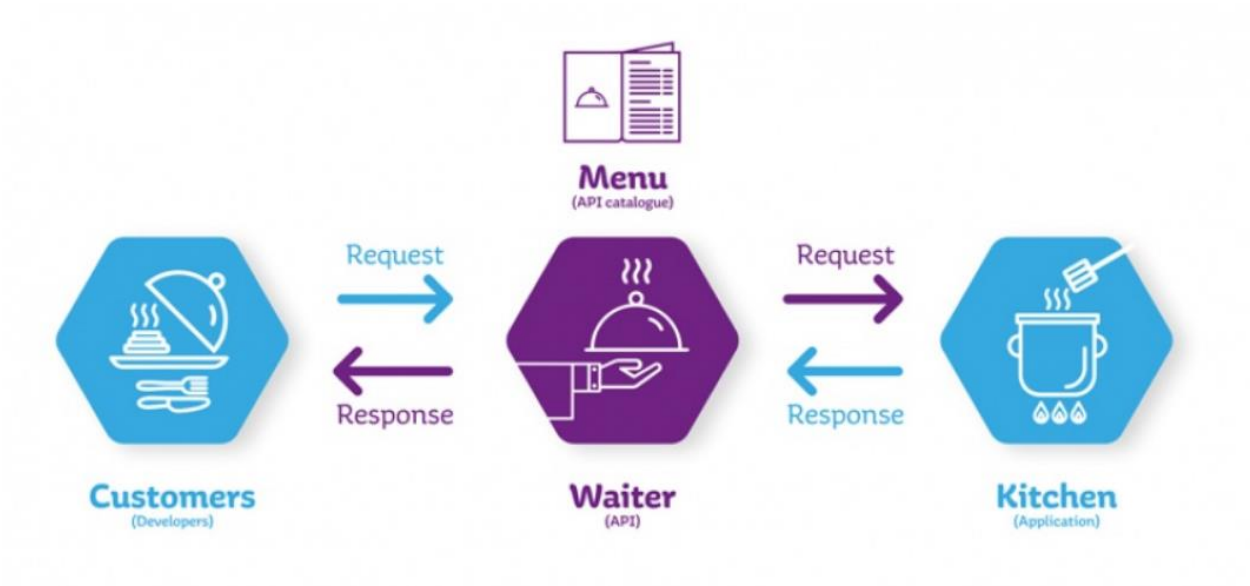

Рисунок 2.1 – Як працює вебзастосунок [16]

Тобто без API в нас є сайт, з якимись елементами, він може бути гарним, зрозумілим, user-friendly, на сайті може бути база даних, де зберігається уся інформація, відбувається валідація даних, але немає ніяких інструкцій між frontend та backend, тож отримуємо непрацюючий сайт. API запевняється, що якщо відвідувач онлайн-магазину захоче подивитись на товари, то при натисканні на певне посилання або кнопку сформується та відправиться запит до серверу цього вебресурсу з проханням про відображення користувачу сторінки з товарами, які будуть взяті з бази даних та відповідь у вигляді вже готової сторінки або файлу з інструкціями, як її відобразити буде надіслана до клієнта, який зробив цей запит (рис. 2.2).

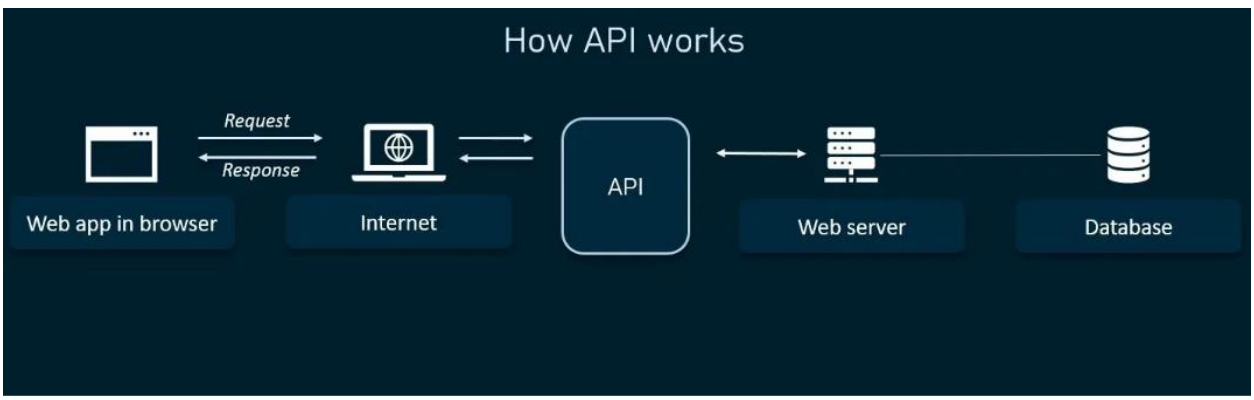

Рисунок 2.2 – Як працює API

2.2 REST API

Існує декілька доволі популярних та розвинутих архітектур щодо розробки API. Ці архітектури – це набір правил, інструкції з програмної реалізації API частини програмного застосунку. Найбільш популярним є REST архітектура, програмні застосунки, які відповідають REST називаються RESTful. Існують й інші, наприклад, GraphQL та SOAP, вони доволі сильно відрізняються одне від одного та мають свої плюси та мінуси.

Як було зазначено, REST API – це архітектурний стиль, який має свої правила, або ж рекомендації. Важливо розуміти, що це все ж таки більше рекомендації і в реальності вебзастосунки, які планувались проєктуватись за REST, іноді порушували певні рекомендації, роблячи свій наче фреймворк.

Отже, кожен HTTP метод відповідає за певну дію і є інтуїтивно зрозумілим:

- GET – отримання даних;
- POST – створення нових даних;
- DELETE – видалення даних;
- PUT – редагування даних (уся сутність);
- PATCH – редагування даних (певна властивість).

Це основні з запитів, існують ще, наприклад, HEAD та OPTIONS, але вони не так часто використовуються та націлені на роботу з Header’ами.

З кожним запитом користувач надсилає заголовки, які несуть мета-інформацію, як кодування, мова, формат стиску, з якого пристрою робиться запит, його розширення екрану і таке інше. Тобто заголовки розроблені для зберігання інформації про користувачів та надання необхідної їм інформації у задовільному вигляді. Також часто з запитом надсилається його тіло, в якому знаходиться інформація, з якою користувач хоче зробити якусь операцію над цими даними. Тіло запиту не завжди присутнє, наприклад, при запиті на видалення достатньо передати ID сутності у URL. Загальний вигляд запиту зображено на рисунку 2.3.

Рисунок 2.3 – Приклад запиту [17]

Кожна відповідь на запит також повинна мати свої заголовки, в яких підтверджується тип кодування, тип стиску та інші налаштування. А також відповіді можуть мати тіло, де повертається та підтверджується інформація про дані, над якими було здійснено певну операцію. Наглядний приклад заголовків та тіла відповіді на запит зображені на рисунках 2.4 та 2.5.

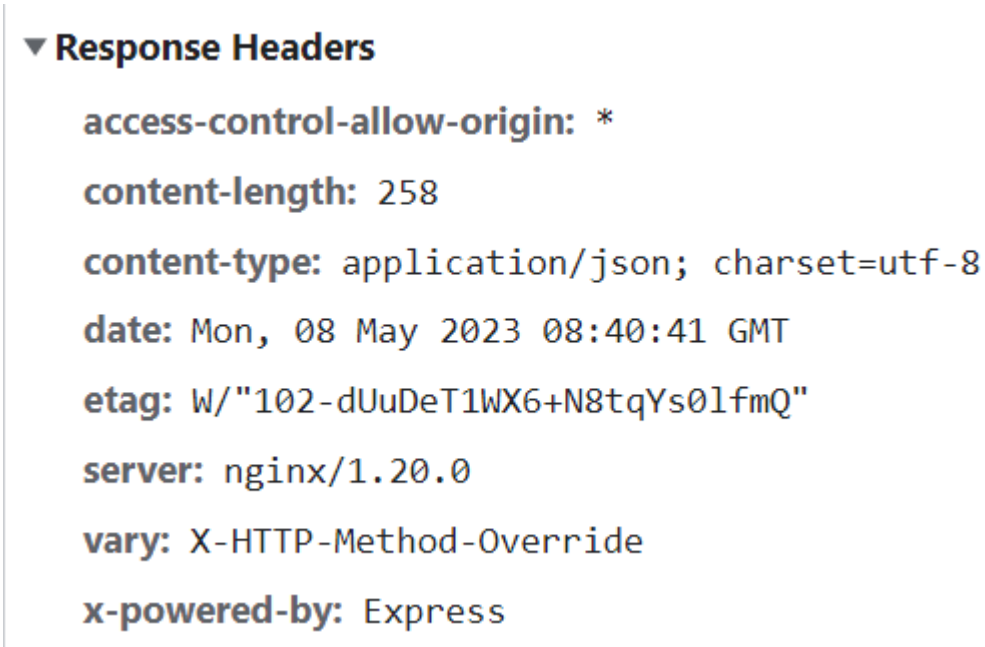

Рисунок 2.4 – Заголовки відповіді на запит

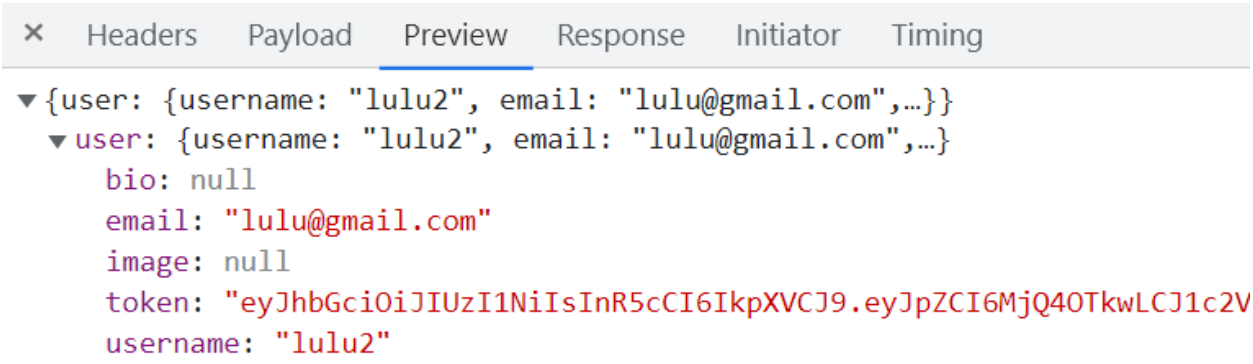

Рисунок 2.5 – Тіло відповіді на запит авторизації

Також варто зауважити, що у відповіді на запит авторизації не надійшов пароль. Це зроблено з піклувань про безпеку, адже таку вразливу інформацію небезпечно відправляти по мережі, до того ж він скоріше за все буде зашифрованим (захешованим), що може заплутати користувача, адже зберігати такі дані як паролі, дані від карток та інші цінні дані у «чистому» вигляді неприпустимо зараз, адже це загрожує втраті особистих даних користувачів [18]. Щоправда зараз й зашифровані дані можна розшифрувати і це іноді взагалі нетривалий процес, особливо, коли на вебресурсі нема ніяких додаткових вимог до паролю на кшталт «повинен містити щонайменше одну цифру, спеціальний символ, складатися принаймні з 7 символів».

Існують так звані «райдужні» таблиці, в яких зібрані хеши найбільш популярних паролів по типу «Qwerty», «Password» та інших, які за допомогою певних інстурментів розшифровуються за дуже швидкий час. Саме тому розробники програмних застосунків додають декілька раз алгоритм шифрування на вразливі дані та додають так звану «сіль». Отже, «сіль» – це спеціальна строка, яка генерується для кожного користувача окремо, додається до хешованого пароля та хешується ще раз (або декілька). Таким чином навіть якщо у декількох користувачів буде однаковий пароль, то у базі даних хеши їх паролів будуть різні. Це робиться задля підсилення захисту даних, адже навіть якщо хакеру вдасться отримати доступ до

хешованих паролів, то від них самих немає сенсу, хешований пароль не буде вважатися валідним при аутентифікації на будь-якому вебресурсі.

Разом з відповіддю надходить і статус код щодо виконаного запиту. Статус код це – номер, який складається з трьох цифр та є сталим. Його завдання – сповістити про результат виконання запиту. Існують наступні статус коди [19]:

– 1** – ті, що починаються на 1 – інформаційні. Наприклад, 101 – «Зміна протоколу»;

– 2** – ті, що починаються на 2 – успішні. Наприклад, 200 – «Ок»;

– 3** – ті, що починаються на 3 – перенаправлення (редіректи). Наприклад, 302 – «Знайдено»;

– 4** – ті, що починаються на 4 – помилки з боку клієнту. Наприклад, 404 – «Не знайдено»;

– 5** – ті, що починаються на 5 – помилки з боку серверу. Наприклад, 503 – «Сервіс недоступний».

Отже статус коди є сталими, 200 це завжди «Ок», увесь їх перелік є у відкритому доступі та REST вимагає, щоб операція над даними відповідала статус коду, який повертається, як результат дії над цими даними. Наприклад, якщо користувач хоче додати якісь дані (пост у соціальній мережі), то у відповідь на цей запит повинен надійти 201 статус код, який відповідає за створення даних.

Також гарною практикою є запобігання використання параметрів у запитах (це коли дані надсилаються у URL після знаку питання, як зазначено тут: */api?type=user&id=1*). Вони можуть поєднуватись за допомогою знаку «&». Це робиться у тому числі з міркувань безпеки, але таким чином можна «забруднити» параметри та надіслати запис з параметрами *?id=1&id=2*. І не зрозуміло який з цих *id* буде обрано, бо наприклад під *id=1* може бути сторінка адміністратора, таким чином можна буде отримати її за умови, якщо розробники не потурбувались про це.

У таблиці 2.1 наведені гарні шляхи до формування ендпоінтів за REST.

Таблиця 2.1 – Гарні ендпоінти за REST

Ендпоінт	HTTP метод	Опис
/api/users	GET	Отримання користувачів
/api/users/1	GET	Отримання користувача з id=1
/api/users	POST	Додання користувача
/api/users/1	PUT	Оновлення даних користувача з id=1
/api/users/1	DELETE	Видалення користувача з id=1

2.3 Токени

HTTP – stateless протокол, що значить, що він не запам’ятовує клієнта. Тобто, якщо ми увійшли на якийсь сайт за допомогою своїх даних та хочемо перейти у розділ «Товари», то сайт не знає, що це саме цей користувач хоче запросити відобразити певну сторінку. Як же тоді серверу дізнатись хто саме надіслав цей запит? Відповідь проста – за допомогою токену.

Коли користувач реєструється на вебресурсі або заходить у свій особистий акаунт, то для цього потрібно надати особисті дані (зазвичай пошта та пароль) для того, щоб аутентифікуватись у системі. Тобто підтвердити, що такий користувач існує. Після того, як вебсервер звернеться до бази даних з SQL запитом та перевірить чи дійсно є у базі даних користувач з такою парою (пошта-пароль), тоді сервер у відповідь на цей запит відправить токен авторизації [20]. Авторизація – це процес підтвердження особи за наданими даними від акаунту та надання певних прав цьому юзеру. Під «певних прав» мається на увазі чи належали надані дані від акаунти адміністратору чи може звичайному користувачу чи може користувачу, який придбав преміум-доступ [21].

Отже, цей токен, який надсилається кожен раз, коли користувач аутентифікує себе – це велика стрічка з довільним змістом, яке не має сенсу само по собі та яка зберігається зазвичай у Cookies. У Cookies зберігається

інформація щодо користувача, його побажань та вподобань. Це робить досвід користування вебресурсу більш привабливим та гарним для користувача. Cookies мають властивість відсилатися разом з кожним запитом на сервер. Тобто, користувач надав свої дані, отримав доступ у систему, авторизувався та отримав своє унікальне значення токену (зазвичай toke, authentication-token або sessionId) та вже при усіх наступних запитах (до моменту виходу з системи) з кожним запитом надсилається цей токен, який зберігається у базі даних на сервері та за допомогою якого сервер може ідентифікувати користувача та надіслати відповідь на певний запит.

Іноді токен може надсилатися в тілі запиту, але зазвичай це окремий заголовок, який називається Authentication, або x-token. Токен може бути зашифрованим, адже, якщо людина з поганими намірами володіє токеном іншого користувача, то це те ж саме, якби ця людина володіла даними від акаунту користувача. Проте це не досить розповсюдження практика, шифрування токенів.

Зазвичай токен має приблизно такий вигляд: *eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MjQ4OTkwLCJ1c2VybmFtZSI6Imx1bHUyIiwiZXhwIjoxNjg4NzE5MjQxLCJpYXQiOjE2ODM1MzUyNDF9.GcHmvHc_E3aDrSBZqCxVJy6dV4fnXPckvIqA4ikwpEw*

Загальна суть токенів зображена на рисунку 2.6.

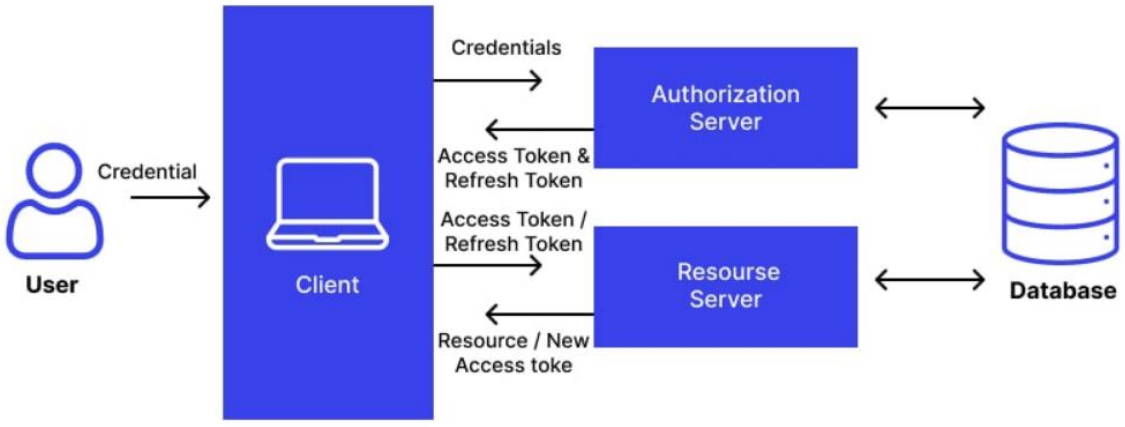

Рисунок 2.6 – Як працює токен [22]

2.4 DOM

DOM – це набір правил, який визначає, як буде відображатись HTML-сторінка у браузері. Він відповідає за елементи, які будуть відображені, які в них будуть стилі, як будуть змінюватися стилі при певних подіях. Тобто DOM – це те, що динамічно змінює структури сторінки при кожній зміні (наприклад, користувач вводить якісь дані у текстове поле).

При написанні вебсайту розробники використовують HTML, який складається з тегів. Кожен тег інтерпретується у JS та таким чином формується об’єкт у DOM дереві. DOM дерево має приблизно такий вид, як зазначено на рисунку 2.7.

Елемент DOM дерева, або ж DOM елемент ще часто називають нодою (node).

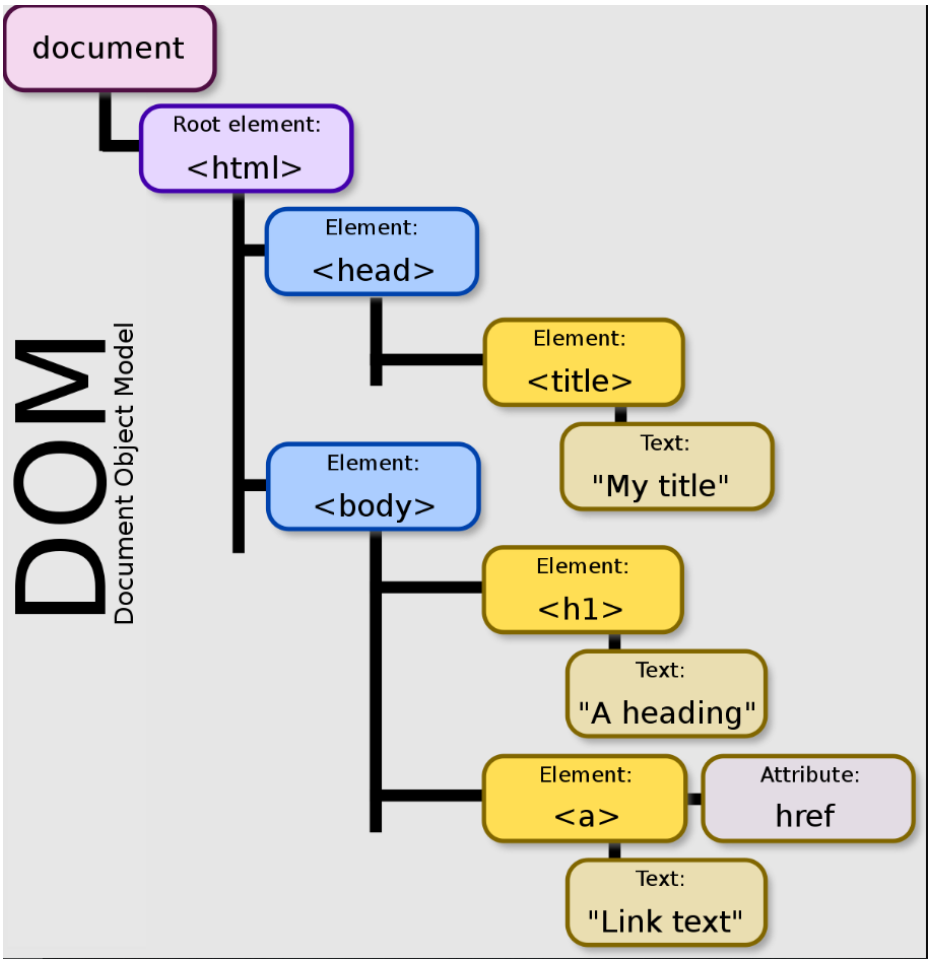

Рисунок 2.7 – DOM дерево [23]

2.5 XSS вразливість

XSS – відома та доволі серйозна вразливість відповідно до OWASP [24]. На даний момент XSS вразливість не входить до найбільш критичних, хоча й досі існує та є доволі актуальною (рис. 2.8). Насправді XSS можна віднести до ін'єкцій, адже суть цієї вразливості полягає у написані та збереженні коду у DOM-елементі або на сервері (відповідно до типу застосованого XSS).

OWASP Top 10 - 2010	OWASP Top 10 - 2013	OWASP Top 10 - 2017
A1 – Injection	A1 – Injection	A1 – Injection
A2 – Cross Site Scripting (XSS)	A2 – Broken Authentication and Session Management	A2 – Broken Authentication
A3 – Broken Authentication and Session Management	A3 – Cross-Site Scripting (XSS)	A3 – Sensitive Data Exposure
A4 – Insecure Direct Object References	A4 – Insecure Direct Object References **[Merged+A7]**	A4 – XML External Entities (XXE) **[NEW]**
A5 – Cross Site Request Forgery (CSRF)	A5 – Security Misconfiguration	A5 – Broken Access Control **[Merged]**
A6 – Security Misconfiguration **(NEW)**	A6 – Sensitive Data Exposure	A6 – Security Misconfiguration
A7 – Insecure Cryptographic Storage	A7 – Missing Function Level Access Control **[Merged+A4]**	A7 – Cross-Site Scripting (XSS)
A8 – Failure to Restrict URL Access	A8 – Cross-Site Request Forgery (CSRF)	A8 – Insecure Deserialization **[NEW, Community]**
A9 – Insufficient Transport Layer Protection	A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards **(NEW)**	A10 – Unvalidated Redirects and Forwards	A10 – Insufficient Logging & Monitoring **[NEW,Comm.]**

Рисунок 2.8 – Топ-10 найбільш вразливих вразливостей за 2010-2017 роки

Існує 2 типи XSS:

– Stored (збережений) – коли шкідливий код зберігається на сервері та відтворюється для будь-якого користувача, хто відвідає сторінку з цим елементом, або якось взаємодіє з ним. Наприклад, якщо у програмному застосунку для написання статей створити статтю та у поле назви написати шкідливий код разом з валідною назвою, в такому разі будь-хто, хто відвідає сторінку цієї статті буже вражений шкідливим кодом;

– Reflected (відображуваний) – коли посилання зі шкидливим кодом у ньому відправляється користувачу. Користувач клікає на нього і вражається шкідливим кодом, який зберігається у DOM-елементі його сторінки.

Тобто збережений тип XSS є більш маштабним, а відображуваний орієнтується на певну групу користувачів або на певного користувача (як фішинг, ще відомий як spear phisfing).

Щодо відображуванного XSS можна подумати, що у посиланні ж буде видно, що там вбудований якийсь шкідливий код. Посилання буде мати якийсь такий вигляд: *www.somesite.com?param=<script>alert('XSS')</script>*. Звісно замість звичайного alert (яке є PoC), яке відповідає за відображення модального вікна, яке несе в собі лише інформацію та відображує певний текст буде щось, що відправить Cookies з браузеру користувача на сервер хакера, або на його пошту, будь-який спосіб підійде (рис. 2.9). Так ось, посилання звісно ж буде захешоване, таким чином, не буде змоги подивитись на нього та зрозуміти, чи є воно шкідливим, чи ні.

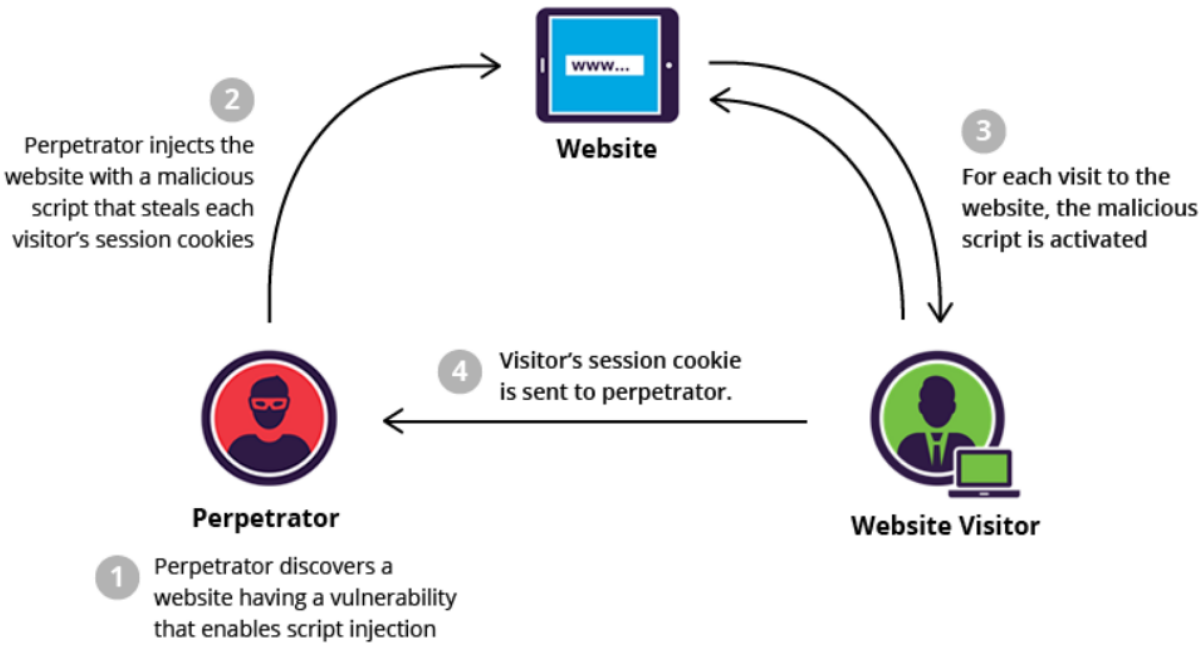

Рисунок 2.9 – Як працює XSS

XSS з'являється там, де є ввод користувача (текстові поля) та там, де нема санітарізації та валідації наданих користувачем даних. Як вже було зазначено XSS – це вбудування хакером додаткового коду у функціонал вебресурсу з поганими намірами авжеж. Існують вебресурси, на яких

зазначені різні скріпти, за допомогою яких можна перевірити свій програмний застосунок на вразливість XSS. Адже просто занести <script> у чорний лист буде недостатньо. Наприклад, можна використати скріпт, де зазначається невірний шлях до картинки і пишеться реалізація шкідливого коду у разі якщо картинка не відобразиться. Виглядати це буде якось так: *<img src=x onerror="alert('XSS')">*. Cookies можна отримати за допомогою скріпту *document.cookie*.

2.6 Асинхронність у програмуванні

При автоматизованому тестуванні доводиться писати код. Були обрані Postman для API тестування та Cypress для E2E тестування. Які самі по собі є асинхронними та підтримують використання JS, який є асинхронною мовою програмування. Асинхронність полягає у тому, що виконання інструкцій (команд) буде не упорядковане, поки одна команда працює та повертає відповідь в цей час можуть бути вже зроблені наступні команди. Це заважає особливо коли наступні команди залежать від результату попередніх (а таке буває дуже часто). Наприклад, гарним тоном у програмуванні є незалежність сутностей. У тестуванні відбувається те ж саме, краще зробити тест незалежним, щоб була змога запустити його самостійно і отримати гарно працюючий тест.

Наприклад, потрібно протестувати, що користувач може зареєструватись на сайті, якщо його дані є валідними. Та наступним тестом потрібно перевірити, що користувач не може зареєструватись з поштою, яку вже використовує будь-який з користувачів. Якщо ми їх запускаємо послідовно, тоді все гарно, ніяких проблем, але, якщо ми змінимо порядок та запустимо спочатку другий, а потім перший тест або запустимо лише другий тест, тоді всі наші тести у цих випадках впадуть.

Перше, що може спасти на думку так це використання тайм-аутів, які на певний час будуть призупиняти роботу програми. Це погане рішення, адже в будь-якому разі програма буде чекати зазначений час, навіть якщо команда виконається за пів часу, який тестувальник відніс під тайм-аут, а по-друге, нема ніяких гарантій, що цього фіксованого часу буде достатньо задля виконання команди.

Тож рішенням цієї проблеми стають проміси (promise). Вони вказують, що певний функціонал є асинхронним, а також, що потрібно почекати на його завершення. Promise має три стани:

– pending – очікування, стан, з яким створюється promise;

– resolved – вирішений, стан, який отримує promise, якщо ніякої помилки не було сформовано під час виконання коду;

– rejected – відхилений, стан, який отримує promise, якщо під час виконання коду була знайдена помилка.

Загальний концепт роботи promise зображено на рисунку 2.10.

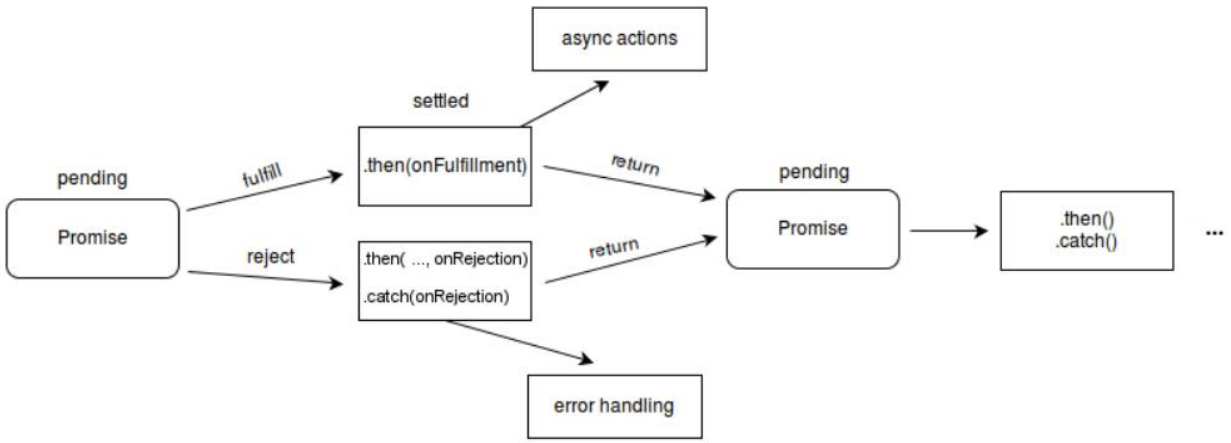

Рисунок 2.10 – Як працює promise [25]

Ще однією перевагою використання promise є можливість обробити виключення (помилку), а також можливість уникнути багатої кількості вложенного коду та фіксованих тайм-аутів. Адже promise повертають promise, що робить можливим chaining (процес виклику методів у цепочку,

один за одним). За допомогою promise також можна працювати з результатом попереднього promise за допомогою методу *then()*.

Лістинг 2.1 Використання promise задля запобігання розробки з асинхроним кодом:

```
eval(globals.login)().then((response) => {
   pm.collectionVariables.set(
       "userName", response.json().user.username);
})
```

У зазначеному вище лістингу спочатку відбувається відпрацювання функції *login*, яка реєструє нового користувача та входить до вебресурсу, використовуючи дані створеного користувача, а після цього отримує з відповіді на запит входу до системи користувацьке ім’я та записує його у змінну.

3 STLC

3.1 Аналіз вимог

У рамках кваліфікаційної роботи був розроблений метод тестування програмного вебзастосунку з перегляду та розміщення статей «Conduit». У ході роботи були пройдені усі стадії STLC.

Так як метою роботи було пройти увесь шлях тестування, то потрібно було почати з аналізу вимог. Вимоги – те, як замовник бачить розробляємий продукт. Варто розуміти, що є явні вимоги (це ті, які зазначаються замовником та корегуються бізнес-аналітиком зазвичай задля того, щоб перекласти ці вимоги у конкретні задачі для розробників) та неявні (це ті, які ніде не прописуються, але є й так очевидними). Неявною вимогою є наприклад, те, що замовник очікує на гарний продукт, який працює без перебоїв та задовольняє потреби користувачів.

Але не завжди вимоги від замовника є правильними. Під «правильними» розуміється наступне [26]:

- вимоги можуть бути не атомарними;
- вимоги можуть перечити одна одній;
- певний функціонал може бути залишений без вимог;
- вимоги можуть бути нелогічними.

При розробці програмного застосунку важливо завжди пам’ятати про валідацію та верифікацію. Верифікація це – розробка згідно до вимог, зазначених замовником, а валідація – це розробка згідно до потреб користувачів, адже може трапитися ситуація, де розроблений продукт не буде користуватись попитом серед користувачів, саме тому важливо перевіряти вимоги та аналізувати їх, щоб уникати ймовірність поганого ісходу.

Вимоги, які надіслав замовник виглядають приблизно так, як зазначено на рисунку 3.1.

Username

- required field;
- should be unique;
- min 3 symbols;
- max 40 symbols;
- should start with the letter;
- should accept Latin letters and digits;
- special characters and non-printing symbols are not allowed;
- no case sensitive.

Рисунок 3.1 – Вимоги від замовника

На рисунку 3.1 зазначені певні вимоги для текстового поля, де користувач може ввести своє бажане ім'я, проте іноді у вимогах є такі неточності, як зображено на рисунку 3.2.

Username	Mandatory field; If empty, "Username can't be blank." error message appears; Contains the username of the user; By default? Must be unique; Min 4 symbols; Max 36 symbols; Username field shows 3 for minimum and 40 for maximum Accepts only Latin characters and special symbols " - _ "; Special characters not allowed according to the Username requirements it also accept digits Username is not case sensitive.

Рисунок 3.2 – Неточні вимоги від замовника

Таким чином на рисунку 3.2 червоним зображені коментарі тестувальника (мене). Вони зображені там, де на мою думку є розбіжності у вимогах, вони є нелогічними або неповними [27]. Як можна помітити деякі вимоги суперечать одна одній на рисунках 3.1 та 3.2.

Загальні вимоги до вимог є наступними:

– атомарність;

– послідовність;

– однозначність;

– повнота (розглянуті усі можливі випадки взаємодії з функціоналом);

– верифікованість (зрозумілість вимог, щоб вони були тестуємі. Наприклад, слова як «пароль надійний», «продуктивність гарна» не є зрозумілими і кожною людиною розуміються по-своєму).

3.2 Тест План

Тест план – це доволі об’ємний документ, який в собі описує стратегію тестування, описує певні крітерії для початку тестування, пояснює, коли можна вважати тестування вдалим і таке інше. У більшості випадків його не пишуть, адже написання такого роду тестового документу вимагає багато часу, а також його складно підтримувати, хоча й безумовна перевага в гарній, чіткій та послідовній документації.

Тест план має у собі багато різних складових, деякі з них зображені на рисунках 3.3 та 3.4.

Тест план окрім цього ще містить:

– функціонал та типи тестування, які будуть протестовані у межах тестування;

– можливі ризики та проблеми, які можуть виникнути під час тестування та можливі шляхи їх поліпшення;

– критерії переходу до нової фази тестування, його зупинки та завершення;

– планування ресурсів, які знадобляться під час тестування;

– людські ресурси (команда, яка буде проводити тестування з чітко зазначенними відповідальностями кожного);

– опис тестового середовища;

– планування та оцінка часу, який знадобиться на певні таски;

– чіткий та точний графік з датами початку виконання певного таску та його кінця;

– тест артефакти, які будуть сформовані після тестування та надані замовнику.

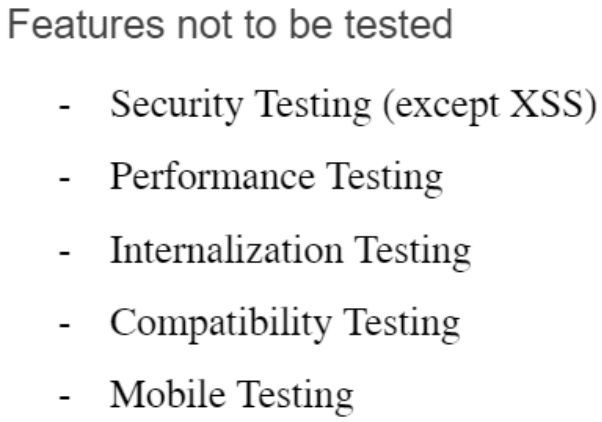

Рисунок 3.3 – Функціонал та тестування, яке не буде проведено

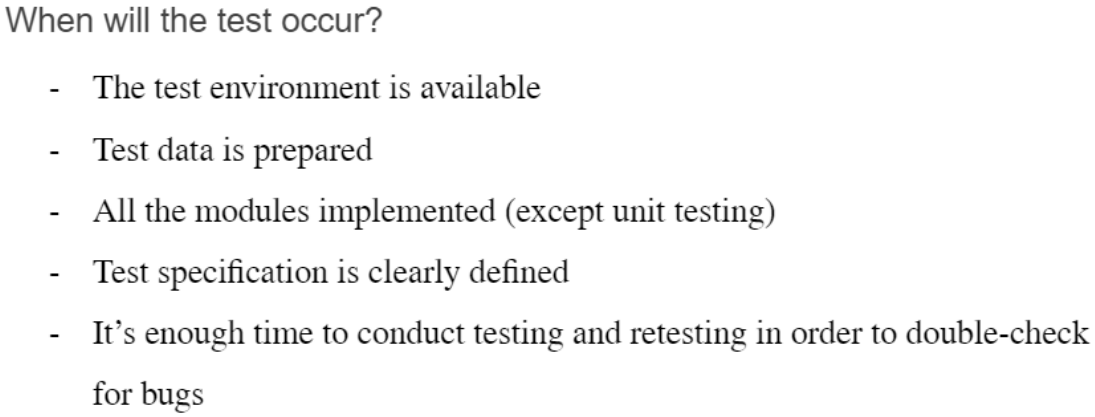

Рисунок 3.4 – Умови початку тестування

3.3 Декомпозиція

Іноді для об’ємних проєктів або задля впевненності у тестуванні, яке усе покриває іноді застосовують декомпозицію. Це такий метод, при якому структуру вебзастосунку розбивають на малі частини. Наприклад, є «шапка сайту», в неї є певні посилання, логотип, може поле пошуку і таке інше. Це робиться задля того, щоб логічно розподілити увесь вебресурс на маленькі шматочки та покрити їх тест-кейсами. Таким чином буде легше прослідити за цим та знайти потрібний модуль, потрібний тест-кейс, адже переглядати багато тест-кейсів, які є у одній купі дуже складно, особливо, коли деякі з них схожі у різних модулях. Наприклад, при написанні і редагуванні статті, вимоги щодо полів будуть майже одні й ті самі.

Багато систем, які дозволяють проводити тест-рани підтримують декомпозицію, наприклад, TestRail, що дійсно полегшує роботу. Загалом декомпозицію ще розподіляють за типами користувачів (залогований, незалогований, адміністратор, преміум та інші). Декомпозиція може бути будь-якого рівня детальності, як буде зручно тестувальникам та які будуть вимоги згідно цього. Приклад декомпозиції наведено на рисунку 3.5.

Рисунок 3.5 – Декомпозиція вебсервісу

3.4 RTM

RTM – ще один артефакт тестування, який допомагає слідкувати за якістю тестування. Зазвичай RTM складається з назви модулю, назви функціоналу, вимог та переліку тест-кейсів, які покривають цей конкретний функціонал.

Загальний вигляд RTM може бути різним в залежності від проєкту, компанії та тестувальників. Розроблена RTM має наступний вигляд, як зображено на рисунку 3.6.

Task	Module	Feature	Requirements	Test Case IDs
CON-11	CON-7	Username	Username required field; should be unique; min 3 symbols; max 40 symbols; should start with the letter; should accept Latin letters and digits; special characters and non-printing symbols are not allowed; no case sensitive.	*T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16*
CON-11	CON-7	Email	Email the format of the email is [name]@[domain].[top-domain]; [name] part accepts Latin letters, digits, and some special characters: plus (+), hyphen (-), underline (_), dot (.) (but not at the beginning and at the end); should be unique; required field.	*T17, T18, T19, T20, T21, T22, T23, T24, T25, T26, T27, T28, T29, T30*
CON-11	CON-7	Password	Password required field; min 8 symbols, max 30 symbols; non-printing symbols are not allowed; should contain at least one special character; should contain at least one digit; should contain at least one capital letter.	*T31, T32, T33, T34, T35, T36, T37, T38, T39, T40, T41, T42, T43, T44, T45*

Рисунок 3.6 – RTM

Отже, RTM допомагає визначити певні модулі, які не покриті тест-кейсами або недостатньо покриті ними. Ще однією перевагою RTM є співставлення функціоналу з тест-кейсами, які його покривають.

Таким чином RTM надає можливість комфортного перегляду актуальної інформації та містить усю необхідну інформацію для аналізу наявного процесу тестування.

Також хочу зауважити, що зазвичай спочатку складається «каркас» таблиці (матриці), в якому не заповнюється останній стовпчик, адже написання тест-кейсів відбувається на наступному етаті STLC. А вже після стадії «Виконання тесту», коли є тест-кейси та було проведено тест-ран, в такому разі можна заповнити RTM актуальними даними.

3.5 Тест-кейси

Наступним логічним етапом є написання тестової документації, згідно якої буде проводитись тестування у подальшому. Існують різні види тестової документації, які мають свої відмінності, переваги та недоліки, тому часто їх поєднують між собою замість того, щоб покривати увесь функціонал лише одним певним типом тестової документації.

Тестова документація потрібна на проєкті, адже вона:

– гарантує якість тестування. На проєктах з багатьма різними модулями, в яких є свій функціонал буде дуже складно тримати у голові ті сценарії, які вже були протестовані, а які ще залишились. Таким чином деяким сценаріям буде приділено більше уваги, аніж іншим, що в кінцевому ітозі може призвести до неякісного тестування програмного застосунку;

– без документації неможливо спланувати час, який знадобиться на тестування. До того ж, якщо на проєкті буде більше за одного тестувальника, як в разі відсутності тестової документації розподілити обов’язки, це буде значно складніше. В такому разі існує ризик неоптимального використання часу, який можна було б витратити на інші речі;

– можна досліджувати прогрес на проєкті. Коли є тестова документація, то зазвичай при тест-рані (коли за тестовою документацію саме відбувається тестування) позначаються вимоги як пройдені, або ні. Таким чином можна побачити стадію, на якій знаходиться команда тестування;

– можна аналізувати модулі та функціонал в якому знаходиться найбільша кількість неправильно працюючого функціоналу або ж багів;

– можна частково або повністю замінити документацію до програмного застосунку. Адже гарна тестова документація дуже часто покриває більшу частину (якщо не всю) функціоналу вебзастосунку.

Існують наступні види тестової документації: тест-кейс, чекліст, use case сценарій, user story.

Тест-кейс – це один з головних артефактів у тестуванні. Зазвичай використовується у якості тестової документації, адже є більш універсальним, а також спеціальні програми, які допомагають організувати тест-рани, як TestRail працюють саме з тест-кейсами.

Отже, тест-кейс може мати в собі різні поля, деякі з них схожі на поля, які застосуються при написанні баг-репорту. Набір цих полів буде залежати від вимог замовника, проєкту, обмежень в часі та через інші можливі причини.

Тест-кейси зазвичай розбивають по модулях, щоб можна було краще орієнтуватися між ними. Загалом, тест-кейс може виглядати як зазначено на рисунку 3.7.

ID	Summary	Priority
	Sign Up Page	
Username		
1	Shouldn't be empty	Critical
2	Should be unique	High
3	Shouldn't accept 2-symbol value	Medium
4	Should accept 3-symbol value	Medium
5	Should accept 7-symbol value	Medium
6	Should accept 40-symbol value	Medium
7	Shouldn't accept 41-symbol value	Medium
8	Should start with a letter	Medium
9	Should accept Latin letters	High
10	Should accept digits	High
11	Shouldn't allow special characters	Medium
12	Shouldn't allow non-printing characters	Medium

Рисунок 3.7 – Тест-кейси

На рисунку 3.7 зображен набір тест-кейсів задля тестового поля для користувацького імені для сторінки реєстрації нового акаунту. Це лише частина тест-кейсів для цього модуля.

Отже, в моїй роботі тест-кейс складається з унікального номеру (id), який потрібен для заповнення RTM та відстеження загальної кількості тест-кейсів, з опису саме того, що повинно перевірятись у певному тест-кейсі та

пріорітет цього функціоналу, або ж «Як важливо, щоб цей функціонал працював?».

Відповідно до проєкту та команди розробки у тест-кейсі можуть бути наявні наступні поля:

– шляхи до відтворення – це інструкції, які покроково та детально описують як саме буде відбуватися тест, щоб інша людина (тестувальник) змогла повторити цей процес та отримати той самий результат;

– тестові дані – це набір значень, які потрібно задля проведення тесту. Наприклад, якщо потрібно перевірити функціонал додання товару у корзину, то для цього потрібно увійти до системи вебзастосунку та зазвичай у тестувальників вже є акаунти з різними правами, які створенні саме для цілей тестування. Тому в таких випадках у тестових данних зазвичай залишають логін та пароль від потрібного користувача;

– очікуванний результат – це опис того, на що ми розраховуємо після проведення цього певного тесту. Наприклад, після додання товару в корзину ми очікуємо що в корзині цей товар з’явиться;

– результат виконання тест-кейсу – відбувається після тестування за тестовою документацією. Може мати наступні стани: Passed, Failed, Blocked, Skipped (Deferred);

– передумови – це додаткові інструції, які потрібні для виконання тесту. Зазвичай робляться, щоб не перенавантажувати шляхи до відтворення та містять у собі інформацію по типу «Користувач потрібен бути у системі», «Користувач повинен мати 5 товарів у корзині». Тобто специфічні вимоги до проведення тестування за тест-кейсом, які є необхідними для 100% відтворення, щоб отримати той самий результат, що отримав тестувальник;

– вкладення – це додаткові файлі або інформація, яка потрібна для проведення тестування. Наприклад, якщо тестується функціонал зміни аватарки у профілі користувача, то в такому разі у вкладеннях тестувальник може залишити файл, який буде використовуватись для проведення цього тесту;

– стан автоматизації – це поле, в якому команда тестувальників планує та міркує який функціонал потрібно автоматизувати, а який краще ні. Автоматизація економить багато часу та є корисним підходом до тестування [28, 29]. Зазвичай автоматизуються ті тести, які проводяться частіше всього, а це smoke-тести (тести, які відносяться до найбільш критичного функціоналу та є позитивними). Стани можуть бути наступні: «автоматизовано», «кандидат на автоматизацію» та «автоматизація не планується»;

– тип тест-кейса – це тип тесту (GUI, Performance, Security, Smoke, тощо).

Також іноді пріорітет може поділятись на два поля – це пріорітет та жорстокість (severity). В такому разі пріорітет відповідає на питання «Наскільки погано буде для програмного застосунку, якщо певний функціонал не буде працювати з бізнес точки зору?», а жорстокість на питання «Наскільки погано буде для програмного застосунку, якщо певний функціонал не буде працювати з технічної точки зору?».

Наприклад, якщо взяти Укрпошту та її марки, присвячені до значних подій протягом війни. Багато людей хоче придбати ці марки та тому сервери можуть не витримувати таке навантаження, адже в звичайний день набагато меньше людей користуються послугами Укрпошти. Тому, якщо впаде сервер – то це дійсно погано, це означає, що ніхто не зможе скористуватися послугами Укрпошти. В такому разі severity буде максимальним, а ось пріорітет буде низьким. Тому що придбання більшої кількості серверів або потужніших допоможе вирішити цю проблему, але як зазначалось, це не буде рентабельним, цей ажіотаж відбувається лише на певні події, які не є такими частими. В звичайний день сервери Укрпошти добре справляються із поставленим завданням.

Також варто зазначити, що відповідно до різних компаній та проєктів можуть бути наявні різні можливі значення цих полів. У моїй роботі були використані наступні значення для priority поля: Critical, High, Medium та

Low. На деяких проєктах замість Critical може бути Highest та також може бути додатково наявним значення Lowest і таке інше.

Як вже було зазначено, іноді ці два показники (priority та severity) поєднують у один і наче комбінують вплив цього функціоналу на систему. Загалом діаграма, на якій зображені можливі приклади щодо комбінації severity та priority зображені на рисунку 3.8.

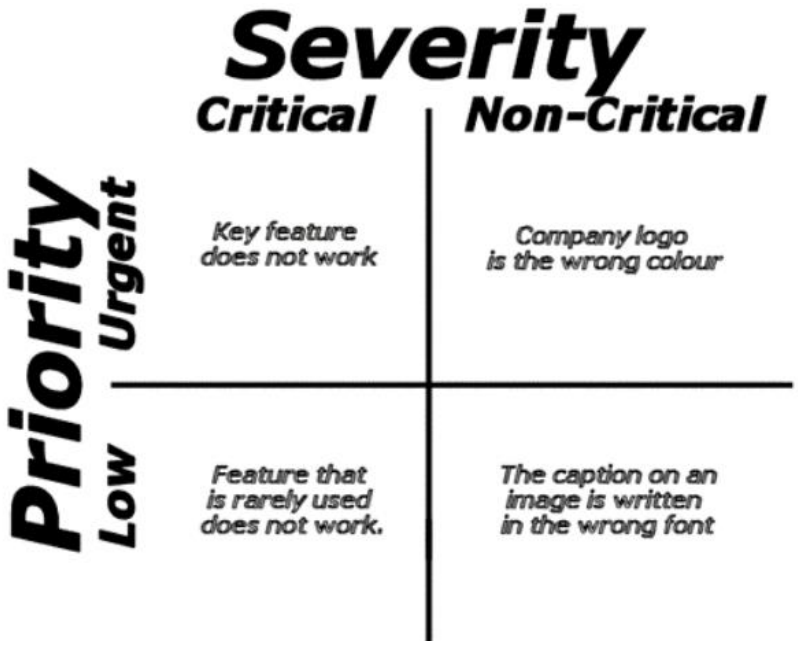

Рисунок 3.8 – Severity vs Priority

3.6 Виконання тест-кейсів (Test run)

Після складання тестової документації та наявності розробленого функціоналу відбувається виконання тест-кейсів (тобто безпосередньо сам процес тестування).

Результат виконання цього етапу потрібно десь зберігати, адже на фоні саме цієї інформації буде формуватися звіт з тестування та майже усі метрики для його наповнення беруться саме з цього етапу. Тому я модифікував таблицю з минулого кроку та тепер вона має такий вигляд, як зазначено на рисунку 3.9.

ID	Summary	Status	
	Sign Up Page		
Username			
1	Shouldn't be empty	Passed	
2	Should be unique	Passed	
3	Shouldn't accept 2-symbol value	Failed	*CON-35*
4	Should accept 3-symbol value	Passed	
5	Should accept 7-symbol value	Passed	
6	Should accept 40-symbol value	Passed	
7	Shouldn't accept 41-symbol value	Failed	*CON-36*
8	Should start with a letter	Passed	
9	Should accept Latin letters	Passed	
10	Should accept digits	Passed	
11	Shouldn't allow special characters	Failed	*CON-37*
12	Shouldn't allow non-printing characters	Passed	

Рисунок 3.9 – Результат виконання тест рану

Таким чином, документ має номер та опис, з нових полів було додано результат виконання певного тест-кейсу (результат може бути одним з наступних значень: Passed, Failed, Blocked та Deferred). Також додатково було додано стовпець-посилання на баг-репорт у JIRA лише у тих тест-кейсів, які не виконались з очікуваним результатом або виконання яких було заблоковано через невірно працюючий функціонал.

3.7 Написання баг-репортів

Паралельно з минулим етапом починаються створюватися баг-репорти на ті тест-кейси, які не проходять тестування та у кінцевому ітозі актуальний результат не відповідає очікуваному. Це краще робити паралельно, адже таким чином знижується ризик того, що якийсь баг загубиться та таким чином не прийдеться ще раз відтворювати тест задля того, щоб залишити усю необхідну інформацію для команди розробки зі зрозумілою інформацією щодо проблеми, що сталася. Саме для цієї потреби й існують баг-репорти. Це такий тип документації, який потрібен команді з ціллю ознайомлення з багом та подальшим його усуненням.

Баг-репорти відповідно до проєкту та компанії можуть відрізнятись, мати якісь додаткові поля або навпаки не мати якихось. Взагалі баг-репорт складається з:

– унікального номеру (коду) задля відстеження проблеми;

– опису проблеми – найголовніше поле, гарний баг-репорт – той, де лише по одному опису зрозуміло в чому полягає проблеми та не має потреби витрачати час та переглядати інші поля;

– шляхи до відтворення – то, як тестувальник знайшов цей баг, які кроки ми робили для цього, кроки потрібні бути точними та зрозумілими, щоб та людина, яка відтворює цей баг отримала саме той результат, який отримав тестувальник;

– очікуваний результат – то, на що тестувальник розрахував згідно до вимог, саме ця дія потрібна виконатись наприкінці тест-кейсу;

– актуальний результат – то, що тестувальник побачив після виконання тест-кейсу, реальний результат;

– пріоритет – показник важливості усунення багу («Наскільки швидко це потрібно зробити?»);

– ПЗ – на якому ПЗ було проведено тестування. Якщо це вебзастосунок, то важливо взяти до уваги браузер, його версію. В залежності від завдання можна також залишати у цьому полі версію застосунку, операційну версію та її версію і таке інше;

– додатки – поле, де тестувальник може залишити логи, які були набуті протягом тестування, посилання додаткові, коментарі, фотоматеріали або відеоматеріали.

В роботі були використані саме наведені вище поля, адже це є допустимий мінімум для того, щоб скласти гарний та змістовний баг-репорт.

Приклад баг-репорту зображено на рисунках 3.10 та 3.11. Даний баг стосується можливості користувачем створити акаунт, який містить у якості імені користувача значення, яке складається лише з двох символів, що не потрібно бути дозволено згідно з вимогами.

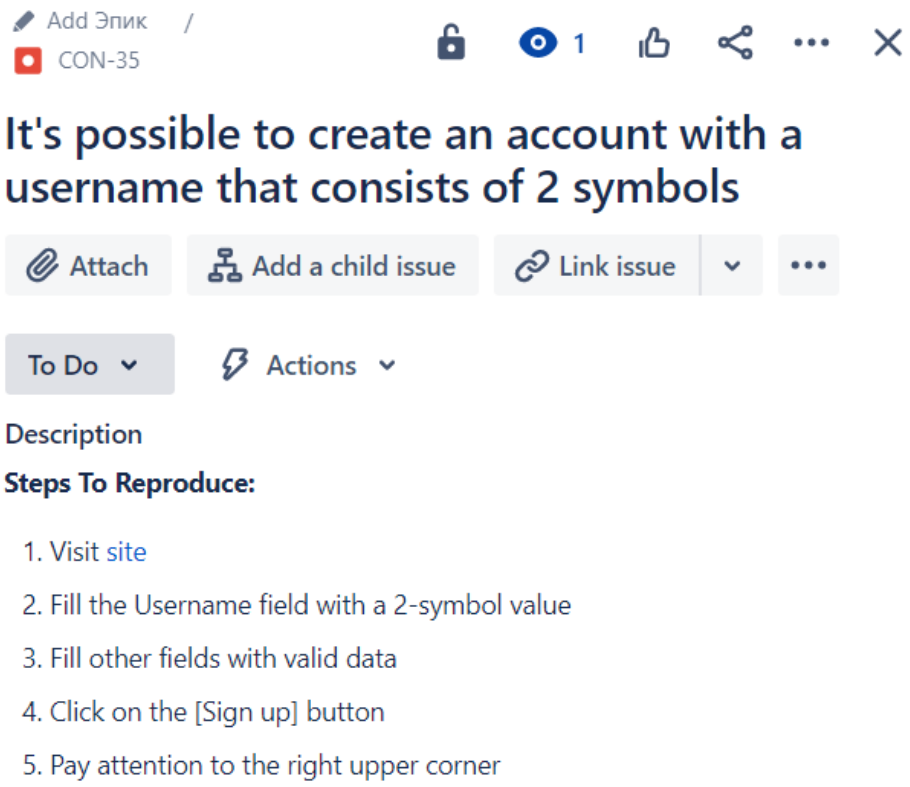

Рисунок 3.10 – Баг-репорт

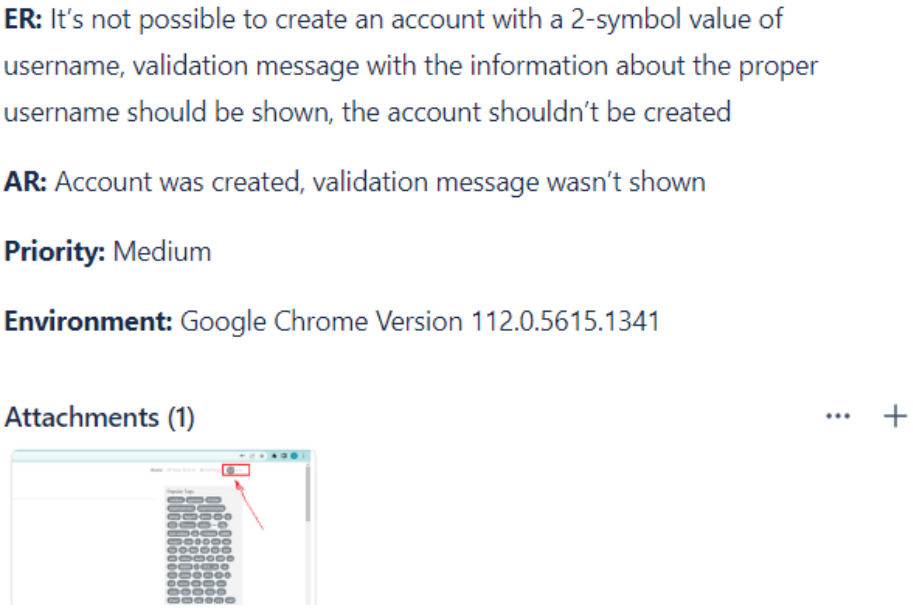

Рисунок 3.11 – Баг-репорт (продовження)

У Attachments знаходиться скріншот з візуалізацією проблеми, можна було також залишити посилання на відео або взагалі залишити це поле порожнім, адже баг не є складним у відтворенні.

3.8 Звіт з тестування (Test report)

На основі актуальних результатів тестування формується звіт з тестування, в якому відображено загальний висновок щодо процесу тестування та його результату. Зазвичай в цьому документі багато візуальної інформації, адже документ такого роду складається здебільшо задля замовників та інформація, яка подана у ньому потрібна бути позбута надмірної кількості термінів, наукових концептів, тобто потрібна бути зрозумілою нетехнічній людині.

Зазвичай за результатами презентації цього документу відбувається рішення про успішність тестування, про потреби внесення певних корективів, повторного проведення тестування (після усунення багів, ще відомо як ретестінг).

Деякі з метрик, отримані за результатами тестування зображені на рисунках 3.12 та 3.13.

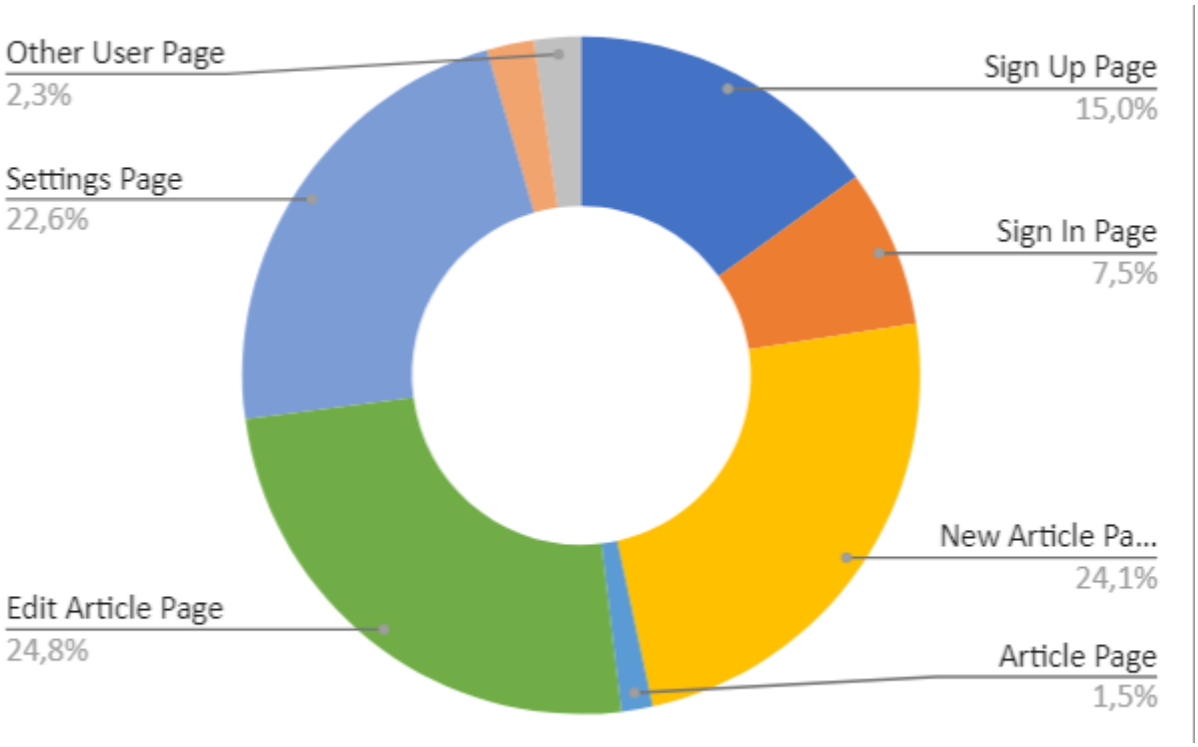

Рисунок 3.12 – Знайдені баги за сторінками

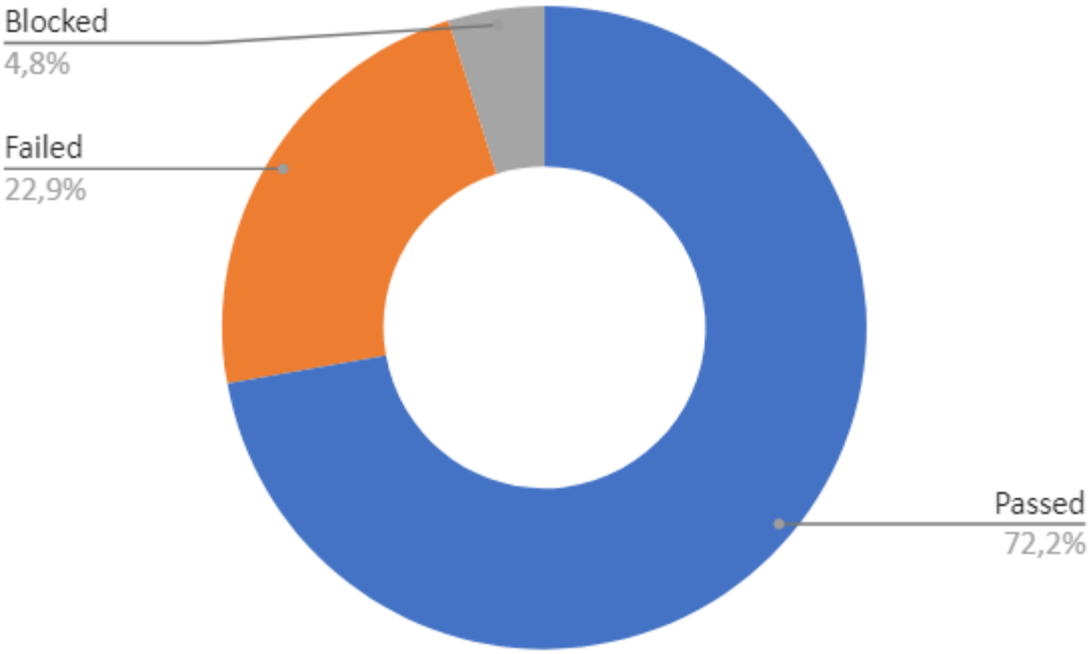

Рисунок 3.13 – Загальний результат тестування

Згідно до отриманих результатів були сформовані наступні висновки. Ці висновки формуються на умовах, які були встановлені замовником та командою тестування на початку проєкту. Отож, умови:

– провести повне тестування (перевірити усі тест-кейси), тобто немає жодного тест-кейсу зі статусом «Deferred» або ж «Відкладено»;

– показник не менше аніж 92% для пройдених тестів;

– усі тести з критичним пріоритетом працюють згідно до вимог.

Дані, які були отримано протягом тестування свідчать про наступне:

– було протестовано 100% тестів;

– 72% від загальної кількості тест-кейсів працюють відповідно до вимог;

– 5 критичних тестів не працюють, як очікувалось.

Таким чином можна вважати, що програмний застосунок містить у собі багато багів, які потребують усунення. Рекомендовано провести баг-фікс та ретестінг протягом наступних двох тижнів.

3.9 API тестування

API тестування відрізняється від мануального тестування за допомогою UI. Воно стає у нагоді, коли API вже запрограмоване, але ще немає UI, на якому можна провести тестування. В такому разі розумно провести саме тестування, посилаючи API запити на сервер та тестуючи їх відповідь.

В моєму випадку було використано Postman. Було реалізовано різні запити з різними методами на різні ендпоінти. Були реалізовані наступні тести:

– POST (реєстрація, логін, створення статті та створення коментаря);
– DELETE (видалення статті та видалення коментаря);
– PUT (зміна користувацького імені та заголовку статті).

Варто зауважити, що у випадку PUT більше підійшло б використання HTTP методу PATCH, адже саме він відповідає за часткову зміну об’єкту, в ту чергу як PUT відповідає за повне перезаписання об’єкту.

Запити були розроблені таким чином, що вони є незалежними від інших запитів. Це стає у нагоді, коли потрібно перевірити певний функціонал і не запускати при цьому увесь набір тестів, який займе багато часу. Також, як вже зазначалось, Postman має асинхронну поведінку виконання коду, а ще й JS вставки, які були використанні задля написання реалізації запитів. Тому потрібно було використовувати Promise.

Лістинг 3.1 Реалізація методу на реєстрацію за допомогою Postman:

```
postman.setGlobalVariable("register", () => {
    return new Promise(function(resolve, reject) {
        const url = pm.globals.get("url");
        const firstName = pm.variables.replaceIn('{{$randomFirstName}}').toLower
Case();
        const rnd = Math.random().toString().slice(2, 6);
        const userName = firstName + rnd;
        const email = userName + '@gmail.com';
        console.log(userName);

        const registrationRequest = {
            url: url + '/api/users',
```

```
            method: 'POST',
            header: {
                'Content-Type': 'application/json',
            },
            body: {
                mode: "raw",
                raw: JSON.stringify({
                    "user": {
                        "username": userName,
                        "email": email,
                        "password": "Test1234!"
                    }
                })
            }
        }

        pm.sendRequest(registrationRequest, function (err, response) {
            if (err) {
                reject(err);
                return;
            }

            console.log(response.json());

            pm.collectionVariables.set("email", response.json().user.email);
            pm.collectionVariables.set("token", response.json().user.token);

            resolve(response);
        });
    });
})
```

Сам запит на реєстрацію має наступний вигляд, як зображено на рисунку 3.14.

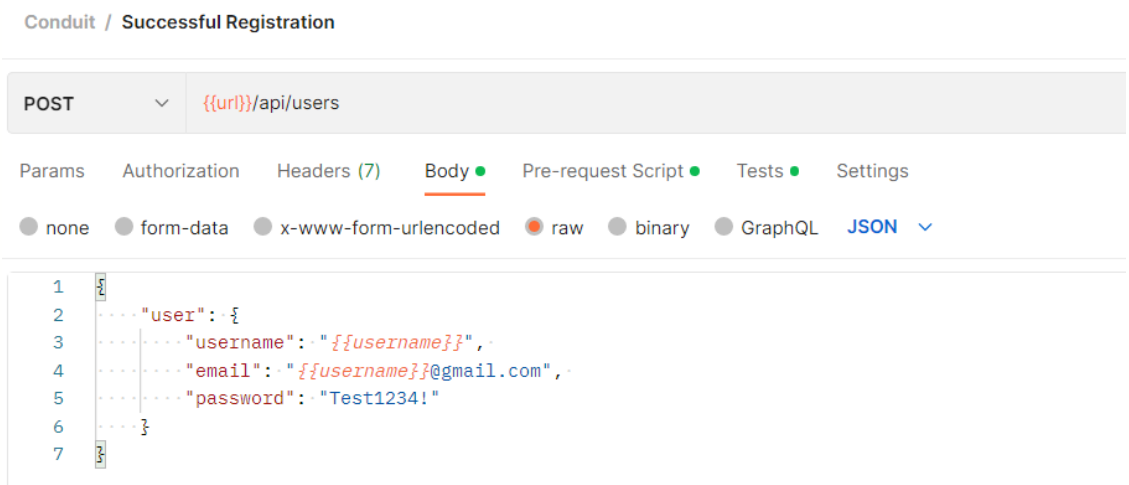

Рисунок 3.14 – API запит на реєстрацію

Також варто пам’ятати, що токени, про які було згадано раніше у роботі також необхідно надсилати разом з запитами, які вимагають цього (наприклад, створення статті або коментаря).

Результат виконання запиту на реєстрацію зображено на рисунку 3.15.

Рисунок 3.15 – Результат API запиту на реєстрацію

Загальний вигляд виконання тестів в усій колекції виглядає наступним чином, як зазначено на рисунку 3.16.

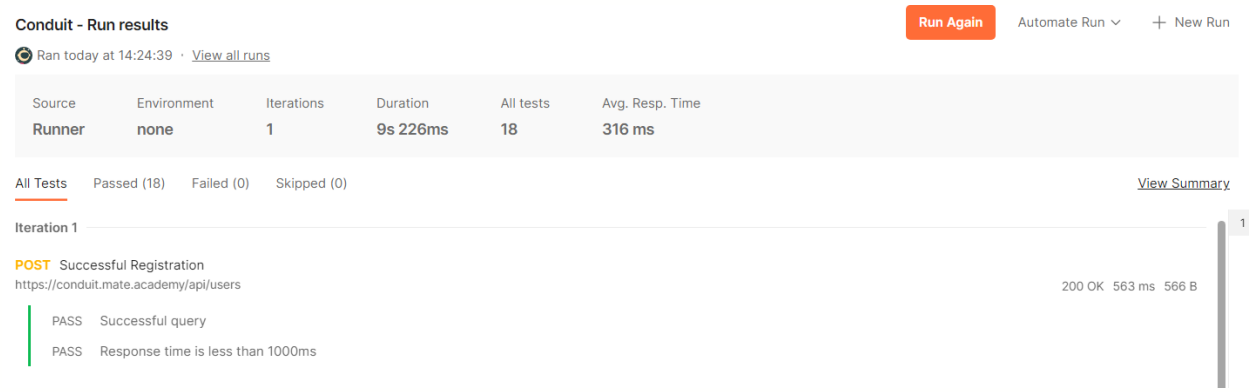

Рисунок 3.16 – Загальний результат прогону Postman колекції

3.10 E2E тестування

Як вже було зазначено, E2E тестування – це імітація дій кінцевих користувачів. Наприклад, створення статті або зміна паролю. Тобто все те, що може зробити користувач під час користування ПЗ. E2E тестування є

автоматизованим, був обран Cypress задля реалізації цього завдання, адже він має інтуїтивно зрозумілий інтерфейс та має гарний функціонал задля автоматизації тестування.

Було розроблено 19 автоматизованих тест-кейсів, які відносяться до різних сторінок.

Лістинг 3.2 Приклад автоматизованого E2E тесту:

```
it('should have links', () => {
    // get all the links in the header and assert their number
    home.assertNumberOfLinks();

    // compare each link that are stored in 'links' with expected links
    home.assertContentOfLinks();
})
```

Було використано Page Object методологію розробки та організування автоматизованих тестів. Ця модель стає у нагоді, адже має за основу принципи ООП. Такти чином тести виглядають «чистими» та легко читаються, а також їх легше підтримувати.

Лістинг 3.3 Реалізація методів у класі *HomePageObject:*

```
expectedLinks = ['conduit', 'Home', 'Sign in', 'Sign up']

get links() {
    return cy.findByClassName('navbar').find('a');
  }
assertNumberOfLinks() {
    cy.allure().startStep("Assert only 4 links are displayed in the
header")
      this.links.as('links')

      cy.get('@links')
        .should('have.length', this.expectedLinks.length);
    cy.allure().endStep()
  }

  assertContentOfLinks() {
    cy.allure().startStep("Assert 'conduit', 'Home', 'Sign in', 'Sign up'
links are present in the header")
      cy.get('@links')
        .each((link, index) => {
```

```
            cy.wrap(link)
              .invoke('text')
              .then((linkText) => {
                const expectedLinkText = this.expectedLinks[index];
                expect(linkText).to.equal(expectedLinkText);
              })
          })
      cy.allure().endStep()
    }
```

Таким чином відбувається перевірка наявності посилань на головній сторінці та перевірці їх контенту. На рисунку 3.17 зображено які самі посилання перевіряються у цьому тестів.

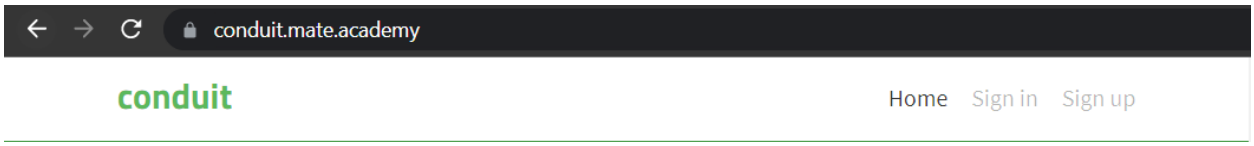

Рисунок 3.17 – Посилання, що перевіряються у тестів

Також протягом роботи з Cypress були розроблені кастомні функції, які допомогали у розробці.

Лістинг 3.4 Реалізація кастомної функції для пошуку елементу на сторінці за впровадженним класом:

```
Cypress.Commands.add('findByClassName', (className) => {
  cy.get(`.${className}`);
})
```

Також, при розробці було впроваджено незалежність до кожного тесту, що робить їх більш зручними у використанні. Адже таким чином можна протестувати певний функціонал і при цьому не запускати інші тести.

При роботі з Cypress були такі ж проблеми, як і у Postman, а саме асинхронність виконання коду. Таким чином запити могли надсилатись у довільній послідовності або відповідь на запит могла ще не прийти до моменту, коли це було потрібно, саме тому були використані Promise.

Лістинг 3.5 Реалізація кастомної команди на логін користувача із застосуванням Promise та API запитів:

```
Cypress.Commands.add('login', (user) => {
  cy.registerNewUser(user).then((response) => {
    cy.request('POST', 'api/users/login', {
      user: {
        email: response.body.user.email,
        password: user.password
      }
    })
  })
  .then((res) => {
    Cypress.env('auth',  res.body.user.token);
  })
})
```

Одна з останніх стрічок код отримує з відповіді серверу токен та записує його у змінну, адже після авторизація з кожним запитом потрібно посилати токен авторизації, щоб дати серверу зрозуміти хто саме робить запит.

Лістинг 3.6 Додання токену авторизації у заголовок авторизації для кожного запиту:

```
cy.intercept('*', (req) => {
    req.headers['Authorization'] = `Token ${Cypress.env('auth')}`;
});
```

Також для формування тестових даних була використана бібліотека faker, яка дозволяє генерувати випадкові.

Лістинг 3.7 Генерація статті за допомогою faker:

```
function generateArticle() {
  return {
    title: faker.random.word(),
    description: faker.random.word(),
    body: faker.random.word()
  };
}
```

Cypress має дуже гарний функціонал, який виконує авто-тести у браузері, можна звернутися до кожного кроку та подивитись, що відбувалось саме на ньому. «Лента» з ходом виконання авто-тесту виглядає так, як зазначено на рисунку 3.18.

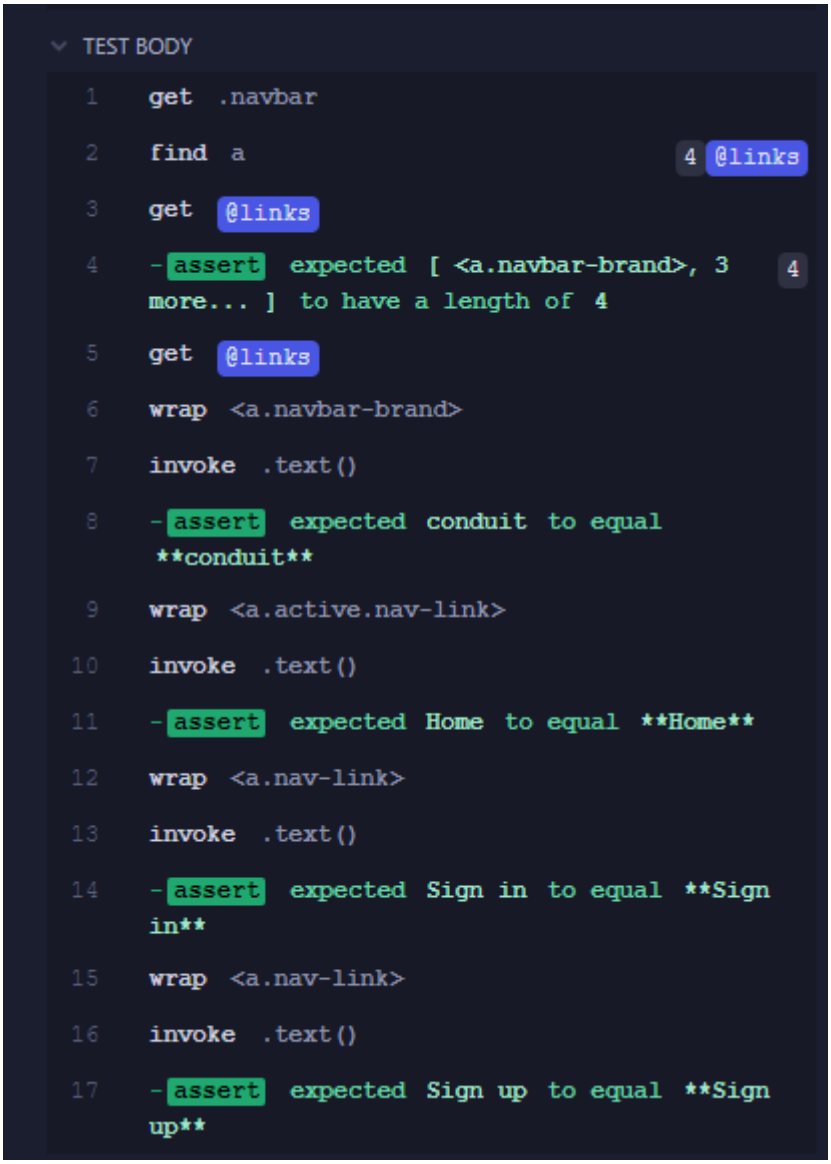

Рисунок 3.18 – Cypress feed

Загалом було розроблено 19 авто-тестів, які розподілені на 7 модулів:

– сторінка реєстрації;
– сторінка логіну;
– сторінка налаштувань;
– сторінка створення статті;

– сторінка редактування статті;
– домашня сторінка;
– сторінка статті.

Cypress можна запустити у headless режимі, тобто у консолі, браузер не буде відкрито, таким чином можна побачити звіт щодо усіх тестів, які були включені у тестування. Результат цієї команди виглядає, як зазначено на рисунку 3.19.

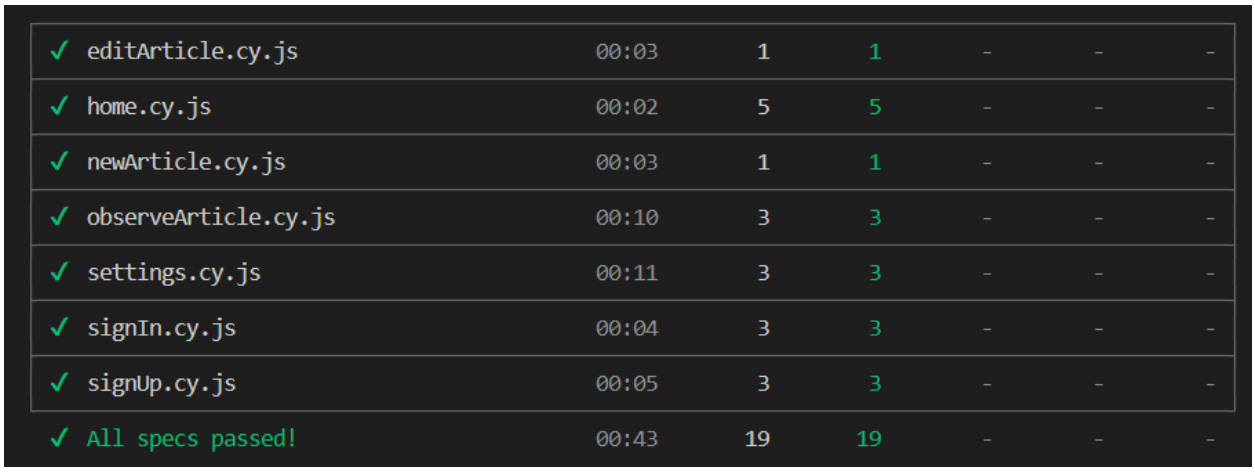

Рисунок 3.19 – Cypress test run

Також важливою функцією Cypress є можливість додавання власних тегів (marks) на тести, які можуть об’єднувати певну групу тестів та запускати лише ці тести при написанні певних команд у терміналі. Це стає у нагоді, наприклад, для розподілення тестів по категоріям таким як smoke, regression [30] та можливо flaky (нестабільний тест).

Результати тестування потрібно десь зберігати, зазвичай компанії дбають про це та купують програми, однією з таких програм є Allure, яка може у облаці зберігати декомпозицію, тести, історію їх прогону, розподіляти тести на категорію (автоматизовані, нестабільні і таке інше). Тож було вирішено скористатися Allure, який локально працює на моєму комп’ютері та зберігає результати саме одного прогону авто-тестів.

Allure легкий у інтеграції та має гарний та user-friendly інтерфейс. Він також має можливість перегляду ходу авто-тестів, тобто щось на кшталт «Шагів до відтворення», якщо казати про тест-кейси або баг-репорти. Цей інструмент пропонує можливість у створення власних кроків, надання їм змістовних назв, що робить Allure звіти легкими для читання як технічними так і нетехнічними людьми.

Загальний вигляд Allure звіту має наступний вигляд, як зображено на рисунку 3.20.

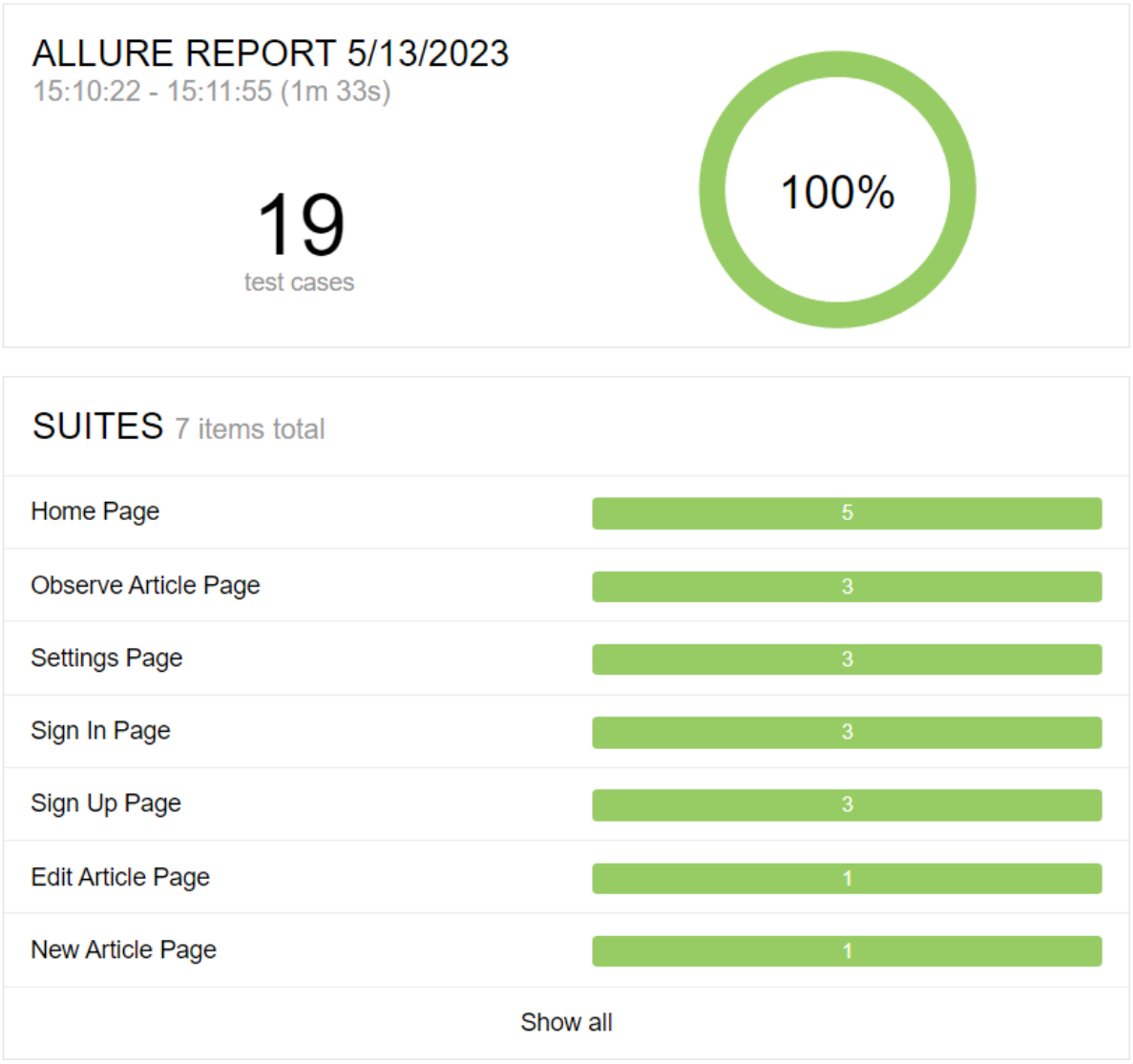

Рисунок 3.20 – Allure звіт

Як вже було зазначено, Allure дозволяє створювати декомпозицію та надавати логічний опис кожному кроку, який відбувається в авто-тесті з ціллю, щоб «перекласти» його для мануального тестувальника.

Лістинг 3.8 Створення кроку у Allure:

```
cy.allure()
    .startStep('Click on the [Publish Article] button and assert the url')
      this.publishArticleBtn.click();
cy.allure().endStep()
```

Лістинг 3.9 Створення кроку декомпозиції:

```
before(() => {
    cy.allure()
      .feature('Edit Article Page')
})
```

Тест-кейс, а саме ход його виконання зображений на рисунку 3.21.

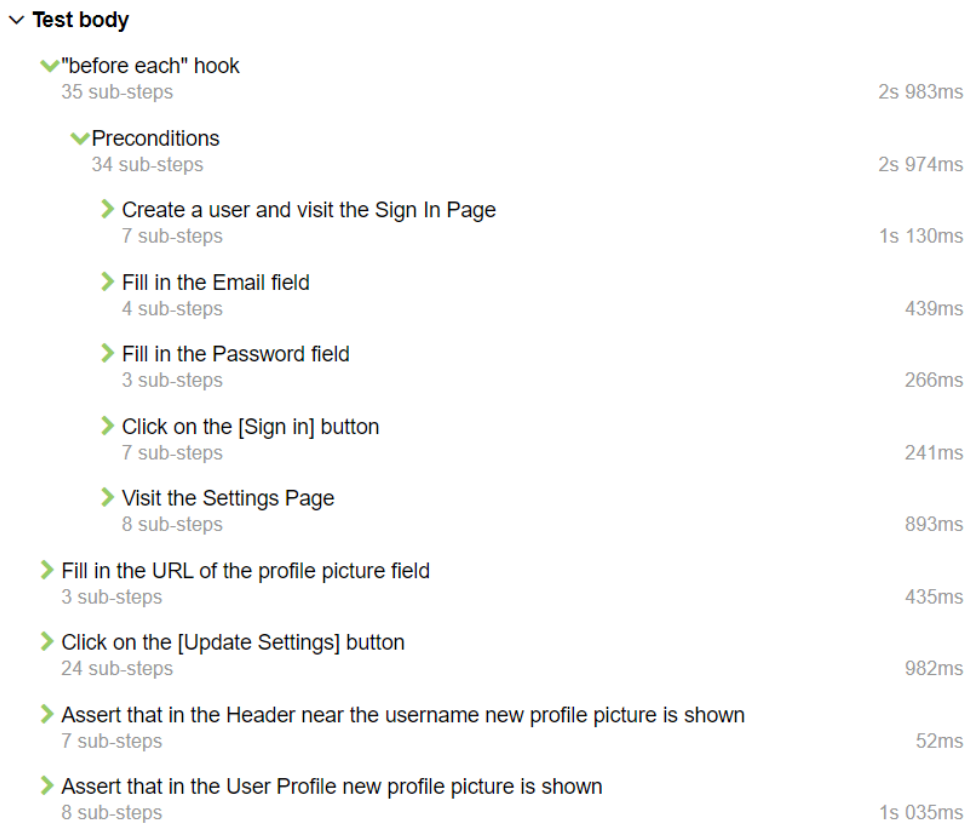

Рисунок 3.21 – Тест-кейс у Allure

3.11 Тестування безпеки (XSS)

Як вже було зазначено, XSS з’являється там, де є користувацький ввод даних, який ніяким чином не валідується перед відправкою на сервер. Тобто завданням було перевірити усі текстові inputs на різні можливі XSS ін’єкції, адже розробники могли занести лише певні слова у «чорний список», не згадавши про інші. Тестування відбувалося за допомогою простих ін’єкцій, які просто виводили певне повідомлення у модальному вікні або ж cookies користувача – це так званий PoC.

Процес тестування безпеки зображено на рисунках 3.22 та 3.23.

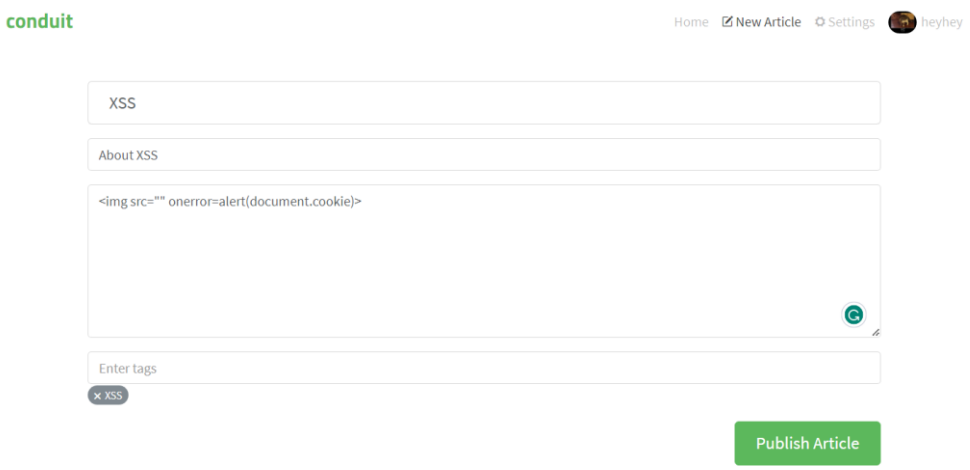

Рисунок 3.22 – Створення статті з XSS

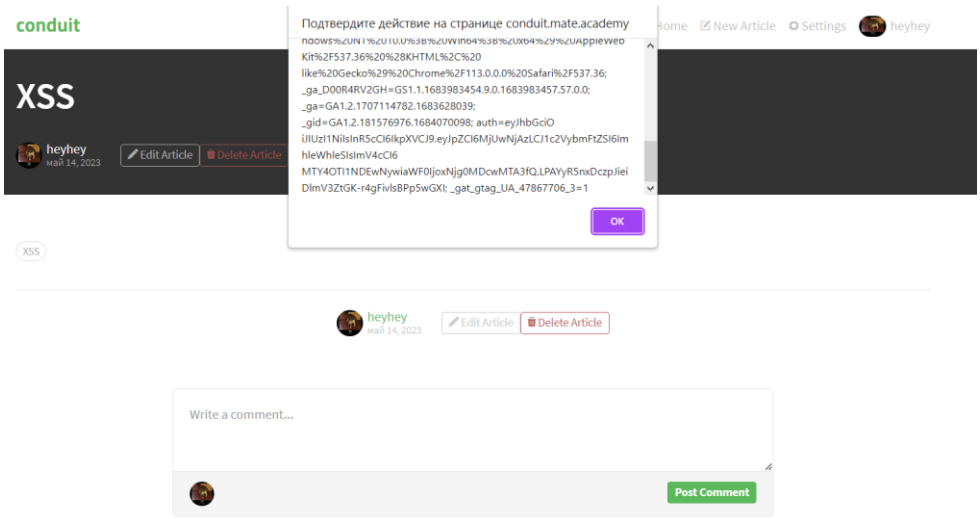

Рисунок 3.23 – Результат XSS-ін’єкції

Таким чином впродовж тестування безпеки була знайдена XSS вразливість. Вона була помічена при створенні статті, де поле, яке відповідає саме за контент статті не було відфільтроване та введений HTML-скріпт був відправлений до серверу та додано на сторінку.

Тепер будь-який користувач, який відвідає сторінку цієї статті побаче таке модальне вікно, що буде свідчити про те, що даний користувач вражений та його дані потенційно могли бути викрадені [31]. Хакер не буде показувати свою присутність та додавати певні модальні вікна, скоріше за все у шкідливому коді буде функціонал, який дістане cookies користувача та надішле токен до поштової скриньки, вказаної хакером.

Тобто такі вразливості можуть залишатись впродовж довгого часу непоміченими. Варто підкреслити, що володіння токеном авторизації – це те саме, що й мати логін та пароль від акаунту користувача, тому ця атака дуже руйнівна по своїй природі та несе багато шкоди.

Таким чином отримавши токен авторизації хакер може вдавати іншу людину та робити усі дії, які дозволено у програмному застосунку від імені своїх жертв.

Варто зазначити, що в даному програмному застосунку, отримавши токен авторизації іншого користувача, хакер може від його імені створювати статті (наприклад, таким чином хакер може вдавати якусь впливову людину та створити статтю з рекламою якихось грошових схем або криптовалют, таким чином обдурити багато людей, які прислухаються до думки впливової людини) або отримати пошту користувача та надсилати фішингові повідомлення саме цьому користувачу або просто змінити пошту чи пароль, тим самим обмежити доступ користувачу до власного профілю.

ВИСНОВКИ

У рамках кваліфікаційної роботи був розроблений і реалізований метод тестування вебресурсу «Conduit». У ході роботи було проведено повний процес STLC від отримання вимог до написання звіту з тестування, а саме: аналіз вимог, отриманих від замовника та їх покращення, написання тест плану, розроблення тестової стратегії, планування застосування різних технік тестування та типів тестів, створення RTM, декомпозиції та тестової документації, перевірка функціоналу за розробленими тест-кейсами, оформлення баг-репортів, проведення тестування безпеки (XSS). Також були оформлені звіти як з тестування безпеки, так і з тестування у цілому.

Здебільшо тестування виконувалось мануальним чином, але також був розроблений функціонал для тестування API за допомогою Postman та e2e тестів за допомогою Cypress. За допомогою Allure та можливостей Postman були сформовані звіти з результатами тестування.

При виконанні роботи було проведено мануальне тестування на 475 тест-кейсах, API тестування на 8 API запитах та e2e тестування на 19 авто-тестів. Також було знайдено 109 багів та 23 тестів, виконання яких було заблоковано через баги, уся детальна інформація щодо них зберігається у JIRA.

У ході роботи було розглянуто та набуто фундаментальні знання з багатьох тематик та технік, серед яких: How the Web Works, мануальне тестування, автоматизоване тестування, тестування API, тестові типи, види тестування, тестова документація, баг-репорти, токени та promises, DOM, тестування безпеки та XSS.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Tvoroshenko, I. S., & Kuznetsov, M. (2021). About the role of testing in process of mobile application development.

2. Творошенко, І. С. (2021). Технології прийняття рішень в інформаційних системах: навч. посібник. *Харків: ХНУРЕ*.

3. What Is Software Testing? All The Basics You Need to Know. URL: https://www.testim.io/blog/software-testing-basics/ (дата звернення 11.04.2023).

4. Software Testing Life Cycle (STLC) – GeeksForGeeks. URL: https://www.geeksforgeeks.org/software-testing-life-cycle-stlc/ (дата звернення 13.04.2023).

5. Iryna, T., & Maksym, K. (2021). Research results of functional, white box and smoke testing methods for mobile applications. *Trends in science and practice of today*, *5*, 418.

6. Tvoroshenko, I., & Mahomet, A. (2021). About classification of the methods in design of medical information systems.

7. Test plan – Wikipedia. URL: https://en.wikipedia.org/wiki/Test_plan (дата звернення 14.04.2023).

8. Test case – Wikipedia. URL: https://en.wikipedia.org/wiki/Test_case (дата звернення 14.04.2023).

9. 7 Principles of Software Testing with Examples – Guru99. URL: https://www.guru99.com/software-testing-seven-principles.html (дата звернення 16.04.2023).

10. Ibrahim, D. Y., Gorokhovatskyi, V., Tvoroshenko, I., & Zeghid, M. (2022). Cluster representation of the structural description of images for effective classification.

11. Agile vs. Waterfall Methodology – Forbes Advisor. URL: https://www.forbes.com/advisor/business/agile-vs-waterfall-methodology/ (дата звернення 17.04.2023).

12. Cherednichenko, O., Vovk, M., Yanholenko, O., & Yakovleva, O. (2020). Towards the technology of employers' requirements collection development. In *Integrated Computer Technologies in Mechanical Engineering: Synergetic Engineering* (pp. 228-239). Cham: Springer International Publishing.

13. Internet – Wikipedia. URL: https://en.wikipedia.org/wiki/Internet (дата звернення 22.04.2023).

14. What is a Server? Definition from WhatIs.com – TechTarget. URL: https://www.techtarget.com/whatis/definition/server (дата звернення 22.04.2023).

15. Gorokhovatskiy, V. A., Vechirska, I. D., & Chetverikov, G. G. (2016). Method for building of logical data transform in the problem of establishing links between the objects in intellectual telecommunication systems. *Telecommunications and Radio Engineering*, *75*(18).

16. API: Definition and application in procurement. URL: https://www.manutan.com/blog/en/glossary/api-definition-and-application-in-procurement (дата звернення 22.04.2023).

17. What Is a REST API? – SitePoint. URL: https://www.sitepoint.com/rest-api/ (дата звернення 24.04.2023).

18. Daradkeh, Y. I., Gorokhovatskyi, V., Tvoroshenko, I., & Zeghid, M. (2022). Tools for Fast Metric Data Search in Structural Methods for Image Classification. *IEEE Access*, *10*, 124738-124746.

19. HTTP Status Codes – REST API Tutorial. URL: https://restfulapi.net/http-status-codes/ (дата звернення 27.04.2023).

20. Tvoroshenko, I., & Andrieieva, A. (2021). Development of web applications for remote learning of English.

21. Authentication vs. Authorization – Okta. URL: https://www.okta.com/identity-101/authentication-vs-authorization/ (дата звернення 29.04.2023).

22. What Is Token-Based Authentication. Types, Pros and Cons. URL: https://www.wallarm.com/what/token-based-authentication (дата звернення 29.04.2023).

23. Document Object Model – Wikipedia. URL: https://en.wikipedia.org/wiki/Document_Object_Model (дата звернення 01.05.2023).

24. OWASP Top Ten. URL: https://owasp.org/www-project-top-ten/ (дата звернення 04.05.2023).

25. Promise – JavaScript – MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (дата звернення 05.05.2023).

26. Творошенко, І. С. (2019). Особливості застосування інструментальних засобів під час інтелектуальної обробки інформації.

27. Вечірська, І. Д. (2013). Аналіз методу побудови та принципів роботи реляційної мережі як багаторівневої структури паралельної дії.

28. Tvoroshenko, I. S., & Maksimenko, H. (2021). To the question of analysis of existing mechanisms of web application testing.

29. Вечірська, І., Кобилін, О., Прокоп'єв, С., Вечірська, А., & Кучеренко, М. (2022). BUILDING A LOGICAL NETWORK FOR SOLVING THE PROBLEM OF CAR RENTAL BY MEANS ALGEBRA OF FINITE PREDICATES. *Computer systems and information technologies*, (2), 78-87.

30. Iryna, T., & Heorhii, M. (2021). Research of regression and modular testing of web applications. *Editorial Board*, 406.

31. Yanholenko, O., Cherednichenko, O., Yakovleva, O., & Arkatov, D. (2020). A Model for Estimating the Security Level of Mobile Applications: a Fuzzy Logic Approach. In *IntelITSIS* (pp. 252-266).