Міністерство освіти і науки України
Університет митної справи та фінансів

Факультет інноваційних технологій

Кафедра комп’ютерних наук та інженерії програмного забезпечення

Кваліфікаційна робота магістра

на тему «Дослідження обмежень автоматизації тестування»

Виконав: студент групи К22-1м

Спеціальність 122 Комп’ютерні науки

Загнієнко Владислав Андрійович
(прізвище та ініціали)

Керівник к.ф.-м.н., доц. Фірсов О.Д.
(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент Університет митної справи та фінансів
(місце роботи)

В.о. завідувача кафедри Кібербезпеки та інформаційних технологій
(посада)

к.т.н., доц. Прокопович-Ткаченко Д.І.
(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2024

АНОТАЦІЯ

Загнієнко В.А. Дослідження обмежень автоматизації тестування.

Кваліфікаційна робота на здобуття освітнього ступеня «магістра» за спеціальністю 122 «Комп'ютерні науки». — Університет митної справи та фінансів, Дніпро, 2024.

Магістерська робота спрямована на дослідження обмежень автоматизованного тестування, які виникають у процесі автоматизації тестування програмного забезпечення. Обмеження автоматизації тестування включають технічні, методологічні та організаційні аспекти, які можуть ускладнювати або унеможливлювати повноцінну автоматизацію тестування. Робота визначає та аналізує основні обмеження, такі як неможливість автоматизувати певні типи тестів, складнощі зі сценаріями, які вимагають взаємодії з людьми або зовнішніми сервісами, та труднощі з динамічними даними.

Для досягнення результатів використовувались інструменти для написання автоматизованних тестів, такі як Cucumber та Cypress.

В заключенні визначається, що автоматизація тестування не є універсальним рішенням і не завжди виправдовує очікування. Вона може бути витратною та трудомісткою задачею, особливо на початкових етапах. Вона не може гарантувати що всі дефекти будуть виявлені, а для її успішної реалізації необхідно встановити стандартизовані підходи та ефективне управління.

ABSTRACT

Exploring Automation Testing Limitations

Qualification work for the degree of "Master" in the specialty 122 "Computer Science". - University of Customs and Finance, Dnipro, 2024.

The master's thesis focuses on investigating the limitations of automated testing that arise in the process of automating software testing. Automation testing limitations include technical, methodological, and organizational aspects that may complicate or prevent full automation of testing. The work identifies and analyzes main constraints, such as the inability to automate certain types of tests, difficulties with scenarios requiring interaction with people or external services, and challenges with dynamic data.

To achieve the results, tools for writing automated tests, such as Cucumber and Cypress, were used.

In conclusion, it is determined that automation testing is not a universal solution and does not always meet expectations. It can be costly and laborious, especially in the initial stages. It cannot guarantee that all defects will be detected, and successful implementation requires standardized approaches and effective management.

ЗМІСТ

ВСТУП

У сучасному програмному забезпеченні якість відіграє визначальну роль у забезпеченні задоволення потреб користувачів та ефективності роботи програмних продуктів. Автоматизація тестування є важливим етапом у забезпеченні високої якості програмних рішень.

Зараз більшість складних сучасних веб-сайтів створюються великими комерційними компаніями. Прибуток компанії може залежати від роботи веб-застосувань. Таким чином, характеристики якості, надійності та відповідності вимогам до продукту є критично важливими під час розробки. Таким чином, постійно з’являються нові інструменти, методи та технології, які полегшують та прискорюють процеси створення та випуску продуктів. Сам процес розробки має бути гнучким, щоб замовник міг швидко змінити свої вимоги, якщо це потрібно. Програмний продукт має працювати без помилок і відповідати вимогам до якості замовника. Для цього зручно, щоб розробники мали проміжні версії продукту, які вже відповідали стандартам якості та працювали коректно, ще під час процесу розробки.

Валідація, верифікація та тестування забезпечують якість і відповідність стандартам. Для більшості клієнтів важливо, щоб їх веб-застосування були легко підтримувати та можна було розвивати та розширювати, якщо це потрібно.

Етап життєвого циклу розробки програмного продукту, відомий як тестування програмного забезпечення, передбачає перевірку відповідності між реальною поведінкою програми та поведінкою, яку очікує клієнт.

На цьому етапі команда може краще зрозуміти, в якому стані зараз знаходиться проект, на що їм слід звернути увагу, і як планувати наступну розробку. На даний момент більшість веб-застосувань розробляються відповідно до цілей групи кваліфікованих працівників.

Автоматичне тестування є частиною контролю якості життєвого циклу розробки програмного забезпечення. Сучасні технології та програмні засоби допомагають у розробці та запуску автоматичних тестів, скорочуючи час і спрощуючи процес.

В рамках цієї роботи буде проведено детальний огляд сучасних підходів та інструментів, що використовуються для автоматизації тестування, а також їх переваги та недоліки. Основними завданнями дослідження буде аналіз і порівняння різних методів автоматизації тестування, розробка стратегії автоматизації для певного програмного продукту та реалізація цієї стратегії з використанням конкретних інструментів.

Метою даної магістерської роботи є дослідження, аналіз та впровадження автоматизованих підходів до тестування програмного забезпечення для підвищення ефективності та швидкості тестування. Робота також ставить перед собою завдання порівняти різні методи автоматизації та розробити стратегію автоматизації для конкретного програмного продукту.

Методами дослідження даної магістерської роботи є проведення літературного аналізу щодо сучасних методів та інструментів автоматизації тестування. Вивчення переваг та недоліків різних методів автоматизації тестування для обрання оптимального підходу. Аналіз характеристик конкретного програмного продукту для визначення оптимальної стратегії автоматизації. Проведення тестування та аналіз отриманих результатів для визначення ефективності впровадженої стратегії.

1 ПРОБЛЕМА ТЕСТУВАННЯ ПЗ

1.1 Теоретична база тестування

Теоретична база тестування програмного забезпечення включає в себе різноманітні концепції, методи та підходи, спрямовані на забезпечення якості ПЗ. Ось деякі ключові аспекти теоретичної бази тестування:

1.1.1 Модель життєвого циклу програмного забезпечення (SDLC):

Модель життєвого циклу програмного забезпечення (SDLC - Software Development Life Cycle) - це концептуальний каркас, що визначає послідовність етапів та процесів, які використовуються для розробки та управління програмним забезпеченням від його концепції до виведення в експлуатацію та підтримки. Різні моделі SDLC можуть відрізнятися за кількістю та послідовністю етапів, а також за варіантами взаємодії між цими етапами. Однак, основні кроки зазвичай залишаються подібними в різних моделях[1].

Основні етапи SDLC:

Специфікація вимог (Requirements Gathering): Визначення та документування вимог до програмного забезпечення. Цей етап включає в себе спілкування з клієнтом або замовником для збирання необхідної інформації.

Аналіз (Analysis): Розробка величезного документу, який включає деталізовану специфікацію вимог, функціональності, архітектури системи та інших характеристик проекту.

Проектування (Design): Створення архітектури програмного забезпечення, визначення компонентів, модулів та інтерфейсів, а також розробка плану тестування.

Розробка (Implementation): Запис програмного коду та його тестування. Цей етап може включати програмування, створення бази даних, розробку інтерфейсу користувача та інше.

Тестування (Testing): Перевірка програмного забезпечення на відповідність вимогам та виявлення помилок чи дефектів. Тести можуть бути проведені на різних рівнях, таких як модульне, інтеграційне, системне та приймальне тестування.

Впровадження (Deployment): Розгортання програмного забезпечення в реальному середовищі. Цей етап може включати перехід від тестового середовища до виробничого, навчання користувачів та інші дії для впровадження програми.

Підтримка та обслуговування (Maintenance): Після впровадження вирішення помилок, вдосконалення та підтримка програмного забезпечення впродовж його життєвого циклу.

Існують різні моделі SDLC, такі як Каскадна модель, Модель спіралі, Модель ітерацій та інші, кожна з яких визначає конкретний спосіб взаємодії між цими етапами. Вибір конкретної моделі залежить від характеристик проекту, вимог та стратегії розробки.

1.1.2 Принципи тестування.

Принципи тестування визначають основні підходи та принципи, які повинні взяти до уваги тестувальники при розробці тестів та виконанні тестових процедур[2]. Ось сім принципів тестування:

Вичерпне тестування неможливе:

Повне тестування неможливе. Натомість нам потрібен оптимальний обсяг тестування, заснований на оцінці ризиків програми. І питання на мільйон доларів полягає у тому, як визначити цей ризик? Щоб відповісти на це питання, давайте виконаємо вправу. На вашу думку, яка операція з найбільшою ймовірністю спричинить збій вашої операційної системи? Я впевнений, що більшість із вас здогадалися: одночасно відкрити 10 різних програм. Отже, якби ви тестували цю операційну систему, ви зрозуміли б, що дефекти, швидше за все, будуть виявлені в багатозадачній діяльності і їх необхідно ретельно протестувати, що підводить нас до наступного принципу.

Кластеризація дефектів:

Кластеризація дефектів, яка стверджує, що невелика кількість модулів містить більшу частину виявлених дефектів. Це застосування принципу Парето до тестування програмного забезпечення: приблизно 80% проблем виявляється у 20% модулів. Досвідченим шляхом можна виявити такі ризиковані модулі. Але цей підхід має свої проблеми. Якщо одні й самі тести повторюються знову й знову, зрештою одні й самі тестові приклади перестануть виявляти нові помилки.

Парадокс пестицидів:

Повторне використання однієї і тієї ж суміші пестицидів для знищення комах у сільському господарстві з часом призведе до того, що у комах розвинеться стійкість до пестицидів. Таким чином, пестициди стануть неефективними щодо комах. Те саме стосується і тестування програмного забезпечення. Якщо буде проведено той же набір тестів, що повторюються, метод буде марний для виявлення нових дефектів. Щоб подолати цю проблему, тестові приклади необхідно регулярно переглядати та переглядати, додаючи нові та різні тестові приклади, щоб допомогти знайти більше дефектів. Тестувальники не можуть просто покладатися на існуючі методи тестування. Він повинен постійно прагнути покращити існуючі методи, щоб зробити тестування ефективнішим. Але навіть після цієї важкої роботи з тестування ви ніколи не зможете заявити, що ваш продукт не містить помилок. Щоб прояснити цю думку, давайте подивимося на це відео публічного запуску Windows 98. Ви думаєте, що така компанія, як MICROSOFT, не стали б ретельно тестувати свою ОС і ризикували б своєю репутацією тільки для того, щоб побачити збій своєї ОС під час її публічного запуску!

Тестування показує наявність дефектів:

Отже, принцип тестування говорить: Тестування свідчить про наявність дефектів і свідчить про відсутність дефектів. тобто. Тестування програмного забезпечення знижує ймовірність того, що у програмному забезпеченні залишаться невиявлені дефекти, але навіть якщо дефекти не виявлено, це не є доказом правильності. Але,

якщо ви докладете максимум зусиль, вживете всіх запобіжних заходів і зробите свій програмний продукт на 99% вільним від помилок. І програмне забезпечення не відповідає потребам та вимогам клієнтів.

Відсутність помилок:

Цілком можливо, що програмне забезпечення, яке на 99% не містить помилок, все ще непридатне для використання. Це може статися у разі, якщо система ретельно тестується щодо неправильного вимоги. Тестування програмного забезпечення – це не просто пошук дефектів, а й перевірка того, що програмне забезпечення відповідає потребам бізнесу. Відсутність помилки є оманою, тобто. пошук та виправлення дефектів не допоможе, якщо складання системи непридатна для використання та не відповідає потребам та вимогам користувача.

Раннє тестування:

Раннє тестування. Тестування слід розпочинати якомога раніше у життєвому циклі розробки програмного забезпечення. Таким чином, будь-які дефекти в вимогах або етапі проектування виявляються на ранніх стадіях. Набагато дешевше виправити Дефект на ранніх стадіях тестування. Але як рано розпочинати тестування? Рекомендується розпочинати пошук помилки з моменту визначення вимог. Докладніше про цей принцип у наступному навчальному посібнику.

Тестування залежить від контексту:

Тестування залежить від контексту, що по суті означає, що спосіб тестування сайту електронної комерції відрізнятиметься від способу тестування готової комерційної програми. Усі розроблені програми не ідентичні. Залежно від типу програми ви можете використовувати інший підхід, методології, методи та типи тестування. Наприклад, тестування будь-якої POS-системи у роздрібному магазині відрізнятиметься від тестування банкомату.

1.1.3 Методи тестування:

Включають в себе чорний ящик, білий ящик, сірі ящики. Це ще один критерій тестування, який відповідає за глибину знання тестувальником системи.

Black box testing – це метод тестування програмного забезпечення, коли тестувальник не має доступу до внутрішнього коду або докладного знання про структуру програми. В цьому методі тестування фокус знаходиться на поведінці програми зі зовнішнього боку[3]. Один з прикладів black box тестування – це тестування веб-сторінки шляхом перевірки правильності виводу результатів за конкретними вхідними даними. Тестувальник не знає, як саме ці результати отримуються, він тільки перевіряє правильність результатів.

Grey box testing – це комбінація black box та white box тестування, коли тестувальник має обмежений доступ до внутрішньої структури програми, але не знає всіх деталей. Цей метод тестування дозволяє тестувальнику отримати більш глибоке розуміння програми і продемонструвати кращі результати, ніж просто black box тестування. Один з прикладів grey box тестування – це тестування програми з доступом до бази даних, але з обмеженою можливістю вносити зміни у її структуру. Тестувальник може спробувати різні варіанти вхідних даних та перевірити, як програма взаємодіє з базою даних та обробляє інформацію.

White box testing – це метод тестування, коли тестувальник має повний доступ до внутрішньої структури програми та знає, як саме вона працює. Цей метод тестування дозволяє перевірити різні компоненти програми, включаючи логіку, алгоритми та код. Один з прикладів white box тестування – це перевірка правильності роботи алгоритму сортування в програмі. Тестувальник може аналізувати код та переконатися, що алгоритм правильно сортує дані за заданими критеріями.

Black, grey та white box testing – це різні методи тестування програмного забезпечення, які забезпечують високу якість та надійність програм. Black box testing сконцентроване на зовнішньому поведінці програми, grey box testing

дозволяє тестувальнику отримати більш глибоке розуміння програми, а white box testing дозволяє перевірити внутрішню структуру та роботу програми.

1.1.4 Метрики тестування:

Метрика — це міра, яка дозволяє отримати числове значення деяких властивостей ПЗ[4].

Метрика — іншими словами це показник поточного стану досягнення цілі. Тому що метрики без цілі, заради того аби щось міряти — не мають змісту. Хоча керівництву, клієнтам є до вподоби гарна аналітика, красиві графіки, показники тощо.

Метрики тестування програмного забезпечення поділяються на два типи:
Метрики процесу (Process metrics). Використовуються у процесі підготовки та виконання тестування.

- Продуктивність підготовки тест-кейсів (Test Case Preparation Productivity): використовується для розрахунку кількості підготовлених тест-кейсів та зусиль (Effort), витрачених на їхню підготовку.

  Test Case Preparation Productivity = (No of Test Case) / (Effort spent for Test Case Preparation)
- Охоплення тестового дизайну (Test Design Coverage): відсоток покриття тест-кейс вимог.

  Test Design Coverage = ((Total number of requirements mapped to test cases) / (Total number of requirements)*100
- Продуктивність виконання тестів (Test Execution Productivity): визначає кількість тест-кейсів, які можуть бути виконані за годину.

  Test Execution Productivity = (No test tests executed) / (Effort spent for execution of test cases)
- Покриття виконаних тестів (Test Execution Coverage): призначене для вимірювання кількості виконаних тест-кейсів порівняно з кількістю

запланованих тест-кейсів. Test Execution Coverage = Test Execution Coverage = (Total no. of test cases executed / Total no. of test cases planned to execute)*100

- Успішні тест-кейс (Test Cases Passed): для вимірювання відсотка пройдених успішно тест-кейсів. Test Cases Pass = (Total no. of test cases passed) / (Total no. of test cases executed) * 100
- Неуспішні тест-кейс (Test Cases Failed): для вимірювання відсотка завалених тест-кейсів. Test Cases Failed = (Total no. of test cases failed) / (Total no. of test cases executed) * 100
- Заблоковані тест-кейс (Test Cases Blocked): для вимірювання відсотка блокованих тест-кейсів. Test Cases Blocked = (Total no. of test cases blocked) / (Total no. of test cases executed) * 100

Метрики продукту (Product metrics):

- Рівень виявлення помилок (Error Discovery Rate): визначення ефективності тест-кейсів.
- Error Discovery Rate = (Додатковий номер звільнених дій / Total no. of test cases executed)*100
- Відсоток виявлення дефектів (DDP): Кількість дефектів, виявлених у фазі тестування, поділена на кількість дефектів, знайдених у цій фазі тестування, а також у всіх наступних фазах. також вислизнув дефект. (ISTQB)
- Рівень виправлення дефектів (Defect Fix Rate): допомагає дізнатися якість складання (build) з погляду усунення дефектів.

  Дефект fix rate = (Все, що не позначається як реакція - Total no. defects reopened) / (Total no of Defects, що опублікована як fixed + Total no. of new Bugs due to fix)*100
- Щільність дефектів (Defect Density): кількість дефектів, виявлених у компоненті або системі, поділена на розмір компонента або системи (виражений у стандартних одиницях вимірювання, наприклад, рядках коду, числі класів або функцій). (ISTQB)

Defect Density = Total no. of defects identified / Actual Size (requirements)

– Виток дефектів (Defect Leakage): використовується для перевірки ефективності процесу тестування перед UAT.

  Defect Leakage = ((Total no. of defects found in UAT)/(Total no. of defects found before UAT)) * 100

– Ефективність усунення дефектів (Defect Removal Efficiency): дозволяє порівнювати загальну (дефекти, виявлені до та після постачання) ефективність усунення дефектів.

– Defect Removal Efficiency = ((Total no. defects found pre-delivery) /((Total no. of defects found pre-delivery)+ (Total no. of defects found post-delivery)))* 100

1.2 Види тестування

Існує багато різних типів тестування, які використовуються для перевірки різних аспектів якості програмного забезпечення. Ось деякі з найпоширеніших типів тестування[5]:

Функціональне тестування - це тип тестування програмного забезпечення, який фокусується на перевірці можливостей та поведінки програми відповідно до заданих вимог. Основна мета функціонального тестування – переконатися, що програмне забезпечення працює правильно, як задумано, та забезпечує бажану функціональність. Ця категорія тестування включає різні методи тестування, в тому числі:

Модульне тестування – це процес тестування окремих компонентів або блоків програмного застосування в ізоляції. Воно в першу чергу спрямоване на перевірку правильності функціональності кожного блоку за допомогою вхідних даних тесту та затвердження, чи вихідний результат тесту відповідає очікуваним результатам. Юніт-тестування є найважливішою практикою виявлення та усунення дефектів на ранніх стадіях процесу розробки, що допомагає скоротити загальні витрати та час виходу на ринок.

Інтеграційне тестування - це процес поєднання різних одиниць або компонентів програмного додатку та тестування їх як єдиної групи. Здебільшого воно спрямоване на перевірку взаємодії між інтегрованими одиницями, гарантуючи, що вони працюють правильно та без проблем. Інтеграційне тестування допомагає виявити та усунути проблеми, пов'язані з потоком даних, комунікацією та залежностями між компонентами програми.

Системне тестування - це процес тестування всього програмного додатку в цілому, що оцінює його загальну функціональність, продуктивність та відповідність заданим вимогам. Основна мета системного тестування - перевірити поведінку програмного застосування у різних умовах та конфігураціях, забезпечуючи безперебійну та задовільну роботу користувача. Системне тестування допомагає виявити та вирішити проблеми, пов'язані з інтеграцією, сумісністю та загальною стабільністю системи.

Регресійне тестування – це практика тестування програмної програми після внесення змін, виправлення помилок чи оновлень. Воно спрямоване на перевірку того, що будь-які зміни, внесені в додаток, не впливають на існуючу функціональність і не створюють нових проблем. Регресійне тестування допомагає підтримувати якість та надійність програмного забезпечення протягом усього процесу розробки, гарантуючи, що будь-які зміни чи вдосконалення не порушать стабільність програми та зручність роботи користувачів.

Приймальне тестування, також відоме як приймальне тестування користувача (UAT), є заключним етапом функціонального тестування, на якому оцінюється, чи програмний додаток відповідає заданим вимогам і потребам користувача. Приймальний тест зазвичай проводиться кінцевими користувачами або клієнтами, які перевіряють функціональність, зручність використання та сумісність програмного забезпечення в реальних сценаріях використання. Основна мета приймального тестування – гарантувати, що програмний додаток забезпечує

бажану функціональність та цінність для гаданих користувачів, тим самим мінімізуючи ризик потенційного невдоволення, відмови чи ескалації.

Нефункціональне тестування - це вид тестування програмного забезпечення, який оцінює критичні аспекти програмного застосування, такі як продуктивність, зручність використання та безпека, сприяючи загальному враженню користувачів та стабільності системи. Нефункціональне тестування спрямоване на оптимізацію поведінки програми, гарантуючи, що вона відповідає заданим показникам продуктивності, забезпечує безпроблемну та інтуїтивно зрозумілу роботу користувача, а також захищає від потенційних загроз безпеці. Деякі ключові методи нефункціонального тестування включають:

Тестування продуктивності - це процес оцінки поведінки програми за різних навантажень та умов, таких як високий трафік, одночасні користувачі та ресурсомісткі завдання. В основному воно спрямоване на оцінку чуйності, масштабованості та ефективності програмного забезпечення, гарантуючи, що воно відповідає заданим контрольним показникам продуктивності та забезпечує задовільний користувальницький досвід. Тестування продуктивності допомагає виявити та усунути вузькі місця, можливості оптимізації та потенційні проблеми, пов'язані з продуктивністю.

Тестування юзабіліті - це процес оцінки користувальницького інтерфейсу програмної програми, простоти використання та загального враження користувача. В першу чергу воно спрямоване на оцінку дизайну, компонування, навігації та взаємодії програми на основі очікувань, переваг та ментальних моделей передбачуваних користувачів. Тестування юзабіліті допомагає виявити та усунути проблеми, пов'язані із задоволеністю користувачів, доступністю та ефективністю, гарантуючи, що програмна програма забезпечує безпроблемний та інтуїтивно зрозумілий користувальницький досвід.

Тестування безпеки - це процес оцінки вразливості програмного застосування до потенційних атак, несанкціонованого доступу та витоку даних. Насамперед воно

спрямоване на оцінку механізмів безпеки додатку, заходів захисту та практик, спрямованих на виявлення та усунення потенційних ризиків та уразливостей безпеки. Тестування безпеки допомагає забезпечити захист програмної програми від потенційних загроз, гарантуючи цілісність, конфіденційність і доступність програми та даних, що лежать в її основі.

Тестування на сумісність - це процес оцінки поведінки та продуктивності програмного застосування на різних платформах, у різних конфігураціях та середовищах. В основному воно спрямоване на оцінку сумісності програми з різними операційними системами, браузерами, пристроями та мережевими умовами, забезпечуючи безперебійну та послідовну роботу користувача у різних сценаріях використання. Тестування на сумісність допомагає виявити та усунути потенційні проблеми, пов'язані з крос-платформною підтримкою, сумісністю та адаптивністю, сприяючи загальному задоволенню та прийняттю програмної програми.

Ручне тестування - це процес тестування програмних додатків людьми, які взаємодіють із додатком та оцінюють його поведінку без підтримки автоматизованих тестових сценаріїв чи інструментів. Ручне тестування, як і раніше, вважається важливою частиною процесу тестування програмного забезпечення, особливо на початкових етапах розробки або коли програма представляється новою цільовою аудиторією. Деякі з основних методів ручного тестування включають:

Дослідницьке тестування: При дослідному тестуванні тестувальники активно вивчають програму, розробляють тестові випадки та одночасно виконують їх. Такий підхід дозволяє тестувальникам виявити дефекти, які, можливо, були передбачені на етапі розробки проекту. Дослідницьке тестування корисне, коли є обмежена кількість документації чи формальних планів тестування.

Тестування зручності використання: Тестування зручності використання в першу чергу спрямоване на оцінку програми з погляду кінцевого користувача, аналізуючи, наскільки легко ним користуватися та орієнтуватися. Тестувальники

оцінюють загальний досвід користувача, включаючи такі аспекти, як інтуїтивний дизайн, навченість і доступність. Цей вид тестування допомагає розробникам покращити користувальницький інтерфейс програми та вирішити будь-які проблеми зі зручністю використання, які можуть вплинути на успіх програми на ринку.

Регресійне тестування: Регресійне тестування спрямоване на забезпечення того, щоб існуючі функціональні можливості програми не постраждали від нових змін, таких як виправлення помилок, розширення функцій або оновлення системи. Тестувальники виконують раніше запущені тестові випадки, щоб переконатися, що зміни не призвели до появи нових проблем і що програма продовжує відповідати заданим вимогам.

Виявлення помилок: При виконанні ручних тестів тестувальники зазвичай наслідують тестові приклади, які охоплюють очікувані функціональні можливості та різні побічні випадки. За допомогою цих тестових прикладів тестувальники можуть знайти помилки, невідповідності та протиріччя у поведінці програми.

Ручне тестування має ряд переваг, таких як здатність виявляти несподівані проблеми, адаптуватися до вимог, що змінюються, і надавати цінні відомості про реальний досвід користувачів. Однак воно має і свої недоліки, наприклад, вимагає багато часу, схильне до людських помилок і потенційно менш ефективно, ніж автоматизовані методи тестування.

Автоматизоване тестування – це процес виконання тестів за допомогою тестових сценаріїв, інструментів та фреймворків. Воно включає автоматизацію повторюваних, що вимагають багато часу завдань, що підвищує загальну ефективність, надійність і точність процесу тестування. Деякі популярні методи автоматизованого тестування включають:

Модульне тестування: Модульне тестування спрямоване на перевірку правильності окремих компонентів або функцій у програмі. Розробники пишуть модульні тести для перевірки того, що їхній код відповідає заданим вимогам.

Популярні платформи для модульного тестування включають JUnit та TestNG для Java, NUnit для .NET та XCTest для iOS.

Інтеграційне тестування: Інтеграційне тестування перевіряє взаємодію між різними модулями або компонентами програми, забезпечуючи їх правильну спільну роботу. Цей тип тестування допомагає виявити проблеми, пов'язані з потоком даних, комунікацією та залежностями між модулями. Відповідні інструменти для інтеграційного тестування включають SoapUI і Postman для тестування API і Selenium і Appium для тестування інтерфейсу користувача.

Функціональне тестування: Автоматизоване функціональне тестування спрямоване на перевірку відповідності функцій та поведінки програми заданим вимогам. Тестувальники розробляють тестові сценарії для імітації дій користувача та перевіряють, чи веде додаток себе так, як очікується в різних умовах. Selenium - інструмент функціонального тестування, що широко використовується, для веб-додатків, а Appium популярний для тестування мобільних додатків.

Тестування навантаження та продуктивності: Тестування навантаження та продуктивності допомагає виявити вузькі місця, використання ресурсів та проблеми масштабованості, які впливають на загальну продуктивність програми та зручність роботи користувачів за різних умов навантаження. Для тестування навантаження та продуктивності зазвичай використовуються такі інструменти, як JMeter, LoadRunner та Gatling.

Автоматизоване тестування дає ряд переваг, таких як швидке виконання, збільшення покриття тестів, зменшення кількості людських помилок та можливість паралельного виконання тестів. Однак воно вимагає великих початкових інвестицій у вигляді часу, зусиль та ресурсів для розробки та підтримки тестових сценаріїв та фреймворків. Крім того, не всі сценарії тестування підходять для автоматизації, особливо коли йдеться про тестування зручності використання та інші аспекти, для ефективної оцінки яких потрібний людський фактор.

Статичне тестування - це тип тестування програмного забезпечення, який включає оцінку коду, дизайну і документації програми без фактичного виконання коду. Основною метою статичного тестування є виявлення проблем, невідповідностей та можливих покращень на ранніх стадіях процесу розробки програмного забезпечення. Деякі поширені підходи до статичного тестування включають:

Огляд коду: Огляд коду - це процес ручного перегляду вихідного коду для виявлення помилок, проблем проектування та невідповідностей, які можуть вплинути на загальну якість програми. Огляди коду сприяють співпраці, обміну знаннями та дотримання стандартів кодування та кращих практик. Вони допомагають розробникам виявити та усунути потенційні проблеми до того, як їх виправлення стане складним та дорогим.

Статичний аналіз: Інструменти статичного аналізу автоматично аналізують вихідний код для виявлення проблем, пов'язаних зі стандартами кодування, передовою практикою та потенційними вразливістю. Ці інструменти допомагають розробникам виявити мертвий код, витік пам'яті, розіменування нульового покажчика та інші поширені проблеми програмування. Популярні інструменти статичного аналізу включають SonarQube, Checkstyle та PMD.

Огляд документації: Огляд документації спрямований на оцінку проектної документації, такої як вимоги, проектна документація та посібники користувача, щодо точності, послідовності та ясності. Цей процес допомагає виявити двозначності, невідповідності та неповну інформацію, які можуть призвести до неправильних інтерпретацій, припущень та дефектів у додатку. Статичне тестування дає безліч переваг, таких як раннє виявлення дефектів, скорочення часу та витрат на розробку, покращення якості коду та документації. Воно допомагає розробникам виявляти та усувати проблеми до того, як вони переростуть у серйозніші проблеми, що знижує ймовірність появи помилок на пізніших етапах життєвого циклу розробки програмного забезпечення.

1.3 Сучасні автоматизовані системи тестування

Сучасні автоматизовані системи тестування надзвичайно різноманітні та інноваційні, пропонуючи широкий спектр інструментів для розробників програмного забезпечення. Ось деякі з них:

1.3.1 Selenium:

Це набір інструментів для автоматизації веб-додатків. Він надає програмістам і тестувальникам інструменти для написання скриптів на різних мовах програмування, таких як Java, C#, Python, Ruby і т. д., які автоматизують дії, які зазвичай виконує користувач у веб-браузері.

Основні компоненти Selenium:

Selenium WebDriver:

Це ключовий компонент, який надає програмному забезпеченню можливість взаємодіяти з веб-браузером. WebDriver дозволяє виконувати різні дії на веб-сторінках, такі як заповнення форм, навігація, клікання на елементи і отримання тексту.

Selenium IDE (Integrated Development Environment):

Це додаток для браузера, який дозволяє записувати, редагувати та відтворювати тестові сценарії. Він оснащений графічним інтерфейсом та призначений для користувачів з мінімальними навичками програмування.

Selenium Grid:

Grid дозволяє одночасно виконувати тести на різних машинах та браузерах, що дозволяє паралельне виконання тестів на різних конфігураціях.

Основні характеристики Selenium:

– Широкий вибір мов програмування: Selenium підтримує багато мов програмування, так що ви можете вибрати ту, яку ви знаєте краще або яка найкраще підходить для вашого проекту.

– Підтримка браузерів: Selenium працює з популярними веб-браузерами, такими як Chrome, Firefox, Safari, Edge, іншими.
– Можливість тестування в реальних браузерах та хедлесс (безголовних) режимах: Тестування може відбуватися як в реальних браузерах, так і в безголовних режимах, що полегшує автоматизацію тестів у фоновому режимі.
– Робота з AJAX-елементами та очікування: Selenium має можливості чекати на завантаження сторінки, наявність або відсутність елементів, що робить його ефективним для тестування динамічних веб-додатків.
– Підтримка тестування на різних операційних системах: Selenium може використовуватися для тестування веб-додатків на різних операційних системах, таких як Windows, Linux, macOS.

Selenium використовується для функціонального, регресійного та навантажувального тестування веб-додатків. Цей інструмент є основним в інструментарії багатьох тестувальників та розробників, які працюють з веб-додатками.

1.3.2 Appium:

Це відкрите програмне забезпечення для автоматизації тестів мобільних додатків. Він дозволяє розробникам та тестувальникам створювати та виконувати тести для мобільних додатків на різних платформах, таких як Android та iOS, використовуючи один і той же набір API.

Основні характеристики Appium:

– Крос-платформеність. Appium є крос-платформеним інструментом, що дозволяє вам автоматизувати мобільні додатки для обох основних платформ - Android та iOS - з використанням одного і того ж коду.
– Підтримка різних мов програмування. Ви можете використовувати різні мови програмування для написання тестів, такі як Java, JavaScript, Python, Ruby, C#,

і PHP. Це дозволяє командам використовувати ту мову, яку вони знають краще або яка найкраще відповідає їхній стеку технологій.

- Підтримка різних типів додатків. Appium підтримує автоматизацію тестів для різних типів додатків, включаючи native, hybrid та mobile web apps.
- Спрощене налаштування. Немає потреби змінювати код додатку для вбудовання Appium. Ви можете використовувати нативні збірки додатків, які вам надали розробники.
- Широкий функціонал для взаємодії з елементами. Appium дозволяє взаємодіяти з елементами інтерфейсу користувача, використовуючи різні дії, такі як натискання, перетягування, введення тексту, встановлення значень тощо.
- Інтеграція з різними інструментами. Appium легко інтегрується з різними фреймворками тестування та засобами звітності.
- Підтримка хедлесс (безголовних) тестів. Appium дозволяє вам виконувати тести в хедлесс (безголовному) режимі, коли немає потреби показувати інтерфейс користувача.

1.3.3 Postman:

Це інструмент для розробки та тестування API. Його використовують для спрощення процесу створення, тестування, обладнання та взаємодії з API (інтерфейсами програмування застосунків). Postman надає інтерфейс для відправлення HTTP-запитів до веб-сервера та отримання відповідей.

Основні характеристики Postman:

- Спрощений відправка HTTP-запитів. Postman дозволяє користувачам легко створювати та відправляти HTTP-запити (GET, POST, PUT, DELETE тощо) без необхідності написання коду. Він надає графічний інтерфейс для введення параметрів та налаштування запитів.

- Створення та організація колекцій запитів. Ви можете створювати та організовувати колекції запитів, щоб легко управляти тестами та повторювати їх.
- Автоматизація тестування API. Postman дозволяє створювати тести для перевірки відповідей від сервера. Ви можете автоматизувати тестові сценарії та використовувати їх для забезпечення правильності роботи вашого API.
- Середовища та змінні. Postman дозволяє вам налаштовувати середовища для ваших API, що дозволяє вам легко перемикатися між різними середовищами (наприклад, тестовим та продакшн). Ви також можете використовувати змінні для динамічного змінювання значень у запитах.
- Генерація коду. Postman може генерувати код для різних мов програмування на основі вашого запиту. Це дозволяє розробникам швидко і легко використовувати API у своїх додатках.
- Спільна робота та обмін даними. Postman надає можливість спільно працювати над проектами та обмінюватися запитами з іншими користувачами.
- Тестування WebSocket. Окрім роботи з HTTP-запитами, Postman також підтримує тестування WebSocket, що дозволяє взаємодіяти з веб-сокетами.

Postman допомагає розробникам, тестувальникам та адміністраторам API простіше взаємодіяти з серверами та впевнено виконувати та тестувати свої API-застосунки.

1.3.4 JUnit/TestNG:

Це фреймворки для тестування в програмуванні на Java. Вони надають засоби для створення, виконання та управління тестовими сценаріями для перевірки правильності функціональності програмного забезпечення.

JUnit:

Основні характеристики:

- Анотації: JUnit використовує анотації Java для позначення тестових методів, фіксації налаштувань та запуску тестів.
- Тестові методи: Тестові методи повинні бути анотовані @Test та визначатися з певними правилами назви (наприклад, починатися з "test").
- Assertions: В JUnit використовуються вбудовані методи для перевірки очікуваних результатів (assertions). Наприклад, assertEquals, assertTrue, assertFalse.

Специфікації JUnit 5:

- JUnit Platform: Платформа, яка надає основну інфраструктуру для виконання тестів та підтримує різні фреймворки для тестування.
- JUnit Jupiter: Новий програмний модуль для написання та виконання тестів на основі JUnit 5.
- JUnit Vintage: Модуль для підтримки запуску тестів, написаних для JUnit 3 та JUnit 4.

TestNG:

Основні характеристики:

- Анотації: TestNG також використовує анотації для позначення тестових методів та налаштувань.
- Групи тестів: Можливість групування тестових методів в логічні групи для виконання окремо.
- Залежності тестів: Визначення порядку виконання тестів та управління залежностями між ними.
- Параметризація тестів: Здатність передавати параметри у тестові методи, що спрощує виконання одного тесту з різними вхідними даними.
- Працює з різними типами тестів: TestNG підтримує різні типи тестів, такі як одиничні тести, тести для сортування, тести відмов та інші.

Архітектура TestNG:

- Test: Представляє тестовий сценарій та може складатися з одного чи декількох класів тестів.
- Class: Визначає клас тесту, який може містити методи тесту та методи налаштування.
- Method: Представляє конкретний метод тесту в класі тесту.

1.3.5 Cypress:

Це фреймворк для автоматизованого тестування, призначений для тестування веб-додатків. Основна його особливість - це можливість виконання тестів прямо в браузері, що відрізняє його від більш традиційних інструментів автоматизованого тестування.

Основні характеристики Cypress:

- Автоматизація в браузері. Тести виконуються прямо в браузері, в якому відбувається реальна інтеракція з веб-додатком. Це дозволяє отримати більш точні та надійні результати.
- Real-time перегляд тестів. Ви можете спостерігати за виконанням тестів у режимі реального часу, переглядаючи події та виводячи повідомлення у консолі браузера.
- Вбудована асинхронність. Cypress побудований на парадигмі асинхронного програмування, що дозволяє легко управляти асинхронними операціями та запитами.
- Легке використання і налаштування. Легко встановлюється та налаштовується для роботи. Включає в себе вбудований інструмент для генерації початкових файлів тестів.
- Зручна API для тестування. Має простий та зрозумілий API для написання тестів, який включає в себе вбудовані методи для емуляції дій користувача та перевірки стану додатку.

- Підтримка для багатьох типів тестів. Підтримує функціональне тестування, інтеграційне тестування та прийомочне тестування (end-to-end).
- Автоматичне очікування. Cypress автоматично чекає на завантаження елементів і виконання асинхронних запитів, що дозволяє уникнути додаткових тайм-аутів та забезпечує стійкість тестів.
- Можливість встановлення пауз та відладки. Ви можете встановлювати паузи у виконанні тестів та використовувати вбудований відладчик для подальшого аналізу тестового середовища.

Cypress використовується для автоматизованого тестування веб-додатків та надає зручний та ефективний інструментарій для розробки та підтримки тестів.

1.3.6 Robot Framework:

Це відкрите програмне забезпечення для автоматизованого тестування та автоматизованого тестування відповідно до концепції роботизованого проектування програмних продуктів (RPA).

Основні характеристики та особливості Robot Framework:

- Легкість вивчення та використання. Robot Framework простий у використанні і швидко вивчається. Його синтаксис використовує ключові слова та анотації для опису тестових сценаріїв.
- Розширюваність. Можливість використовувати власні бібліотеки на різних мовах програмування, таких як Python, Java, C#, для створення власних ключових слів.
- Тестування крок за кроком. Легко організувати тести у вигляді кроків, що полегшує розуміння тестових сценаріїв.
- Підтримка прийомочного тестування (Acceptance Test-Driven Development - ATDD). Робота із зацікавленими сторонами для формалізації вимог та тестів на початковому етапі розробки.Підтримка бібліотек та плагінів. Широка підтримка стандартних бібліотек для тестування HTTP, баз даних, файлів,

GUI, мереж та інше. Можливість використовувати сторонні плагіни та бібліотеки.

- Графічний інтерфейс та інтеграція з CI/CD. Має графічний інтерфейс для використання та відлагодження тестів. Легко інтегрується з інструментами для продовження інтеграції (CI) та постійного вивільнення (CD).
- Зручність у використанні веб-додатків. Можливість легко взаємодіяти з елементами веб-сторінок та виконувати тести для веб-додатків.
- Гнучкість при використанні мов програмування. Можливість використовувати бібліотеки на різних мовах програмування для реалізації функцій та ключових слів.

1.3.7 LoadRunner:

Це інструмент для проведення тестів на навантаження та продуктивності в різних типах програмного забезпечення, зокрема, веб-додатків. Цей продукт був розроблений компанією Micro Focus і використовується для моделювання та імітації тиску, якому піддається система під час роботи в реальних умовах.

Основні характеристики та можливості LoadRunner:

- Протоколи підтримки. LoadRunner підтримує широкий спектр протоколів для виконання тестів, таких як HTTP, HTTPS, Web Services, Ajax, JDBC, Citrix, Microsoft .NET, Oracle NCA, та інші.
- Моделювання навантаження. Дозволяє створювати та виконувати тести для визначення, як система веде себе під різними рівнями навантаження та об'ємами даних.
- Аналіз продуктивності. Забезпечує засоби для аналізу продуктивності системи, включаючи вимірювання часу відгуку, використання ресурсів, побудову графіків і звітів.

- Сценарії імітації користувачів. Дозволяє створювати скрипти, які імітують поведінку реальних користувачів, включаючи взаємодію з веб-інтерфейсом, запити до сервера, та інші дії.
- Моніторинг системи. Дозволяє проводити моніторинг системи в режимі реального часу під час виконання тестів, збираючи дані про ресурси, що використовуються.
- Інтеграція з іншими інструментами. Можливість інтеграції з іншими інструментами для тестування, засобами продовження інтеграції, та системами моніторингу.
- Сценарії високого рівня. Дозволяє створювати сценарії високого рівня, що полегшує процес створення та моделювання тестів.
- Масштабованість. Здатність виконувати тести на великому масштабі з великою кількістю віртуальних користувачів.

1.3.8 TestComplete:

Це інтегроване середовище для автоматизованого тестування різних типів програмного забезпечення, включаючи веб-додатки, настільні застосунки, мобільні додатки та веб-сервіси. Він розроблений компанією SmartBear Software і володіє рядом функцій для створення, виконання та аналізу автоматизованих тестів.

Основні характеристики та можливості TestComplete:

- Підтримка різних платформ. TestComplete підтримує тестування програмного забезпечення на різних платформах, включаючи Windows, macOS та Linux.
- Підтримка різних типів додатків. Здатність автоматизувати тести для веб-додатків, настільних застосунків, мобільних додатків (iOS та Android), веб-сервісів і т. д.
- Запис та відтворення тестів. Вбудована функція запису та відтворення тестів для швидкого створення автоматизованих сценаріїв.

- Скриптоване та скриптоване тестування. Підтримка як скриптованого (з використанням мов програмування, таких як JavaScript, Python, VBScript), так і скриптованого тестування (без написання коду, використовуючи вбудовані ключові слова).
- Підтримка різних мов програмування. Можливість використовувати різні мови програмування для написання автоматизованих тестів.
- Інтеграція з системами керування версіями. Зручна інтеграція з різними системами керування версіями, такими як Git та SVN.
- Тестування мобільних додатків. Забезпечує можливість автоматизованого тестування мобільних додатків на платформах iOS та Android.
- Засоби реалізації тестових сценаріїв. Можливість використовувати дерево об'єктів для навігації та редагування об'єктів в програмному забезпеченні.
- Генерація звітів. Автоматизована генерація звітів та результатів тестування.

2 РОЗРОБКА ТА ВИКОРИСТАННЯ СИСТЕМИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ НА ОСНОВІ ФРЕЙМВОРКУ CYPRESS

Cypress - це фреймворк автоматизованого тестування для веб-додатків. Він використовується для створення, запуску та виконання тестів, спрямованих на перевірку функціональності та взаємодії користувача з веб-додатком.

Cypress володіє кількома особливостями, які роблять його популярним серед розробників:

- Простота використання. Його синтаксис легко зрозуміти та використовувати, що дозволяє швидко створювати тести.
- Зручний інтерфейс користувача. Cypress має гарний графічний інтерфейс для перегляду результатів тестів та налагодження тестових сценаріїв.
- Live Reload. Здатність оновлювати результати тестів у реальному часі, що дозволяє швидко бачити та аналізувати зміни в тестах.
- Підтримка модернізації. Він підтримує сучасні фреймворки та технології, такі як React, Angular, Vue та інші.
- Автоматизоване виявлення помилок. Cypress може автоматично виявляти проблеми у веб-додатку та надавати докладні повідомлення про помилки.
- Зручна інтеграція з іншими інструментами. Він добре інтегрується з іншими інструментами для тестування та CI/CD системами, такими як Jenkins, CircleCI, TeamCity тощо.
- Підтримка відлагоджування. Можливість використовувати відладку під час виконання тестів, що спрощує виявлення та виправлення проблем.

Його можна використовувати для різних типів тестів, таких як функціональні, інтеграційні, тестування API та інші.

2.1 Вибір методів та інструментів автоматизації

Автоматизація тестування з використанням Cypress передбачає використання різних методів та інструментів, які надає сам Cypress. Ось кілька ключових методів

та інструментів, які ви можете використовувати при написанні автоматизованих тестів з Cypress:

2.1.1 Cypress Commands:

- cy.visit(): це команда в фреймворку Cypress, яка використовується для переходу на вказану веб-адресу (URL). Ця команда дозволяє вам відкривати сторінки вашого веб-додатка або сайту та починати виконання тестів з конкретної точки.
- cy.get(): це команда в фреймворку Cypress, яка використовується для отримання посилань (збірників) на DOM-елементи на сторінці. Ця команда є ключовою для взаємодії з елементами сторінки під час написання тестів.
- Після виклику cy.get, ви можете використовувати інші команди Cypress для взаємодії з отриманим елементом, такими як click, type, should, тощо.
- cy.click(): В фреймворку Cypress команда .click() використовується для симуляції кліку на вибраний елемент на веб-сторінці. Це дозволяє автоматизовано виконувати дії, які користувач робить під час взаємодії зі сторінкою, такі як клік на кнопці, гіперпосиланні, чекбоксі, тощо.
- cy.type():Команда cy.type() використовується для введення тексту в поле вводу на веб-сторінці. Це дозволяє автоматизовано вводити дані, які користувач зазвичай вводить в текстові поля форм.
- cy.contains(): Команда cy.contains() використовується для знаходження елементу на сторінці за його текстовим вмістом. Ця команда дозволяє легко взаємодіяти з елементами, які мають певний текст, і використовується для автоматизованого пошуку елементів за їхнім вмістом.
- cy.should():У фреймворку Cypress команда cy.should() використовується для встановлення умов (assertions) щодо елементів на сторінці. Вона дозволяє перевіряти різні аспекти стану елементів та взаємодіяти з ними тільки тоді, коли вони задовольняють певні умови.

– Параметризація та динамічні дані:
– cy.fixture() використовується для завантаження фікстур (файлів даних) під час тестування. Фікстури - це заздалегідь підготовлені дані, які можна використовувати у ваших автоматизованих тестах.

Пошук елементів. CSS або XPath селектори використовуються для "пошуку" (або вибору) HTML-елементів, які ви хочете стилізувати.

Можна розділити CSS селектори на п’ять категорій:

– Прості селектори (вибір елементів на основі імені, ідентифікатору, класу)
– Комбінаторні селектори (вибір елементів на основі визначених відношень між ними)
– Селектори псевдокласів (вибір елементів на основі визначеного стану)
– Селектори псевдоелементів (вибір і стилізація частини елемента)
– Селектори атрибутів (вибір елементів на основі атрибуту або значення атрибуту)

Основні види CSS-селекторів:

– Елементи: element - вибір за тегом елемента (наприклад, div, p, a).
– Ідентифікатор: #id - вибір за ідентифікатором елемента (наприклад, #header).
– Клас: .class - вибір за класом елемента (наприклад, .btn).

Загальний синтаксис атрибуту:

– [attribute] - вибір за наявністю атрибуту (наприклад, [required]).
– [attribute=value] - вибір за значенням атрибуту (наприклад, [type="text"]).

Комбінатори:

– element element - вибір всередині іншого елемента (наприклад, div p).
– element > element - вибір дочірнього елемента (наприклад, ul > li).
– element + element - вибір елемента, що йде після іншого елемента на тому самому рівні (наприклад, h2 + p).
– element ~ element - вибір всіх елементів того ж рівня, що йде після даного елемента (наприклад, p ~ span).

Псевдокласи:

- :hover - стиль для елемента при наведенні.
- :focus - стиль для елемента, який отримав фокус.
- :active - стиль для елемента під час натискання на нього.
- :first-child - вибір першого дочірнього елемента батьківського елемента.
- :last-child - вибір останнього дочірнього елемента батьківського елемента.
- :nth-child(n) - вибір n-го дочірнього елемента батьківського елемента.

XPath (XML Path Language) - це мова запитів для вибору вузлів з XML-документів. У контексті фреймворку Cypress, XPath також може використовуватися для вибору елементів на веб-сторінках. В Cypress ви можете використовувати як CSS-селектори, так і XPath для знаходження елементів.

Основні види XPath-селекторів:

- Вибір елемента за тегом: cy.xpath('//div'); // Вибрати всі елементи
- Вибір елемента за ідентифікатором: cy.xpath('//*[@id="elementId"]'); // Вибрати елемент з ідентифікатором "elementId"
- Вибір елемента за класом: cy.xpath('//div[@class="elementClass"]'); // Вибрати елемент з класом "elementClass"
- Вибір елемента за текстовим вмістом: cy.xpath('//*[contains(text(), "Some text")]'); // Вибрати елемент з текстом "Some text"
- Використання позначень: cy.xpath('//input[@name="username"]'); // Вибрати <input> елемент з атрибутом name="username"
- Використання псевдокласів та псевдоелементів: cy.xpath('//div[@class="elementClass"]:first-child'); // Вибрати перший дочірній елемент з класом "elementClass"

Використання XPath може бути менш читабельним і більш складним у порівнянні з CSS-селекторами, тому рекомендується використовувати їх у крайніх випадках, коли CSS-селекторів не вистачає або їх недостатньо. Вибір між CSS і

XPath залежить від конкретного випадку використання та ваших власних уподобань.

2.1.2 Асинхронні операції:

У фреймворку Cypress багато операцій виконуються асинхронно, оскільки тестування веб-додатків часто включає в себе роботу з асинхронним JavaScript, таким як запити AJAX, завантаження ресурсів, очікування на події і т. д. Щоб ефективно працювати з асинхронним кодом в Cypress, слід використовувати вбудовані засоби та патерни.

Ось кілька основних аспектів асинхронних операцій у Cypress:

– Команди then і should. Багато команд у Cypress повертають об'єкти, які ви можете подальше аналізувати. Для цього використовуються методи then та should. Вони дозволяють вам виконувати асинхронні перевірки та маніпуляції з результатами команд наприклад: cy.get('.some-element').should('be.visible')
– Асинхронні команди. Деякі команди, такі як cy.request, являють собою асинхронні операції напряму. Вони повертають об'єкти обіцянок (Promises), і ви можете працювати з ними, використовуючи then або await.
– Наприклад: cy.request('https://example.com/api/data').then((response) => {
  // Робота з отриманою відповіддю
  });

2.1.3 Паралельні тести:

Запуск паралельних тестів в тестуванні означає виконання кількох тестових сценаріїв або тестових уитестів одночасно, замість того, щоб виконувати їх послідовно. Це може значно збільшити ефективність виконання тестів, особливо в умовах великої кількості тестів чи тестування розподілених систем.

Основні переваги запуску паралельних тестів включають:

– Економія часу. Паралельний запуск дозволяє виконувати тести одночасно, що значно зменшує час виконання всього тестового набору.

- Збільшення ефективності. Запуск тестів паралельно забезпечує оптимальне використання ресурсів тестового середовища, зменшуючи очікування на виконання.
- Швидке виявлення проблем. Якщо ви запускаєте тести паралельно і один з них видає помилку, це дозволяє швидше виявити та виправити проблему.
- Масштабованість. Паралельний запуск легко масштабується для використання в різних середовищах та на різних рівнях тестування.

Запуск паралельних тестів можна реалізувати за допомогою спеціальних фреймворків для управління тестами, контейнерів, або засобів управління виконанням тестів. Багато тестових фреймворків мають вбудовану підтримку паралельного виконання. Залежно від вибору інструментів та технологій у вашому тестовому стеку, існують різні підходи до організації паралельного виконання тестів.

2.1.4 Інтеграція з Page Object Model (POM):

Page Object Model (POM) - це патерн проектування в області тестування програмного забезпечення, особливо популярний у тестуванні веб-додатків. Ідея POM полягає у тому, щоб визначити веб-сторінки та їх елементи в окремих класах або моделях (Page Objects) для подальшого використання в автоматизованих тестах.

Основні концепції Page Object Model:

- Page Object - це клас або об'єкт, який представляє конкретну веб-сторінку. У цьому класі зазвичай знаходяться локатори елементів на сторінці та методи для взаємодії з цими елементами. Page Object ізолює логіку взаємодії зі сторінкою від логіки тестування.
- Локатори елементів. Локатори елементів визначають, як ізольовані елементи на сторінці ідентифікуються. Вони можуть бути представлені за допомогою CSS-селекторів, XPath, ідентифікаторів, класів тощо.

– Методи взаємодії з елементами. У Page Object визначають методи, які використовують локатори для взаємодії з елементами на сторінці. Це може включати дії, такі як натискання на кнопку, заповнення форм, отримання текстового вмісту тощо.
– Розділення логіки. POM дозволяє розділити логіку тестування та логіку взаємодії зі сторінкою. Такий підхід робить тести більш читабельними, підтримуваними та можливими до перевикористання.
– Спрощення тестових сценаріїв. За допомогою POM тестові сценарії можуть бути більш зрозумілими та простими. Тестерам не потрібно займатися деталями взаємодії зі сторінками, оскільки ця логіка вже зосереджена в Page Objects.
– Підтримка масштабованості. Page Object Model дозволяє легко розширювати та змінювати тестовий код при зміні або додаванні нових функцій до веб-додатка.

Рисунок 2.1 Приклад використання Page Object Model

2.2 Визначення тестових сценаріїв для автоматизації

Визначення тестових сценаріїв для автоматизації за допомогою Cypress включає в себе розгляд функціональностей вашого веб-додатку та визначення кроків, які потрібно протестувати. Нижче подано кроки для визначення тестових сценаріїв:

2.2.1 Аналіз функціональностей:

Відноситься до оцінки та розгляду функціональностей, які входять у склад роботи відділу або команди з якості (QA - Quality Assurance). Такий аналіз може включати в себе розгляд різноманітних аспектів, таких як процеси тестування, автоматизація, взаємодія з розробкою, відслідковування багів, тест-дизайн і так далі.

Нижче розглянуто деякі ключові аспекти, які можна включити в аналіз функціональностей QA:

- Процес тестування. Оцінка методології тестування (наприклад, Agile, Scrum, Waterfall). Аналіз етапів тестового циклу та їх оптимізація. Визначення стратегії тестування (функціональне, навантаження, безпека тощо).
- Автоматизація тестування. Оцінка потреби у впровадженні автоматизації тестування. Вивчення використовуваних інструментів автоматизації.Розгляд покриття автоматизованих тестів.
- Взаємодія з розробкою (DevOps). Аналіз інтеграції QA в процес розробки (DevOps). Розгляд ролі QA у впровадженні Continuous Integration та Continuous Deployment (CI/CD).
- Управління багами. Оцінка системи відслідковування багів. Розгляд процесу виявлення, документації та вирішення багів. Перевірка використання стандартів у форматі опису багів.

- Тест-дизайн. Оцінка процесу розробки тест-кейсів та сценаріїв. Розгляд покриття тестування та виявлення можливих прогалин. Вивчення тестової документації.
- Тестування безпеки. Аналіз стратегії тестування безпеки. Визначення участі QA в аналізі загроз та вразливостей.
- Навчання та розвиток. Оцінка програми навчання для QA. Вивчення можливостей розвитку кар'єри в області тестування.

2.2.2 Вибір пріоритетних сценаріїв:

Вибір пріоритетних сценаріїв QA є важливою частиною стратегії тестування. Це допомагає спрямовувати зусилля на тести, які мають найбільший вплив на якість продукту або на тести, які представляють найбільший ризик. Нижче подано кілька кроків для вибору пріоритетних сценаріїв QA:

- Розуміння бізнес-процесів. Ознайомлення з бізнес-процесами та функціональністю продукту. Зверніть увагу на ті частини системи, які є критично важливими для користувачів або для виконання ключових бізнес-задач.
- Оцінка ризиків. Визначення потенційних ризиків та найбільш вразливих частин продукту. Карта користувача (User Journey Map). Створення карти користувача, що відображає шляхи користувачів через систему. Визначення критичних точок, де користувачі взаємодіють з ключовою функціональністю.
- Визначення критичних функцій. Оцінка функцій, які є критичними для основної функціональності продукту.
- Аналіз статистики використання. Збір даних про використання продукту, якщо вони доступні. Визначення найбільш популярних функцій та операцій, які використовуються користувачами найчастіше.

– Узагальнені тести. Розгляд тестів, які можуть виявити проблеми на різних частинах продукту. Ідентифікація тестів, які включають в себе різноманітні типи взаємодії та операції.
– Узгодження з командою розробників. Обговорення пріоритетів тестування з командою розробників. Оцініть їхню точку зору та отримайте інформацію про ті частини системи, які є найбільш важливими для їхньої розробки.

2.2.3 Створення тестових сценаріїв:

Створення тестових сценаріїв QA - це важливий етап у процесі тестування, який дозволяє визначити, як система повинна поводитися в різних умовах. Нижче наведено загальний підхід до створення тестових сценаріїв.

– Визначте вхідні дані, кроки виконання тесту та очікувані результати для кожного сценарію.
– Розуміння вимог. Ознайомлення з документацією, що містить вимоги до системи. Зверніть увагу на функціональні та нефункціональні вимоги.
– Визначення областей для тестування. Визначення тих частин системи, які потрібно протестувати. Оцінка критичних та чутливих областей, де можливі помилки можуть мати найбільший вплив.
– Вибір типів тестів. Визначення типів тестів, які підходять для кожної області (наприклад, функціональні тести, тести на навантаження, тести безпеки тощо).
– Визначення передумов та умов. Визначення передумов, які повинні бути виконані перед початком тесту. Визначення умов, які повинні бути задоволені для виконання тестового сценарію.
– Написання кроків тестування. Розробка конкретних кроків, які тестувальник повинен виконати для виконання сценарію. Зробіть кроки чіткими, зрозумілими та послідовними.

- Введення тестових даних. Специфікація необхідних тестових даних для кожного кроку. Переконайтеся, що тестові дані дозволяють вам перевірити різні варіанти використання системи.
- Визначення очікуваних результатів. Опис очікуваних результатів для кожного кроку тесту. Зазначте, які повинні бути виходи, і враховуйте можливі варіації.
- Оцінка покриття. Визначте, яке покриття тестів надає кожен сценарій вимогам та функціональностям.
- Підтримка тестових скриптів. Забезпечте можливість легко підтримувати та оновлювати тестові сценарії з плином часу.
- Тестування в реальних умовах. Використовуйте тестові сценарії для тестування продукту в реальних умовах перед випуском.

2.2.4 Робота з різними сценаріями:

Робота з різними сценаріями QA передбачає розгляд різних варіантів тестування для забезпечення повноцінної перевірки якості продукту. Нижче наведено різні види сценаріїв QA та їх характеристики:

- Функціональні сценарії. Перевірка основної функціональності продукту. Визначення того, як система повинна працювати за стандартних умов.
- Сценарії на межі. Тестування межових значень та екстремальних ситуацій. Виявлення проблем при використанні максимальних або мінімальних значень вхідних даних.
- Позитивні та негативні сценарії. Тестування сценаріїв з правильними та неправильними вхідними даними. Перевірка, як продукт реагує на коректне та некоректне введення.
- Сценарії відновлення (Recovery Scenarios). Відновлення системи після збою чи виникнення помилки. Перевірка того, як система обробляє та відновлюється після непередбачуваних ситуацій.

- Тестування взаємодії користувача. Моделювання різних способів взаємодії користувача з системою. Визначення, як користувачі використовують продукт та як система реагує на їхні дії.
- Тестування продуктивності. Моделювання роботи системи при великому навантаженні. Оцінка швидкодії, ефективності та стабільності системи за різних умов.
- Тестування сумісності. Використання різних конфігурацій апаратного та програмного забезпечення. Визначення того, як продукт взаємодіє з різними платформами та середовищами.
- Тестування безпеки. Виявлення потенційних загроз та вразливостей. Забезпечення надійності та стійкості системи до атак та несанкціонованого доступу.
- Тестування оновлень та виправлень. Перевірка, як система взаємодіє з новими версіями або виправленнями. Як гарантування того, що оновлення не порушать роботу системи та вирішують виявлені проблеми.

Комбінація різних сценаріїв дозволяє виявляти різноманітні види проблем та покращувати якість продукту.

2.2.5 Визначення критеріїв успішності:

Критерії успішності тестового сценарію визначаються на основі очікуваних результатів та вимог до функціональності продукту. Нижче наведено кілька загальних критеріїв, які можна використовувати для оцінки успішності тестового сценарію:

- Відповідність Функціональності. Сценарій вважається успішним, якщо він виконує всі передбачені функції та завдання відповідно до вимог.
- Коректність Результатів. Результати, отримані в ході виконання сценарію, повинні бути правильними та відповідати очікуваним.

- Стабільність Тестів. Сценарій повинен бути стійким та надійним, і він не повинен призводити до системних або критичних помилок.
- Час Виконання. Час виконання тестового сценарію повинен бути в межах прийнятного обсягу та не перевищувати встановлені ліміти.
- Сумісність з Різними Платформами та Браузерами. Якщо тестовий сценарій призначений для роботи на різних платформах або браузерах, він повинен взаємодіяти коректно на всіх цих середовищах.
- Відсутність Помилок. Сценарій вважається успішним, якщо він не призводить до появи критичних помилок або суттєвих дефектів.
- Зручність для Користувача. Якщо сценарій включає в себе взаємодію з користувачем, важливо переконатися, що ця взаємодія є зручною та легко зрозумілою.
- Відповідність Специфікаціям. Сценарій повинен відповідати всім визначеним специфікаціям та вимогам, встановленим для функціональності, яку він тестує.
- Заповнення Логів та Звітності. Успішний сценарій повинен заповнювати логи та генерувати звіти, що дозволяє виявити та вирішити проблеми, якщо вони виникають.
- Відсутність Побічних Ефектів. Тестовий сценарій не повинен впливати на інші частини системи та не викликати непередбачувану поведінку.

Ці критерії допомагають забезпечити, що тестові сценарії виконують своє завдання ефективно та відповідають очікуванням щодо якості продукту.

2.2.6 Створення даних для тестів:

Створення даних для тестів — це важлива частина процесу тестування, оскільки тестові дані визначають вхідні параметри, на яких буде виконуватися тест. Цей процес може включати в себе наступні етапи:

- Визначення Потрібних Даних. Зрозумійте, які дані необхідні для виконання тестового сценарію чи тестової кейсу. Це можуть бути дані введення, параметри конфігурації, стан системи тощо.
- Створення Тестових Даних. Створіть або згенеруйте тестові дані, які будуть використовуватися для виконання тестів. Це може включати в себе створення записів у базі даних, введення у форми, завантаження файлів тощо.
- Використання Зразків Даних. Використовуйте зразки даних, які покривають різні варіації тестових сценаріїв. Наприклад, якщо тестуєте форму реєстрації, створіть дані для тестування різних видів введення (валідних та невалідних).
- Генерація Реальних Даних. Використовуйте реальні дані там, де це можливо, для наближення тестового середовища до реального використання продукту.
- Збереження Тестових Даних. Забезпечте можливість зберігати тестові дані, щоб їх можна було використовувати у майбутніх тестах або сценаріях.
- Ініціалізація Даних перед Тестами. Впевніться, що тестові дані ініціалізуються вірно перед виконанням тестів. Це може включати в себе встановлення певних станів системи чи завантаження попередньо визначених даних.
- Автоматизація Створення Даних. У випадках, коли тестові дані можуть бути генеровані автоматично, розгляньте можливість автоматизації цього процесу. Це може бути корисним у великих проектах або при регулярних випусках тестів.
- Очищення Даних після Тестів. Після завершення тестів важливо очистити тестові дані, щоб забезпечити чистоту тестового середовища та уникнути впливу тестів один на одного.

Забезпечення належного створення та управління тестовими даними є важливим елементом ефективного тестування, оскільки правильні тестові дані дозволяють точніше оцінити функціональність продукту та виявити потенційні проблеми.

2.2.7 Підтримка та оновлення:

Підтримка та оновлення автоматизованих тестів - це важливий етап у життєвому циклі тестового проекту. Це дозволяє забезпечити стабільність тестової сутності під час змін у програмному продукті або його середовищі. Ось кілька ключових практик для ефективної підтримки та оновлення автоматизованих тестів:

- Регулярне Оновлення Тестового Коду. Важливо періодично переглядати та оновлювати тестовий код, особливо при внесенні змін до функціональності продукту.
- Застосування Прийомів Рефакторингу. Використовуйте прийоми рефакторингу для поліпшення структури та читабельності тестового коду без зміни його поведінки.
- Ретельне Збереження Ідентифікаторів Елементів. Якщо тести використовують селектори для знаходження елементів на сторінці, важливо вести облік змін у структурі HTML та CSS, щоб уникнути зламу тестових сценаріїв через зміни в макеті.
- Аналіз Помилок та Видозміни Тестів. Розглядайте звіти про помилки та фіксуйте тестові сценарії, які виявляють проблеми. Важливо вносити зміни в тестовий код, щоб покрити новий функціонал чи виправити виявлені помилки.
- Оновлення Залежностей та Бібліотек. Періодично перевіряйте та оновлюйте версії залежностей та бібліотек, які використовуються в тестовому проекті. Це дозволяє користуватися останніми функціональностями та виправленнями помилок.
- Створення Служби Підтримки. Розглядайте можливість створення служби або групи, відповідальної за підтримку та розвиток тестових скриптів.
- Запуск Тестів в Режимі CI/CD. Інтегруйте автоматичний запуск тестів в режимі Continuous Integration (CI) або Continuous Delivery (CD). Це дозволяє вам вчасно виявляти проблеми зі зламом тестів під час розробки.

- Автоматизована Генерація Звітів. Використовуйте автоматизовані засоби для генерації звітів про виконання тестів, які детально описують стан тестів та їх результати.
- Супровід Документації. Додавайте та оновлюйте документацію до тестів, щоб інші члени команди могли легко розуміти їх призначення та використання.
- Тестування на Різних Конфігураціях. Впевніться, що тестові сценарії проходять на різних конфігураціях (різні браузери, операційні системи), якщо це необхідно для вашого продукту. Належна підтримка та оновлення тестів допомагають забезпечити ефективну та стабільну тестову базу при розвитку програмного продукту.

Визначення тестових сценаріїв є ключовим етапом у процесі автоматизації тестування, оскільки це дозволяє створити систематичний та повний покриття для вашого веб-додатку.

3 ОБМЕЖЕННЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

Існує кілька причин, чому автоматизація може не виправдати своїх очікувань. І всі вони пов’язані з неправильними рішеннями в управлінні чи інженерії, а іноді й у обох.

найбільші помилки:

- спроба зменшити витрати на автоматизаторів. Менеджер не правий, якщо думає, що може відправити своїх тестувальників на курси Selenium і отримати автоматизацію.
- спроба впровадити автоматизацію без чітко продуманої стратегії та планування Крім того, ситуація може погіршитися, коли автоматизація з метою автоматизації: «Незрозуміло навіщо, але потрібно».
- Занадто пізній старт: тести починають автоматизувати тільки тоді, коли тестувальники вже зовсім загинаються

«Інженерні рішення» - це рішення, які інженери приймають під час розробки та впровадження стратегії автоматизації. Це включає вибір інструментів, типів тестування, архітектури та інші елементи[7].

Розглянемо кілька питань з інженерної точки зору. Чому автоматизація тестів користувача лише шкідлива:

Рішення робити автоматизацію тестів лише за допомогою графічного інтерфейсу є найпоширенішою помилкою. В момент прийняття такого рішення здається, що воно зовсім не погане. Іноді воно займає досить тривалий час, щоб завершити певні завдання. Якщо продукт вже знаходиться в стадії підтримки і більше не розвивається, це іноді може бути цілком достатньо. Але, як правило, це не найкращий підхід для проектів, які активно розвиваються в довгостроковій перспективі.

UI-тести — це те, що роблять тестувальники, це природний шлях тестування програми. Крім того, це демонструє, як користувачі будуть користуватися

додатком. Здавалося б, це найкращий варіант і саме той, який слід використовувати в автоматизації перш за все. Але є певна деталь, як кажуть:

– UI-тести нестабільні;
– UI-тести повільні.

Вони нестабільні через те, що тести залежать від «верстки» інтерфейсу програми. Тести можуть зламатися, якщо змінити розташування кнопок на екрані або додати або видалити елемент. Логіка тесту зміниться, якщо інструмент автоматизації не зможе знайти необхідний елемент або натисне неправильну кнопку.

У міру того, скільки таких тестів ви проходите, тим більше часу доводиться витрачати на виправлення та підтримку. Як наслідок, через часті помилково-позитивні спрацьовування довіра до результатів таких тестів знижується. Автоматизатор весь час витрачає на відновлення пошкоджених скриптів, і більше нічого не створюється.

Ці тести повільні через повільний інтерфейс програми, це вимагає перемальовування, перезавантаження ресурсів, очікування появи даних і так далі. Чекати для тестового скрипта займає більшу частину часу. Крім того, чекати — це марнотратство часу. Крім того, тест може бути невдалим, оскільки він вже намагається використати елемент, який ще не встиг промальюватися на швидкій інтерфейсній інтерфейсі. Таку автоматизацію дуже складно використовувати як показник якості в щоденній практиці, оскільки прогін UI-сценаріїв займає дві доби, навіть за умови запуску незалежних груп тестів одночасно на декількох серверах.

Що робити для стабілізації? Незважаючи на те, що проблема нестабільності не складна для вирішення, проте автоматизатори часто навіть не намагаються її вирішити.

У будь-якому випадку першим кроком має бути розмова з розробниками, щоб вони не забували прописати для елементів унікальні характеристики, за допомогою яких інструмент автоматизації може їх точно визначити. Таким чином, ви повинні

максимально відмовитися від п’ятиповерхових xPath-виразів або CSS-селекторів і, якщо це можливо, використовувати унікальні ідентифікатори, назви та інші ідентифікатори в будь-якому місці. Це повинно бути чітко описано в девелопмент-гайдах і включатися в опис виконаного для розробників.

Якщо нестабільність не так складна, проблема повільних тестів повинна вирішуватися повністю, оскільки вона впливає на весь процес розробки.

Перше і найпростіше рішення, яке може прискорити процес, — це деплоїти програму та запускати тести на більш «швидкому» залізі. Це дозволяє уникнути проблем, пов’язаних із затримками мережі, які сповільнюють взаємодію між програмою та тестом. Таким чином, проблему можна «вирішити» за допомогою заліза та архітектури тестового стенду. Це вже може забезпечити значну економію за часом у кілька разів.

З самого початку необхідно включити можливість паралельного та незалежного запуску в дизайн тестового фреймворку та тестових кейсів. Паралельність тестових запусків значно зменшує час виконання. Насправді, тут теж існують обмеження. По-перше, логіка тестової програми не завжди дозволяє йому тестувати в кілька потоків. Хоча такі ситуації досить незвичайні та рідкісні, вони відбуваються. По-друге, неможливо паралелити до безкінечності, що робить все залежним від заліза.

Третє, і найбільш радикальне рішення, — створити якомога менше UI-тестів. Зменшивши кількість тестів, ми можемо швидше дізнатися про результати їхнього прогону.

Піраміда — це дуже зручна метафора, вона наочно показує бажану кількість автоматизованих тестів щодо кожного з рівнів архітектури системи. Повинно бути багато низькорівневих юніт-тестів і зовсім мало високорівневих UI-тестів. Питання в тому, чому саме так і чому всі так носяться з цією пірамідою[7]?

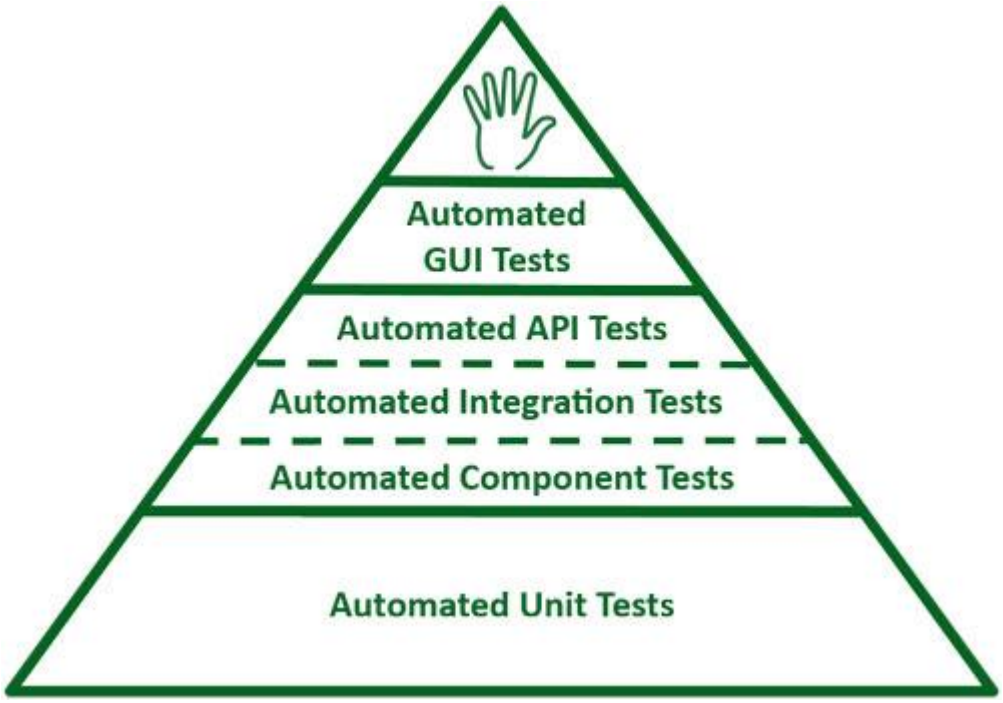

Рисунок 3.1 – Піраміда тестування

Це дуже просто. Згадаймо, як зазвичай виглядає процес знаходження і виправлення проблеми в програмі, коли його тестують вручну. Розробник спочатку вносить нові зміни в код. Тестувальник чекає, коли новий білд буде зібраний і поставлений на тестовий стенд. Тестувальник проводить тестування, визначає проблему та запускає тікет у системі відстеження багів. Розробник реагує на цей тікет і виправляє проблему негайно. Це нові кодові зміни, а потім білд, деплой і ретест. Тікет відкривається, коли все добре. Деякі проблеми вимагають від декількох годин до декількох діб, якщо не тижнів, щоб вирішити.

Інша історія, коли є автоматизовані тести, які використовують API для спілкування з бек-ендом додатки. Тут вже є цікаві варіанти:

– Усі тести проводяться на повністю задеплоєному додатку, якому відключено всі зовнішні системи. У чистих UI-тестах набагато менше хибно-позитивних спрацьовувань, що значно скорочує час виконання та аналіз результатів. Все інше схоже на UI-тести.

- Тести проводяться після складання білда, але без деплою на тестовий стенд; для зовнішніх систем використовуються заглушки. Розробник запускає тести в середовищі збірки білда, і якщо проблеми виникають, вони часто не вимагають створення тікетів. Це відбувається тому, що розробник змінює код і фіксує результати негайно. У цьому випадку швидкість виявлення та вирішення проблеми є значною.
- Крім того, безсумнівно, «вершки» включають компонентні та юніт-тести. Вони працюють миттєво після компіляції модуля, не виходячи з улюбленої IDEшки, і не вимагають збірки всього проекту. У хвилинах вимірюється час, необхідний для внесення змін і виправлення потенційних проблем.
- Вочевидь, відповідні автотести будуть виконуватися швидше, чим нижче спускатися пірамідою. Таким чином, є можливість проганяти набагато більше тестів одночасно. Відповідно, ефективніші тести, що стосуються часу відгуку та площі покриття, можна створювати на рівні нижче.

Важливо розуміти, що юніт-тести тестують код, щоб розробники могли бути впевнені, що частина коду працює так, як передбачено, і, найважливіше, що код розробника не ламає логіку інших кодів. Це тому, що юніт-тести включені в код колеги, і розробник запускає ці тести перед коммітом у репозиторій. UI-тести перевіряють систему в цілому, тобто те, що використовує користувач. Критично важливо мати доступ до таких тестів.

Загальна порада тут одна: на кожному рівні потрібно проводити достатню кількість різних типів автотестів. Тоді є шанс отримати хорошу віддачу від таких тестів.

Таким чином, швидкі тести та чудове покриття за нормальних витрат розробки та підтримки можна отримати, якщо виконувати рекомендації піраміди.

3.1 Вартість та трудомісткість автоматизації тестування

Вибір інструментів:

– Вартість інструментів автоматизації тестування може коливатися від безкоштовних (наприклад, Selenium, JUnit) до комерційних (наприклад, HP UFT, Ranorex). Комерційні інструменти можуть вимагати значних витрат.

Трудомісткість:

– Розгортання та навчання з використанням нових інструментів також може вимагати додаткових зусиль та часу.

Вибір підходів до тестування:

– Вартість може змінюватися в залежності від того, чи використовується ручне тестування, автоматизоване тестування на рівні інтерфейсу користувача, автоматизоване тестування на рівні API, чи інші підходи.
– Впровадження нових методологій або підходів (наприклад, DevOps, Continuous Testing) може знадобитися додатковий час та зусилля для забезпечення їх успішної реалізації.

Автоматизація ручних тестів чи створення нових:

– Автоматизація вже наявних ручних тестів може бути вартісною, але створення нових автоматизованих тестів може вимагати більше часу та ресурсів.
– Переписання та адаптація існуючих тестів для автоматизації може вимагати ретельного вивчення їх структури та логіки.

Супровід та підтримка:

– Витрати на супровід і підтримку автоматизованих тестів можуть включати в себе витрати на їхнє поновлення під час змін в програмному продукті або середовищі.
– Навчання членів команди на підтримку та розробку нових тестів може бути трудомістким завданням.

Інфраструктура та ресурси:

– Розгортання та управління інфраструктурою для виконання автоматизованих тестів, таких як сервери, хмарні ресурси, може бути вартісним завданням.
– Підтримка та масштабування інфраструктури для тестування може вимагати зусиль команди.

Тестування нових функцій чи змін в програмному продукті:

– Автоматизація тестування нових функцій або вносити зміни в програмний продукт може вимагати великих витрат, оскільки потрібно розробити та підтримувати нові тести.
– Адаптація існуючих тестів до нового функціоналу може бути трудомісткою задачею.

Навчання та підтримка команди:

– Витрати на навчання команди щодо використання нових інструментів та методологій.
– Навчання та підтримка членів команди може займати значний час.

3.2 Автоматизація тестування не гарантує 100% пошук дефектів

Автоматизація тестування є потужним інструментом для покращення ефективності тестування, зменшення часу виконання тестів та покращення точності, але важливо розуміти, що вона не може гарантувати 100% виявлення всіх дефектів у програмному продукті. Деякі ключові причини цього обмеження включають:

Неавтоматизовані аспекти тестування:

– Автоматизовані тести часто орієнтовані на певні аспекти програми, зазвичай ті, які можна легко автоматизувати. Проте, багато аспектів тестування, таких як оцінка інтерфейсу користувача, взаємодія з реальними користувачами, тестування продуктивності, не завжди можна автоматизувати в повному обсязі.

Неспроможність виявлення нових та унікальних дефектів:

– Автоматизовані тести зазвичай спрямовані на виявлення відомих проблем та згідно з заданими сценаріями. Вони можуть не бути ефективними у виявленні нових та унікальних дефектів, які можуть виникнути в процесі реального використання програмного продукту.

Відсутність тестування в реальних умовах:

– Деякі дефекти можуть виявитися лише в реальних умовах експлуатації, які важко або неможливо автоматизувати. Наприклад, помилки, пов'язані з мережевим взаємодією, масштабованістю, довгостроковою стабільністю тощо.

Природа змін у програмному продукті:

– Програмний продукт постійно розвивається, і зміни в коді можуть впливати на тести. Навіть дрібні зміни можуть викликати невдачі в автоматизованих тестах, і їх потрібно постійно оновлювати для відповідності новим вимогам.

Якщо ви виявили баг, не варто негайно заводити його в баг трекер з описом «нічого не працює!». Відтворіть дефект повторно. Знову відтворюється? А тепер мінімізуйте кількість кроків для відтворення, та переконайтесь, що немає нічого зайвого [6].

Якщо використовуються певні вхідні дані, переконайтесь що вони не містять зайвої інформації. Коли ви зрозуміли, які саме дані та які ваші дії призводять до проблеми, коротко сформулюйте її суть - придумайте тему (опис) баг-репорта.

«Кроки відтворення» – основне поле для заповнення в баг-репорті. Запишіть кроки, які було визначено. Як вже було сказано, кількість кроків має бути необхідною та достатньою для відтворення проблеми. Зайві - не пишіть, необхідні – теж не пропускайте.

Щоб заводити дефекти правильно, необхідні технічна кваліфікація та розуміння архітектури продукту, що тестується. Якщо ми говоримо про веб-тестування, то слід вказати в баг-репорті код помилки, який повертає сервер, подивитись подробиці за допомогою «Firebug» та надати детальну інформацію.

Ви знайшли баг, але не можете повторно його відтворити. Що робити в таких ситуаціях? Заводити такий дефект чи ні?

По-перше, ви повинні бути впевнені в тому, що насправді виконуєте ті ж кроки, що й минулого разу. Можливо щось було упущено.

Основні причини, чому дефект не відтворюється повторно:

– Дефект відтворюється тільки при першому запуску додатка. Для повторного відтворення необхідно знову виконати інсталяцію додатка.
– Використовуються старі дані системи, в які не були внесені зміни в процесі тестування.
– Швидкість відтворення дефекта. Швидкість виконання кроків для відтворення повинна бути ідентичною.

Бувають ситуації, коли у вас дефект відтворюється з вірогідністю 100%, але розробник не може відтворити у себе даний дефект. В таких випадках необхідно звернути увагу на оточення тестованого продукта:

– версію системи;
– версію компонентів;
– стан системи та даних.

Незважаючи ні на що, дефекти, які не відтворюються повторно, все ж таки потрібно заводити в баг-трекер. Важливо при цьому не забувати вказувати відповідне значення відтворюваності дефекта (Reproducibility).

Для запобігання проблем з відтворенням дефектів та наданням команді розробників максимальної кількості інформації про дефект, необхідно використовувати різні варіанти моніторингу дій:

– Скрінкаст (англ. screen - екран, broadcasting - трансляція) – це запис відеозображення з екрана комп’ютера чи іншого цифрового пристрою. Це один із найбільш ефективних варіантів поділитися тим, що відбувається на екрані монітора. Таким чином тестувальнику легко наочно показати помилки в роботі будь-якого програмного продукту. Знімають скрінкасти

спеціальними програмами, наприклад: Snagit, Movavi Screen Capture, Jing, Reflector, ADB Shell Screenrecord, AZ Screen Recorder.

– Live-логування – це зняття системних логів в режимі реального часу. Лог-файли (журнал подій, Log) - це файли, що містять системну інформацію роботи сервера або комп'ютера в хронологічному порядку, в які вносяться певні дії користувача або програми. Для цього можуть використовуватися наступні програми: Visual Studio для Windows, iMazing для iOS, XCode для MacOS, Android Debug Monitor для Android.
– Створюйте нотатки. Зберігайте все, що ви знаєте про дефект, який виник. Тоді вся потрібна інформація буде у вас під рукою, коли ви підключите розробників до роботи над цією проблемою.
– Рекордер дій. Існують програми, які дозволяють повторювати всі записані рухи мишки та дії, виконані на клавіатурі ПК. Прикладом такої програми є Advanced Key and Mouse Recorder. Нажаль даний метод актуальний лише для десктопних платформ, для мобільних платформ аналогічних програм поки що немає.

Помилки в програмному забезпеченні, які не відтворюються або відтворюються випадково, ускладнюють роботу тестувальників.

Як правило, мало дефектів можуть бути невідтворюваними для кваліфікованого, досвідченого тестувальника, але для тестувальника-початківця пошук пояснення і кроків відтворення помилки, яка випадково відтворюється може бути досить складним завданням. Однак деякі програмні помилки, знайдені під час виконання тестування програмного забезпечення, важко відтворити та пояснити навіть спеціалістам в області тестування. В цьому випадку рекомендується передивитися свою роботу і спробувати подивитися на ситуацію з нової точки зору.

У випадках, коли здається, що помилку відтворити неможливо, необхідно:

– відкинути теорії, які не підходять;
– розробити нові теорії або змінити існуючі;

– подивитись всі записи та шукати те, що було упущено або проігноровано в попередніх теоріях.

Корисно також використовувати деякі наслідкові методи виконання тестування програмного забезпечення.

Згідно одному з них потрібно додавати або виключати один або декілька факторів за раз, та дивитись як це впливає на частоту виникнення помилки, яка випадково відтворюється.

Таким чином, дотримуючись цих простих кроків з відтворення дефектів, ви зможете не тільки виявити дефект, але й звузити область пошуку помилки розробником для успішного подальшого виправлення.

Обмеження в дизайні тестів:

– Якщо тестові сценарії не відповідають реальним умовам використання програмного продукту, то автоматизовані тести можуть бути обмеженими у здатності виявляти дефекти в реальних умовах.

Підтримка крос-платформенних або крос-браузерних тестів:

– Автоматизація тестів для різних платформ та браузерів може бути складною через різницю в їхніх характеристиках та взаємодії.

Помилки у розробці тестових скриптів:

– Навіть якщо сам програмний продукт бездоганний, помилки в тестових скриптах або їхній логіці можуть призводити до неправильних результатів.

Автоматизація тестування повинна розглядатися як складова частина повного стратегічного підходу до тестування, а не як універсальний інструмент для виявлення всіх можливих дефектів. Комбінування автоматизованих тестів із ручним тестуванням та іншими техніками дозволяє забезпечити більш повну та ефективну покриття тестування.

3.3 Стандартизація та управління автоматизацією в тестуванні

Стандартизація та управління автоматизацією в тестуванні визначають важливі принципи та практики для забезпечення ефективності, стабільності та доступності автоматизованих тестів на проекті. Ця тема включає в себе розробку і впровадження стандартів у вигляді спільних правил, процедур та інструментів для всього відділу тестування.

Стандартизація коду:

– Визначення спільних стилів та конвенцій для написання коду автоматизованих тестів. Це сприяє зрозумілості коду, полегшує обмін інформацією між членами команди та робить код більш схожим за структурою та стилем.

Вибір та впровадження єдиної платформи:

– Установлення стандартів для вибору та використання єдиних інструментів автоматизації тестування. Це включає в себе фреймворки, бібліотеки, інструменти для тестування інтерфейсу користувача, API, тощо.

Розробка загальних стратегій тестування:

– Розробка та визначення загальних стратегій автоматизації тестування для впровадження на всьому проекті. Це включає в себе підходи до тестування різних рівнів, обсяг тестування, покриття тестів та інші важливі аспекти.

Введення систем управління конфігурацією:

– Впровадження систем управління конфігурацією для зберігання та відстеження версій автоматизованих тестів та вихідного коду. Це дозволяє забезпечити консистентність та стабільність усіх компонентів тестування.

Навчання та взаємодія команди:

– Встановлення стандартів для навчання та взаємодії членів команди в питаннях автоматизації тестування. Це може включати проведення тренінгів, обмін досвідом та встановлення ефективних комунікаційних каналів.

Моніторинг та звітність:

– Розробка стандартів моніторингу та звітності за результатами автоматизованих тестів. Це включає в себе визначення метрик, які вказують на ефективність тестування, час виконання, покриття коду тестів та інші ключові показники.

Забезпечення змінності спеціалістів:

– Розробка системи, що забезпечує сменяємість спеціалістів, які працюють з автоматизацією тестування. Це включає в себе документацію, яка описує правила та процедури, а також докладні коментарі в коді тестів.

Підтримка розвитку та апгрейду:

– Створення системи для підтримки постійного розвитку та апгрейду автоматизованих тестів. Це може включати в себе рутинне оновлення тестових скриптів для відповідності змінам у програмному продукті та виправлення помилок.

Завдяки стандартизації та управлінню автоматизацією в тестуванні, команди можуть досягати більшої стабільності, швидкості та ефективності в процесі розробки та тестування програмного продукту.

3.4 Труднощі з тестуванням асинхронних або динамічних аспектів в автоматизованому тестуванні

Асинхронні або динамічні аспекти в програмному продукті можуть стати викликом при автоматизації тестування через їхню непередбачуваність та змінливість станів. Такі аспекти включають асинхронні запити, динамічне завантаження елементів інтерфейсу користувача, анімації та інші взаємодії, які виникають в реальному часі. Ось деякі труднощі, які можуть виникнути при тестуванні таких аспектів:

Очікування завантаження елементів:

– Динамічне завантаження елементів важко передбачити, і тестувальник може стикнутися з труднощами визначення часу очікування завантаження. Це може призвести до невдач тестів через недоступність елементів для взаємодії.

Обробка асинхронних подій:

– Зміни стану асинхронних об'єктів або подій можуть бути складні для відстеження в тестових сценаріях. Неправильне управління асинхронними подіями може викликати непередбачувані результати.

Визначення часових меж:

– Встановлення часових меж для очікування або взаємодії з елементами може бути складним завданням через змінливий час завантаження або затримки в асинхронних операціях.

Тестування асинхронних операцій на фоні:

– Сценарії, які виконують асинхронні операції на фоні, можуть ускладнити контроль за їхнім станом та правильністю виконання. Тестування в таких умовах може вимагати додаткових стратегій.

Динамічне оновлення сторінки або інтерфейсу:

– Анімації та динамічні оновлення можуть впливати на виявлення елементів та їх стан під час виконання тестів. Це може вимагати додаткового розуміння механізмів оновлення сторінок та коректного управління очікуванням.

Взаємодія з динамічно генерованим контентом:

– Тестування динамічно генерованих елементів або контенту може бути важким, оскільки їхні структури та ідентифікатори можуть змінюватися з кожним запуском тестів або під час взаємодії з користувачем.

Обробка винятків та помилок:

– Асинхронні операції можуть викликати винятки та помилки, які можуть бути важко передбачити. Ефективне оброблення таких ситуацій у тестових сценаріях є важливим аспектом.

Труднощі при відлагодженні:

– Відлагодження асинхронного коду може бути складним. Знаходження точки, де виникає проблема, та відстеження зміни стану можуть вимагати додаткових інструментів та стратегій.

Розв'язання цих труднощів вимагає ретельного планування та використання відповідних технік тестування, таких як встановлення коректних очікувань, використання правильних стратегій очікування асинхронних подій та врахування непередбачуваних затримок при розробці автоматизованих тестів.

3.5 Техніки тест дизайну

Розглянемо кілька основних методик, проте пам'ятатимемо, що найчастіше їх використовують у комплексі. Однієї техніки може бути недостатньо, оскільки вона не забезпечить максимального охоплення тестових сценаріїв[11].

Еквівалентне розбиття:

Метод еквівалентного розбиття дозволяє мінімізувати число тестів, не створюючи сценарій для кожного можливого значення, а вибравши лише одне значення з цілого класу і прийнявши за аксіому, що для всіх значень цієї групи результат буде аналогічним.

Наприклад, ми тестуємо функціональність програми, що дозволяє купувати авіа та залізничні квитки онлайн. Вартість квитка залежатиме від віку пасажира, оскільки діти, студенти та пенсіонери належать до пільгових категорій.

У нас є чотири вікові групи: молодше 15 років, від 15 до 25 років, старше 25 і молодше 60 років і люди старше 60. При цьому в полі для введення віку міститься всього два символи, тому вказати вік понад 99 років технічно неможливо.

QA-фахівцеві не потрібно писати 99 тестів для кожного віку, вистачить п'яти: по одному для кожної вікової групи (скажімо, 10, 18, 35 та 75 років) та один для випадку, якщо вік людини перевищує 99 років. Так, останній тест на практиці нездійсненний (оскільки в полі віку неможливо ввести більше двох знаків), та все ж не слід забувати про цю перевірку.

Рисунок 3.2 – Приклад еквівалетного розбиття

Граничні значення:

Техніка граничних значень базується на припущенні, що більшість помилок може виникнути на межі еквівалентних класів. Вона тісно пов'язана із вищеописаною технікою еквівалентного розбиття, через що часто використовується з нею в парі. Тоді для прикладу з попереднього пункту кордонами будуть значення 0, 15, 25, 60 і 99. Граничними значеннями будуть 0, 1, 14, 15, 16, 24, 25, 26, 59, 60, 61, 98, 99, 100 .

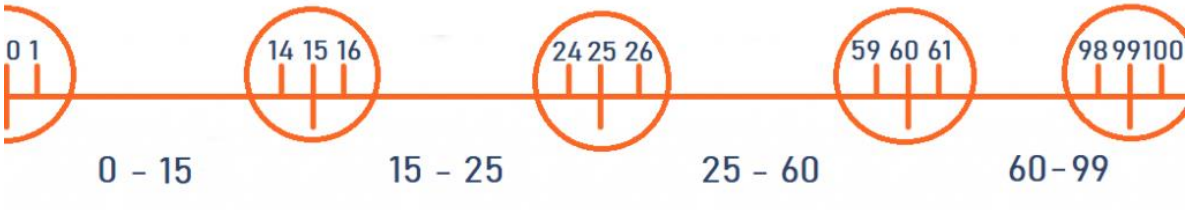

Рисунок 3.3 – Приклад граничних значень

Часто складнощі виникають, якщо вікові категорії вказані «внахлест», наприклад, 0-12, 12-25 років і т.д.

Таблиця прийняття рішень:

Інша назва методу – матриця ухвалення рішень. Ця техніка підходить для складніших систем, наприклад – двофакторної аутентифікації. Припустимо, щоб увійти в систему, користувачеві потрібно ввести спочатку логін і пароль, а потім підтвердити свою особу надісланим в смс кодом.

Які можливі сценарії:

– Правильний логін та правильний пароль.
– Правильний логін, неправильний пароль.
– Неправильний логін, правильний пароль.
– Неправильний логін, неправильний пароль.

Перший із цих сценаріїв супроводжується або правильним, або неправильним введенням смс-коду, разом у нас виходить 5 тестів. При цьому лише один із сценаріїв призведе до позитивного результату (користувач успішно авторизується), а решта закінчиться невдачею.

Однак, може бути так, що система видає різні повідомлення в залежності від того, на якому етапі була допущена помилка, скажімо: invalid login, invalid password. Відповідно, груп знадобиться більше, а таблиця стане більшою.

Цей метод хороший тим, що він показує відразу всі можливі сценарії у формі, зрозумілій навіть фахівця.

Условия	Значения	Правила				
Логин	Верный, Неверный	Верный	Верный	Верный	Неверный	Неверный
Пароль	Верный, Неверный	Верный	Верный	Неверный	Неверный	Верный
Код	Верный, Неверный, Не пришел	Верный	Неверный	Не пришел	Не пришел	Не пришел
Действия	Пользователь вошел на сайт Пользователь не авторизован	Пользователь вошел на сайт	Пользователь не авторизован	Пользователь не авторизован	Пользователь не авторизован	Пользователь не авторизован

Условия	Значения	Правила			
Логин	Верный, Неверный	Верный	Неверный	Верный, Неверный	Верный
Пароль	Верный, Неверный	Верный	Верный, Неверный	Неверный	Верный
Код	Верный, Неверный, Не пришел	Верный	Верный, Неверный, Не пришел	Верный, Неверный, Не пришел	Неверный
Действия	Пользователь вошел на сайт Пользователь не авторизован	Пользователь вошел на сайт	Пользователь не авторизован	Пользователь не авторизован	Пользователь не авторизован

Рисунок 3.4 – Приклад таблиці прийняття рішень

Передбачення помилок:

Використовуючи свої знання про систему, QA-фахівець може «передбачити», за яких умов є ризик помилок. Для цього важливо мати досвід, добре знати продукт та вміти побудувати комунікації з колегами.

Наприклад, у специфікації зазначено, що поле має приймати код із чотирьох цифр. Серед можливих тестів:

- Що станеться, якщо не ввести код?
- Що станеться, якщо не запровадити спецсимволи?
- Що станеться, якщо не ввести цифри, а інші символи?
- Що станеться, якщо запровадити не чотири цифри, а іншу кількість?

Переваги:

- Ця перевірка ефективна як доповнення до інших технік.
- Виявляє тестові випадки, які ніколи не повинні трапитися.

Недоліки:

- Техніка значною мірою заснована на інтуїції.
- Необхідний досвід у тестуванні подібних систем.
- Мале покриття тестами.

Зрозуміло, цей список далеко не повний і дає тільки найзагальніше уявлення про принципи тестування та техніки тест-дизайну. Наприклад, вичерпне тестування, що покриває всі можливі сценарії та виявляє всі помилки, існує лише теоретично. Це пов'язано з тим, що перевірка всіх параметрів та станів займе надто багато часу. Однак, чим досвідченіший QA-фахівець, тим кращих результатів він може досягти, і для цього важливо вміти правильно підбирати та комбінувати техніки.

Отже автоматизація тестування не є універсальним рішенням на всі випадки і не завжди виправдовує очікування. Вона може бути витратною та трудомісткою задачею, особливо на початкових етапах. Потрібно враховувати витрати на розробку та підтримку автоматизованих тестів. Вона не може гарантувати що всі

дефекти будуть виявлені, а для її успішної реалізації необхідно встановити стандартизовані підходи та ефективне управління. Також тестування динамічних або асинхронних елементів може бути складною задачею для автоматизованного тестування. Важливою частиною в автоматизованному тестуванні є використання правильних технік тест-дизайну, які допомагають створювати ефективні та надійні тестові набори для максимального покриття функціональності програмного забезпечення.

Завдяки знанням теорії та технікам тест дизайну, які наведені вище, було створено тест кейси, які дали змогу покрити весь необхідний функціонал та зменшити кількість тест кейсів, що значно заощадило час виконання обчислювального експерименту.

4 ОБЧИСЛЮВАЛЬНИЙ ЕКСПЕРИМЕНТ

4.1 Тестування логін форми

Тестування логін форми є критично важливою частиною тестування веб-додатків, оскільки логін - це один з основних етапів взаємодії користувача з системою. Вдале тестування логін форми допомагає забезпечити високу якість програмного продукту, зменшити ризики і підвищити надійність системи.

Розробити та виконати автоматизовані тести для перевірки функціональності логіну на веб-сайті. Впевнитися, що логін-функціонал працює коректно та надійно, а введені користувачем дані правильно обробляються системою.

Тестування успішного логіну:

- Натиснути кнопку “Sign in with email”
- Ввести коректний логін в поле “Email”.
- Натиснути кнопку “Next”.
- Ввести коректний пароль в поле “Password”.
- Натиснути кнопку “Sign in”.
- Перевірити, що користувач вдало увійшов на сайт.
- Перевірити коректність відображення аватару користувача після входу.

Тестування невдалого логіну (неправильний пароль або ім'я користувача):

- Натиснути кнопку “Sign in with email”
- Ввести коректний логін в поле “Email”.
- Натиснути кнопку “Next”.
- Ввести неправильний пароль в поле “Password”
- Натиснути кнопку “Sign in”.
- Користувач має отримати повідомлення “The email and password you entered don't match”
- Перевірити, що аватар користувача не відображається через помилку входу.

Очікувані результати:

- Всі тести успішно виконуються без помилок.

– Система коректно повідомляє про стан логіну (успішний чи невдалий).

Виконання дослідження:

Перший крок у виконані нашого автоматизованного тесту буде відкриття веб сайту який ми тестуємо, після чого ми можем побачити логін форму з якою нам доведеться працювати.

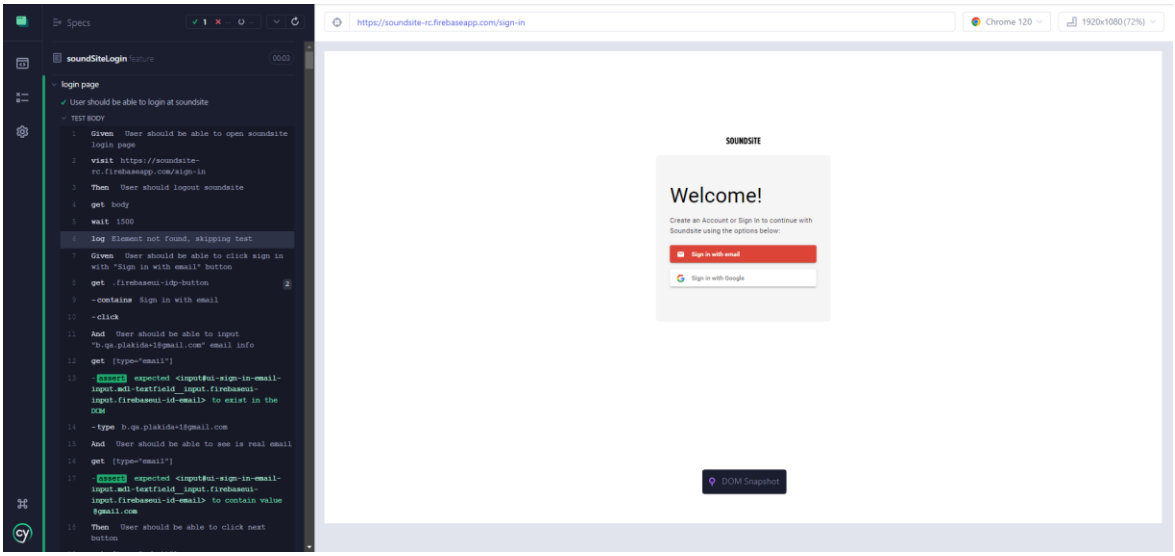

Рисунок 4.1 – Логін сторінка сайту

Після цього скрипт натисне кнопку “Sign in with email” та ми отримаємо нове вікно в якому побачимо два поля “Email” та “Password”. Скрипт автоматично заповнить ці два поля коректними даними які ми вказали в ході написання коду, після чого тасине кнопку “Sign in”.

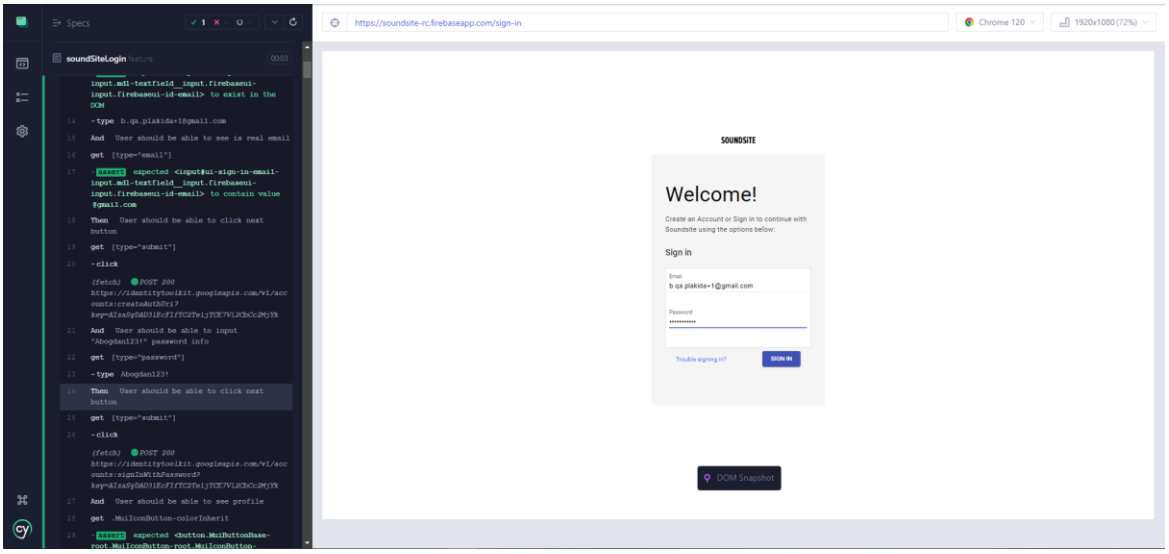

Рисунок 4.2 – Введення коректних даних

Після успішної авторизації ми опинимось на головній сторінці з нашими проектами. Це ми і можемо вважати успішним проходженням тесту для авторизації з коректними даними.

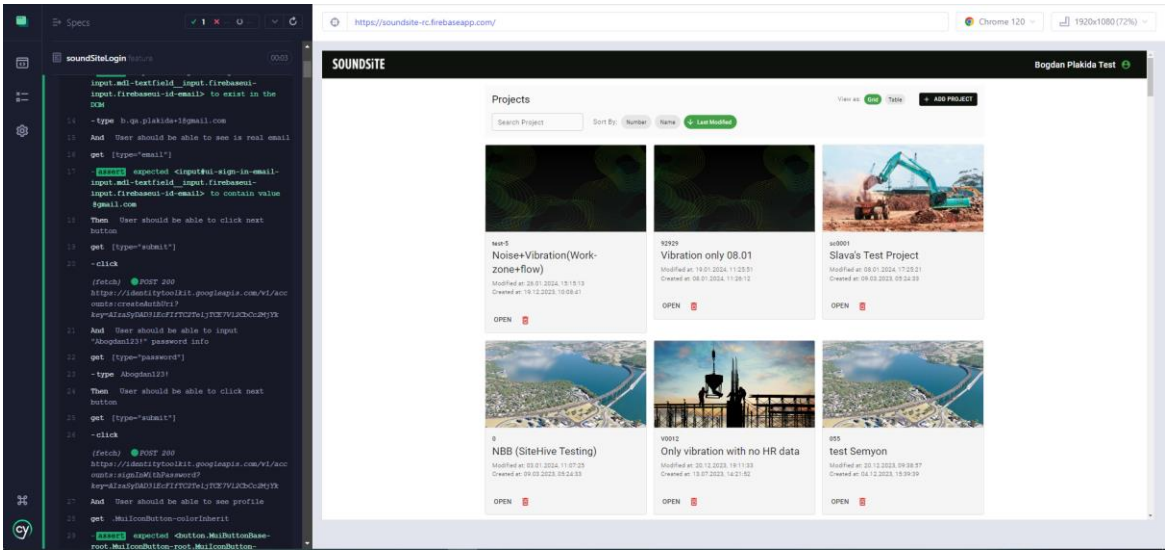

Рисунок 4.3 – Успішна авторизація

Так як, нам потрібно протестувати весь функціонал логін форми, ми також виконуємо схожі маніпуляції з кнопкою “Cancel”. Скрипт автоматично заповнить

поле “Email”, але в цей раз натисне кнопку “Cancel” після чого ми опинимось на головній сторінці логін форми, що і означатиме успішне проходження тесту.

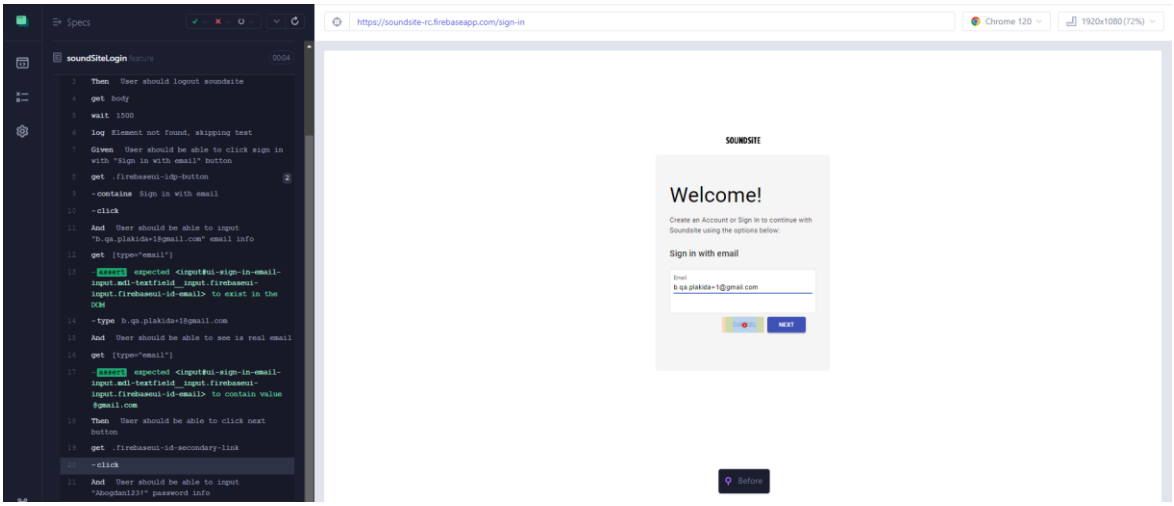

Рисунок 4.4 – Відміна авторизації

Дуже важливим кроком є перевірка негативного тест кейсу, в якому ми будем вводити некоректні дані, в нашому випадку некоректний пароль. Після того як наш скрипт заповнить всі поля та натисне кнопку “Sign in” ми отримаємо помилку, яка вкаже нам що пароль введено невірно, це свідчить про те що наш негативний тест пройшов успішно.

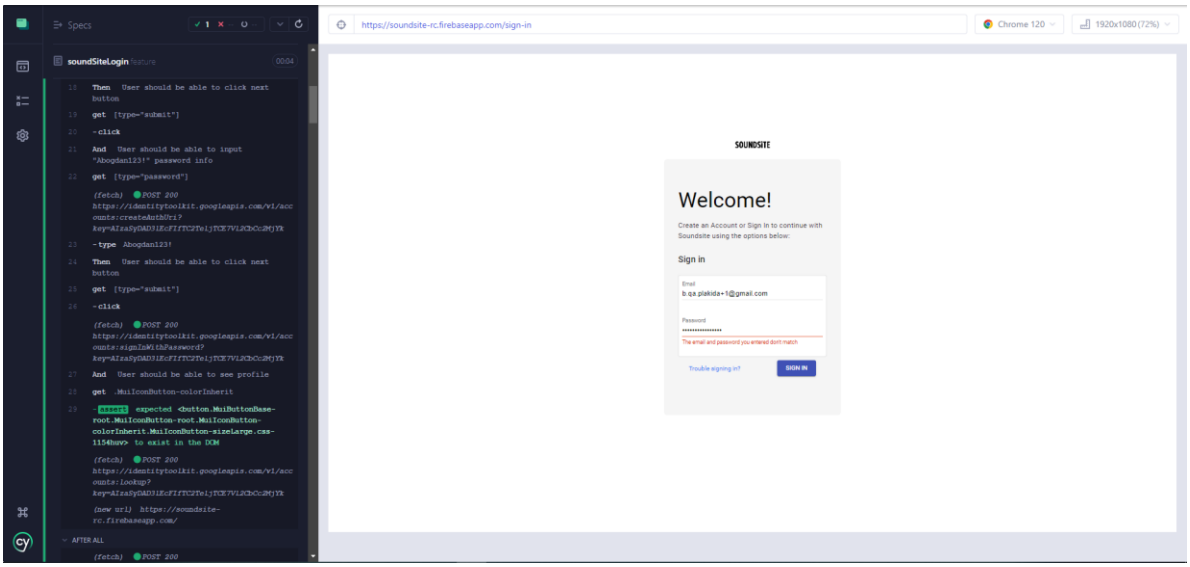

Рисунок 4.5 – Помилка при введенні некоректних даних

Останнім кроком в тестуванні логін форми буде введення неіснуючого логіна. Через специфіку сайту та умов його написання, ми побачимо вікно регістрації, яке запропонує нам створити аккаунт з таким логіном. Зазвичай ми б мали отримати помилку що такого юзеру неіснує, але враховуючи специфіку сайту, будемо вважати за успішне проходження тесту вікно з пропозицією створити аккаунт.

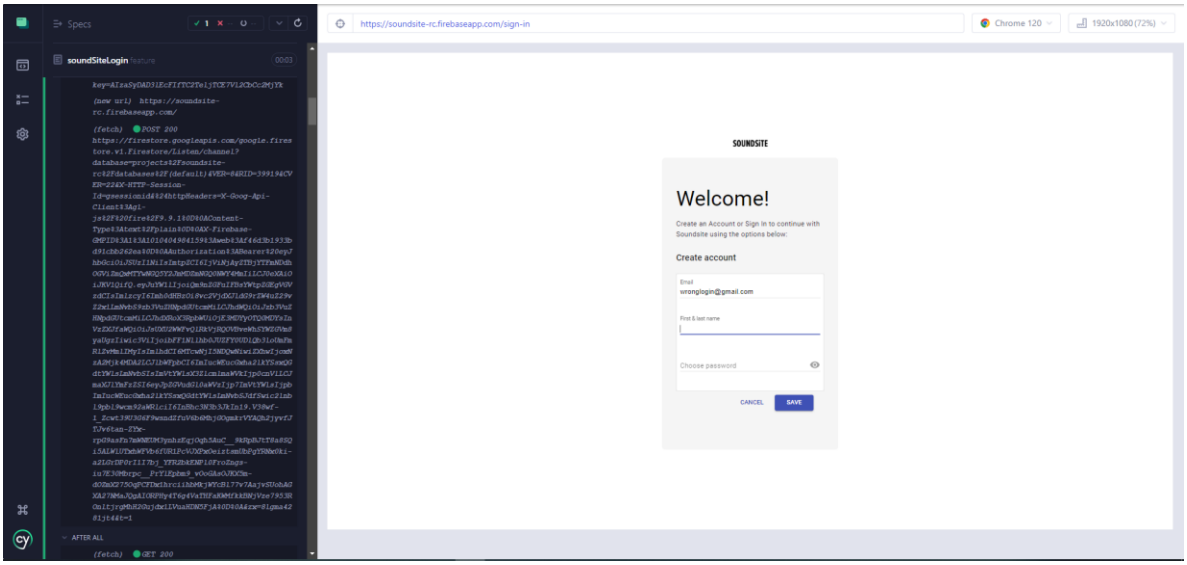

Рисунок 4.6 – Спроба входу з неіснуючим логіном

Метою дослідження буде питання як ми можемо зрозуміти що перед нами саме строка Email. Перше що ми можемо помітити, це звичний для нас email-домен, а саме @gmail.com. Для цього в одному з тесів ми використовуємо перевірку:

```
Given("User should be able to see email is real", () => {
    soundSiteLoginPage.getEmailField().should("contain.value", "@gmail.com")
});
```

Яка перевіряє наявність домену “@gmail.com” в нашому логіні використовуючи cypress команду “should” з умовою “("contain.value", "@gmail.com")”.

Друга деталь, це те що ми можемо побачити на скріншоті нижче.

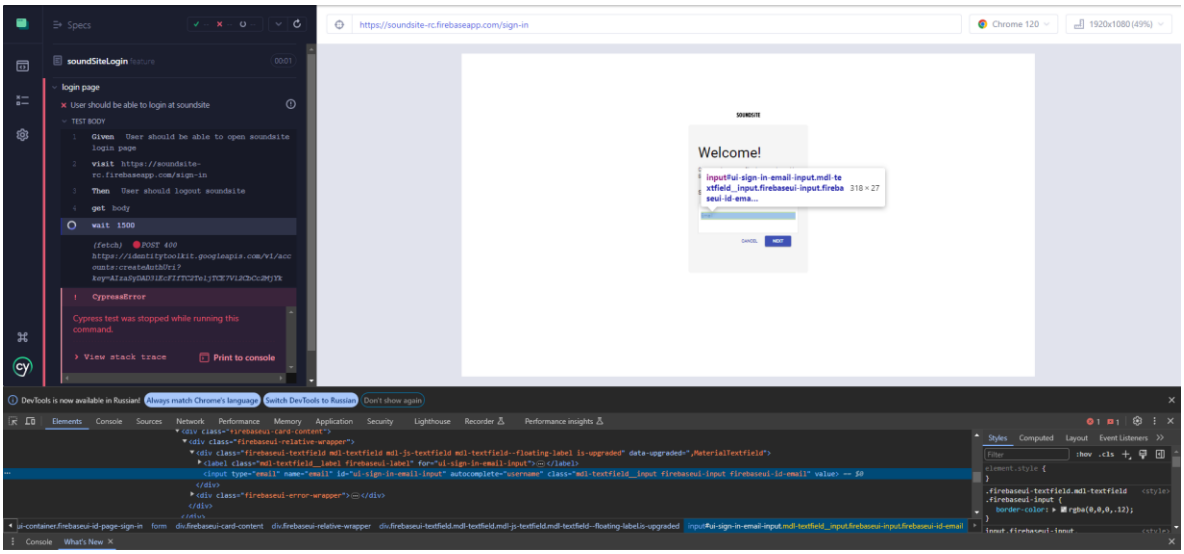

Рисунок 4.7 – Html Input в строкі Email

Якщо відкрити Chrome DevTools, то ми можемо побачити, що виділене Email поле містить в своєму DOM-дереві input з типом “email” <input type=”email” … value >

Елементи <input> типу email (електронна пошта) використовуються, аби дати користувачам змогу вводити й редагувати адресу електронної пошти, або, якщо заданий атрибут multiple, список таких адрес[8].

Значення поля автоматично валідується, аби пересвідчитись, що воно або порожнє, або є коректно відформатованою адресою електронної пошти (або списком таких адрес), перед поданням форми. Автоматично застосовуються псевдокласи CSS :valid і :invalid – аби візуально позначити те, чи є поточне значення поля дійсною адресою електронної пошти.

Атрибут value елемента <input> містить рядок, що автоматично валідується на відповідність синтаксисові електронних пошт. Якщо конкретніше, то є три можливі формати значень, що пройдуть валідацію:

Порожній рядок, котрий позначає, що користувач не ввів значення, або що значення було прибрано.

Одна коректно відформатована адреса електронної пошти. Це не обов'язково означає, що вона існує, але вона щонайменше коректно відформатована. Простіше кажучи, це щось типу username@domain або username@domain.tld. Звісно, тут набагато більше нюансів, дивіться регулярний вираз, що відповідає алгоритмові валідації електронної пошти, в розділі Валідація.

Якщо і лише якщо заданий атрибут multiple, то значення може бути списком коректно відформатованих розділених комами адрес електронної пошти. Всі пробіли на початку та в кінці кожної адреси в списку – прибираються.

Базова валідація:

Браузери автоматично надають валідацію, аби пересвідчитись, що лише текст, що відповідає стандартному форматові адрес електронної пошти Інтернету, був уведений в поле[9]. Браузери використовують алгоритм, рівносильний наступному регулярному виразові:

/^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$/;

Такий самий Input викрористовується і для поля з паролем, а саме:

<input type=”password” … >

<input> елементи типу "password" надають користувачеві можливість безпечного введення пароля. Елемент представлений як однорядковий текстовий редактор, в якому текст затінений, щоб його не можна було прочитати, як правило, шляхом заміни кожного символу іншим символом, таким як зірочка ("*") або точка ("•"). Цей символ змінюватиметься залежно від user agent та OS[10].

Особливості роботи процесу введення можуть відрізнятись від браузера до браузера; мобільні пристрої, наприклад, часто відображають символ, що вводиться на мить, перш ніж закривати його, щоб дозволити користувачеві бути впевненим, що вони натиснули клавішу, яку вони хотіли натиснути; це корисно з огляду на невеликий розмір клавіш і легкість, з якою може бути натиснута неправильна, особливо на віртуальних клавіатурах.

Параметри строки вводу (Input Parameters) відносяться до характеристик та властивостей, які можна задати для елементів <input> у HTML. Ці параметри визначають тип введення, вигляд та поведінку текстового поля або іншого елементу вводу на веб-сторінці. В нашому випадку використовується параметр типу type. Тип введення, такий як "text", "password", "checkbox", "radio", "email" і т.д. Цей параметр дозволяє розширити функціонал і вигляд елементів введення та надають можливості для керування введенням користувача на веб-сторінці.

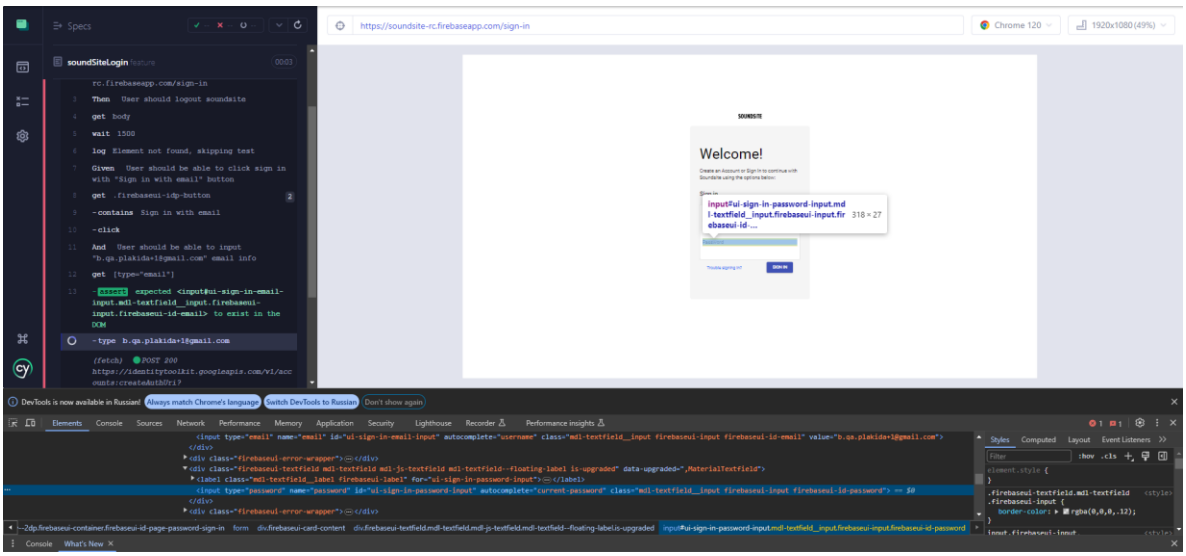

Рисунок 4.8 - Html Input в строкі Password

4.2 Тестування кнопок сортування

Кнопки сортування Sort By: Number, Name, Last modified.

– Натиснути кнопку “Number”.
– Натиснути кнопку “Number” другий раз.
– Натиснути кнопку “Name”.
– Натиснути кнопку “Name” другий раз.
– Натиснути кнопку “Last modified”.
– Натиснути кнопку “Last modified” другий раз.

Виконання дослідження:

На сайті який тестується при створенні проекта необхідно вказувати номер. В залежності від приципа за яким треба відсортувати проекти, буде залежати які кнопки будет натискати юзер. Наприклад, якщо треба відсортувати проекти за їх номером юзеру будет потрідно натиснуты кнопку Sort By: “Number”. В цьому випадку алгоритм сортування відсортує їх в порядку від 0 до Z, при другому натисканні ми отримаємо порядок від Z до 0. На перший погляд дивно, що сортування по номерам використовує також літери, але враховуючи специфіку сайту, та те що при створенні проекту вказуючи його номер ми не обмежені тільки цифровими значеннями, це виглядає логічним. Саме через це на сайті використовується лексикографічний метод сортування. Лексикографічне сортування порівнює символи за їхніми ASCII-кодами або Unicode-кодами і використовує цю інформацію для визначення порядку сортування. Зазвичай при цьому цифри вважаються меншими за літери верхнього та нижнього регістрів.

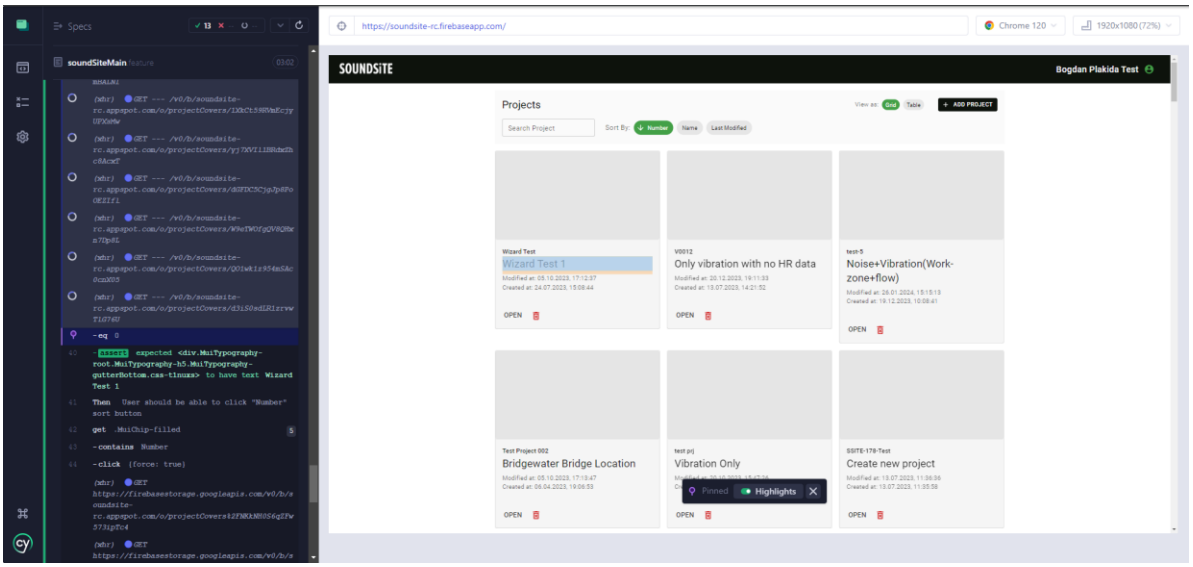

Рисунок 4.9 – Сортування від більшого номеру до меншого

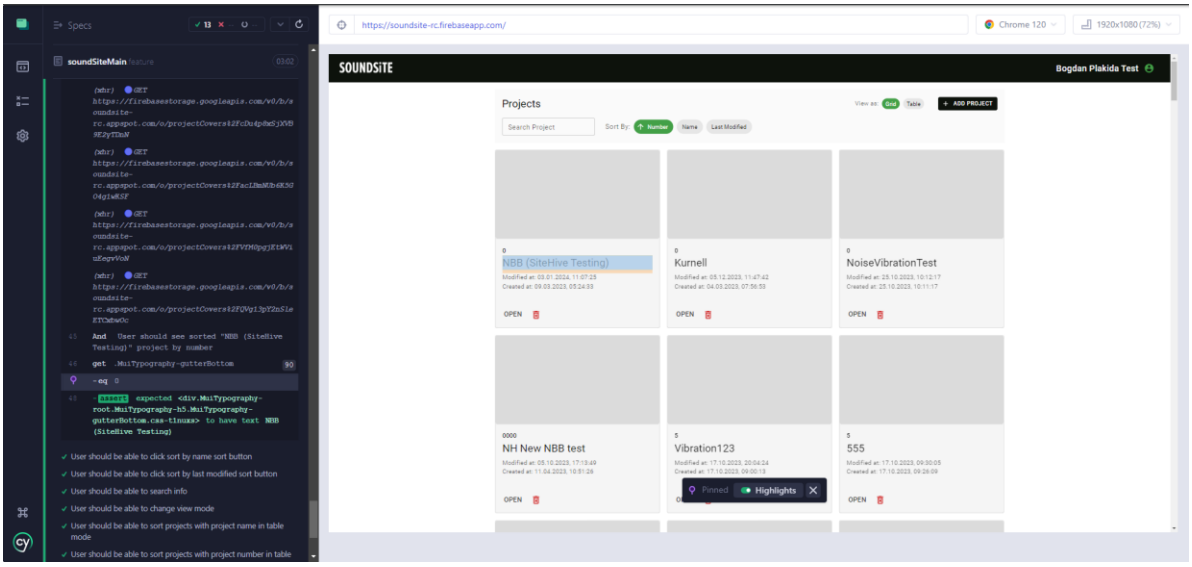

Рисунок 4.10 – Сортування від меншого номеру до більшого

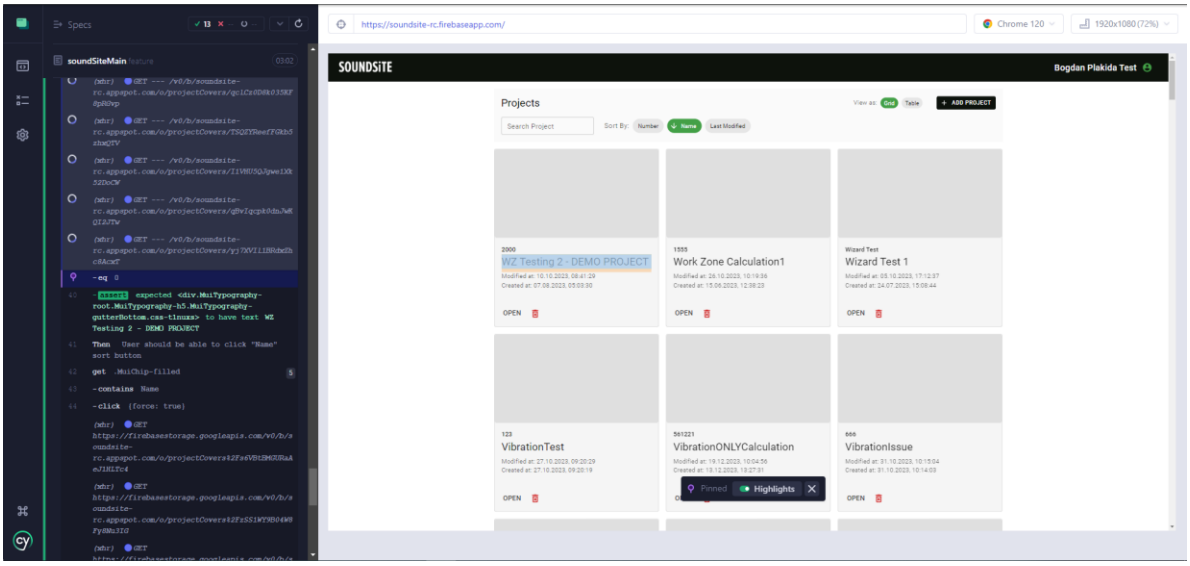

Рисунок 4.11 – Сортування від останньої до першої літери алфавіту

На рисунку 4.12 можна побачити, що назви проектів починаються с цифр, це відбувається через те, що цифрові значення є меншими у цьому виді сортування.

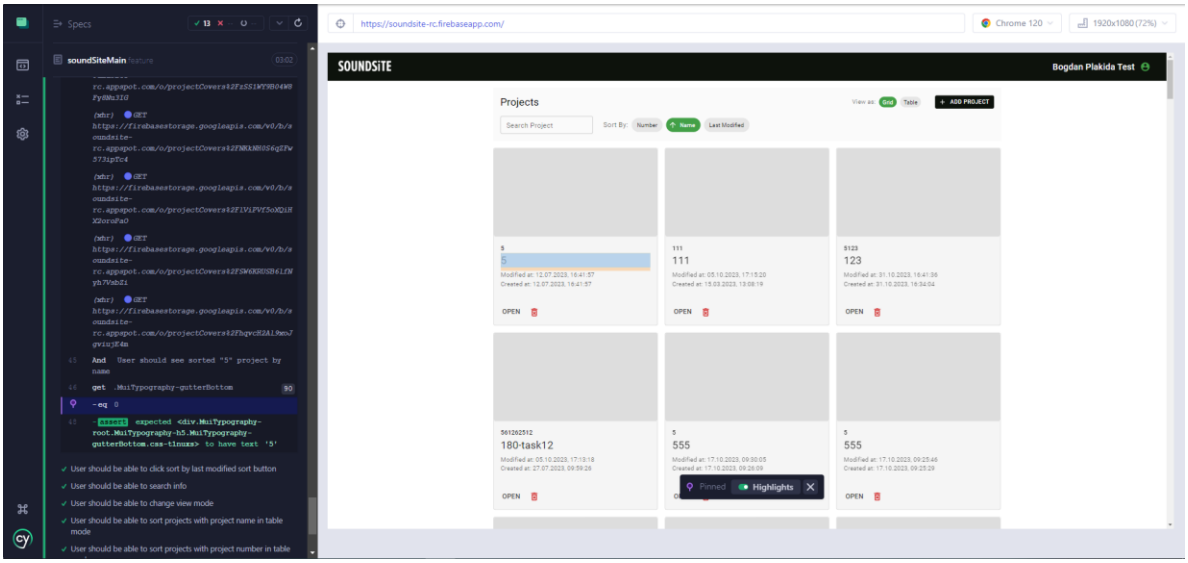

Рисунок 4.12 - Сортування від першої до останньої літери алфавіту

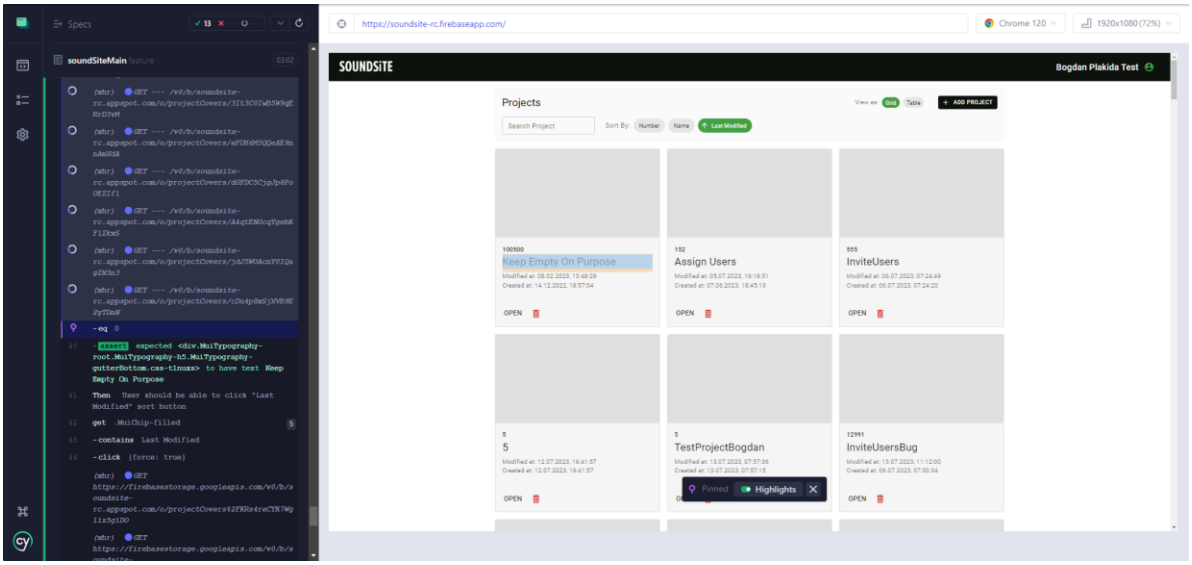

Рисунок 4.13 – Сортування за найстарішими змінами

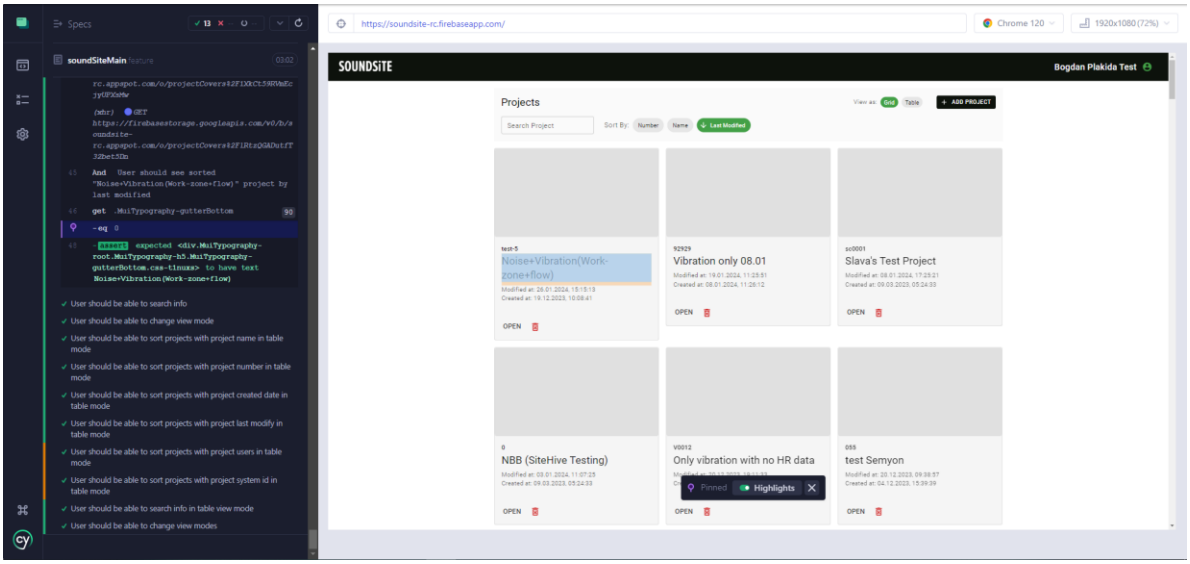

Рисунок 4.14 – Сортування за найновішими змінами

Також в нашому випадку всі кнопки мають атрибут “enabled”, який дозволяє користувачу в будь-який момент взаємодіяти з нею. Також важливою частиною є час виконання сортування. Провівши деякі досліди, отримав час за який виконується кожне сортування. Отримати ці значення стало можливим додавши до тесту таймер.

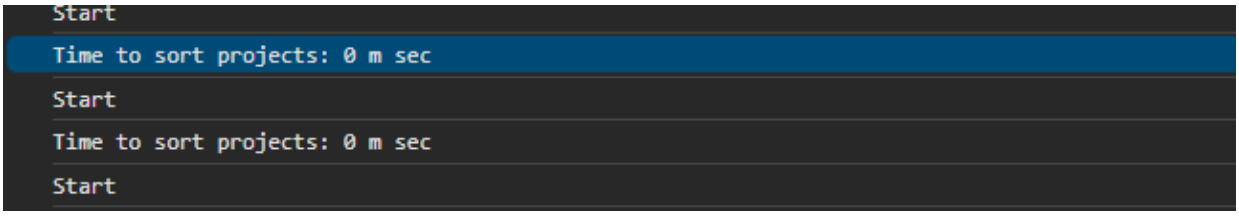

Рисунок 4.15 – Час сортування

Приклад коду, який використовується для заміру швидкості сортування:

```
Given("User should be able to click {string} sort button", (sortButton: string) => {
    console.log("Start");
    cy.then(() => {
        const startTime = new Date().getTime();
```

```
            soundSiteMainPage.getSortButtons().contains(sortButton).click({force:
true});
            const endTime = new Date().getTime();
            const executionTime = endTime - startTime;
            console.log(`Time to sort projects: ${executionTime} m sec`);
        })
        cy.log("End of test")
    });
```

4.3 Дослідження поля вводу

Поле вводу Search Project

- Натиснути на поле вводу, після чого воно стане активним.
- Ввести назву проекту який необхідно знайти.

Виконання дослідження:

На сайті який досліджується для пошуку проектів є окреме поле для вводу пошуку. Це зроблено для того, щоб користувачеві було зручніше та швидше перемикатися між своїми проектами. Юзеру потрібно ввести назву проекту в поле “Search Project”, після чого на сторінці буде тільки один проект з назвою яку він ввів. Виникає питання: чи можемо ми дослідити, що перед нами дійсно поле вводу?

Відповісти на це питання дуже легко, треба відкрити Chrome DevTools та обрати наше поле. Наш елемент має html-тег <input type="text">, що і вказує на те, що перед нами строка для вводу текста. Також нам необхідно перевірити, що текст який ми вводимо є валідним та немає помилок. Для цього в нашому тесті ми використовуємо перевірку яка виглядає наступним чином:

```
Given("User should see sorted {string} project by number", (projectByNumber:
string) => {
    soundSiteMainPage.getProjectCards().eq(0).should("have.text",
projectByNumber)
```

});

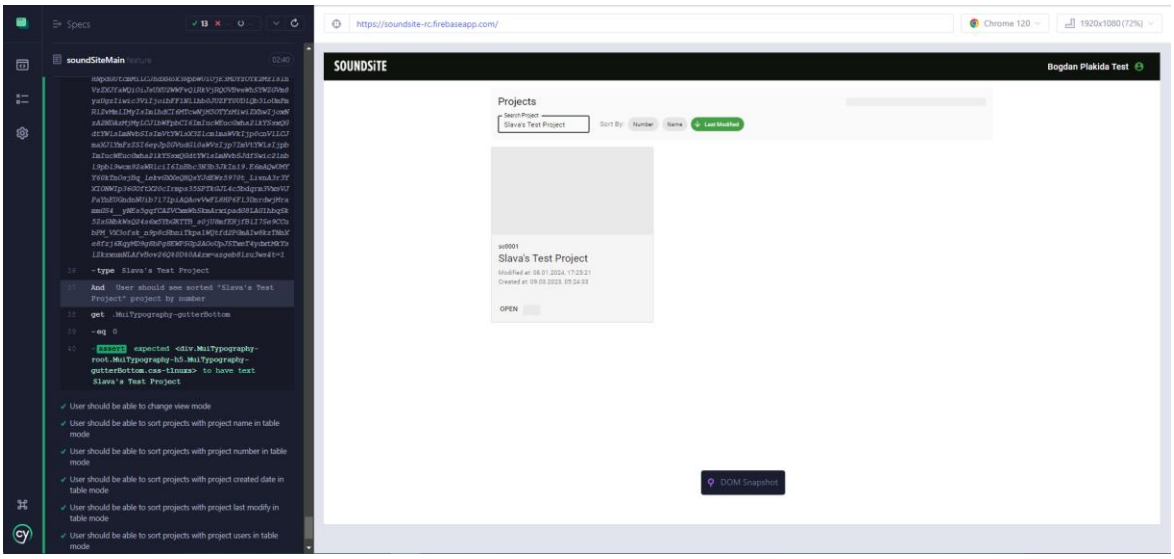

Рисунок 4.16 – Пошук проекту за назвою

4.4 Створення проекту

– Натиснути кнопку “ADD PROJECT”.
– Ввести номер та назву проекту у відповідні поля.
– Натиснути кнопку “NEXT STEP”.
– Натиснути кнопку “SAVE AND GO TO NEXT STEP”.
– Завантажити необхідні файли.
– Натиснути кнопку “NEXT STEP”
– Ввести почту юзера якого хочете додати до проекту в поле вводу.
– Натиснути кнопку “SAVE AND FINISH PROJECT SETUP”.
– Користувач має отримати побачити створенний проект з певними даними.

Виконання дослідження:

Сайт який досліджується специфікується на створенні проектів, в яких люди можуть вимірювати рівень шуму, та відстань на яку він може поширюватись. Отже можливість створити проект є однією з найважливіших функцій цього веб сайту. Кроки як створити проект можна побачити в тест-кейсі

вище. Перевірити чи дійсно створився проект можна задопомогою поля ввода, про яке я писав до цього, або задопомогою перевірки, яка виконується в одному із тестів, а саме:

```
Given("User should see created {string} project name", (projectName: string) => {
    soundSiteMainPage.getSearchAlert().eq(1).should("have.text", projectName)
});
```

Таким чином ми перевіряємо, що проект з назвою яку ми задали було створено. Це можна побачити на рисунку 4.22.

Також при створенні проекту було звернуто увагу на одну цікаву річ, а саме функцію AutoComplete, яка використовується у фінальному кроці створенні проекта, коли ми обираємо які юзери матимуть доступ до цього самого проекта.

Як зрозуміти, що це саме функція AutoComplete? На це вказувало дві речі: по-перше AutoComplete - це функціональність, яка автоматично доповнює введення користувача на основі раніше введених даних або доступних варіантів. Це може виявитися у вигляді випадаючого списку або автоматично вставленого тексту. В нашому випадку, це автоматичне доповнення з випадаючим списком. По-друге поле вводу має атрибут <input autocomplete=”on” type=”text” aria-autocomplete=”both”>. Це і вказує на те, що наше поле має функцію автозаповнення.

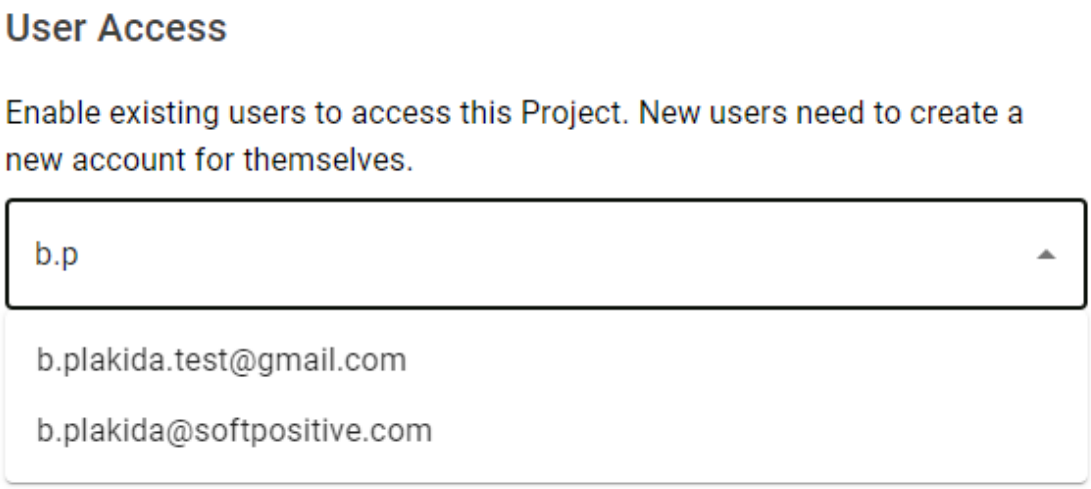

Рисунок 4.17 – AutoComplete

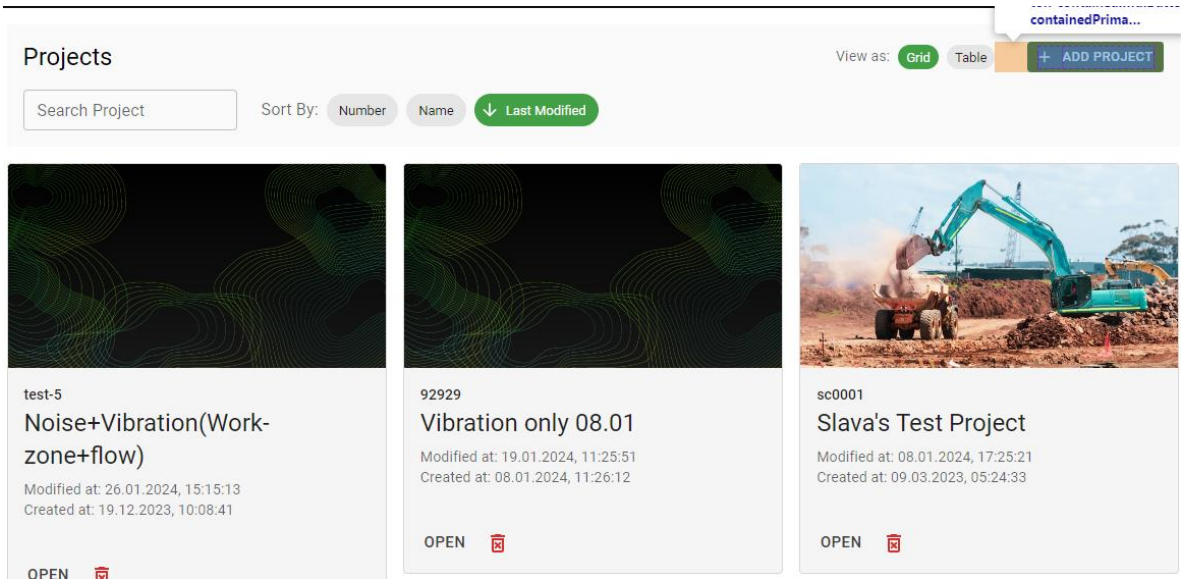

Рисунок 4.18 – Кнопка ADD PROJECT

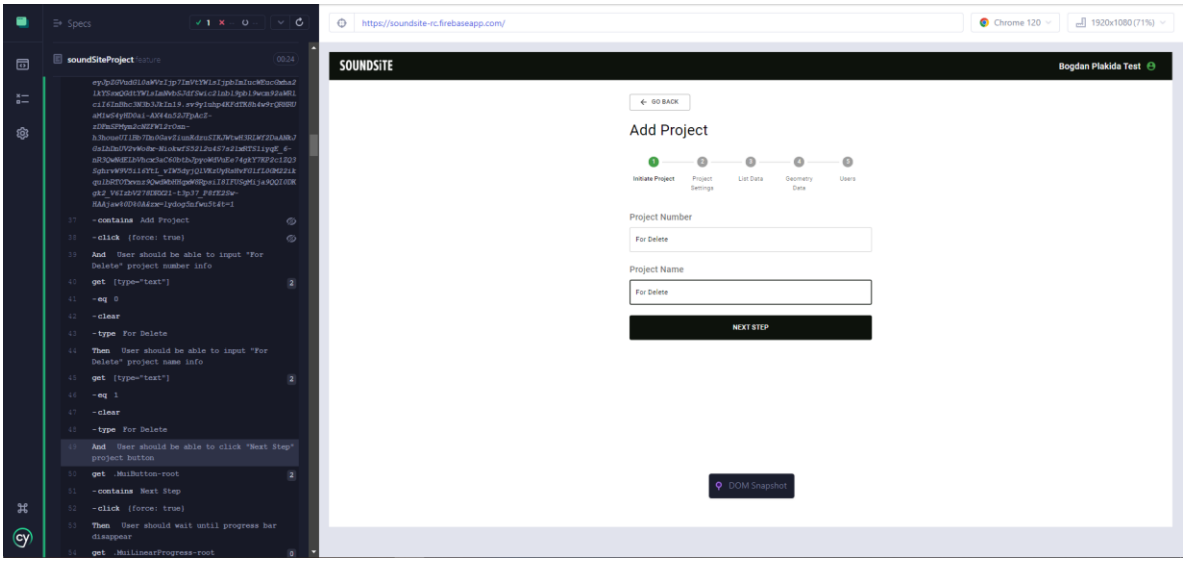

Рисунок 4.19 – Initiate Project

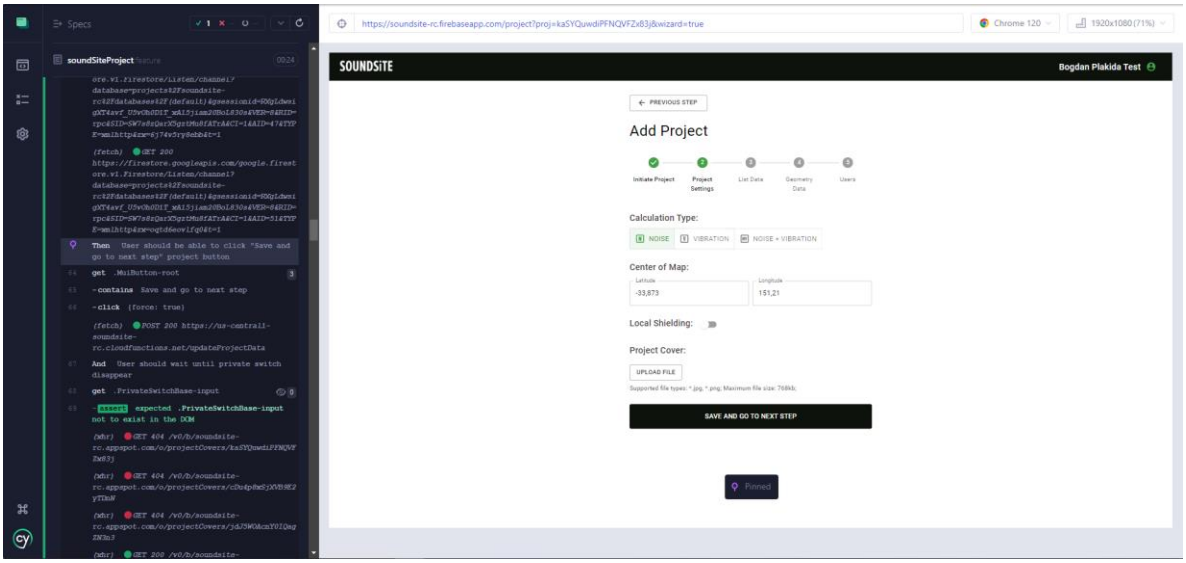

Рисунок 4.20 – Project Settings

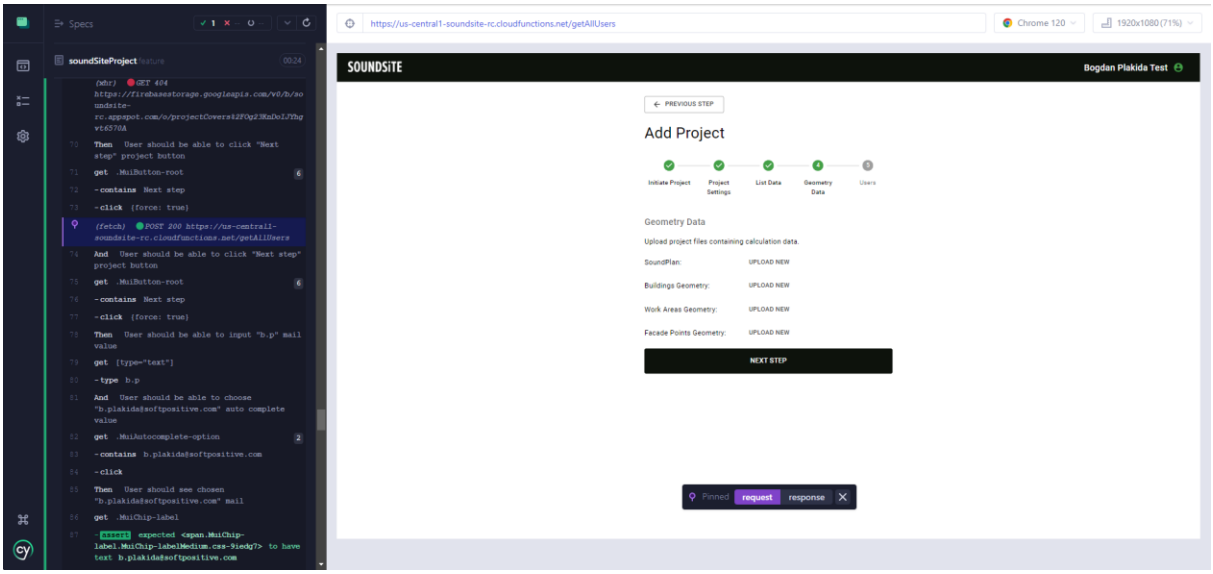

Рисунок 4.21 – Geometry Data

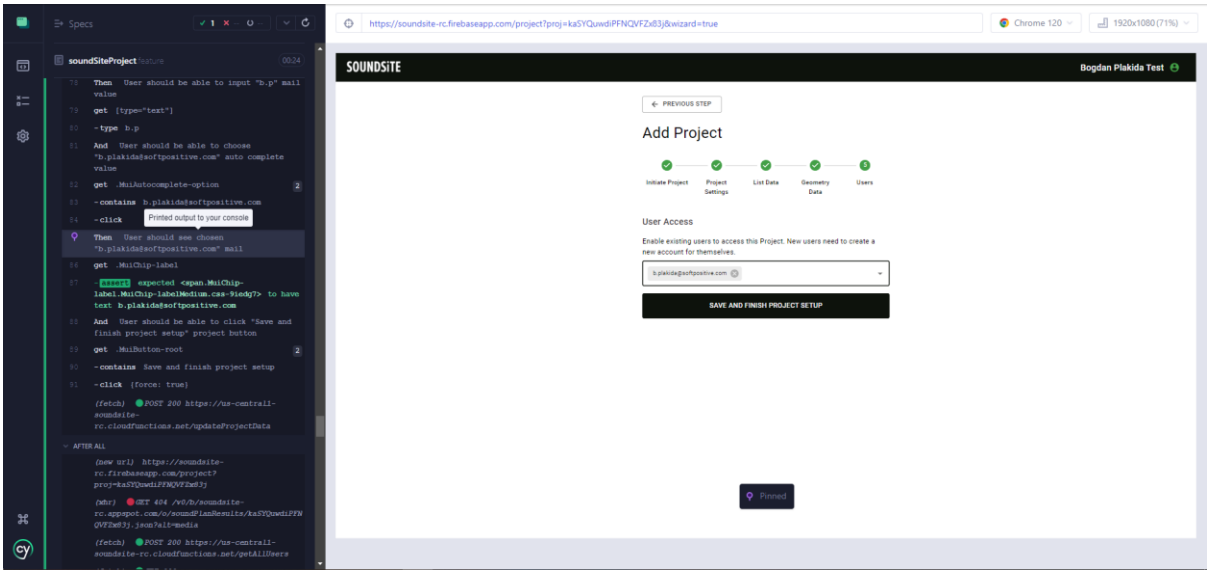

Рисунок 4.22 – Users

Рисунок 4.23 – Успішне створення проекту

Рішення тестувати саме цей функціонал виникло по ряду причин.

По-перше: логін форма є важливою частиною процесу тестування веб-додатків, тому що є одним із механізмів захисту доступу до конфіденційної інформації. Успішний логін є частиною більшого процесу взаємодії з веб-додатком, без входу на сайт юзер не зможе користуватись функціями які надає сайт взагалі.

По-друге: сортування та пошук проектів за назвою є зручною функією для користувачів, яка дає змогу швидко та ефективно знайти потрібний проект, або переключатись між декількома проектами, що значно економить час та позитивно впливає на досвід використання додатку.

По-третє: створення проекту та додавання потрібних файлів, які знадобляться у подальшому для роботи є основною функціє даного веб-сайту, яка має бути протестована однією із перших, що дасть змогу запустит сайт у бета-користування, щоб юзери могли якомога скоріше почати ним користуватись.

В ході виконання дослідження було виявлено ряд обмежень автоматизованного тестування, а саме:

- Неможливість автоматизувати проходження захисту від ботів(CAPTCHA). Через те, що вона являє собою механізм, який розробленний для захисту від автоматизованних систем, таких як боти або скрипти, а в основі автоматизованного тестування лежать скрипти, які виконують автоматичну перевірку функціональності.
- Неможливість автоматизувати вхід на сайт через інші сервіси, такі як Google, через те, що це потребує взаємодії із зовнішніми сервісами та введення даних користувачем. Такі послуги зазвичай використовують механізми автентифікації, які не можуть бути автоматизовані без ризику порушення політики безпеки або безпеки даних. Крім того, такі сервіси можуть використовувати капчі або інші механізми захисту, які складно чи неможливо автоматизувати через свою природу.
- Обмеження у роботі з динамічними данними у строкі пошуку. Через те, що поле введення призначене для пошуку динамічних даних, в автоматизованному тестуванні виникає обмеження в керуванні цими даними та перевірці їх коректності.

ВИСНОВКИ

Метою роботи було дослідження обмежень автоматизованого тестування під час тестування веб-сайту. Проаналізовано найбільш поширені види тестування в сукупності з сучасними автоматизованими системами тестування. Детально описаний фреймворк Cypress, а також обрані методи та інструменти автоматизації.

В ході роботи були розроблені автоматизовані тести, для тестування логін форми, сортування, створення проектів та пошуку серед них.

Для досягнення мети були поставлені такі задачі:

- Виявлення дефектів, яке полягає у пошуку помилок та проблем у програмному забезпеченні.
- Підтвердження якості, яке допомагає впевнитися, що програмне забезпечення відповідає вимогам та стандартам якості.
- Забезпечення безпеки, яке допомагає виявити потенційні проблеми безпеки та захистити програмне забезпечення від зловмисних атак.

Для написання тестів було використано інструмент для автоматизованого функціонального тестування Cucumber разом з фреймворком Cypress, задопомогою яких здійснювались тестування та дослідження веб-сайту, були обрані найбільш приоритетні сценарії для автоматизації.

Значимість даної роботи полягала у висвітленні недоліків та обмежень автоматизованного тестування, а саме:

- Неможливість автоматизувати проходження захисту CAPTCHA.
- Неможливість автоматизувати вхід на сайт через інші сервіси.
- Обмеження у роботі з динамічними данними у пошукових строках.

Складено план проведення експерименту. Випробувана робота тестів на прикладі реального веб-сайту. Виконано детальний аналіз результату роботи тестів. Завдяки аналізам результатів було виявлено ряд обмежень в автоматизованному тестуванні, які значно впливають на його ефективність, точність результатів та швидкість розробки.

Практично була примінена техніки тест-дизайну матриця прийняття рішень. При тестуванні логін форми було використано техніку матриці прийняття рішень. Рішення використовувати саме цю техніку було принято через те, що логін форма має тільки дві змінні, завдяки цьому можна перевірити всі можливі варіанти заповнення вхідних строк, а саме:

- Введення вірного логіну та паролю.
- Введення вірного логіну та невірного паролю.
- Введення неіснуючого логіну.

Були проведені порівняння з очікуваними результатами. Результати збігаються з очікуваними, це може бути підтвердженням того, що тести валідні. Була проведена перевірка тестів на реальних даних, які відображають сценарії використання продукту.

Таким чином, можна зробити висновок, що використання автоматизованного тестування на проектах дозволить збільшити швидкість перевірки на можливі баги, але якщо проект націлений на довгострокову перспективу.

Отже автоматизація тестування не є універсальним рішенням на всі випадки і не завжди виправдовує очікування. Вона може бути витратною та трудомісткою задачею, особливо на початкових етапах. Вона не може гарантувати що всі дефекти будуть виявлені, а для її успішної реалізації необхідно встановити стандартизовані підходи та ефективне управління.

ПЕРЕЛІК ПОСИЛАНЬ

1. Життєвий цикл програмного забезпечення – [Електрон. ресурс]. - https://uk.wikipedia.org/w/index.php?title=Життєвий_цикл_програмного_забезпечення&oldid=36239602
2. 7 принципов тестирования программного обеспечения с примерами – [Електрон. ресурс]. - https://www.guru99.com/ru/software-testing-seven-principles.html
3. ЩО ТАКЕ BLACK/GREY/WHITE BOX TESTING 2024. – [Електрон. ресурс] - https://reporter.zp.ua/shho-take-black-grey-white-box-testing.html
4. Метрики тестирования (Software Test Metrics) – [Електрон. ресурс] - https://vladislaveremeev.gitbook.io/qa_bible/testovaya-dokumentaciya-i-artefakty-test-deliverablestest-artifacts/metriki-testirovaniya-software-test-metrics
5. Различные виды тестирования программного обеспечения – [Електрон. ресурс] - https://appmaster.io/ru/blog/vidy-testirovaniia-programmnogo-obespecheniia
6. Локалізація дефекта та його повторне відтворення 2023. – [Електрон. ресурс] - https://training.qatestlab.com/blog/technical-articles/defect-localization-and-replay/
7. Автоматизація тестування: як уникнути поширених помилок 2023. – [Електрон. ресурс] - https://www.globallogic.com/ua/insights/blogs/qa-automation-2/
8. Елементи <input> типу email – [Електрон. ресурс] - https://webdoky.org/uk/docs/Web/HTML/Element/input/email/#znachennia
9. Валідація елементу <input> типу email – [Електрон. ресурс] - https://webdoky.org/uk/docs/Web/HTML/Element/input/email/#validatsiia
10. Елементи <input> типу password 2024. – [Електрон. ресурс] - https://developer.mozilla.org/ru/docs/Web/HTML/Element/input/password
11. Техники тест-дизайна и их предназначение 2019. – [Електрон. ресурс] - https://www.simbirsoft.com/blog/tekhniki-test-dizayna-i-ikh-prednaznachenie/

ДОДАТОК А

```
      class SoundSiteLoginPage {
          signInButton = ".firebaseui-
idp-button";
          emailField =
"[type=\"email\"]";
          nextButton =
"[type=\"submit\"]";
          passwordField =
"[type=\"password\"]";
          profileButton =
".MuiIconButton-colorInherit";
          profileActions = ".MuiMenu-
list";
          getSignInButton() {
              return
cy.get(this.signInButton)
          }
          getEmailField() {
              return
cy.get(this.emailField)
      class SoundSiteMainPage {
          sortButtons = ".MuiChip-
filled";
          projectCards =
".MuiTypography-gutterBottom"
          tableHeader = ".MuiDataGrid-
columnHeaders";
          sortTableHeader =
".MuiDataGrid-columnHeaderTitle";
          sortByNameInTable = "[data-
field=\"name\"]";
          sortByNumberInTable =
"[data-field=\"number\"]"
          sortByCreatedInTable =
"[data-field=\"projectCreated\"]";
          sortByLastModifiedInTable =
"[data-field=\"lastModified\"]";
          sortByUsersInTable = "[data-
field=\"users\"]";
          sortBySystemIdInTable =
"[data-field=\"id\"]";
          searchFieldInTableViewMode =
"[type=\"search\"]";
          mitigationInput =
"[placeholder=\"Mitigation\"]";
          descriptionField =
"[placeholder=\"Description\"]";
          addIcon = "[data-
testid=\"AddIcon\"]";

          enterCategory =
"[placeholder=\"Enter category\"]";
          mitigationDropDown = "[data-
testid=\"ArrowDropDownIcon\"]";
          uploadNewButton =
"[role=\"button\"]";
          }
          getNextButton() {
              return
cy.get(this.nextButton)
          }
          getPasswordField() {
              return
cy.get(this.passwordField)
          }
          getProfileButton() {
              return
cy.get(this.profileButton)
          }
          getProfileActions() {
              return
cy.get(this.profileActions)
          }
      }
      export default new
SoundSiteLoginPage();
          deleteProject = "[data-
testid=\"DeleteForeverOutlinedIcon\"]"
;
          fileSuccessfullyUploaded =
".MuiAlert-message";
          projectSettingsCheckbox =
"[type=\"checkbox\"]";
          assignedUsersSelect =
"#tags-standard";
          noRows = ".MuiDataGrid-
overlay";
          skeleton = ".MuiSkeleton-
root";
          searchAlert =
".MuiTypography-root";
          openProject =
".MuiButtonBase-root";
          scenarioListHeader =
".MuiTypography-h4";
          noResults = ".MuiTypography-
h5";
          getSortButtons() {
              return
cy.get(this.sortButtons)
          }
          getProjectCards() {
              return
cy.get(this.projectCards)
          }
          getTableHeader() {
              return
cy.get(this.tableHeader)
          }
          getSortTableHeader() {
              return
cy.get(this.sortTableHeader)
          }
```

```
          getSortByNameInTable() {
              return
cy.get(this.sortByNameInTable)
          }
          getSortByNumberInTable() {
              return
cy.get(this.sortByNumberInTable)
          }
          getSortByCreatedInTable() {
              return
cy.get(this.sortByCreatedInTable)
          }

getSortByLastModifiedInTable() {
              return
cy.get(this.sortByLastModifiedInTable)
          }
          getSortByUsersInTable() {
              return
cy.get(this.sortByUsersInTable)
          }
          getSortBySystemIdInTable() {
              return
cy.get(this.sortBySystemIdInTable)
          }


getSearchFieldInTableViewMode() {
              return
cy.get(this.searchFieldInTableViewMode
)
          }
          getMitigationInput() {
              return
cy.get(this.mitigationInput)
          }
          getDescriptionField() {
              return
cy.get(this.descriptionField)
          }
          getAddIcon() {
              return
cy.get(this.addIcon)
          }
          getEnterCategory() {
              return
cy.get(this.enterCategory)
          }
          getMitigationDropDown() {
              return
cy.get(this.mitigationDropDown)
          }
      class SoundSiteProjectPage {
          progressBar =
".MuiLinearProgress-root";
          privateSwitch =
".PrivateSwitchBase-input";
          chooseMail = "[type=\"text\"]";
              getUploadNewButton() {
                  return
cy.get(this.uploadNewButton)
              }
              getDeleteProject() {
                  return
cy.get(this.deleteProject)
              }
             getFileSuccessfullyUploaded()
{

                  return
cy.get(this.fileSuccessfullyUploaded)

              }

              getProjectSettingsCheckbox()
{

                  return
cy.get(this.projectSettingsCheckbox)

              }
              getAssignedUsersSelect() {
                  return
cy.get(this.assignedUsersSelect)
              }
              getNoRows() {
                  return
cy.get(this.noRows, {timeout: 20000})
              }
              getSkeleton() {
                  return
cy.get(this.skeleton)
              }
              getSearchAlert() {
                  return
cy.get(this.searchAlert)
              }
              getOpenProject() {
                  return
cy.get(this.openProject)
              }
              getScenarioListHeader() {
                  return
cy.get(this.scenarioListHeader)
              }
              getNoResults() {
                  return
cy.get(this.noResults)
              }
          }
          export default new
SoundSiteMainPage();
          autoComplete =
".MuiAutocomplete-option";
          completedMailField = ".MuiChip-
label";
          scenarioTypeSelect =
"#ScenarioTypeSelect";
```

```
        startAndEndDate =
"[type=\"tel\"]";

        date = "[role=\"gridcell\"]";

        time = "[type=\"time\"]";

        timePeriod = "#timeSelect";

        timePeriodDropDown =
"[role=\"listbox\"]";

        descriptionInNewScenario =
".MuiInputBase-input";

        editScenario = "[aria-
label=\"Edit Selected Scenario\"]";
        saveAsScenario = "[aria-
label=\"Save Scenario As\"]";
        deleteScenario = "[aria-
label=\"Delete Scenario\"]";
        dropdownOption =
"[role=\"listbox\"] li";
        projectButton = ".MuiButton-
root";
        searchField =
"[type=\"text\"]";
        cancelIcon = "[data-
testid=\"CancelIcon\"]";
        changeAssignedUser =
"[id=\"tags-standard\"]";
        assignedUsersLabel = ".MuiChip-
label";
        scenarioListSearch =
"[type=\"search\"]";
        scenarioName = "[data-
field=\"name\"]";
        sortInScenarioList =
".MuiDataGrid-columnHeaderTitle";
        scenarioStartDate = "[data-
field=\"startDate\"]";
        scenarioStartTime = "[data-
field=\"startTime\"]";
        scenarioStopDate = "[data-
field=\"stopDate\"]";
        scenarioStopTime = "[data-
field=\"stopTime\"]";
        scenarioTimePeriod = "[data-
field=\"timePeriod\"]";
        scenarioDescriptionOfWorks =
"[data-field=\"description\"]";
        scenarioCreatedBy = "[data-
field=\"createdBy\"]";
        scenarioCreationTime = "[data-
field=\"creationTime\"]";
        scenarioModifiedBy = "[data-
field=\"modifiedBy\"]";
        scenarioModifiedTime = "[data-
field=\"modifiedTime\"]";
        baseButtons = ".MuiButtonBase-
root";
        getProgressBar() {
            return
cy.get(this.progressBar)
        }
        getPrivateSwitch() {
            return
cy.get(this.privateSwitch)
        }
        getChooseMail() {
            return
cy.get(this.chooseMail)
        }
        getAutoComplete() {
            return
cy.get(this.autoComplete)
        }
        getCompletedMailField() {
            return
cy.get(this.completedMailField)
        }
        getScenarioTypeSelect() {
            return
cy.get(this.scenarioTypeSelect)
        }
        getStartAndEndDate() {
            return
cy.get(this.startAndEndDate)
        }
        getDate() {
            return cy.get(this.date)
        }
        getTime() {
            return cy.get(this.time)
        }
        getTimePeriod() {
            return
cy.get(this.timePeriod)
        }
        getTimePeriodDropDown() {
            return
cy.get(this.timePeriodDropDown)
        }
        getDescriptionInNewScenario() {
            return
cy.get(this.descriptionInNewScenario)
        }
        getEditScenario() {
            return
cy.get(this.editScenario)
        }
        getSaveAsScenario() {
            return
cy.get(this.saveAsScenario)
        }
        getDeleteScenario() {
            return
cy.get(this.deleteScenario)
```

```
        }
        getDropdownOption() {
            return
cy.get(this.dropdownOption)
        }

        getProjectButton() {
            return
cy.get(this.projectButton)
        }
        getCancelIcon() {
            return
cy.get(this.cancelIcon)
        }
        getChangeAssignedUser() {
            return
cy.get(this.changeAssignedUser)
        }
        getAssignedUsersLabel() {
            return
cy.get(this.assignedUsersLabel)
        }
        getScenarioListSearch() {
            return
cy.get(this.scenarioListSearch)
        }
        getScenarioName() {
            return
cy.get(this.scenarioName)
        }
        getSortInScenarioList() {
            return
cy.get(this.sortInScenarioList)
        }
        getScenarioStartDate() {
            return
cy.get(this.scenarioStartDate)
        }
        getScenarioStartTime() {
            return
cy.get(this.scenarioStartTime)
        }
        getScenarioStopDate() {
            return
cy.get(this.scenarioStopDate)
        }
        getScenarioStopTime() {
            return
cy.get(this.scenarioStopTime)
        }
        getScenarioTimePeriod() {
            return
cy.get(this.scenarioTimePeriod)
        }
        getScenarioDescriptionOfWorks()
{
            return
cy.get(this.scenarioDescriptionOfWorks)
        }
        getScenarioCreatedBy() {
            return
cy.get(this.scenarioCreatedBy)
        }
        getScenarioCreationTime() {
            return
cy.get(this.scenarioCreationTime)
        }
        getScenarioModifiedBy() {
            return
cy.get(this.scenarioModifiedBy)
        }
        getScenarioModifiedTime() {
            return
cy.get(this.scenarioModifiedTime)

        }
        getSearchField() {
            return
cy.get(this.searchField)
        }
        getBaseButtons() {
            return
cy.get(this.baseButtons)
        }
    }
    export default new
SoundSiteProjectPage();
```

ДОДАТОК Б

```
      import {
          Given
      } from "@badeball/cypress-
cucumber-preprocessor";
      import soundSiteLoginPage from
"../../pageObjects/soundSiteLoginPage"
      import soundSurferLoginPage from
"../../pageObjects/soundSurferLoginPag
e";
      Given("User should be able to
click sign in with {string} button",
(chooseSignIn: string) => {
      soundSiteLoginPage.getSignInButt
on().contains(chooseSignIn).click();
      });
      Given("User should be able to
open soundsite login page", () => {
          cy.visit("https://soundsite-
rc.firebaseapp.com/sign-in");
      });
      Given("User should be able to
input {string} email info", (email:
string) => {
soundSiteLoginPage.getEmailField().sho
uld("be.exist").type(email);
      });
      Given("User should be able to
click next button", () => {

soundSiteLoginPage.getNextButton().cli
ck();
      });
      Given("User should be able to
input {string} password info",
(password:string) => {
      soundSiteLoginPage.getPasswordFi
eld().type(password);
      });
      Given("User should be able to
click soundsite profile button", () =>
{
soundSiteLoginPage.getProfileButton().
click();
      });
      Given("User should be able to
click {string} profile action button",
(action:string) => {
      soundSiteLoginPage.getProfileAct
ions().contains(action).click();
      });
      Given("User should be able to
see profile", () => {
soundSiteLoginPage.getProfileButton().
should("be.exist")
      });
      Given("User should logout
soundsite", () => {
          cy.get('body').then((body)
=> {
              cy.wait(1500).then(() =>
{
                  if
(body.find("[data-
testid=\"AccountCircleIcon\"]").length
> 0) {
                      cy.log('Element
found, proceeding with test')
soundSiteLoginPage.getProfileButton().
click({force:true});
soundSiteLoginPage.getProfileActions()
.contains("Logout").click();
                      cy.wait(1500);
                  } else {
                      cy.log('Element
not found, skipping test')
                  }
              })
          })
      });
      Given("User should be able to
see email is real", () => {
soundSiteLoginPage.getEmailField().sho
uld("contain.value", "@gmail.com")
      import {
          Given
      } from "@badeball/cypress-
cucumber-preprocessor";
      import soundSiteMainPage from
"../../pageObjects/soundSiteMainPage"
      import soundSiteProjectPage from
"../../pageObjects/soundSiteProjectPag
e";
      Given("User should be able to
click {string} sort button",
(sortButton: string) => {
          console.log("Start");
          cy.then(() => {
              const startTime = new
Date().getTime();

soundSiteMainPage.getSortButtons().con
tains(sortButton).click({force:
true});
              const endTime = new
Date().getTime();
              const executionTime =
endTime - startTime;
              console.log(`Time to
sort projects: ${executionTime} m
sec`);
      })  cy.log("End of test")});
```

```
      Given("User should see sorted
{string} project by number",
(projectByNumber: string) => {
      soundSiteMainPage.getProjectCard
s().eq(0).should("have.text",
projectByNumber)
      });
      Given("User should see sorted
{string} project by name",
(projectByName: string) => {
      soundSiteMainPage.getProjectCard
s().eq(0).should("have.text",
projectByName)
      });
      Given("User should see sorted
{string} project by last modified",
(projectByLastModified: string) => {
      soundSiteMainPage.getProjectCard
s().eq(0).should("have.text",
projectByLastModified)
      });
      Given("User should be able to
search {string} text", (search:
string) => {
soundSiteProjectPage.getSearchField().
clear().type(search);
      });
      import {
          Given
      } from "@badeball/cypress-
cucumber-preprocessor";
      import soundSiteProjectPage from
"../../pageObjects/soundSiteProjectPag
e"
      import soundSiteMainPage from
"../../pageObjects/soundSiteMainPage";
      import soundSurferLoginPage from
"../../pageObjects/soundSurferLoginPag
e";
      Given("User should be able to
click {string} project button",
(projectButton: string) => {
soundSiteProjectPage.getProjectButton(
).contains(projectButton).click({force
: true});
      });
      Given("User should be able to
input {string} project number info",
(projectNumber: string) => {
soundSiteProjectPage.getSearchField().
eq(0).clear().type(projectNumber);
      });
      Given("User should be able to
input {string} project name info",
(projectName: string) => {
      soundSiteProjectPage.getSearchFi
eld().eq(1).clear().type(projectName);
      });
      Given("User should be able to
click {string} project button",
(projectButton: string) => {
      soundSiteProjectPage.getProjectB
utton().contains(projectButton).click(
{force: true});
      });
      Given("User should wait until
progress bar disappear", () => {
      soundSiteProjectPage.getProgress
Bar().should("not.exist")
      });
      Given("User should wait until
settings spinner disappear", () => {
soundSiteSettingsPage.getSettingsSpinn
er().should("not.exist")
      });
      Given("User should wait until
skeleton disappear", () => {
      soundSiteMainPage.getSkeleton().
eq(0).should("not.exist")
      });
      Given("User should wait until
private switch disappear", () => {
soundSiteProjectPage.getPrivateSwitch(
).should("not.exist")
      });
      Given("User should be able to
input {string} mail value", (value:
string) => {
soundSiteProjectPage.getChooseMail().t
ype(value)
      });
      Given("User should be able to
choose {string} auto complete value",
(autoCompleteValue: string) => {
      soundSiteProjectPage.getAutoComp
lete().contains(autoCompleteValue).cli
ck();
      });
      Given("User should see chosen
{string} mail", (mailText: string) =>
{
soundSiteProjectPage.getCompletedMailF
ield().should("have.text", mailText)
      });
      Given("User should see scenario
dropdown", () => {

soundSiteProjectPage.getScenarioTypeSe
lect().should("be.visible")
          cy.wait(5000);
      });
      Given("User should see created
{string} project name", (projectName:
string) => {
      soundSiteMainPage.getSearchAlert
().eq(1).should("have.text",
projectName)})
```

ДОДАТОК В

```
      Feature: login page
        Background:
          Given User should be able to
open soundsite login page
          Then User should logout
soundsite
        Scenario: User should be able
to login at soundsite
          Given User should be able to
click sign in with "Sign in with
email" button
          And User should be able to
input "b.qa.plakida+1@gmail.com" email
info
          And User should be able to
see email is real
          Then User should be able to
click next button
          And User should be able to
input "Abogdan123!" password info
          Then User should be able to
click next button
          And User should be able to
see profile
      Feature: main page
        Background:
          Given User should be able to
open soundsite login page
          Then User should logout
soundsite
        Scenario: User should be able
to click sort by number sort button
          Given User should be able to
click sign in with "Sign in with
email" button
          And User should be able to
input "b.qa.plakida+1@gmail.com" email
info
          Then User should be able to
click next button
          And User should be able to
input "Abogdan123!" password info
          Then User should be able to
click next button
          And User should be able to
see profile
          Then User should be able to
click "Number" sort button
          And User should see sorted
"Wizard Test 1" project by number
          Then User should be able to
click "Number" sort button
          And User should see sorted
"NBB (SiteHive Testing)" project by
number
        Scenario: User should be able
to click sort by name sort button
          Given User should be able to
click sign in with "Sign in with
email" button
          And User should be able to
input "b.qa.plakida+1@gmail.com" email
info
          Then User should be able to
click next button
          And User should be able to
input "Abogdan123!" password info
          Then User should be able to
click next button
          And User should be able to
see profile
          Then User should be able to
click "Name" sort button
          And User should see sorted
"WZ Testing 2 - DEMO PROJECT" project
by name
          Then User should be able to
click "Name" sort button
          And User should see sorted
"5" project by name
        Scenario: User should be able
to click sort by last modified sort
button
          Given User should be able to
click sign in with "Sign in with
email" button
          And User should be able to
input "b.qa.plakida+1@gmail.com" email
info
          Then User should be able to
click next button
          And User should be able to
input "Abogdan123!" password info
          Then User should be able to
click next button
          And User should be able to
see profile
          Then User should be able to
click "Last Modified" sort button
          And User should see sorted
"Keep Empty On Purpose" project by
last modified
          Then User should be able to
click "Last Modified" sort button
          And User should see sorted
"Noise+Vibration(Work-zone+flow)"
project by last modified
        Scenario: User should be able
to search info
```

```
          Given User should be able to
click sign in with "Sign in with
email" button
          And User should be able to
input "b.qa.plakida+1@gmail.com" email
info
          Then User should be able to
click next button
          And User should be able to
input "Abogdan123!" password info
          Then User should be able to
click next button
          And User should be able to
see profile
          Then User should be able to
search "Slava's Test Project" text
          And User should see sorted
"Slava's Test Project" project by
number
      Feature: Project
        Background:
          Given User should be able to
open soundsite login page
          Then User should logout
soundsite
        Scenario: User should be able
to create project
          Given User should be able to
open soundsite login page
          Then User should be able to
click sign in with "Sign in with
email" button
          And User should be able to
input "b.qa.plakida+1@gmail.com" email
info
          Then User should be able to
click next button
          And User should be able to
input "Abogdan123!" password info
          Then User should be able to
click next button
          And User should be able to
see profile
          Then User should be able to
click "Add Project" project button
          And User should be able to
input "For Delete" project number info
          Then User should be able to
input "For Delete" project name info
          And User should be able to
click "Next Step" project button
          Then User should wait until
progress bar disappear
          Then User should wait until
settings spinner disappear
          And User should wait until
skeleton disappear
          Then User should be able to
click "Save and go to next step"
project button
          And User should wait until
private switch disappear
       Then User should be able to
click "Next step" project button
      And User should be able to click
"Next step" project button
          Then User should be able to
input "b.p" mail value
          And User should be able to
choose "b.plakida@softpositive.com"
auto complete value
          Then User should see chosen
"b.plakida@softpositive.com" mail
          And User should be able to
click "Save and finish project setup"
project button
          Then User should see
scenario dropdown
          And User should see created
"For Delete" project name
```