МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Сумський державний університет
Центр заочної, дистанційної та вечірньої форм навчання
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
_____ Ігор ШЕЛЕХОВ
(підпис)
____ _____ 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 – Комп'ютерних наук,

освітньо-професійної програми «Інформатика»

на тему: Інформаційна технологія автоматизованого тестування універсальної інформаційної веб-системи

здобувача групи ІН.мз-21с Хоруженко Наталії Петрівна

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Наталія ХОРУЖЕНКО
_____
(підпис)

Керівник,
старший викладач,
кандидат технічних наук
Олег БЕРЕСТ

_____
(підпис)

Суми – 2023

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри

Ігор ШЕЛЕХОВ

_____
(підпис)

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістр

зі спеціальності 122 - Комп'ютерних наук, освітньо-наукової програми «Інформатика»
здобувача групи ІН.мз-21с Хоруженка Наталія Петрівна

1. Тема роботи: «Інформаційна технологія автоматизованого тестування універсальної інформаційної веб-системи»
затверджую наказом по СумДУ від *«20» листопада 2023 р. наказ №1308-VI*

2. Термін здачі здобувачем кваліфікаційної роботи *до 14 грудня 2023 року*

3. Вхідні дані до кваліфікаційної роботи

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Аналіз проблеми предметної області, постановка й формування завдань дослідження.
2) Дослідження підходів та технологій, що використовуються для автоматизованого тестування універсальної інформаційних веб-систем. 3) Проведення автоматизованого тестування з урахуванням результатів проведення дослідження 4) Аналіз результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти до проекту (роботи), із зазначенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» _____ 20 ___ р.

Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	Аналіз проблеми предметної області, постановка і формулювання завдань дослідження	17.10.23-20.10.23	
2	Аналіз рівнів та видів тестування	23.10.23-25.10.23	
3	Аналіз типів автоматизованого тестування	26.10.23-31.10.23	
4	Порівняльний аналіз популярних фреймворків, що використовуються для автоматизованого тестування	01.11.23-03.11.23	
5	Якісна характеристика обраних інструментів для автоматизованого тестування	06.11.23-08.11.23	
6	Розробка модулю тестів для End-to-End тестування	09.11.23-13.11.23	
7	Розробка модулю тестів для REST API тестування	14.11.23-17.11.23	
8	Розробка модулю тестів для бекенд тестування	20.11.23-24.11.23	
9	Тестування універсальної веб-системи	27.11.23-29.11.23	
10	Оформлення пояснювальної записки до кваліфікаційної роботи	30.11.23-01.12.23	

Здобувач вищої освіти _____ (підпис)

Керівник _____ (підпис)

АНОТАЦІЯ

Записка: 84 сторінки, 3 таблиці, 41 рисунок, 47 літературних джерел, 1 додаток.

Об'єкт дослідження — інформаційна технологія тестування веб-системи.

Мета роботи — розробити концепцію швидкого старту автоматизації для універсальних веб-систем, де не буде грати роль певний стек технологій. Даний підхід базується на скороченні часових, людських та матеріальних витрат, необхідних для проведення тестування, щоб забезпечити ефективну та високоякісну перевірку функціональності веб-застосунків.

Результати — розглянуто види тестування та концепцію автоматизації тестування веб-систем. Проведено аналіз фреймворків для автоматизованого тестування та прийнято рішення про інструменти, які будуть застосовані для проведення автоматизованого застосування веб-застосунку.

Здійснений вибір веб-застосунку flaskBlog для автоматизованого тестування та розроблено модель інформаційної системи. Дана робота охоплює в першу чергу набір функціональних тестів системного рівня та проведення наступних видів тестів: функціональне за допомогою Pytest, End-2-End тестування за допомогою Cypress та API тестування за допомогою Postman, Database testing.

ІНФОРМАЦІЙНА СИСТЕМА, АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ, ТЕСТУВАННЯ API, POSTMAN, END-2-END ТЕСТУВАННЯ,УПРАВЛІННЯ ТЕСТОВИМ СЕРЕДОВИЩЕМ, API, ТЕСТОВІ ЗВІТИ

ЗМІСТ

ВСТУП

Обґрунтування вибору теми роботи та її актуальність. Вибір теми обумовлений сучасними викликами та тенденціями в розробці програмного забезпечення. З поглибленням діжиталізації та ростом складності веб-систем, автоматизація тестування стає критично важливою для забезпечення високої якості програмного продукту.

Тестування програмного забезпечення вважається найважливішим етапом у життєвому циклі розробки програмного забезпечення (SDLC). Його головна мета - розкрити не відкриті і приховані помилки в програмному забезпеченні.

З кожним роком веб-розробка набуває нових розмірів та вимог, а універсальні інформаційні веб-системи стають основою цього екосистемного зростання. Концепція "клієнт-сервер" передбачає взаємодію різних клієнтських складових (веб-браузера, скрипту, який викликає API і т. д.) та програмного забезпечення на сервері, розширюючи при цьому різноманіття використовуваних технологій та шаблонів проектування.

Саме тому автоматизація тестування виходить на новий рівень і поширюється між проектами з кожним роком. Окрім unit-тестів, інженери з автоматизації тестувань проектують системні тести, які імітують дії користувача на сайтах, взаємодіють з викликами API та аналізують швидкість відповідей веб-сервісу в залежності від умов середовища та об'єму оброблюваних даних.

Актуальність даної роботи виникає із необхідності впровадження автоматизованого тестування веб-додатків. При кожному запуску автоматизований тест виконує певні дії в однаковому порядку для усунення людських помилок, таких як невідомі помилки у введених даних, що можуть впливати на результат тесту.

В той же час, на створення автоматизованих тестів від QA-інженера

вимагається знання мов програмування, шаблонів автоматизації та підходів до налаштування середовища. Наприклад, розробка функціональних автоматизованих тестів через інтерфейс веб-браузера включає організацію взаємодії з елементами сторінок для стабільної роботи тесту в різних браузерах, версіях та операційних системах. При змінах у дизайні сторінки тест повинен легко адаптуватися до нових умов. Це в свою чергу потребує високої кваліфікації QA-інженерів, але в довгостроковій перспективі це швидко окупиться.

Предмет дослідження. Засоби та методи, що використовуються для автоматизації тестування веб-системи.

Об'єкт дослідження. Інформаційна технологія тестування веб-системи.

Мета. *О*птимізація витрат ресурсів, спрямованих на тестування веб-додатків, з метою підвищення ефективності цього процесу. Даний підхід базується на скороченні часових, людських та матеріальних витрат, необхідних для проведення тестування, щоб забезпечити ефективну та високоякісну перевірку функціональності веб-застосунків. Оптимізація проходить через вибір ефективних інструментів автоматизації, раціональне планування тестових процесів та комплексне покриття різних аспектів тестування, зокрема бекенд тестування, енд-ту-енд тестування та API тестування.

Основними завданнями кваліфікаційної роботи є:

- розкрити теорію тестування веб-системи;
- провести аналіз існуючих методів тестування веб-системи;
- визначити алгоритми системи автоматизованого тестування веб-системи;
- здійснити вибір інструментарію автоматизованого тестування веб-системи;
- розробити тестові сценарії для автоматизації тестування веб-системи з використанням обраних засобів автоматизації;
- провести тестування системи автоматизованого тестування веб-системи;
- викласти отримані результати дослідження.

Методи дослідження. В рамках роботи були використані наступні методи:

аналіз літературних джерел, експериментальне тестування, аналіз результатів, порівняння, вимірювання.

Структура. Робота структурована відповідно до основних завдань.

Практичне значення. Результати дослідження та розроблені методи автоматизованого тестування веб-системи можуть бути використані для підвищення ефективності розробки та підтримки універсальної інформаційної веб-системи. Практичне значення роботи полягає у покращенні процесів тестування, зниженні витрат часу та ресурсів на ручне тестування, а також у вдосконаленні якості програмного продукту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність питання

Якість програмного продукту - одна з основних мір успіху продукту. Розробники програмного забезпечення, інженери з забезпечення якості, керівники проекту та інші зацікавлені сторони спільно працюють над створенням програмних систем, які легко сприймаються та використовуються, але при цьому мають мінімальну кількість помилок.

Враховуючи, що все більша кількість систем інтегрується з програмним забезпеченням, чи то система, критична для життя, де відмова програмного забезпечення може призвести до смерті або значних збитків для навколишнього середовища, чи інші системи, що можуть призвести до втрати грошей і часу. Згідно з звітом компанії Synopsys (2022), витрати, що виникають через погану якість програмного забезпечення, оцінюються в 2,41 трильйона доларів США тільки для США в 2022 році [1]. Щоб уникнути цих витрат і критичних збитків серед інших інженерних заходів, пов'язаних з програмним забезпеченням, програмне забезпечення повинно бути належним чином протестоване. Дослідження оцінюють, що загальна вартість тестувальних заходів в проекті розробки програмного забезпечення перевищує 40% від загальної вартості проекту. Це показує, наскільки важливою є фаза тестування в життєвому циклі розробки програмного забезпечення (SDLC) [2].

Протягом останніх років веб-сервіси, які реалізовані на основі клієнт-серверної архітектури, стали основними гравцями у розвитку розподілених інформаційних систем. У той же час, десктопні додатки, що потребують встановлення на окрему робочу станцію, виявили ряд суттєвих недоліків, таких як необхідність в процесі інсталяції, управління оновленнями та проблеми з підробкою ліцензій. Це призвело до природного переходу до клієнт-серверної технології, яка використовує веб-інтерфейс для надання

користувачам інформаційних послуг.

Зростання функціональності та складності сучасних веб-систем вимагає вдосконалених методів та засобів тестування. Автоматизоване тестування дозволяє ефективно впроваджувати тести для різноманітних сценаріїв взаємодії користувача, переконливо перевіряти взаємодію складних компонентів та забезпечувати високу якість веб-систем.

Потреба в кращій якості означає більше тиску на тестування програмного забезпечення та на інженерів-тестувальників, які про це дбають.

В такому контексті збільшується відповідальність QA-команди, враховуючи, що зазвичай замовники рідко розширюють команду тестувальників, то варто щоб команда вже мала розуміння або вже готову архітектуру автоматизації системи, для уникнення перебору підходів, фреймворків та шаблонів автоматизації, так як це може вплинути не лише на фінансову сторону, але і на якість самого продукту в цілому. Наша мета - знайти ефективний та оптимальний підхід до автоматизації, який відповідатиме вимогам проекту та забезпечує високу якість продукту.

1.2 Сутність поняття тестування програмного забезпечення

Тестування програмного забезпечення є важливою частиною верифікації та валідації, і воно відіграє ключову роль у забезпеченні відповідності реалізації програмного забезпечення заданій специфікації[4]. Відмінність між верифікацією та валідацією можна розглядати через постановку двох простих запитань (рис. (1.1):

- Verification — чи правильно ми створюємо продукт?
- Validation — чи створюємо ми правильний продукт?

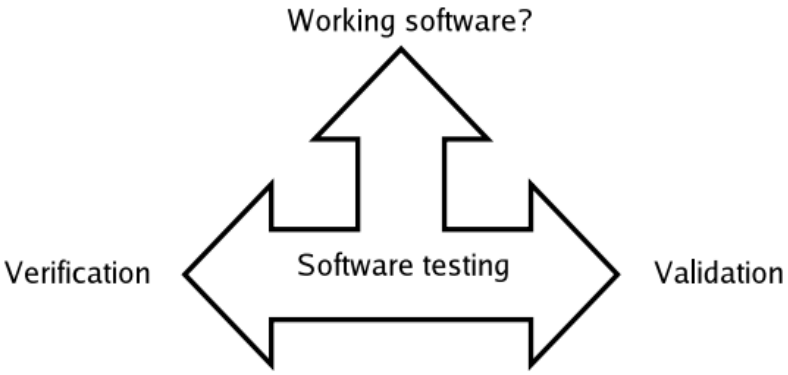

Рисунок 1.1 – Верифікація та валідація в тестуванні програмного забезпечення

Валідація програмного забезпечення оцінює, чи відповідає продукт потребам клієнта. Зазвичай вона відбувається в кінці проекту для оцінки того, чи програмне забезпечення відповідає очікуванням користувачів. Верифікація програмного забезпечення визначає, чи продукт належним чином виконує всі задумані функції, не виконуючи жодних функцій, які погіршують продуктивність системи. Це забезпечує якість програмних продуктів, і, коли включено в тестування, верифікація програмного забезпечення відбувається протягом усього циклу розробки.

Для забезпечення якості продукту програмного забезпечення важливо спочатку зрозуміти, з чого вона складається. Згідно зі стандартом ISO/IEC 25010, модель якості програмного забезпечення включає широкий спектр характеристик, таких як функціональна придатність, зручність використання, безпека, здатність до утримання тощо [5]. Кожна характеристика має кілька атрибутів, які можуть бути оцінені для отримання уявлення про рівень якості для цих характеристик (рис. 1.2).

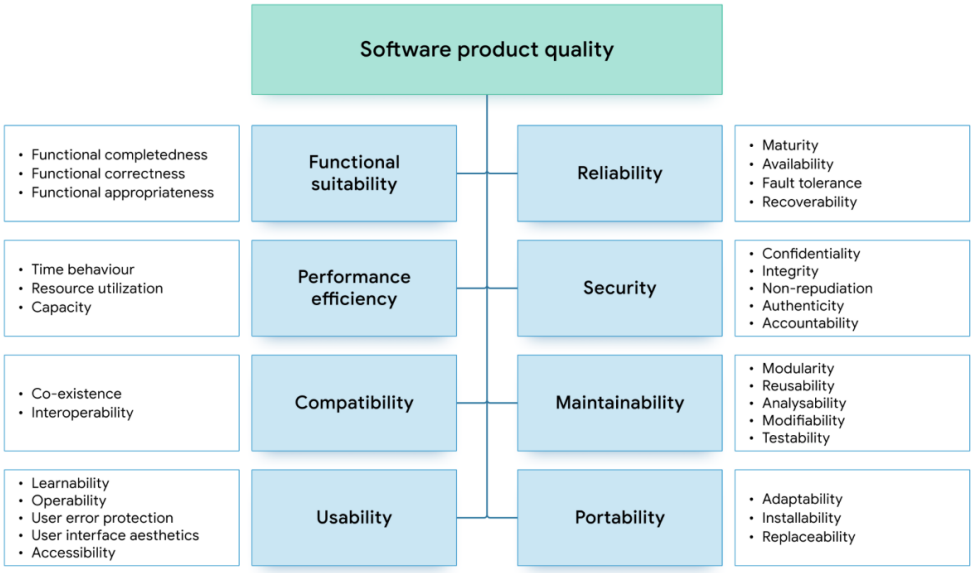

Рисунок 1.2 – Характеристики якості продукту програмного забезпечення

Оскільки ці характеристики є дуже різноманітними та обширними, їх неможливо тестувати подібним чином. Оскільки ці аспекти якості виникають з різних джерел, таких як інтерфейс, сервіси та база даних, пропонується комбінувати ці характеристики в три шари: інтерфейс користувача, сервіси та база даних (рис. 1.3).

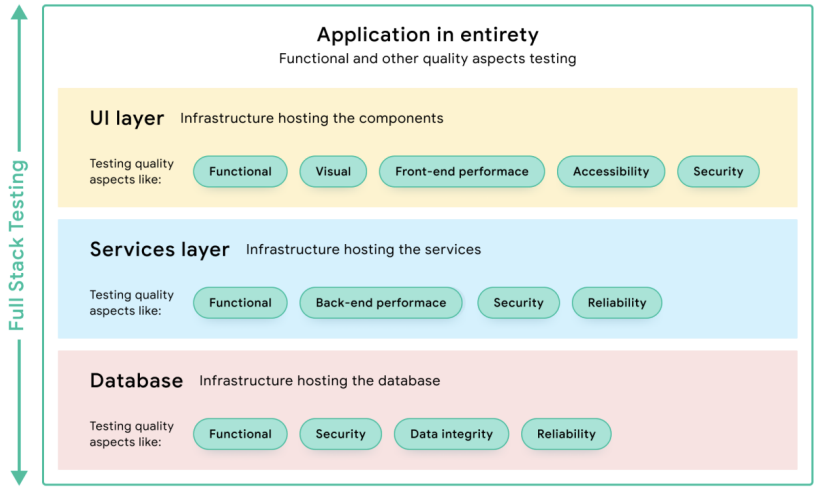

Рисунок 1.3 – Характеристики якості програмного продукту

Тестування з використанням описаних стеків дозволяє тестувальнику спрямовувати увагу на конкретні джерела даних за допомогою тестових стратегій, які найкраще працюють з ними, та виявляти вади схожого походження, що полегшує виявлення помилки та її виправлення. Це робить важливим при визначенні обсягу тестових сценаріїв в майбутньому [6].

1.3 Рівні та методи тестування програмного забезпечення

Оскільки тестування найбільш ефективне, коли воно виконується якнайскоріше, більшість практик вказує на те, що тестування слід паралельно інтегрувати в процес розробки, незалежно від того чи тестування відбувається ітеративно чи лінійно [6]. Фази розробки можуть змінюватися в залежності від конкретного програмного забезпечення, але типові рівні тестування, які перевіряють ці фази, включають модульне, інтеграційне, системне та приймальне тестування, яке може бути альфа- та бета-тестування (рис. 1.4).

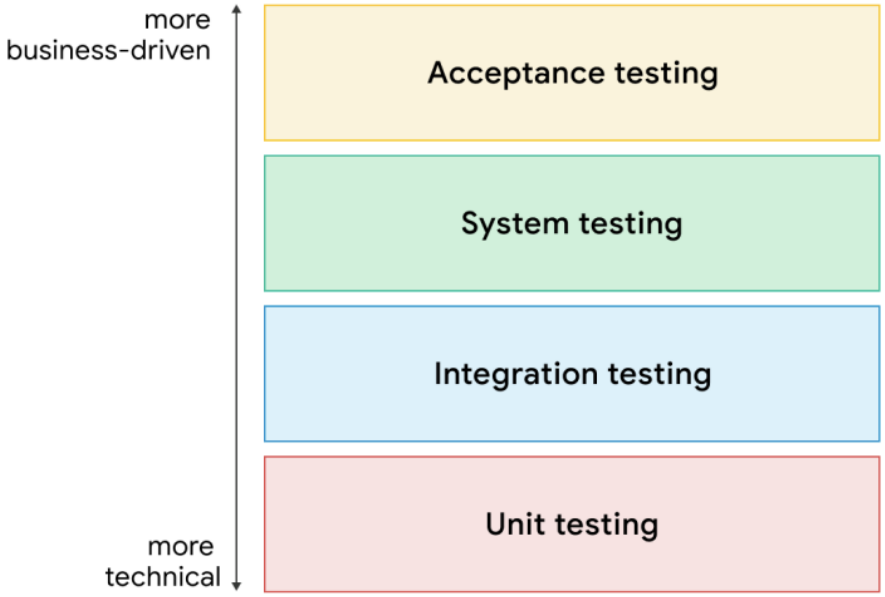

Рисунок 1.4 – Ієрархія типових рівнів тестування

Модульне тестування - це найбільш детальний рівень тестування програмного забезпечення. Код програмного продукту зазвичай складається з найменших компонентів – юнітів. Зазвичай вони будуються ізольовано, щоб бути інтегрованими на пізнішому етапі. Перед інтеграцією з іншими блоками кожен блок можна перевірити на відповідність вимогам якості. Тести можуть бути створені після написання коду або, за методом розробки, відомим як Test-Driven Development, тести можуть бути написані першими. Код будується, тестується і змінюється до тих пір, поки він не пройде всі тести [7].

Наступний рівень - це **інтеграційне тестування**. Як тільки кілька модулів було інтегровано в більшу систему, слід виконати інтеграційне тестування для виявлення дефектів у взаємодії інтегрованих компонентів, які не можуть бути виявлені на рівні модульного тестування. Вважається найкращою практикою тестувати інтеграції поетапно, не після об'єднання всіх модулів разом (відомого як підхід "великого вибуху"), оскільки це дозволяє легше виявляти проблеми інтеграції і зменшує ризик виявлення проблем пізніше в процесі. Цей рівень може бути розглянутий як інтеграційне тестування компонентів спрямоване на взаємодію між проектами, тоді як системне інтеграційне тестування вирішує питання взаємодії між системами, службами та зовнішніми організаціями [8].

Третій рівень тестування - **системне тестування**. Оскільки описані вище тести не враховують умови живого середовища або взаємодії користувача в системі в цілому, вони навряд чи виявляють будь-які збої, спричинені цими аспектами. На цьому рівні тестується програмна система в цілому. Цей етап служить для перевірки відповідності продукту функціональним і технічним вимогам і загальним стандартам якості [7].

Тестування системи має виконуватися в середовищі, яке максимально наближене до реального (продакшен).

Приймальне тестування. Це останній етап процесу тестування, на якому продукт перевіряється на відповідність вимогам кінцевого користувача та на

точність. Цей останній крок допомагає команді вирішити, готовий продукт до релізу чи ні. Хоча невеликі проблеми слід виявляти та вирішувати раніше в процесі, цей рівень тестування зосереджується на загальній якості системи, від вмісту та користувацького інтерфейсу до проблем з продуктивністю. За етапом прийняття може бути альфа- та бета-тестування, що дасть змогу невеликій кількості фактичних користувачів випробувати програмне забезпечення до його офіційного випуску [9].

Альфа - це тестування продукту обмеженою кількістю людей, можливо членами компанії (але не членам команди розробки). **Бета** - це тестування, яке проводиться обмеженою кількістю кінцевих користувачів, які записалися в програму бета-тестування.

Існують різні техніки, які можуть бути використані для тестування програмного забезпечення, такі як тестування чорної скриньки, тестування білої скриньки та тестування сірої скриньки.

Тестування чорної скриньки - це техніка тестування без будь-якого знання внутрішньої роботи застосунку, і тестувальник не знає архітектури системи і не має доступу до вихідного коду [8]. Як правило, під час виконання тесту чорної скриньки тестер взаємодіє з інтерфейсом користувача системи, надаючи вхідні дані та перевіряючи вихідні дані, не знаючи, як і де вони обробляються (рис.1.5).

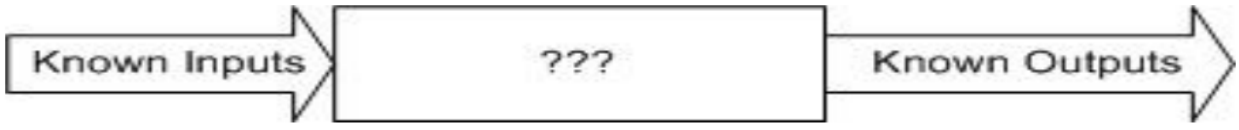

Рисунок 1.5 – Графічне зображення тестування чорної скриньки

Метою тестування чорної скриньки є перевірка правильності поведінки програмного забезпечення, яке безпосередньо підтримує щоденну

бізнес-діяльність. Цю мету часто називають охопленням поведінки. Вимоги до тестування чорної скриньки – це вимоги до програмного забезпечення, use cases та лише виконувана програма та її дані. Зазвичай ці вимоги виконуються в середній стадії циклу розробки, коли великі частини коду взаємодіють або коли програмне забезпечення придбане у вендора. Це має сенс для будь-кого (розробник або тестувальник), хто виконує тестування чорної скриньки, якщо той, хто виконує тестування, не є тим, хто написав код [13]. Тестувальники збільшують значущість тестування чорної скриньки, плануючи та виконуючи як позитивні, так і негативні тести поведінки, знаючи, що більшість виявлених користувачем помилок виникає внаслідок неочікуваної поведінки.

Тестування білої скриньки - це техніка тестування, яка перевіряє внутрішню структуру, дизайн та кодування програмного забезпечення для перевірки введення-виведення та покращення дизайну, зручності використання та безпеки. Під час тестування білої скриньки код видно тестувальникам, тому його також називають тестуванням прозорої скриньки [1.10]. Метою тестування білої скриньки є перевірка правильності інструкцій, шляхів коду, умов, циклів та потоків даних програмного забезпечення. Цю мету часто називають покриттям логіки (рис.1.6).

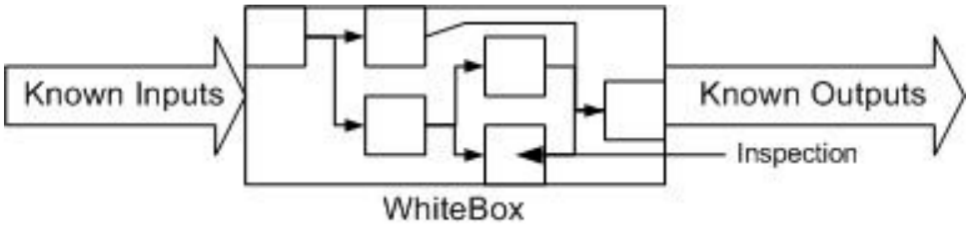

Рисунок 1.6 – Графічне зображення тестування білої скриньки

Передумовами для тестування білої скриньки є вимоги до програмного забезпечення, юз кейси, виконуваний програмний код, його дані та вихідний код. Розробник програмного забезпечення зазвичай виконує тестування білої скриньки

як розширення дій з відлагодження коду на ранній стадії циклу розробки. Розробники програмного забезпечення, як правило, спрямовані на те, щоб код працював відповідно до випадків використання, що призводить їх до відлагодження лише того коду, який, на їхню думку, працює (вибіркове покриття логіки). Чим більше покриття логіки ви досягаєте під час відлагодження, тим менше помилок буде виявлено пізніше іншими видами тестування.

Тестування сірої скриньки - це техніка тестування програмного продукту або застосунку з частковим знанням внутрішньої структури програми. Метою тестування сірої скриньки є пошук та ідентифікація помилок, що виникають через неправильну структуру коду або невірне використання додатків. Цей процес зазвичай виявляє контекст-специфічні помилки, які стосуються веб-систем. В інженерії програмного забезпечення тестування сірої скриньки відкриває можливість тестування обох сторін застосунку: презентаційного рівня, який є інтерфейсом користувача застосунку, а також рівня коду, інструменти розробника в браузері (DevTools). Це особливо корисно при інтеграційному тестуванні та penetration тестування [10].

1.4 Види тестування

Система, яка не працює належним чином, є значним витоком терпіння, грошей і часу, а також створює негативний досвід для клієнтів, і саме тому тестування програмного забезпечення становить таку важливу частину процесу розробки. Достатньо однієї помилки від однієї людини, і у клієнтів може з'явитися дефект або помилка в програмному забезпеченні. Помилка може бути в коді самому, в інструкціях або в вимогах, які спочатку були передані розробникам [6].

За ступенем автоматизації програмного забезпечення тестування буває: мануальне та автоматизоване.

Мануальні тести - ті тести, які виконуються без автоматизації. Не варто автоматизувати (або автоматизовувати в останню чергу) той функціонал, який

часто змінюється та ті тести, які не вдається зробити стабільними. А також тестові сценарії, які неможливо автоматизувати повністю, але є ті які можно проводити як вручну так і в автоматизованому режимі, наприклад візуальне тестування.

Сьогодні візуальне тестування відіграє важливу роль у тестуванні програмного забезпечення. Візуальне тестування — це спосіб порівняння фактичного зображення, що відображається на екрані, яким його бачить користувач, із вже збереженим базовим зображенням. Хтось може задатися питанням, якщо візуальне тестування — це порівняння двох зображень, і якщо наші очі достатньо точні для перевірки, навіщо нам автоматизація? Ну, неозброєним оком можна помітити видимі зміни сторінки, але оку важче помітити дрібні деталі, такі як зміни властивостей CSS, які перемістили вхідний елемент на кілька пікселів, або мінімальні піксельні зміни. Візуальне тестування передбачає візуальну перевірку того, що зміни, внесені в продукцію, не впливають на функціональність користувацького інтерфейсу [11]. Ручне тестування може бути хорошим і придатним для невеликих змін інтерфейсу користувача, але воно не дуже точне для перевірки програми з багатьма сторінками та великими змінами або різними вікнами перегляду в інтерфейсі користувача. Давайте порівняємо наведені нижче зображення, опубліковані Atlantide Photo Travel [12], на яких показано порівняння Ейфелевої вежі, і спробуйте знайти візуальні відмінності (рис.1.7).

Рисунок 1.7 – Виявлення різниці на зображенні Ейфелевої вежі

Дивлячись на деталі двох зображень, можна побачити такі відмінності: візерунки птахів, зниклих людей і навіть зниклі хмари, і на перший погляд неможливо відрізнити ці два зображення. Якби ручні тести застосовувалися до великих програм, деталі, швидше за все, були б упущені, і було б важко знайти відмінності під час порівняння.

Під час **автоматизованого тестування** для перевірки стабільності інтерфейсу користувача нам потрібно визначити наше початкове зображення (базове зображення) перед тестуванням. Потім, коли ми запускаємо тест, робиться знімок екрана поточного інтерфейсу веб-сайту та порівнюється з початковим зображенням (базовим зображенням), яке ми маємо. Тест повідомляє нам, чи є якісь відмінності між двома зображеннями, показуючи відсоткове порогове значення, яке вказує, наскільки вони відрізняються [12].

Програмне забезпечення для автоматизації тестування може вводити тестові дані в систему під час тестування, як в прикладі вище, порівнювати очікувані та фактичні результати та генерувати докладні звіти про тести. Однак в наступних циклах розробки потрібно буде повторно виконувати той самий набір тестів. За допомогою інструменту автоматизації тестування ви можете записати цей тестовий набір і відтворювати його за потребою.

Також слід враховувати, що інструменти автоматизації можуть мати додаткові витрати, і що люди, які працюють над скриптами автоматизованого тестування, зазвичай є спеціалізованими та очікують значно вищих зарплат, ніж ручні тестувальники. Однак чим більше разів автоматизовані тести можуть бути виконані, тим менше часу та зусиль людини потрібно для їх виконання порівняно з ручними тестами, і тим більше вигоди, що робить його загалом дуже корисним для будь-якого проекту середньої і довгострокової тривалості [14].

Автоматизовані тести можна виконати різними способами, такими як взаємодія з програмним забезпеченням через графічний інтерфейс користувача (GUI), використанням API або безпосередньо на рівні коду. Це означає, що одна й та ж система може бути протестована шляхом імітації введення користувача, такого як кліки мишкою, взаємодії безпосередньо з точками входу API або автоматизації кожного кодового компоненту за допомогою модульних тестів [12].

1.5 Типи автоматизованого тестування

Автоматичні тести можна застосовувати до програмного забезпечення на різних рівнях. Одним із способів ілюстрації різних рівнів автоматизації тестування є піраміда автоматизації тестування. Згідно з його оригінальною концепцією, тестова піраміда складається з трьох рівнів, які повинні включати в себе тестовий набір (від верху донизу): рівень одиниць, рівень інтеграції та рівень E2E або GUI тестування, необхідний для створення широкого охоплення системи в цілому. У цій моделі, чим вищий рівень, тим більш інтегровані та дороговартісні тести [15]. Таким чином, більшість тестів можна автоматизувати, і слід реалізувати більше тестів на нижньому рівні (рис.1.8).

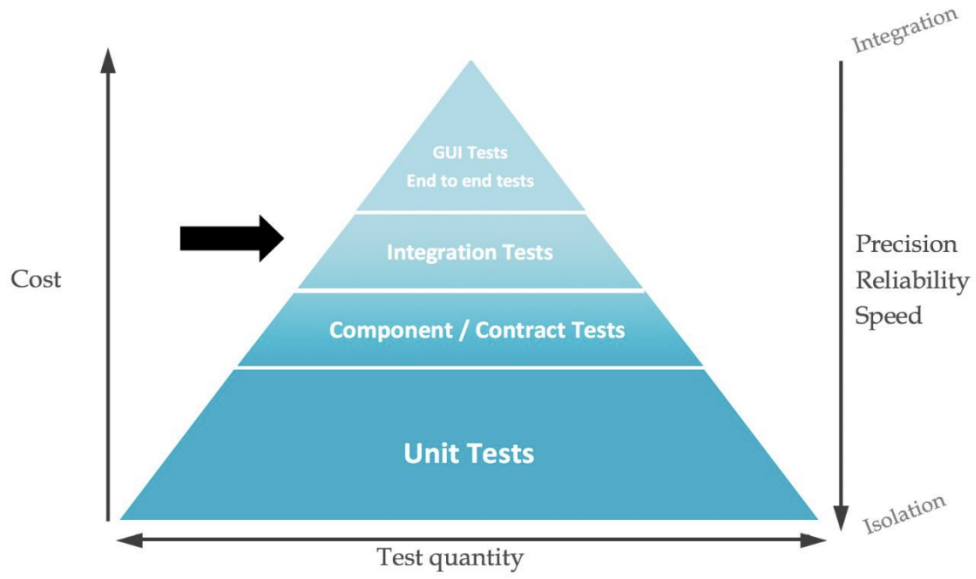

Рисунок 1.8 – Піраміда автоматизації тестування ілюструє розподіл ресурсів між різними підходами

Дані підходи до тестування реалізуються в різних співвідношеннях, і немає загальних вказівок, однак Google пропонує розділити 70/20/10: 70 % для модульних тестів, 20 % для інтеграційних тестів і 10 % для End-to-End тестування [16].

1.5.1 Автоматизація на рівні коду

Автоматизація тестування на рівні коду в основному відноситься до модульного тестування. Тестування виконується за допомогою фреймворку, який ізолює кожний модуль та перевіряє його. Одним із найчастіше використовуваних фреймворків є сім'я xUnit. Сім'я xUnit має однаковий архітектурний дизайн та підтримується в багатьох мовах програмування, таких як Java (JUnit), C (CUnit), C++ (CppUnit) та Python (pyUnit, pyTest) [18].

Pytest - це потужний фреймворк для написання та виконання тестових сценаріїв, який дозволяє розробникам ефективно перевіряти функціональність

програмних компонентів. На рівні коду тестування здійснюється шляхом написання автоматизованих тестів, які перевіряють роботу окремих функцій, методів чи класів програмного коду [19]. Pytest забезпечує зручний синтаксис тестування, просте використання та широкий набір можливостей (рис.1.9).

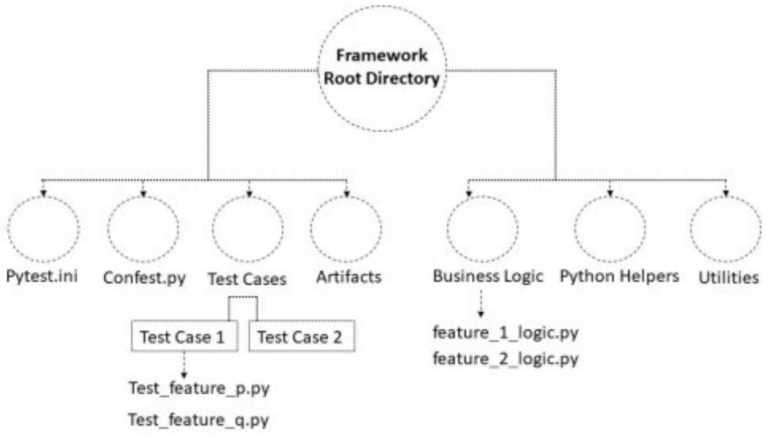

Рисунок 1.9 – Коренева структура Pytest

Як показано вище в структурі, бізнес-логіка основних компонентів фреймворку повністю незалежна від компонентів Pytest. Pytest використовує основну структуру так само, як створення екземплярів об’єктів і виклик його функцій у тестовому сценарії. Ім’я файлу тестового сценарію має починатися з «test_» або закінчуватися на «_test». Ім'я тестової функції також має бути в такому ж форматі. Звітність у Pytest може здійснюватися за допомогою звітності Pytest-html [20].

1.5.2 Автоматизація функціонального тестування через API

Мета тестування та перевірки REST сервісів полягає в тому, що клієнт ініціює HTTP-запит до кінцевої точки API, а потім аналізує отриману відповідь,

перевіряючи не лише очікувані значення, але й відповідність схемі відповіді, наявність заголовків та іншої службової інформації. Як частина інтеграційних тестів, тестування API визначає, чи відповідає логіка API очікуванням з точки зору безпеки, зручності використання, надійності, тестованості та масштабованості. Це дозволяє додаткам взаємодіяти з системами на боці сервера. Зазвичай ці додатки повертатимуть дані через свої веб-сторінки на мові розмітки гіпертексту (HTML). Коли користувачі реагують на веб-сторінку, що означає, що вони відправляють запит на користувацький інтерфейс (UI), цей запит відправляється до запитаної служби бази даних. Потім служба повертає запитані дані у форматі XML (eXtensible Markup Language) або JSON (JavaScript Object Notation). Причина у тому, що база даних повертає дані не у випадковій формулі, а в їх власних структурах [21].

Тестування API стає критичним у сфері автоматизованого тестування, особливо за методологією Agile, оскільки самі API виступають основним інтерфейсом до логіки служби чи додатка (рис.1.10 та рис.1.11).

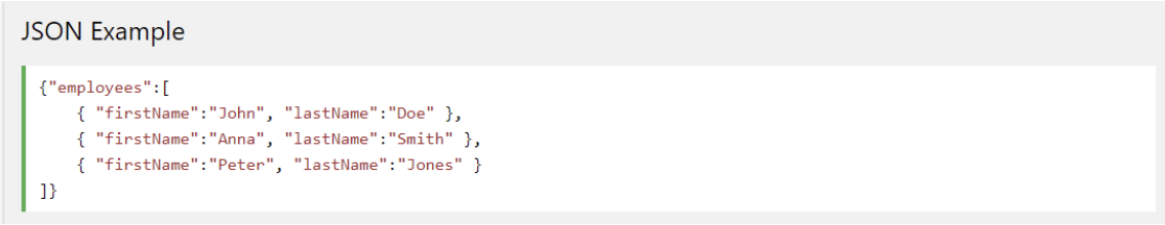

Рисунок. 1.10 – Приклад формату JSON

Рисунок 1.11 – Приклад формату XML

Ці два формати допомагають упаковувати та пересилати бази даних, стан об'єктів. Ці ресурси отримуються на основі так званого CRUD, що впорядковано означає створення, читання, оновлення, видалення. За допомогою лише методів протоколу передачі гіпертексту (HTTP) користувачі можуть створювати, читати, оновлювати, видаляти ресурси. Ось як працює API представлення стану (REST). Ще одним типом API є простий протокол доступу до об'єктів (SOAP), який викликає послуги за допомогою методу видалених процедур (RPC). Навпаки, REST API просто викликає послуги через шлях однорідного ресурсу (URL). Подібно до REST API, яке передає базу даних через HTTP, SOAP також виконує цю роботу, використовуючи HTTP, простий протокол передачі пошти (SMTP) та протокол передачі файлів (FTP). В наши дні багато компаній перейшли від SOAP API до REST API. Причина полягає в тому, що продуктивність SOAP API не така потужна, як продуктивність REST API. Переваги REST API порівняно з SOAP API - це менший код, легше викликати з JavaScript та менше завантаження центрального процесора (CPU) [22].

1.5.3 End-to-End тестування

E2E тестування - це випробування всього додатка як єдиного цілого. Ідея полягає в імітації поведінки реального кінцевого користувача під час використання додатка. Це означає, що налаштування тестування повинно бути значно складніше. Для створення тестових випадків використовуються хедлес браузери, тобто браузери без графічного інтерфейсу. Хедлес (веб браузер без графічного інтерфейсу користувача) браузери полегшують виконання тестових випадків в автоматизованому тестовому середовищі [12].

Існують два типи тестування E2E [23]: вертикальне та горизонтальне:

Найбільш поширеним є **горизонтальне тестування**, яке охоплює робочий процес чи транзакцію користувача через кілька додатків від початку до

кінця, щоб переконатися, що кожен етап відбувається, як очікується. У тестовій піраміді горизонтальне тестування E2E відбувається на найвищому рівні, тестуючи систему в цілому так, як це робить реальний користувач.У **вертикальному тестуванні** E2E: Тестування здійснюється на всіх рівнях піраміди, починаючи від модульних тестів та продовжуючи до тестів на рівні інтеграції та користувацького інтерфейсу (рис.1.12).

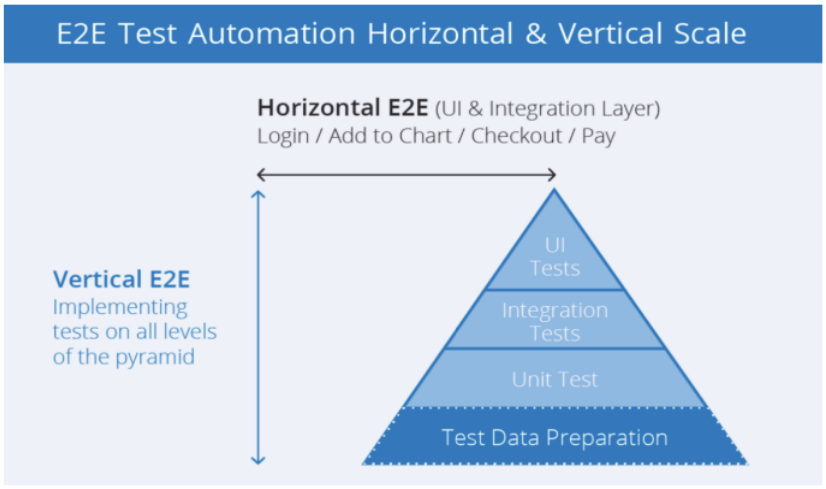

Рисунок 1.12 – Вертикальне та горизонтальне тестування E2E

Тест E2E складається з послідовності дій користувача, які імітують дії, виконані користувачем у його браузері (відкрити сторінку, клацнути кнопку, заповнити форму і т. д.). Крім того, тест E2E складається з перевірок (assertions), які перевіряють очікувані результати [23].

Наприклад, на малюнку 1.12 показана веб-сторінка flaskBlog для створення нової публікації, а на малюнку 1.13 представлено його тест E2E, який складається з 5 дій користувача (написаних за допомогою фреймворка Cypress). Цей тест E2E написаний для перевірки функціональності створення нової публікації (рис.1.13).

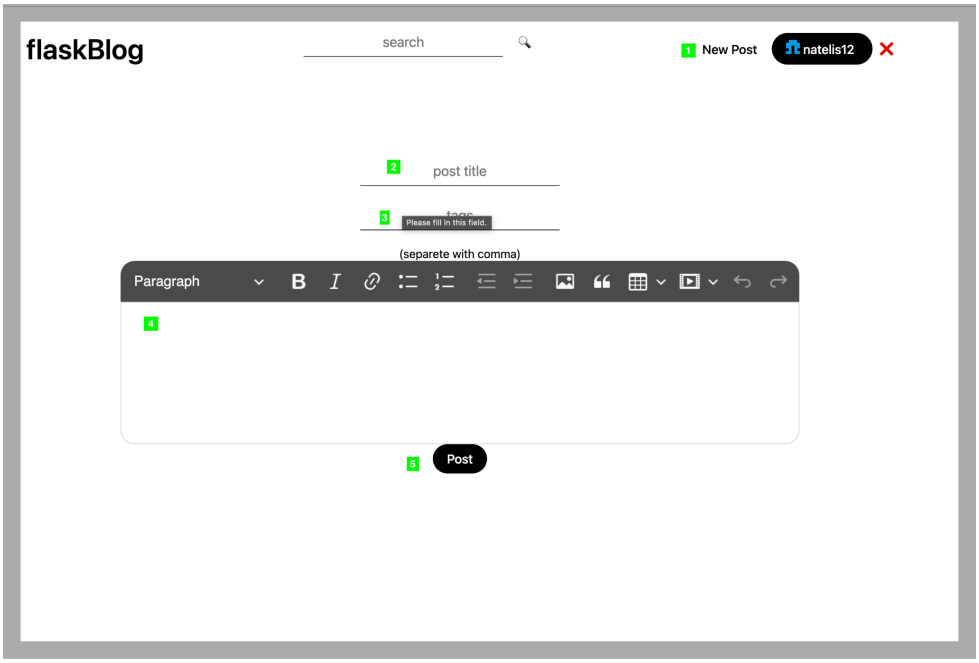

Рисунок 1.13 – Веб-сторінка flaskBlog для створення нової публікації

Під час виконання цього тестового випадку браузер автоматично запускається і переходить на цільову веб-сторінку. Цей простий приклад показує тест E2E, який може імітувати поведінку користувачів при взаємодії з веб-сторінками (рис.1.14).

```
cy.get('[href="/createpost"] > .btn').should('contain.text', 'New Post').click()
cy.get('[placeholder="post title"]').as('post title');
cy.get('#postTitle').type('Good morning');
cy.get('[placeholder="tags"]').as('post title');
cy.get('#postTags').type('#test');

cy.get('.ck-editor__editable').then(($editable) => {
  // @ts-ignore
  const editor = $editable[0].ckeditorInstance;
  editor.setData('Hello, this is a test test text.');
});

cy.get('.form > .btn').click();
cy.get('#flash').should('contain.text', 'You earned 20 points by posting');
cy.get(':nth-child(10) > .title').click()
cy.get('[href="/deletepost/9/redirect=&"]').click()
```

Рисунок 1.14 – Тестовий сценарій для створення нової публікації

2 ПОСТАНОВКА ЗАДАЧІ

2.1 Вибір мови програмування для створення тестових автоматизованих фреймворків

Вибір мови програмування для створення автоматизованих тестів — це стратегічне рішення, яке може суттєво вплинути на ефективність та обслуговуваність тестового набору. Вибір мови залежить від кількох факторів, таких як тип проекту, технічний стек, навички команди тестування, інструменти, що вже використовуються, та інші фактори. Однак, для пов'язаного тестування бекенду і фронтенду, особливо у великих та складних системах, важливо обрати мову програмування, яка дозволяє легко взаємодіяти з обома складовими.

Мови, такі як Python, JavaScript, є популярними для написання автоматизованих тестів, оскільки вони мають широкий спектр бібліотек та інструментів для тестування. Вони також дозволяють легко взаємодіяти з HTTP-запитами, базами даних та іншими аспектами бекенду [35].

Python - це високорівнева, інтерпретована, загально призначена мова програмування. Розроблено Гвідо ван Россумом та вперше випущено у 1991 році. Однією з ключових особливостей Python є чистий та лаконічний синтаксис, який полегшує читання та розробку коду. Його не тільки легко освоїти та використовувати, але його об’єктно-орієнтований підхід робить його чудовим вибором для написання автоматизованих тестів. Python також відомий своєю здатністю швидко розгортати та прототипувати програми. Це робить мову ідеальним вибором для компаній, які потребують швидкої реалізації проєктів і постійно вдосконалюють свої продукти. Python в комбінації з популярним веб фреймворком Django став де-факто стандартом для розробки вебзастосунків.
Python відомий як скриптова мова програмування, що означає
можливість швидкого написання та виконання коротких скриптів або програм без необхідності компіляції. Це робить його ідеальним для швидкого розробки та

вивчення програм, особливо в сфері автоматизації завдань та моделювання.

Python може бути використаний для тестування різних компонентів, включаючи веб-додатки, мобільні додатки, API та інше. Обширна бібліотека підтримує широкий спектр різноманітних інфраструктур та інструментів тестування, таких як PyTest, Selenium WebDriver, Robot Framework та Appium. Python дозволяє виконувати тести паралельно, що зменшує час виконання тестового набору. Python ідеально підходить для інтеграції в процеси CI/CD, забезпечуючи автоматизоване виконання тестів під час розробки та впровадження [36].

Python, яка інтерпретується, що означає безпосереднє виконання коду по рядку. У випадку помилки вона зупиняє подальше виконання та повідомляє про її наявність. Мова показує лише одну помилку, навіть якщо у програмі їх кілька, що спрощує процес налагодження.

JavaScript - це потужна мова програмування, яка використовується для розробки динамічних веб-сайтів. Вона має ефективні структури даних високого рівня та простий, але ефективний підхід до об'єктно-орієнтованого програмування. JavaScript володіє елегантним синтаксисом та динамічною типізацією, що робить його ідеальним для створення веб-застосунків і реагує на події користувача на сторінках.

Інтерпретатор JavaScript вбудований у всі сучасні веб-браузери, що робить його доступним для використання на більшості платформ без необхідності встановлення. Він також використовується для розробки серверних застосунків, завдяки платформі Node.js.

JavaScript має широкий спектр бібліотек і фреймворків, таких як React, Angular та Vue.js, які полегшують розробку великих та складних веб-застосунків. Ця мова також підтримує асинхронне програмування, що дозволяє ефективно взаємодіяти з сервером та оновлювати вміст сторінки без перезавантаження.

JavaScript може бути використаний для створення розширень для браузерів,

мобільних додатків та навіть для розробки ігор. Відкрите спільнота та велика кількість ресурсів роблять JavaScript однією з найбільш популярних та використовуваних мов програмування [37].

JavaScript є однією з найулюбленіших мов програмування для автоматизації тестування та веб-розробки, особливо інтерфейсу. JavaScript може бути використаний для емуляції різних взаємодій користувача, таких як кліки, наведення курсору, введення тексту тощо. Він пропонує безліч бібліотек з відкритим вихідним кодом, які полегшують автоматизоване тестування на всіх рівнях, включаючи тестування інтерфейсу користувача/UX за допомогою таких інструментів, як Cypress або Playwright, а також тестування на рівні API за допомогою бібліотек Supertest або RESTClient. Такі фреймворки, як Jest, Mocha та Jasmine, можуть ще більше розширити можливості JavaScript для цілей автоматизації тестування [36].

2.2 Аналіз існуючих систем автоматизації

У цей сучасний етап розробки програмного забезпечення тестування веб-сайтів стало невід'ємною частиною. Запит на надійний та безпечний інструмент автоматизації тестування зростає з кожним днем.

Розглядаються інструменти, призначені для тестування веб-сервісів, юніт-тестування коду, тестування API, неперервної інтеграції та безпеки. Визначення основних характеристик кожної системи допомагає розробникам та тестувальникам обрати оптимальний інструмент для власного проекту.

Таблиця 2.1 - Порівняння фреймворків для автоматизованого тестування

Система	**Тип системи**	**Мова програмування**	**Можливості**

Selenium WebDriver	UI	Java, Python, C#	Автоматизація дій користувача на веб-сторінці
Cypress	UI	JavaScript	Автоматизація веб-інтерфейсів та їхніх функціональностей
JUnit	Unit Testing	Java	Тестування окремих одиниць коду (юніт-тести)
Pytest	Unit Testing	Python	Тестування окремих одиниць коду (юніт-тести)
Postman	API Testing	JavaScript	Тестування API за допомогою HTTP запитів
SoapUI	API Testing	Groovy, JavaScript	Тестування веб-служб за допомогою SOAP та REST протоколів
Jenkins	CI/CD	-	Неперервна інтеграція та постачання
OWASP ZAP	Security Testing	Java	Тестування вразливостей веб-додатків

Аналізуючи існуючі системи автоматизації тестування, виявляється, що на сьогоднішній день не існує універсального рішення, яке б ідеально відповідало всім видам тестування. Кожен інструмент має свої переваги та недоліки, а також відрізняється за мовою програмування та підтримкою браузерів.

Для QA-інженерів, які розглядають можливість розпочати автоматизацію

тестування, важливо усвідомлювати, що кожній команді доводиться самостійно реалізовувати програмний проект для автоматизації тестів. Тому великого значення набуває можливість легко налаштовувати інструмент для використання в різних середовищах, і обрані інструменти не повинні вимагати складної конфігурації, а також повинні бути легкими для впровадження з нуля та використання взагалі.

Для тестування систем з постійно змінюючим API важливо, щоб обрані інструменти мали можливість читати документ OpenAPI та генерувати структуру на його основі. Це полегшить роботу з можливими змінами в API та сприятиме ефективній роботі з постійно оновлюваними інтерфейсами програмного забезпечення.

2.3 Характеристика інструменту автоматизації тестування Cypress

Cypress - це інструмент автоматизації тестування для автоматизації кінцевого до кінця графічного інтерфейсу веб-додатків. Інструмент в першу чергу призначений для спрощення роботи розробників та інженерів забезпечення якості. Розробники можуть легко синхронізувати та вирішувати проблеми за допомогою Cypress. Навіть якщо його часто порівнюють із Selenium (портативний фреймворк для тестування веб-додатків), обидва вони мають різну архітектуру [27].

Cypress використовує JavaScript для написання тестових кейсів, оскільки інструмент побудований на Node.js. Він постачається як модуль npm. Крім того, у Cypress є численні вбудовані команди для написання тестів. Ці команди дуже зручні та зрозумілі для користувачів. Він також містить методи jQuery для ідентифікації компонентів користувацького інтерфейсу та спрощення обходу та маніпулювання DOM, HTML-деревом. Крім того, він спрощує використання CSS, Ajax та обробники подій.

Цей інструмент може тестувати все, що може бути виконано в браузері. Інші

інструменти можуть працювати поза браузером та виконувати віддалені команди. Але двигун Cypress працює безпосередньо всередині браузера. Таким чином, тестувальник може бачити, як це бачать користувачі. Cypress також надає можливість подорожі у часі. Під час виконання тестових кейсів Cypress робить знімки окремих кроків. Це дуже корисно, оскільки тестувальник може навести курсор на кожну команду в тестовому засобі та переглядати історію дій [26].

Cypress спрямований на усунення невідповідностей в тестах, забезпечуючи можливість написання, відлагодження та запуску тестів у браузері без потреби додаткової конфігурації чи пакетів [28]. Cypress працює як автономний додаток і може бути встановлений на операційні системи macOS, Unix/Linux та Windows за допомогою як графічного інтерфейсу, так і інструментів командного рядка. Cypress призначений переважно для розробників, які пишуть свій код на JavaScript, оскільки він може використовуватися для тестування будь-якого додатка, що працює у браузері. Наведена нижче фігура демонструє, як працює Cypress на практиці (рис.2.1).

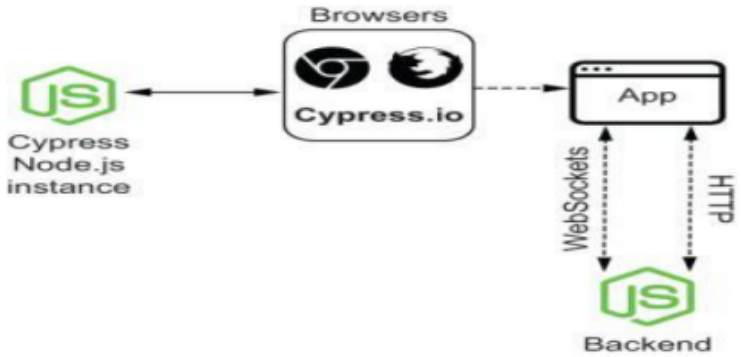

Рисунок 2.1 – Як працює Cypress на практиці

Більше того, після встановлення та ініціалізації Cypress все готово до роботи без подальшої конфігурації, і команда може розпочати написання автоматизованих тестів, щоб переконатися, що покриті найважливіші функції

додатка. Окрім того, Cypress пропонує відмінний досвід розробки, такий як режим перегляду, що означає повторне виконання тестів після збереження файлу тесту, та "подорож у часі", яка допомагає розробникам відлагоджувати тести, зберігаючи знімки додатка перед та після кожної дії і дозволяючи їм повертатися в часі, щоб точно побачити, що трапилося під час виконання тесту. Cypress також робить знімки екрану та відео, коли тест не вдається, і надає функцію автоматичного очікування за замовчуванням, що означає чекання видимості елементів перед спробою взаємодії з ними, а також чекання завершення запитів перед продовженням тестів. Фреймворк Cypress робить знімки на час виконання тесту, що дозволяє розробникам навести курсор на конкретну команду в журналі команд, щоб точно побачити, що трапилося на цьому кроці [12].

2.4 Характеристика інструменту автоматизації тестування Postman

Postman - це універсальний інструмент для розробки та автоматизації тестів API з графічним інтерфейсом користувача. Основна функція Postman - це дружелюбний для початківців робочий процес з графічним інтерфейсом користувача для створення, відправлення та отримання відповідей на HTTP-запити. Фактично Postman реалізує функціональність хедлес браузера. Всі запити та їх відповіді, разом з даними, що містяться в них, зберігаються для подальшого аналізу, і відповідні заголовки для всіх запитів генеруються автоматично або за введенням користувача [32].

Одиночні запити можна зберігати та об'єднувати в ієрархічні структури, які в Postman називаються колекціями. Колекції - це спосіб Postman керувати більшими операціями, виконуваними з API, зберігаючи всі необхідні HTTP-запити в потрібному порядку для подальшого використання [31]. Обробка динамічних даних реалізована за допомогою змінних середовища, які використовуються або глобально, або в межах окремих капсульованих середовищ, які можуть бути застосовані до колекцій (рис.2.2).

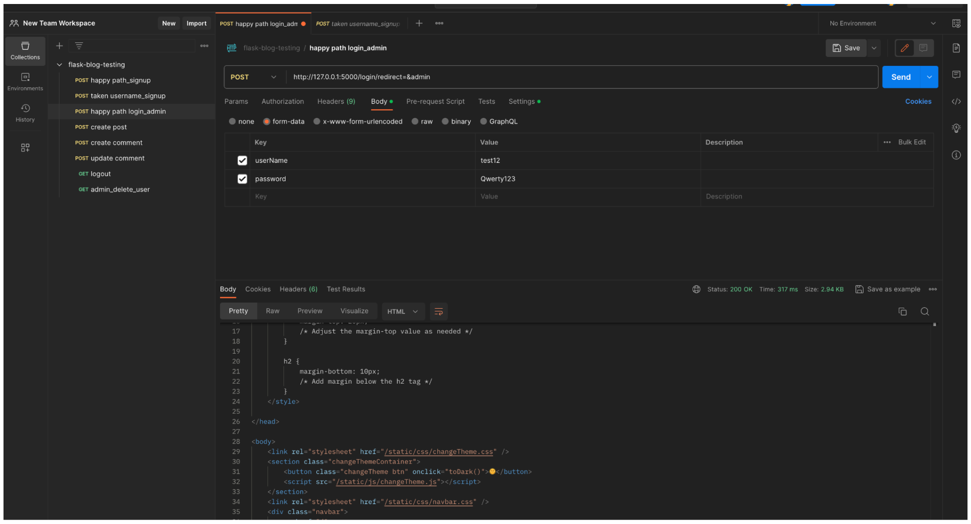

Рисунок 2.2 – Головне вікно. Postman із зразком колекції, відкритим у лівій частині екрана

Колекції можуть автоматично генеруватися з документа OpenAPI, заповнені зразками запитів для кожної доступної операції, які групуються в структуру папок з іменем, що відповідає пов'язаному кінцевому пункту API. Усі метадані, записані в документі API, також відображаються в інтерфейсі, роблячи його одночасно засобом візуалізації документації.

Postman пропонує механізм виконання скриптів, написаних на JavaScript, перед відправленням запиту та після отримання відповіді. Скрипт виконується в захищеному відокремленому середовищі із доступом до відповідних змінних середовища і, у випадку скрипта після відповіді, повної інформації про відповідь. У робочому процесі Postman автоматичне виконання тестів відбувається у скриптах після відповіді, де отриману відповідь можна проаналізувати, щоб визначити, чи була вона відправлена з очікуваними даними [32].

Скрипти можуть опціонально надаватися для всієї колекції та окремої папки всередині колекції. Ці скрипти застосовуються до всіх запитів у колекції або папці і виконуються автоматично разом з іншими скриптами. Це дозволяє автоматично виконувати деякі загальні операції для кожного запиту у колекції. Порядок

виконання скриптів починається зі скрипта на рівні Колекції (рис.2.3).

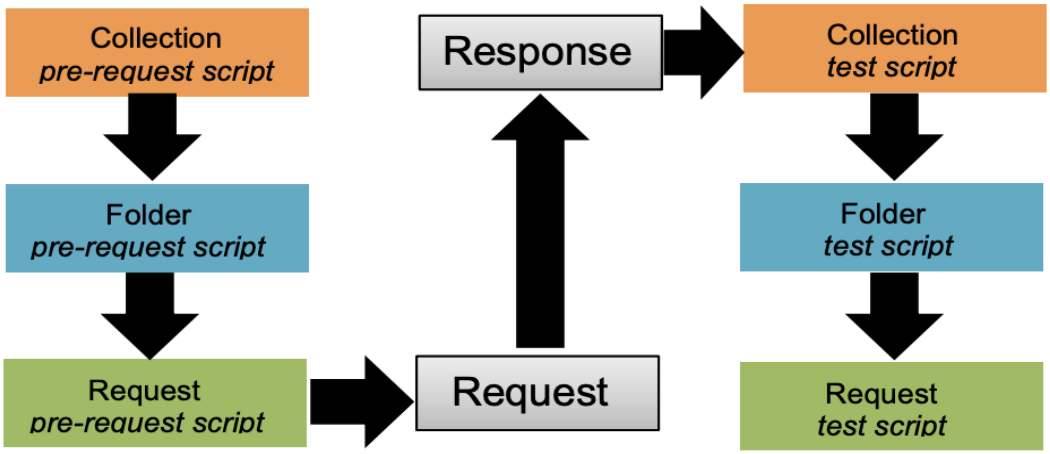

Рисунок 2.3 – Порядок виконання сценарію Postman

Редактор скриптів у Postman містить ряд часто використовуваних фрагментів тестових тверджень, які можна вставити в скрипт за один клік, щоб уникнути необхідності вводити їх вручну кожен раз. Ці скрипти є основним методом для динамічного управління середовищем, що дозволяє зберігати інформацію, отриману відповідями, в змінні для подальших запитів.

Це корисно, наприклад, при роботі з авторизаційною схемою на основі токенів, яка вимагає включення дійсного токена в усі запити. У такому сценарії новий токен авторизації може бути отриманий окремим запитом та збережений у змінну середовища для успішного відправлення подальших запитів [33].

Створення та написання наборів тестів для тестування програмного забезпечення називається дизайном тестів. Визначення архітектури тестів та ідентифікація умов тестування надають загальну ідею для тестування, яка застосовується до різних сценаріїв. Однак, коли мова йде про створення конкретного тестового випадку, він повинен бути більш конкретним. У цьому проекті тестові файли були створені на основі окремих функцій додатка.

Перш ніж перейти до правил автоматизації тестування API, важливо

пояснити деякі технічні терміни, такі як "spec", "endpoint". У термінах тестування автоматизованих API, "spec" визначається як технічна специфікація, яка роз'яснює набір вимог для заданого коду. "Endpoint" - це унікальна адреса сторінки Всесвітньої мережі в Інтернеті. Вона веде користувачів до ресурсів даних, і користувачі взаємодіють з цими ресурсами різними методами, такими як GET, PUT, POST і DELETE [31].

Варто зазначити, що специфікація повинна походити від одного ендпоінту і одного методу (GET, PUT, POST, DELETE). Причина в тому, що легше пов'язувати специфікації та посилання на REST API та переглядати покриття тестування автоматизації, переглядаючи список специфікацій. Назви специфікацій повинні виглядати так: URL ендпоінту + метод. Іноді ті ж самі ендпоінти можуть бути зовнішніми та внутрішніми без автентифікації. У цьому випадку громадські ендпоінти будуть охоплені всіма тестовими випадками, а до того ж до тієї ж самої специфікації для внутрішніх ендпоінтів додається один або кілька позитивних тестів, щоб переконатися, що вони працюють без авторизації [33].

2.5 Характеристика інструменту автоматизації тестування Pytest

Засоби тестування бекенду з використанням Pytest включають в себе різні компоненти для ефективного та комплексного тестування. Pytest - це популярний фреймворк для автоматизованого тестування в мові програмування Python, який надає зручний і ефективний спосіб написання та виконання тестів.

Поза широкою спільнотою, pytest має кілька чинників, які роблять його одним із найкращих інструментів для виконання вашого автоматизованого тестового набору на Python [34]. Філософія та можливості Pytest спрямовані на поліпшення досвіду розробника під час тестування програмного забезпечення.

Один з способів, яким творці Pytest досягли цієї мети, полягає в значному зменшенні кількості коду, необхідного для виконання загальних завдань, та

можливості виконувати складні завдання за допомогою розширених команд та плагінів. Причини використання Pytest включають наступне:

- **Простота вивчення:**

Pytest дуже простий у вивченні: якщо ви розумієте, як працює ключове слово assert мови Python, ви вже добре крокуєте до володіння фреймворком. Тести за допомогою pytest - це функції Python із префіксом "test_" у назві функції - хоча можна використовувати клас для групування декількох тестів.

- **Фільтрація тестів:**

Ви, можливо, не захочете запускати всі свої тести при кожному виконанні, особливо якщо ваш тестовий набір зростає. Іноді вам захочеться виокремити кілька тестів для нової функції, щоб отримати швидку зворотню зв'язку під час розробки, а потім запустити повний набір, якщо ви впевнені, що все працює, як заплановано. У pytest є три способи ізоляції тестів: 1) фільтрація за назвою, яка вказує pytest виконати лише ті тести, імена яких відповідають заданому шаблону; 2) обмеження директорією, що є налаштуванням за замовчуванням, яке вказує pytest виконати лише ті тести, які знаходяться в або під поточною директорією; та 3) категоризація тестів, яка дозволяє визначати категорії тестів, які pytest повинен включати або виключати.

- **Параметризація:**

У Pytest є вбудований декоратор під назвою parametrize, який дозволяє параметризувати аргументи для функції тесту. Таким чином, якщо функції, які ви тестуєте, обробляють дані або виконують загальне перетворення, вам не потрібно писати кілька схожих тестів. Для вимірювання покриття тестами використовується Pytest-cov, який допомагає визначити, наскільки тестовий код охоплює функціональність бекенду.

2.6 Метрики

Для вимірювання ефективності застосованого тестування важливо збирати

метрики до та після впровадження нових тестів. Метрики - це кількісні величини, які використовуються для оцінки та приблизної оцінки атрибутів, таких як якість чи прогрес процесу. Згідно з Guru99, коли показники успіху встановлені на ранній стадії планування, легше відслідковувати прогрес та оцінювати стан та відповідність кінцевого результату.

Оскільки існує багато можливих метрик для відстеження, важливо вибрати ті, які є найбільш інформативними та корисними в даному випадку. Щоб мати змогу виміряти успішність тестування, метрики повинні бути похідними, що означає, що ефективність тестування можна розділити на використані ресурси. Таким чином, метрики, зібрані до та після змін у тестуванні, можуть бути порівняні [12].

Серед похідних метрик, які можуть бути використані для оцінки успішності проведених тестів, слід відзначити наступні:

- **Зусилля витрачені на тестування.** Метрики для вимірювання можуть включати кількість пройдених, створених або переглянутих тестів на одиницю часу, або кількість виявлених дефектів на один тест.
- **Ефективність тестування.** Вказує на те, наскільки успішним є кожен з розроблених тестів. Вона обчислює відсоток від загальної кількості виявлених помилок, який припадає на кожен окремий тест, що повинен дати показник якості та цінності тесту.
- **Покриття тестування.** Показує, наскільки програмне забезпечення піддається тестуванню. Його можна визначити на основі відсотка вимог або кількості коду, покритого тестами. Якщо вимірюється обсяг покриття коду, цю метрику можна назвати покриттям коду. Це дає розуміння того, чи достатньо тестів в наборі тестів, чи чи є потреба в додаванні нових тестів.
- **Метрика економії тестів.** Допомагають зрозуміти, скільки було витрачено на тестування і можуть допомогти планувати розподіл ресурсів у майбутньому. Ці метрики можна визначити порівнянням часу чи витрат,

виділеного на тестування, із фактичними витратами. Ці різниці можна визначити для кожного проекту, тестового випадку, виправлення помилки чи періоду часу [13].

- **Розподіл дефектів.** Визначає, які дефекти виявлені, що допомагає в моніторингу помилок та плануванні їх виправлень. Розподіл може бути розділений за причиною, функцією, серйозністю, типом і т.д.

Вибір основної метрики для даної роботи базуються на вартості проведення вимірювань. Оскільки вже було встановлено, що процес тестування є витратним, і ресурси, які використовуються, не повинні перевищувати ризики витрат, логічно використовувати ті метрики, які вже можна легко знайти.

Оскільки покриття тестів можна легко розрахувати, вибравши одиниці програмного забезпечення та порівнявши кількість покритих тестами з повною кількістю, його можна використовувати без додаткових даних. Деякі середовища тестування пропонують автоматичне визначення покриття коду для юніт-тестів, що дозволяє знизити вартість використання покриття коду як метрики до мінімуму. Акцентування на покритті тестів також допомагає оцінити, чи достатньо тестів створено, чи чи є потреба в більш високому рівні покриття або покритті інших одиниць програмного забезпечення [8].

3 ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ

3.1 Опис проєкту

Проєкт, що тестується - це веб-застосунок "flaskBlog". Як випливає з назви веб-застосунку. Це мікро-блог, написаний з використанням популярного фреймворку для мови програмування Python - Flask. Цей блог спроектовано у відповідності з архітектурним підходом REST та з використанням HTTP(S) для обміну повідомленнями (даними) між клієнтом і сервером.

Основні компоненти проєкту знаходяться у директорії "app".

1. **Мови програмування:**

- Python: Основна мова програмування для бекенду.
- JavaScript + HTML, CSS: Використовуються для розробки фронтенду та стилізації.

2. **Основні функції та можливості:**

- Сторінка користувача з можливістю логіну, видалення, зміни імені та ін.
- Адмін-панель та інструментарій для управління додатком.
- Система авторизації, скидання та зміни паролю користувача.
- Використання CKEditor 5 для редагування постів.
- Пошукова стрічка, можливість редагування, видалення та створення постів.
- Система коментарів та можливість їх видалення.
- Логування, відлагоджувальні повідомлення та інші розвинені функції.

3. **Вимоги:**

- Flask, Passlib, WTForms.
- Використання SQLite як DB engine.
- Python версії 3.10 або новіше (рис 3.1).

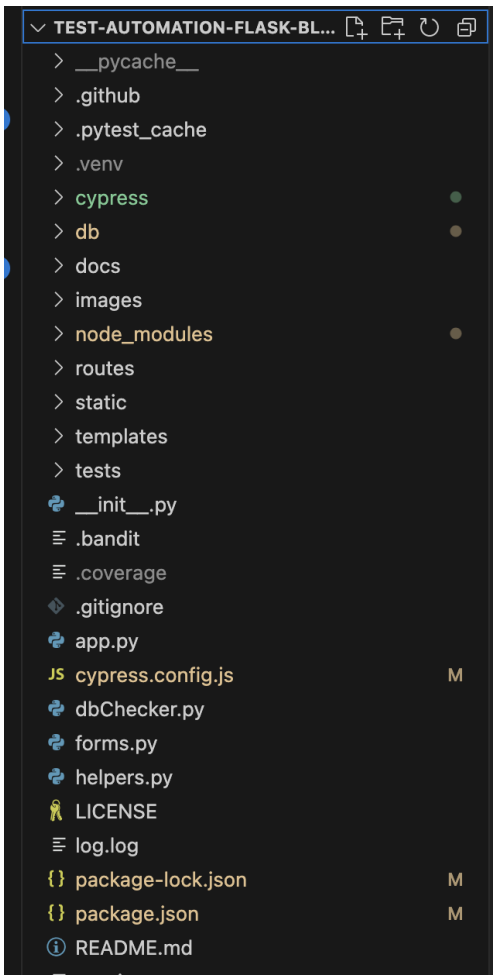

Рисунок 3.1 - Структура проєкту веб-застосунка.

3.2 Модель інформаційної системи

FlaskBlog є блоговим застосунком, розробленим з урахуванням сучасних вимог та потреб користувачів. Цей веб-додаток спрямований на створення надійного та зручного середовища для обміну інформацією, висловлення ідей та підтримки спільноти.

Нижче наведено детальну модель інформаційної системи FlaskBlog, включаючи ключові можливості.

Таблиця 2.2 - Модель функціональності FlaskBlog

Категорія	Функція	Опис
Користувач	User Sign Up	Процес реєстрації нового користувача.
	User Login	Автентифікація зареєстрованих користувачів.
	User Points	Відображення балів або рейтингу користувача.
	User Delete	Видалення облікового запису користувача.
	User Log Out	Безпечний вихід із системи.
	User Page	Сторінка користувача з відображенням основної інформації.
	User Name Change	Зміна імені користувача з урахуванням безпеки.
	User Settings Page	Сторінка налаштувань для персоналізації облікового запису.
	User Profile Pictures	Завантаження та зміна фотографії профілю.
Адміністратор	Admin Panel	Панель адміністратора з розширеними можливостями.
Dashboard	Dashboard Page	Загальний огляд даних та статистика для користувача.
Безпека	Password	Зміна паролю з урахуванням безпеки.

	Change	
	Password Reset	Відновлення паролю через електронну пошту.
Створення та Редагування Змісту	CKEditor 5	Використання текстового редактора для створення постів.
	Post Edit	Редагування створених постів з урахуванням форматування.
	Post Views	Відображення кількості переглядів постів.
	Post Delete	Видалення постів з можливістю відміни.
	Post Creation	Створення нових постів з можливістю прикріплення зображень.
Коментування	Comment	Додавання коментарів до постів для взаємодії з користувачами.
	Comment Delete	Видалення коментарів з можливістю відновлення.
Інші Можливості	Dark/Light Themes	Вибір теми оформлення для зручності користувача.
	Responsive Design	Адаптивний дизайн для оптимального відображення на різних пристроях.

3.3 Реалізація бекенд тестування за допомогою pyTest

Бекенд веб-застосунку Flask Blog написаний на мові програмування Python[39], та використовує популярний фреймворк Flask [38] у якості основного рішення для створення шляхів, що забезпечують комунікацію користувача з бекенд частиною застосунку. Для валідації вхідних даних від користувача та рендеренгу форм для вводу даних використовується бібліотека WTForms [40]. У якості СУБД - SQLite [41].

Одним із методів досягнення впевненості в коректності взаємодії між компонентами бекенду веб-застосунку є здійснення функціонального тестування.

Для мови програмування Python, яка є доволі популярною у веб-розробці, однією із самих відомих бібліотек для проведення різних видів тестування (функціонального та юніт включно) є pytest [34], її я і використаю для проведення тестування бекенд частини застосунку.

Вона дозволяє доволі швидко створювати тест-кейси, а також має широкий набір інструментів для максимальної автоматизації процесу тестування. Ця бібліотека має можливість гнучкого налаштування - через консоль або ж використанням файла конфігурації, наприклад, ***setup.cfg***.

Там ми можемо зазначати рівні ***verbosity,*** тобто наскільки детально інформацію про процес тестування ми бажаємо отримувати в консоль під час самого процесу тестування, або ж режим дебагінгу (***pdb - python debugge***r), який дозволяє “зупинятися” на тій строці коду, яка призвела до помилки, додавати змінні оточення для тестів, що дозволяє імітувати наявність, наприклад, секретів, які застосунок отримує як значення змінних оточення реального серверу, та багато іншого.

Крім того, для визначення рівня покриття тестами коду веб-застосунку я використовую бібліотеку coverage [41], а також плагін для pytest - ***pytest-cov*** [42], що дозволяє цим бібліотекам працювати разом максимально ефективно (рис.3.2).

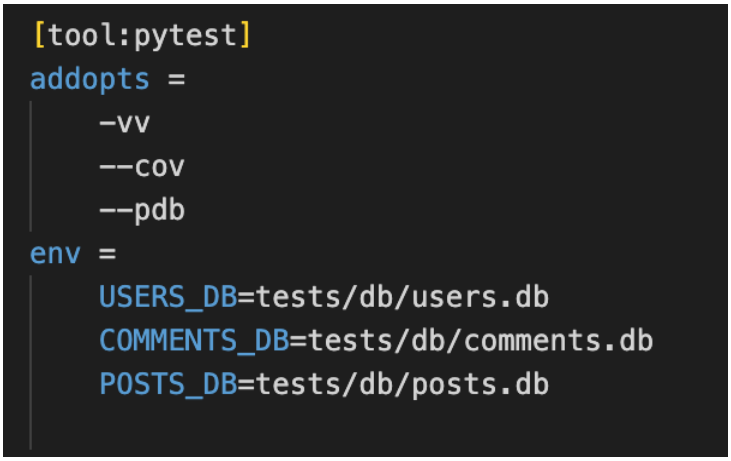

Рисунок 3.2 - Конфігурація бібліотеки pytest

Coverage як і pytest також має можливість конфігурації як через консоль під час запуску, так і через файл конфігурації (setup.cfg в моєму випадку).

Враховуючи, що “серцем” бекенд частини RESTful веб-застосунку, тобто такого, що побудований на клієнт-серверній архітектурі із обміном повідомленнями (даними) через HTTP протокол, є його шляхи, я вважаю, що функціональне тестування в першу чергу треба провести саме для них. Бо саме коректність їх поведінки буде вцілому визначати коректність поведінки бекенд частини веб-застосунку.

Для того, щоб визначити, що функціональне тестування буде проводитися для каталогу, в якому зберігаються модулі в яких описані (оголошені) шляхи (routes) веб-застосунку, я зазначу цей шлях в конфігурації бібліотеки coverage (рис.3.3).

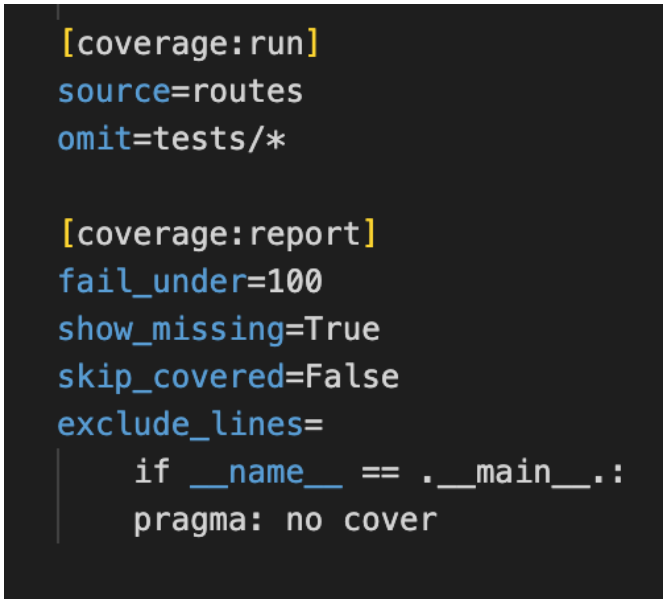

Рисунок 3.3 - . Конфігурація бібліотеки coverage (тільки файли в папці “routes”)

Крім того, я зазначу, що якщо покриття тестами коду (логіки) у визначеному шляху буде нижче 100% - бібліотека coverage видасть помилку (fail_under).

Для того, щоб повністю покрити тестами увесь код модулів, що описують API шляхи, мені знадобилося написати 81 тест.

Ось так, наприклад, виглядає тест, що перевіряє шлях “/accountsettings”.

```
from tests.conftest import app
def test_account_settings_authenticated_user(mocker):
        mocker.patch('app.getProfilePicture',
        return_value='/static/default_profile_picture.png')
        with app.test_client() as client:
          with client.session_transaction() as sess:
            sess['userName'] = 'test_user'
          response = client.get('/accountsettings')
        assert response.status_code == 200
        assert b'change your username' in response.data
        assert b'changepassword' in response.data
```

```
assert b'delete your account' in response.data
```

Мета цього тесту (як і йому подібних для інших логіки/модулів застосунку) - переконатися що фактична поведінка елементу логіки веб-застосунку відповідає очікуваній. Більш конкретно - при створенні запиту до локально піднятого серверу з блогом по шляху “/accountsettings” методом GET ми маємо отримати код відповіді 200, та певні елементи у даних відповіді (response.data), такі як: “change your username”, “changepassword”, “delete your account”.

Слід зазначити, що цей конкретний тест кейс тестує випадок, коли користувач авторизований. Це перевіряється в імплементації шляху через наявність імені користувача у сесії, для чого ми штучно додаємо цю інформацію в сесію тестового клієнта:

```
with client.session_transaction() as sess:
    sess['userName'] = 'test_user'
```

Хотілося б додати, що тестовий клієнт вбудований в клієнт (application) Flask, який ініціалізується наступним чином: with app.test_client() as client

Крім того, можна побачити що app імпортується зі модуля conftest,py, про який я би додати трохи більше інформації. Зазвичай цей модуль використовується бібліотекою pytest для “швидкого” пошуку та імпорту його фікстур (fixtures) - механізмів, які дозволяють підготовувати інфраструктуру для виконання тестів. Крім того, у цьому файлі зоручно агрегувати інші функції, що використовуються в різних модулях з описаними тест кейсами.

В моєму випадку існує всього одна фікстура, яка видаляє всі дані із тестової бази даних перед та після виконання кожного теста:

```
pytest.fixture(scope='function', autouse=True)
def truncate_databases():
  # Truncate the databases before running tests
  truncate_database(USERS_DB)
```

```
    truncate_database(COMMENTS_DB)
    truncate_database(POSTS_DB)
    # Run the tests
    yield
    # Truncate the databases after running tests
    truncate_database(USERS_DB)
    truncate_database(COMMENTS_DB)
    truncate_database(POSTS_DB)
```

Також у цьому файлі описані інші вспоміжні функції, наприклад така, що дозволяє швидко створювати користувача безпосередньо у базі даних (а не через виклик шляху), що пришвидшує процес розробки, та певною мірою підвищує ізоляцію функціоналу одного шляху від функціоналу іншого.

Наприклад, якщо мені потрібно створити користувача для перевірки функції логіну (аутентифікації), то мені не потрібно попередньо викликати шлях “/login/redirect=<direct>”.

```
def create_test_user(user_name, email, password, role=None):
    connection = sqlite3.connect(USERS_DB)
    cursor = connection.cursor()
    hashed_password = sha256_crypt.hash(password)
    cursor.execute(
        'INSERT INTO users (userName, email, password, role) VALUES (?, ?, ?, ?)',
        (user_name, email, hashed_password, role)
    )
    connection.commit()
    connection.close()
```

Повертаючись до питання фікстур, та попередньо описаного тесту “def test_account_settings_authenticated_user”, хотілося б зазначити, що в цьому тесті використана фікстура mocker з плагіну pytest-mock [44], яка по своїй суті є більш зручною обгорткою довколо патч-API - mock з стандартної бібліотеки Python - unittest [45]. Вона автоматично “прокидається” в тест і не потребує додаткового імпортування на рівні модуля (один з плюсів використання фікстур). В цьому (та інших тестах), вона використовується для того, щоб ізолювати код від непотрібних залежностей під час тестування (ті які не мають відношення до суті тесту):

```
mocker.patch('app.getProfilePicture',
return_value='/static/default_profile_picture.png')
```

Тут ми підмінили (mocked) реальну функцію “getProfilePicture” заглушкою, тобто кожного разу колу вона буде виконуватися застосунком, вона не буде запускати код, описаний в тілі функції, а буде повертати вже заготовлену відповідь - “/static/default_profile_picture.png” (шлях до картинки). Цей прийом позбавляє нас від необхідності генерувати додаткові дані (які не мають прямого відношення до суті тесту) в базі даних при підготовці тесту (інакше застосунок видасть помилку).

Тестування з використанням pytest відбувається після введення в консолі команди “pytest . ” та виглядає наступним чином (рис.3.4):

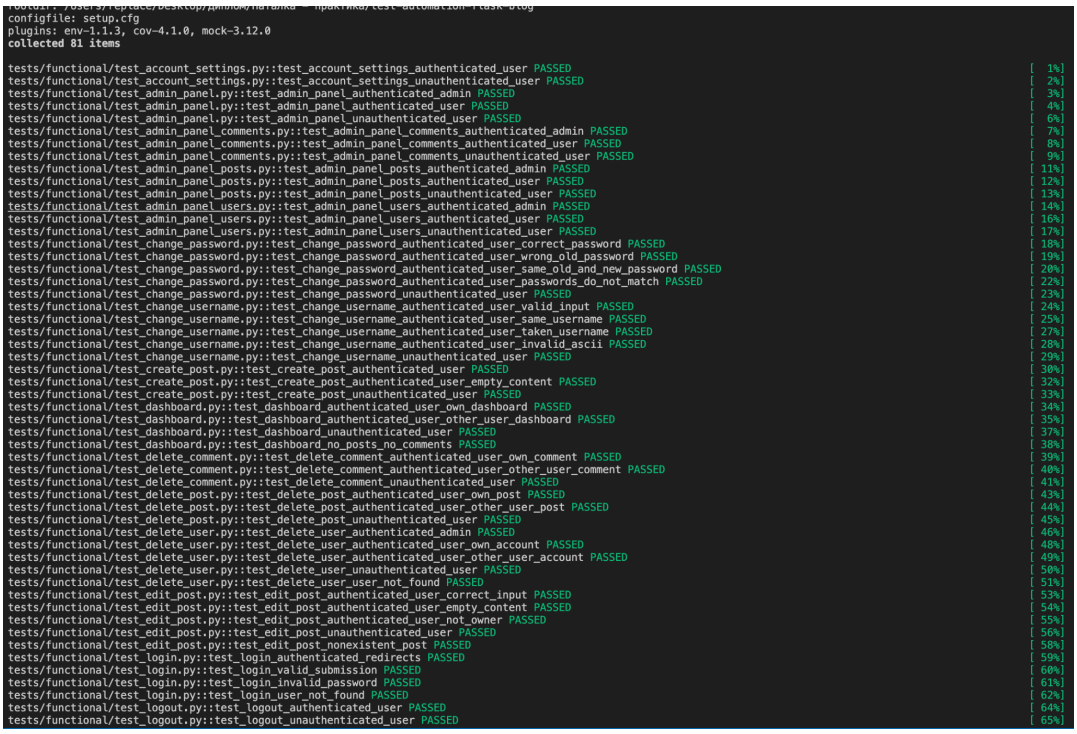

Рисунок 3.4 - Процес виконання тестів pytest

За результатами виконання тестів ми маємо 100% покриття та 0 помилок (рис.3.5):

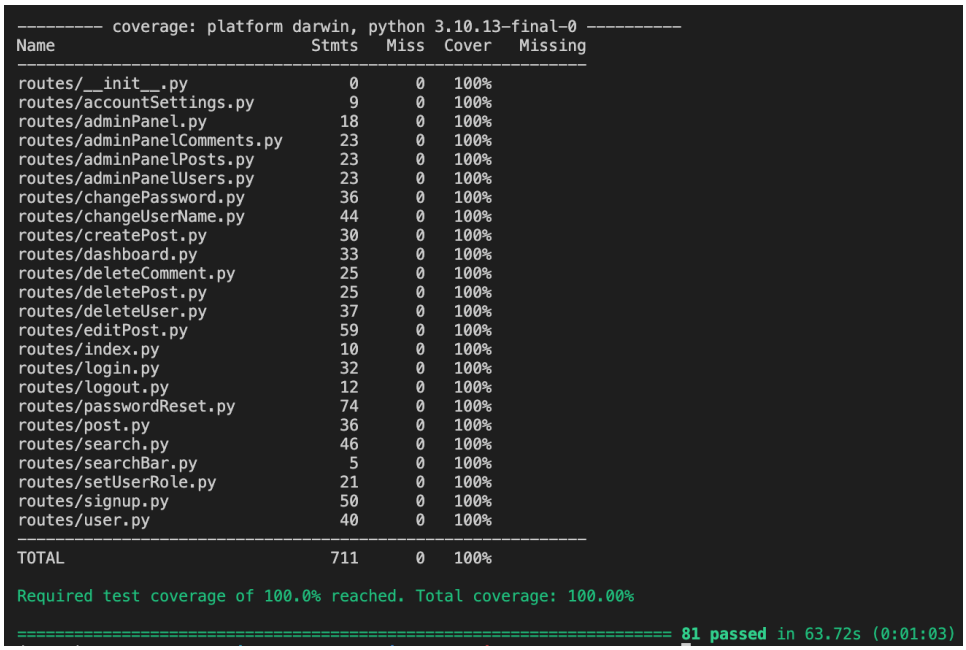

Рисунок 3.5 - Результати виконання тестів pytest

Для перевірки коректності роботи інших модулів (в частині яка не покрита функціональними тестами) я використаю ***юніт (unit) тестування***, тобто тестування ізольованих частин програмного забезпечення, таких як функції, класи або методи. Цей методі особливо стає в нагоді, коли деякий функціонал не вписується в “звичайний” workflow застосунку.

Наприклад, завдяки юніт тестуванню я можу перевірити випадок, коли функція “dbFolder” з модуля “dbChecker.py” не може знайти папку “db” (де зберігаються у файловій системі бази даних SQLite), що не характерно навіть для функціонального тестування, а видаляти спеціально каталог було б на мою думку надлишковим.

```
def test_db_folder_not_found(mocker):
    mocker.patch('dbChecker.exists', return_value=False)
    mocker.patch('dbChecker.mkdir')
    mock_message = mocker.patch('dbChecker.message')
    dbFolder()
    mock_message.assert_called_with("2", 'Folder: "/db" CREATED')
    mock_message.call_args_list
```

Такий тест допомагає пройтися по гільці логіки функції під “case False”:

```
def dbFolder():
    match exists("db"):
        case True:
            message("6", 'Folder: "/db" FOUND')
        case False:
            message("1", 'Folder: "/db" NOT FOUND')
            mkdir("db")
            message("2", 'Folder: "/db" CREATED')
```

Тут ми здійснюємо перевірку чи викликався функція “message” з аргументами “"2", 'Folder: "/db" CREATED"”. Якщо це так, то ми переконаємося що “case False” дійсно виконується коректно з точки зору даних, які ми можемо “перехопити”.

Для повного покриття веб-застосунку тестами додатково до вже написаних 81 тестів, що покривали роути, мені знадобилося дописати ще 8 юніт тестів (рис.3.6 та 3.7).

```
[coverage:run]
source=.
omit=tests/*

[coverage:report]
fail_under=100
show_missing=True
skip_covered=False
exclude_lines=
    if __name__ == .__main__.:
    pragma: no cover
```

Рисунок 3.6 - Оновлена конфігурація бібліотеки coverage (усі файли)

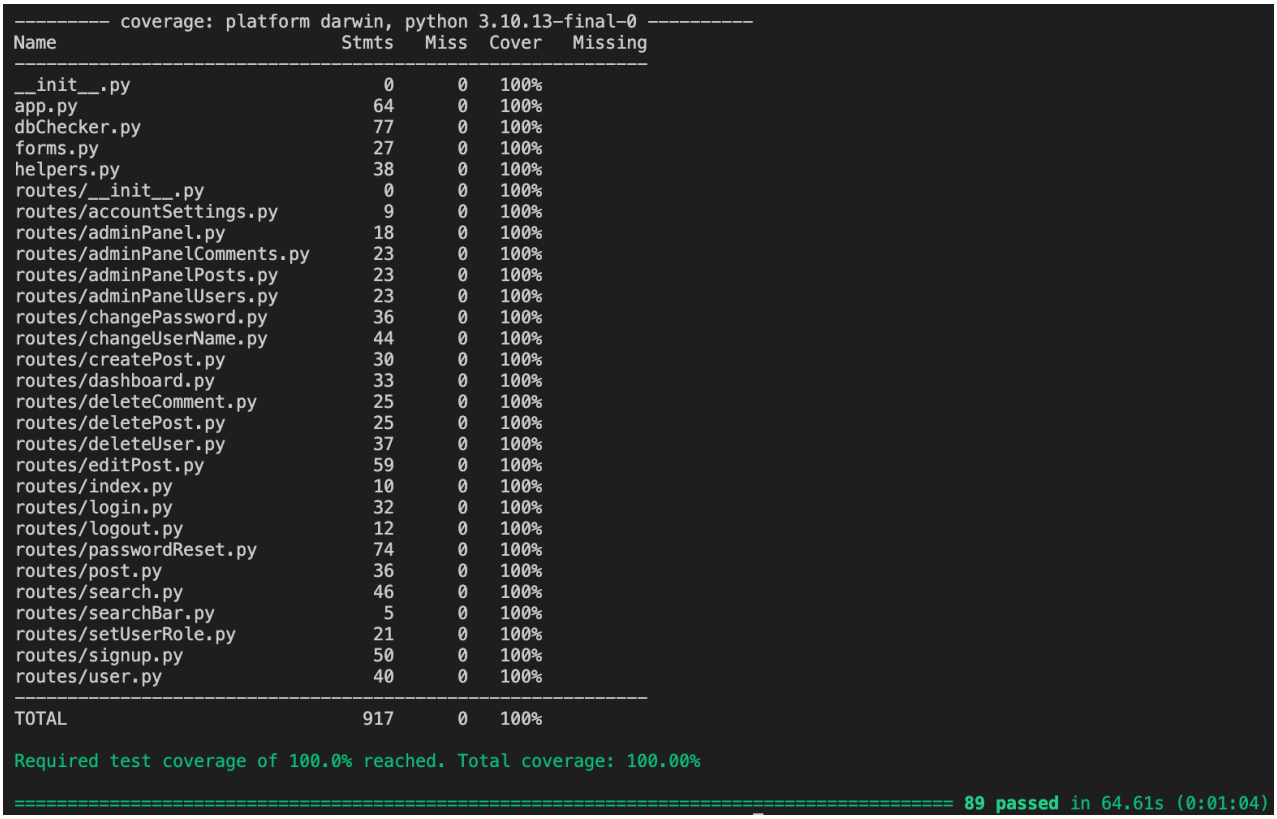

Рисунок 3.7 - Результати повторного виконання тестів pytest

Як можна побачити Python (бекенд) частина веб-застосунку повністю покрита тестами, що має значним чином підвищити стабільність його функціонування.

Також хочу зазначити, що робота над тестами pytest разом з бібліотекою coverage допомагає виявити одна строку коду у роуті “deletePost.py” ніколи не викликається. Навіть при описі всіх можливихи варіантів використання застосунку у тестах, coverage показував, що ця строка коду не покрита тестами. Додатково я перевірила і підтвердила це через UI блогу. Після цього “зайва” строка була прибрана з коду застосунку (рис.3.7).

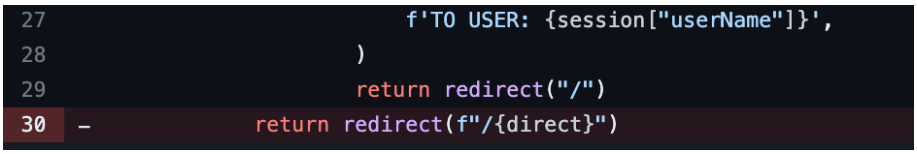

Рисунок 3.7 - Видалення частини коду, що не використовується (redundant)

Крім того, я інтегрувала pytest в мій ***CI/CD*** воркфлоу (workflow) через Github Actions додавши етап проходження тестів “Tests with pytest” в конфігураційний файл “main.yml” (рис.3.8).

```yaml
name: Python package      Nataliia Khoruzhenko, 6 months a

on: [push]

jobs:
  build:

    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: ["3.10"]

    steps:
      - uses: actions/checkout@v3
      - name: Set up Python ${{ matrix.python-version }}
        uses: actions/setup-python@v4
        with:
          python-version: ${{ matrix.python-version }}
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Linting
        run: |
          flake8 && isort --check --diff .
      - name: Test with pytest
        run: |
          pytest .
      - name: Security check
        run: |
          bandit -r .
```

Рисунок 3.8 - Зміст main.yml

Тепер для того, щоб зміни могли бути залиті (merged) в головну гілку (branch) проекту - main, потрібно щоб відповідний Pull Request успішно пройшов всі етапи, перелічені в “main.yml”, в тому числі і тести pytest (рис.3.9, 3.10).

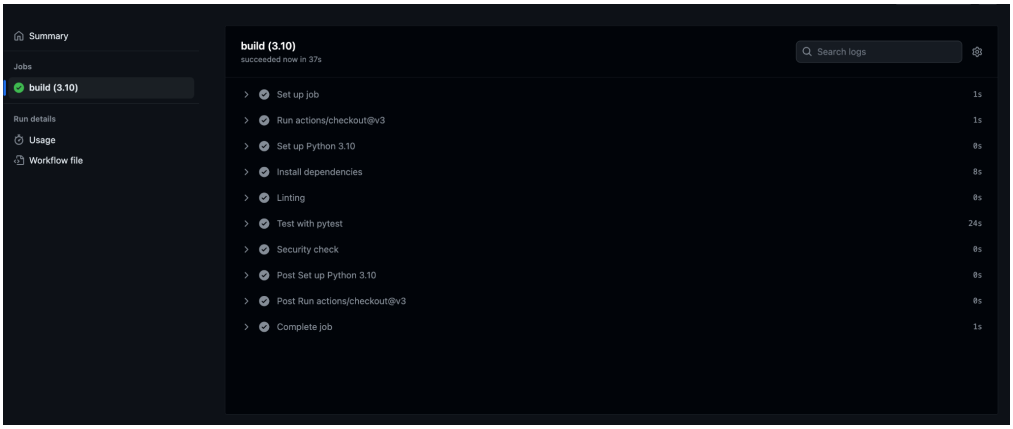

Рисунок 3.9 - Успішне завершення обов’язкового білду (build) на Github Actions

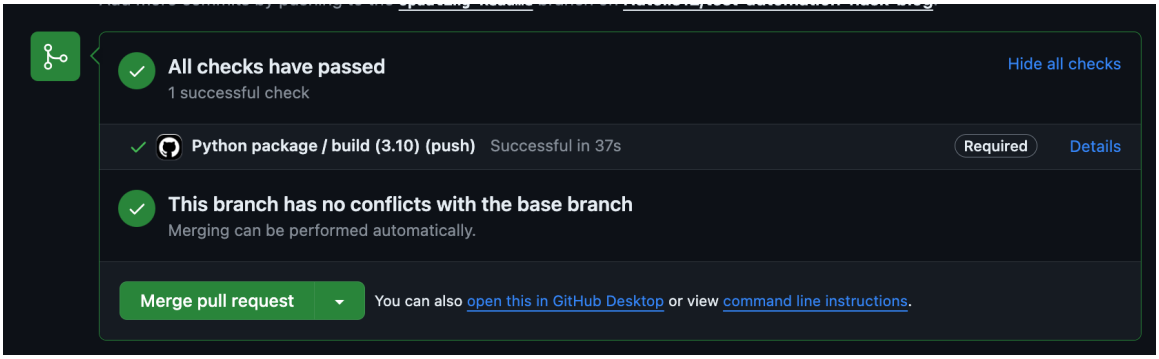

Рисунок 3.10 - Всі обов'язкові перевірки пройшли успішно і PR можливо залити (merge) в гілку main

3.4 Реалізація модуля API тестування

Postman - це один з найбільш важливих інструментів для тестування API, який надає величезний потенціал для дослідження, налагодження та валідації HTTP запитів (в т.ч. GraphQL, REST, SOAP) та WebSockets.

Організація запитів в колекції Postman дозволяє ефективно керувати запитами для подальшого використання, уникнення зайвого витрачання часу на створення нових запитів. Також у колекціях можна включати JavaScript-код для зв'язування запитів або автоматизації загальних робочих процесів. За допомогою сценаріїв можна візуалізувати відповіді API у вигляді діаграм та графіків.

Postman надає зручні інструменти для створення та запуску тестів безпосередньо в інтерфейсі користувача, або як частину конвеєра CI/CD. Можливості Postman включають написання функціональних тестів, інтеграційних тестів, регресійних тестів та інших.

В рамках даної роботи я продемонструю тестування API з використанням інструменту Postman на прикладі функціональних тестів.

Розпочнемо з налаштування середовища. В рамках колекції ми призначаємо змінну з ім'ям host та значенням "http://127.0.0.1:5000" в local Environments, та зберігаємо. Тепер можемо використовувати цю змінну "host" у запитах, вказавши "{{host}}" в полі URL для кожного запиту. Це спростить управління та

масштабування, дозволяючи змінювати базовий URL в одному місці, а вони автоматично оновлюватимуться в усіх запитах, що використовують цю змінну (рис.3.11).

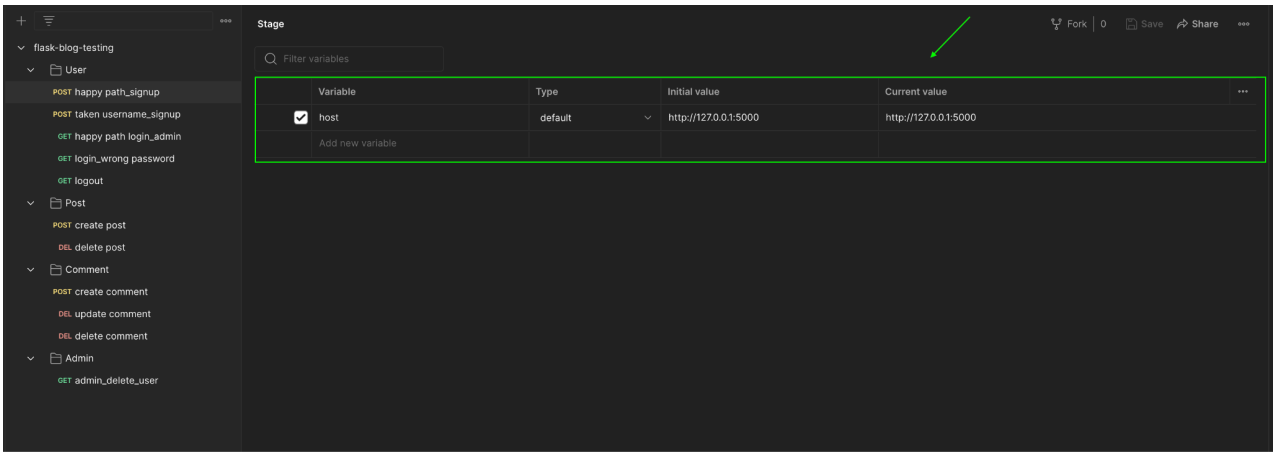

Рисунок 3.11 - Налаштування середовища

Наступним кроком визначаємо мінімальний набір тестових сценаріїв, який, в свою чергу, покриває основний функціонал програми.

Таблиця 3.3 Тестові випадки для проходження в Postman

Section	**Query**	**Functionality**	**Test case**	**Description**
User	POST	Sign up	happy path	user can successfully sign up
	POST	Sign up	registration with taken username	ensures proper handling when a user tries to sign up with an already taken username

	GET	Login	happy path	user can successfully log in
		Login	login with wrong password	verifies the system's response when a user attempts to log in with an incorrect password
Blog	POST	Create post	user should be able to create a post	validates the user's ability to create a new post
	DELETE	Delete post	user should be able to delete a post	confirms that users can delete their own posts
	PUT	Update post	user should be able to update a post	verifies the user's capability to edit and update their existing posts
Comments	POST	Create comment	user should be able to create a comment	tests the user's ability to add comments to posts
	DELETE	Delete comment	user should be able to delete a comment	confirms that users can remove their own comments
Admin	DELETE	Settings	admin should be able to delete a specific user	ensures that an admin has the authority to delete a particular user

Далі відправимо POST-запит для створення користувача. Для початку створимо запит з body form-data який має такі ключі та значення:

- userName
- email
- password
- passwordConfirm

Після отримання відповіді від сервера відбувається перевірка статус коду та тіла відповіді (response body). Надалі ця перевірка буде відбуватися для всіх інших тестових сценаріїв (рис.3.12).

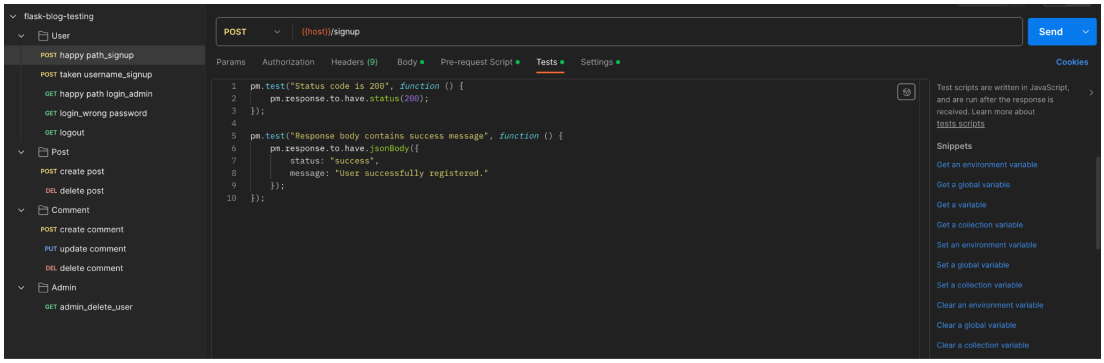

Рисунок 3.12 - Структура опису тестів в Постман

Як бачимо користувач був успішно створений і всі написані тести пройшли без помилок (рис.3.13).

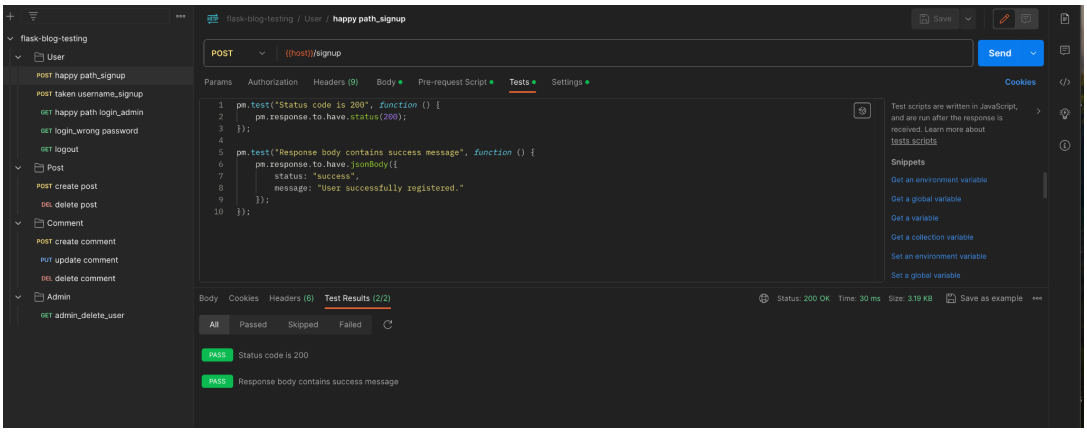

Рисунок 3.13 - Результати проходження тестів реєстрації

Так виглядає результат проходження усього набору тестів для основних компонентів блогу (рис.3.14).

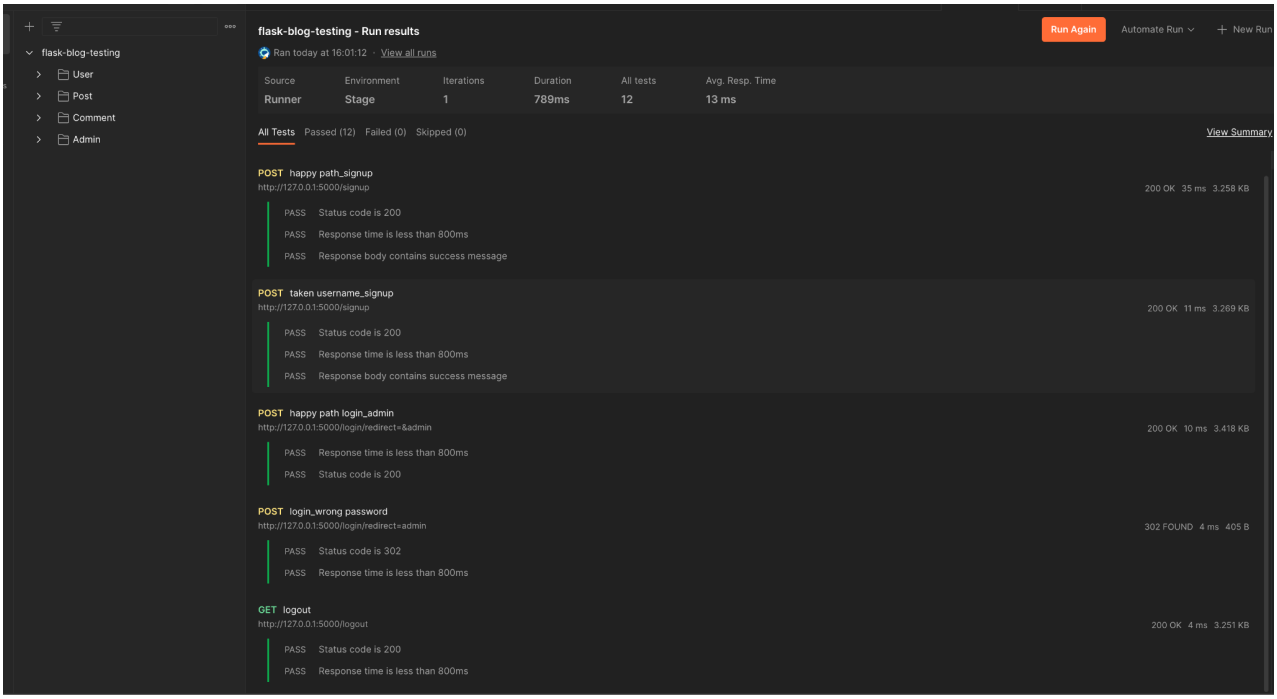

Рисунок 3.14 - Результати тестів по 4 групам шляхів (routes).

3.5 Реалізація модуля End-to-End тестування

End-to-End тестування використовується для оцінки та перевірки взаємодії системи від початку до кінця. Це включає в себе автоматизоване тестування всіх компонентів системи разом, симулюючи реальні сценарії взаємодії з кінцевим користувачем. Основна мета цього тестування - впевнитися, що всі частини системи працюють вірно і ефективно в умовах, які найбільше наближені до реального використання. Тестові файли були створені на основі компонентів.

Кожен компонент має різну функціональність, однак усі тести мають спільну структуру. Наведений нижче фрагмент коду демонструє шаблон тестів, який в подальшому адаптується до відповідного сценарію. В ньому використовується

функція describe(), яка охоплює весь тест, і beforeEach(), яка охоплює функцію, що виконує вхід в систему. (рис 3.15)

```
describe('test section', () => {
  beforeEach(() => {
  cy.visit('base_URL');
})

  it('some test script, for exapmle: user should be allow to login', () => {
    cy.get
  })
```

Рисунок 3.15 - Загальна структура тестів для компонентів

Щоб запустити тест, можна скористатися командою в терміналі npx cypress open. Тести можна виконувати як з допомогою графічного інтерфейсу так і без графічного інтерфейсу користувача, залежно від команд, які використовуються для запуску програми з терміналу. Оскільки графічний інтерфейс полегшує розуміння того, що відбувається зі сторони кінцевого користувача, тести в виконувалися з використанням такого інтерфейсу під час написання цієї роботи.

При відкритті інтерфейсу користувача Cypress представлено список всіх доступних тестів, згрупованих за видом тестового сценарію. Їх можна виконувати окремо чи всі разом у веб-браузері. Браузери, доступні для використання, - Chrome і Electron (браузер на основі Chromium, який працює в режимі хендлес виконання). Тести виконувались з допомогою браузера Chrome [48].

Для створення тестових наборів було обрано найбільш критичний функціонал. Загалом було створено 11 тестових сценаріїв, які входять в смоук набір тестування. Кожен тестовий випадок складається з опису умов, за яких виконується тест, кроків, які виконуються для досягнення кінцевого результату, і очікувань від результату.

Якщо очікування відповідають результату, тест вважається успішним. Якщо результат відрізняється від очікуваного результату, тест вважається неуспішним.

Тестові випадки розроблені таким чином, щоб покрити всі основні функції flaskBlog, при цьому деякі подібні функції були згруповані у відповідні спільні тестові сценарії, щоб зменшити повторюваність коду.

Таблиця 3.3 - Тестові набори та відповідні тестові випадки описані в таблиці

№	**Page/Test Suite**	**Test case**
1	Sign up/Sign page	should be able to Sign up
2	Login/Login page	should be allow to Login
		should not be allow to Login with wrong password
3	Post/Create post	should create post
		should show flash
4	Comment/Create comment	should create post
	Comment/Delete comment	should delete comment
5	Home page	should have list of blog
		should have light/dark toggle
		should have Login/Signup button
6	Search/Search	should be able to search for a specified post

	functionality	
7	Account settings/Change password	should be able to change password
8	Account settings/Change username	should be able to change username
9	Logout/Logout page	should log out successfully
10	Account/Delete account	should be able to delete account
11	Account/Settings page	should have access to settings

Нижче на рисунку можна побачити набір тестів для виконання їх безпосередньо через інтерфейсі клієнта Cypress (рис.3.16).

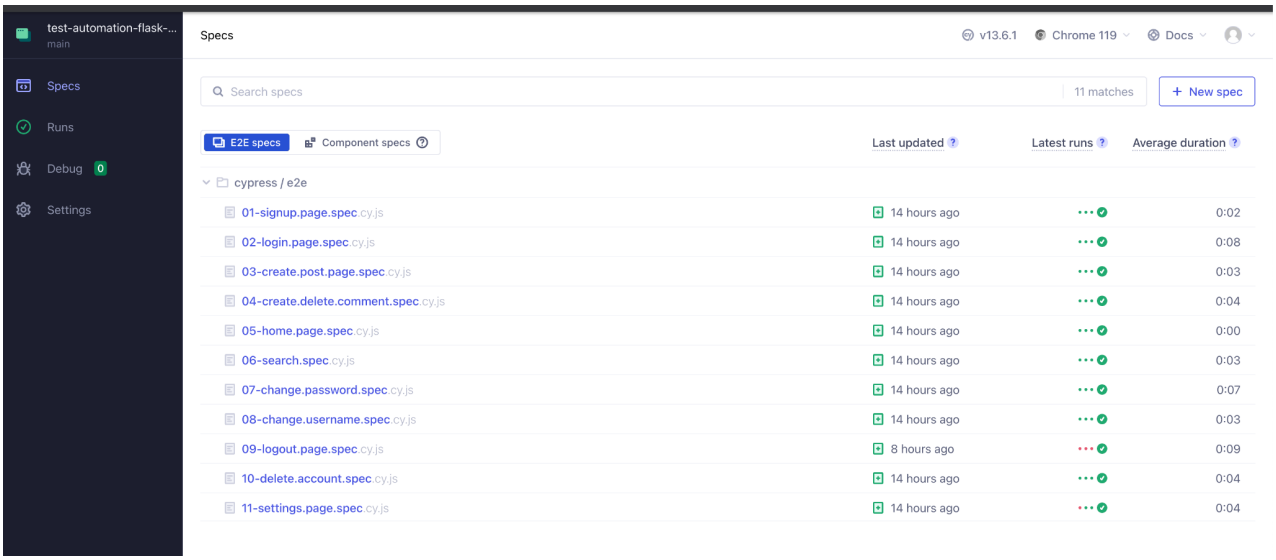

Рисунок 3.16 - Набір тестів для виконання в Cypress

Під час виконання тестів з використанням графічного інтерфейсу його права частина також демонструє інтерфейс веб-застосунку, що тестується. Область

перегляду інтерфейсу може бути визначена в коді для окремих тестів або у конфігураційному файлі.

У лівій частині інтерфейсу відображаються тестові випадки, вибрані для цього тестового запуску. Над списком є загальна кількість успішних і невдалих тестів, час, витрачений на тести, і кнопка для повторного запуску всіх тестів у списку. У випадку виникнення помилки під час виконання тестового сценарію, код тесту для невдалих кроків відобразиться червоним кольором у лівій частині бічної панелі інтерфейсу, яка візуалізує помилковий крок (рис.3.17).

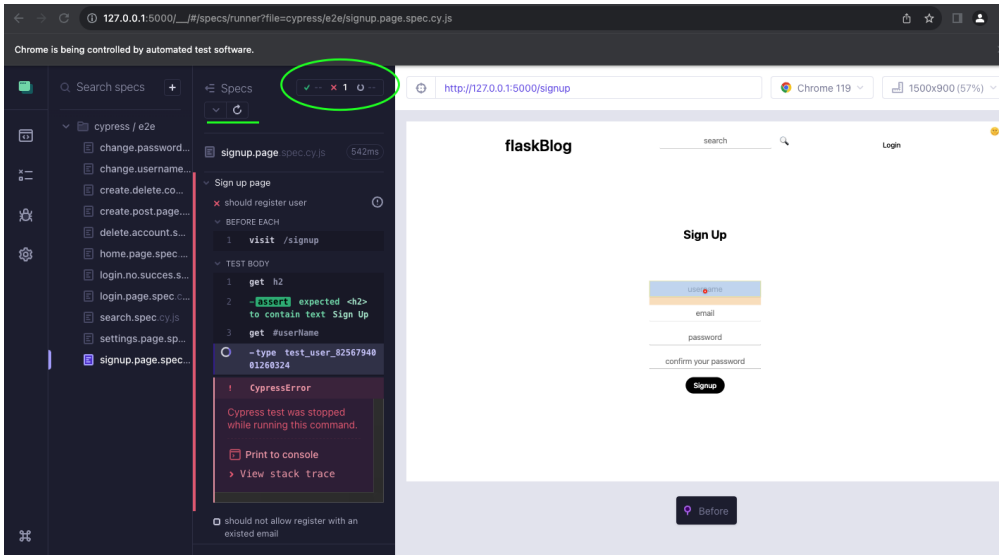

Рисунок 3.17 - Загальний огляд інтерфейсу Cypress та візуалізація помилки під час тестування

Більше детально процес проходження тестових сценаріїв я продемонструє на прикладі тестового сценарію "Sign up page". Цей сценарій є важливою частиною нашого тестового покриття та включає ряд кроків для перевірки функціоналу реєстрації нового користувача.

Перед початком кожного тесту здійснюється попереднє відвідування сторінки реєстрації за допомогою команди cy.visit('/signup'). Далі, для

забезпечення унікальності тестових даних, генерується випадковий номер, який додається до імені користувача та адреси електронної пошти.

У цьому тесті перевіряється успішність реєстрації користувача (рис.3.18).

Розглянемо окремо кожен крок цього тесту:

1. ***beforeEach(() => { cy.visit('/signup');*** }): Цей фрагмент коду встановлює початковий стан перед кожним тестом. В даному випадку, він відвідує сторінку /signup.
2. Генерація випадкових даних:

   const randomNumber = Math.random().toString().slice(2);

   const userName = `test_user_${randomNumber}`

   const email = `${userName}@gmail.com`;

   Тут генеруються випадкові дані для імені користувача та електронної пошти.
3. Перший тест (it('should register user', () => { ... })):

- Перевірка, що заголовок сторінки містить текст 'Sign Up'.
- Введення ім'я користувача, електронної пошти, пароля та його підтвердження.
- Клік на кнопку реєстрації.
- Перевірка, що URL сторінки дорівнює базовому URL, визначеному в конфігурації Cypress. Це говорить про те, що користувач був успішно зареєстрований і переадресований на головну сторінку після успішної реєстрації.

```
const randomNumber = Math.floor(1000 + Math.random() * 9999).toString();
const userName = `test_user_${randomNumber}`;
const email = `${userName}@gmail.com`;
const password = 'password123';

export const signUpTestData = {
  userName,
  password,
};
describe('Sign up page', () => {
  beforeEach(() => {
    cy.visit('/signup');
  });
it('should register user', () => {
cy.get('h2').should('contain.text', 'Sign Up')
cy.get('#userName').type(userName);
cy.get('#email').type(email);
cy.get('#password').type(password);
cy.get('#passwordConfirm').type(password);
cy.get('.form > .btn').click();
cy.url().should('equal', Cypress.config().baseUrl)
});
```

Рисунок 3.18 - Тест з Cypress (registration_spec.js)

Тепер пропоную розглянути, що в цей момент відбувається в інтерфейсі Cypress:

1. Запускається тест, що передбачає відвідування сторінки реєстрації за допомогою команди cy.visit('/signup'). На екрані в лівій частині розташована консоль, де виводиться проходження кожного кроку тесту. У центральній частині екрана відображається візуальна реалізація кожного кроку тесту, де можна спостерігати за взаємодією з елементами сторінки.
2. Після успішного проходження кроків консоль виводить Passed", що свідчить про успішне проходження тестів та виконання всього тесту зайняло - 4 секунди.
3. На екрані також відображається новий URL, що є індикатором того, що користувач потрапив на сторінку блогу.
4. Окрім того, можно швидко перемикатися між кроками в консолі за допомогою курсора миші, що дозволяє робити перевірку кожного етапу (рис. 3.19).

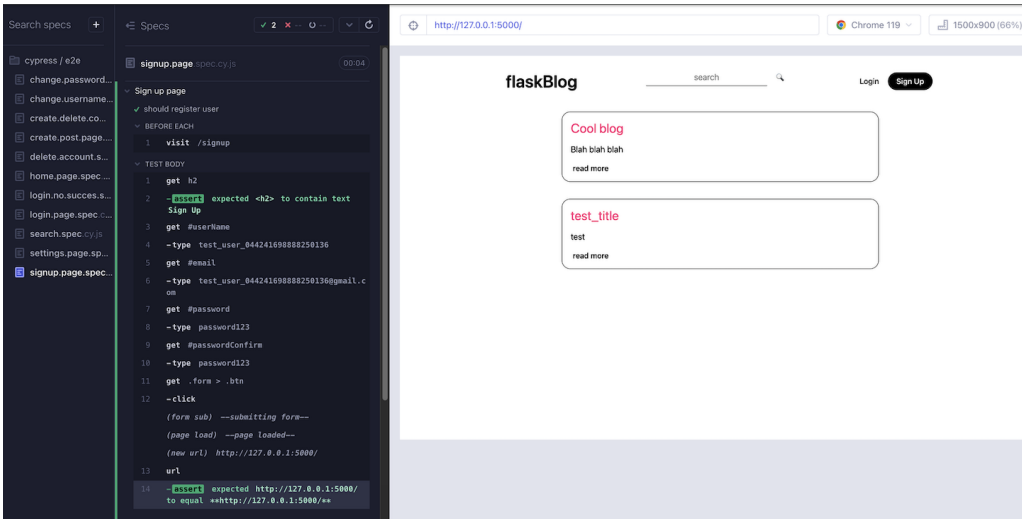

Рисунок 3.19 - Зведення результатів тестів для процесу реєстрації користувача

Процес тестування проходить наступним чином - результат порівнюється зі станом, описаним в утвердженні, і якщо вони співпадають, тести проходять.

Якщо результат порівняння відрізняється, порівняння виконується знову до закінчення попередньо визначеного періоду тайм-ауту (за замовчуванням 5000 мс), і або результат все ж таки співпаде з очікуванням, або тестовий випадок вважається неуспішним. Тестовий випадок може містити кілька очікуваннь за потреби. Тестові випадки виконуються в порядку з'явлення, кожен запускається окремо від інших випадків, так щоб невдачі попередніх тестів не впливали на процес або результати наступних тестових випадків.

Щоб переглянути візуалізацію результатів тестування без необхідності клонування репозиторію та налаштовування середовища, в каталозі є папка з відеофайлами, які автоматично генеруються Cypress під час виконання тестів. Також доступний каталог із знімками екрану та відео записом,які відображаються як проходить процес тестування і в разі помилки ви можете чітко її локалізувати.

Окрім того, під час рану тестового випадку можно переглянути детальну інформацію про викликані методи HTTP та кроки, виконані до, під час та після тестових випадків. Кроки можна окремо вибирати для візуалізації у лівій частині

інтерфейсу. Коли виникає помилка, причина невдачі, відповідна інформація та код тесту, який відображається у правій частині екрану, виокремлюються для поліпшення процесу відлагодження (рис.3.20).

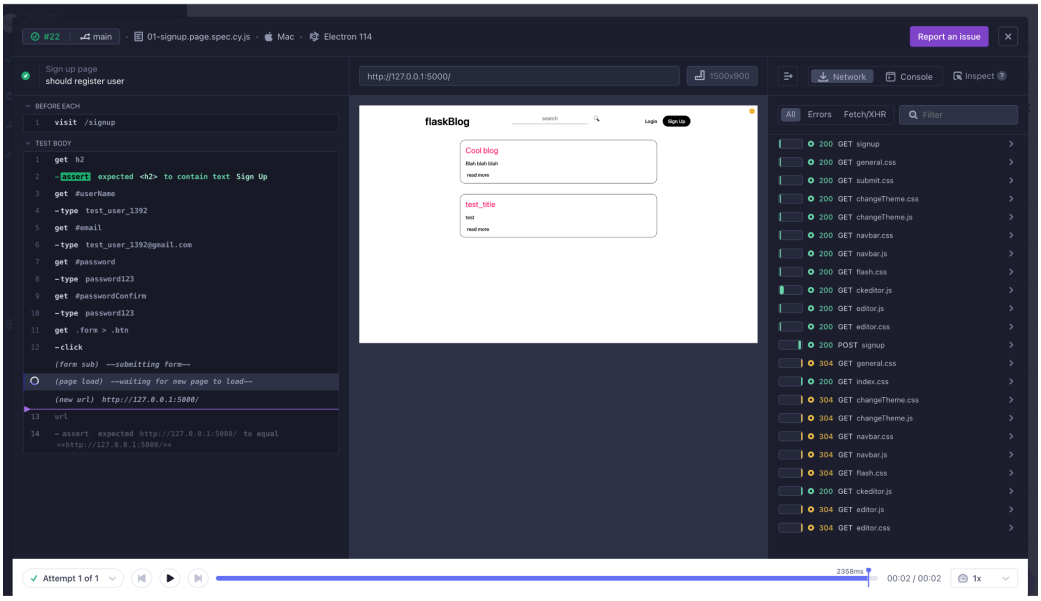

Рисунок 3.20 - Відображення етапів проходження процесу реєстрації

Створені тести слід запускати якнайчастіше. Згідно зі стандартами неперервної інтеграції, тести повинні бути інтегровані в кодову базу для автоматичного виконання кожного разу, коли базовий код оновлюється. В даному випадку використовувався GitHub для управління своєю кодовою базою і CI/CD, і саме там ці тести будуть інтегровані. Візуалізацію автоматичних тестів, що виконуються в GitHub, можна побачити на рисунку 3.21.

(Run Finished)

	Spec		Tests	Passing	Failing	Pending	Skipped
✔	01-signup.page.spec.cy.js	00:02	1	1	-	-	-
✔	02-login.page.spec.cy.js	00:03	3	2	-	1	-
✔	03-create.post.page.spec.cy.js	00:03	2	2	-	-	-
✔	04-create.delete.comment.spec.cy.js	00:04	2	2	-	-	-
✔	05-home.page.spec.cy.js	00:01	3	3	-	-	-
✔	06-search.spec.cy.js	00:03	2	2	-	-	-
✔	07-change.password.spec.cy.js	00:05	2	2	-	-	-
✔	08-change.username.spec.cy.js	00:07	3	3	-	-	-
✔	09-logout.page.spec.cy.js	00:08	4	4	-	-	-
✔	10-delete.account.spec.cy.js	00:09	4	4	-	-	-
✔	11-settings.page.spec.cy.js	00:08	4	4	-	-	-
✔	All specs passed!	00:57	30	29	-	1	-

Рисунок 3.21 - Візуалізація завершеного запуску тестів в GitHub

Результати тестування підтвердили успішне виконання всіх тестових випадків, що є підтвердженням правильності функціоналу веб-додатка. Загальний час витрачений на виконання всього функціоналу, що тестується становить лише 1,34 секунди. Це вражаючий показник, який підкреслює швидкість та ефективність автоматизованих тестів у Cypress. Результати негайно доступні і відображені у графічному інтерфейсі (рис. 3.22).

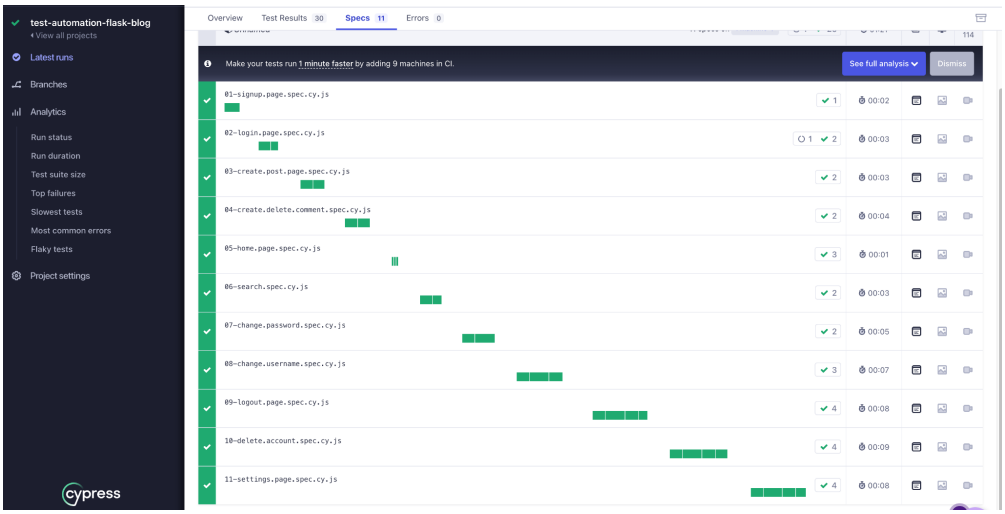

Рисунок 3.22 - Візуалізація завершеного запуску тестів як частини неперервної інтеграції.

ВИСНОВКИ

В ході виконання даної кваліфікаційної роботи було досліджено низку методів та алгоритмів автоматизації тестування веб-додатків, зокрема, функціонального тестування, енд-ту-енд тестування та бекенд тестування.

Проаналізувавши концептуальні підходи до організації автоматизованого тестування та використовуючи їх під час розробки системи для реалізації тестових сценаріїв, можно впевнено стверджувати, що автоматизація тестування відіграє важливу роль в оптимізації роботи тестувальника.

В результаті було проведено глибокий аналіз сучасних методів та інструментів автоматизації тестування веб-додатків, та обрано такі, щоб дозволили провести тестування інформаційної веб-систем - flaskBlog максимально повно та ефективно. У якості мов програмування для автоматизації процесу тестування було обрано Python (так як бекенд системи, що тестується написаний на цій мові) та JavaScript (ця мова використовується для написання тестів фреймворку Cypress, крім того фронтенд системи, що тестується також використовує цю мову).

Порівняльний аналіз існуючих бібліотек та інструментів автоматизації дозволив обрати сучасні та ефективні інструменти, які задовольняють усім потребам інженерів-автоматизаторів.

Окремий акцент приділено проектуванню інформаційної технології, що дозволила протестувати бекенд (за допомогою бібліотеки pytest та тестового клієнта Flask), а також фронтенд частину веб-застосунку і систему в цілому (з використанням Postman та Cypress).

Бекенд тестування дозволило впевнитися в правильній роботі серверної частини та оптимізації взаємодії з базою даних.

Впровадження тестування з використанням Postman дозволило створити механізми швидкої та зручної перевірки коректності роботи API шляхів веб-застосунку (відсутня необхідність “підіймати” проект локально для

проведення тестування, на відміну від бекенд тестування).

Комплесне (End-to-End) тестування, що охоплює як бекенд, так і фронтенд частину, дозволило переконатися, що досліджуємий веб-застосунок - flaskBlog, функціонує коректно не тільки, як набір окремих компонент (фронт, бек), а й як готовий продукт, що має забезпечити максимально позитивний досвід користуванням ним у майбутньому.

Таким чином, вважаю, що проведені під час написання цієї роботи дослідження, а також результати їх застосування - створена мною інформаційна технологія автоматизованого тестування універсальної веб-системи, мають значне практичне значення, адже можуть будуть адаптовані та успішно використані для перевірки коректності роботи різноманітних сучасних веб-застосунків, побудованих з використанням HTTP протоколу та REST архітектури, що одним з найбільш актуальних підходів до проектування веб-систем на момент написання цієї роботи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cost of poor software quality in the US:A 2022 report [Електронний ресурс] – Режим доступу до ресурсу: https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/
2. How Much Software Testing Cost? [Електронний ресурс] – Режим доступу до ресурсу: https://intersog.com/blog/software-testing-percent-of-software-development-costs/
3. Static Estimation of Test Coverage [Електронний ресурс] – Режим доступу до ресурсу: https://www.researchgate.net/publication/220703742_Static_Estimation_of_Test_Coverage
4. Differences between Verification and Validation [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/differences-between-verification-and-validation/
5. ISO/IEC 25010 [Електронний ресурс] – Режим доступу до ресурсу: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010
6. FOUNDATIONS OF SOFTWARE TESTING [Електронний ресурс] – Режим доступу до ресурсу: https://www.utcluj.ro/media/page_document/78/Foundations%20of%20software%20testing%20-%20ISTQB%20Certification.pdf
7. Types of Software Testing [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/types-software-testing/
8. Different Testing Types with Details [Електронний ресурс] – Режим доступу до ресурсу: https://www.softwaretestinghelp.com/types-of-software-testing/
9. Types of Testing: Different Types of Software Testing in Detail [Електронний ресурс] – Режим доступу до ресурсу: https://www.browserstack.com/guide/types-of-testing
10. Types of Software Testing (100 Examples) [Електронний ресурс] –

Режим доступу до ресурсу: https://www.guru99.com/types-of-software-testing.html/

11. What is Visual Testing? [Електронний ресурс] – Режим доступу до ресурсу: https://applitools.com/blog/visual-testing/

12. Automated Testing for React Web Application with Cypress [Електронний ресурс] – Режим доступу до ресурсу: https://www.theseus.fi/bitstream/handle/10024/785012/Al-Ajily_Mohamed.pdf?sequence=2

13. ISTQB Glossary of Testing Terms 2.3x [Електронний ресурс] – Режим доступу до ресурсу: https://bystqb.org/files/content/bystqb/downloads/ISTQB_Glossary_English_v2.3.pdf

14. How Much Automated Testing Should Be Done in Software Development? [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@m-zimmermann1/how-much-automated-testing-should-be-done-in-software-development-c445445320aa

15. What Is Automation Testing Pyramid? [Електронний ресурс] – Режим доступу до ресурсу: https://qatestlab.com/resources/knowledge-center/test-automated-pyramid/

16. Test for value, not vanity. The testing “pyramid” is dead. [Електронний ресурс] – Режим доступу до ресурсу: https://ed-burton.medium.com/test-for-value-not-vanity-the-testing-pyramid-is-dead-5abd75c0498a

17. Just Say No to More End-to-End Tests [Електронний ресурс] – Режим доступу до ресурсу: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html

18. Unit Test Frameworks: Tools for High-Quality Software Development [Електронний ресурс] – Режим доступу до ресурсу: https://www.researchgate.net/publication/274074619_Unit_Test_Frameworks_Tools_for_High-Quality_Software_Development

19. Python testing in Visual Studio Code [Електронний ресурс] – Режим доступу до ресурсу: https://code.visualstudio.com/docs/python/testing

20. PyTest with Django REST Framework: From Zero to Hero [Електронний ресурс] – Режим доступу до ресурсу: https://dev.to/sherlockcodes/pytest-with-django-rest-framework-from-zero-to-hero-8c4

21. What Is API Testing? | Definition from TechTarget [Електронний ресурс] – Режим доступу до ресурсу: https://www.techtarget.com/searchapparchitecture/definition/API-testing

22. HTTP request methods - MDN Web Docs [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

23. What is End To End Testing? [Електронний ресурс] – Режим доступу до ресурсу: https://www.browserstack.com/guide/end-to-end-testing

24. What is Agile Project Management (APM)? [Електронний ресурс] – Режим доступу до ресурсу: https://www.techtarget.com/searchcio/definition/Agile-project-management

25. Agile Methodology: What is Agile Model in Software Testing? [Електронний ресурс] – Режим доступу до ресурсу: https://www.guru99.com/agile-scrum-extreme-testing.html

26. What is Cypress Testing? What It Is and How to Get Started [Електронний ресурс] – Режим доступу до ресурсу: https://www.perfecto.io/blog/cypress-testing

27. How Cypress Works | End to end and component testing [Електронний ресурс] – Режим доступу до ресурсу: https://www.cypress.io/how-it-works

28. Let's Dive Into Cypress For End-to-End Testing [Електронний ресурс] – Режим доступу до ресурсу: https://www.smashingmagazine.com/2021/09/cypress-end-to-end-testing/

29. Welcome to Flask [Електронний ресурс] – Режим доступу до ресурсу:

https://flask.palletsprojects.com/

30. API testing with Postman [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@krizten/api-testing-with-postman-5fc39f9e5dd7

31. Testing API With Postman In 2023 [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@dees3g/testing-api-with-postman-in-2023-18fcd021a9d2

32. Postman Tutorial – How to use for API Testing? [Електронний ресурс] – Режим доступу до ресурсу: https://www.guru99.com/postman-tutorial.html

33. How to automate API tests with Postman [Електронний ресурс] – Режим доступу до ресурсу: https://blog.logrocket.com/how-automate-api-tests-postman/

34. PyTest: Python Testing Framework for Backend Engineers [Електронний ресурс] – Режим доступу до ресурсу: https://www.digitalocean.com/community/tutorials/pytest-python-testing-framework-for-backend-engineers

35. Selecting the Scripting Language [Електронний ресурс] – Режим доступу до ресурсу: https://support.smartbear.com/testcomplete/docs/scripting/selecting-the-scripting-language.html

36. Python Test Automation: Seven Options for More Efficient Tests [Електронний ресурс] – Режим доступу до ресурсу: https://www.testim.io/blog/python-test-automation/

37. JavaScript Unit Testing Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://www.browserstack.com/guide/unit-testing-in-javascript#:~:text=JavaScript%20Unit%20Testing%20is%20a,organized%20in%20the%20test%20suite

38. Flask [Електронний ресурс] – Режим доступу до ресурсу: https://flask.palletsprojects.com/en/3.0.x/

39. Python [Електронний ресурс] – Режим доступу до ресурсу:

https://www.python.org/

40. WTForms [Електронний ресурс] – Режим доступу до ресурсу: https://wtforms.readthedocs.io/en/3.1.x/

41. What Is SQLite [Електронний ресурс] – Режим доступу до ресурсу: https://www.sqlite.org/index.html

42. Coverage.py [Електронний ресурс] – Режим доступу до ресурсу: https://coverage.readthedocs.io/en/7.3.2/#coverage-py

43. Welcome to pytest-cov's documentation! [Електронний ресурс] – Режим доступу до ресурсу: https://pytest-cov.readthedocs.io/en/latest/

44. Pytest-mock 3.12.0 [Електронний ресурс] – Режим доступу до ресурсу: https://pypi.org/project/pytest-mock/

45. Unitest [Електронний ресурс] – Режим доступу до ресурсу: https://docs.python.org/3/library/unittest.html

46. Postman documentation overview [Електронний ресурс] – Режим доступу до ресурсу: https://learning.postman.com/docs/introduction/overview/

47. Cypress documentation: [Електронний ресурс] – Режим доступу до ресурсу:https://docs.cypress.io/guides/guides/command-line

ДОДАТОК А. КОД РЕАЛІЗАЦІЇ

End-2-End тестування

signup.page.spec.cy.js

```
const randomNumber = Math.floor(1000 + Math.random() * 9999).toString();
const userName = `test_user_${randomNumber}`;
const email = `${userName}@gmail.com`;
const password = 'password123';

export const signUpTestData = {
  userName,
  password,
};
describe('Sign up page', () => {
  beforeEach(() => {
    cy.visit('/signup');
  });
it('should register user', () => {
cy.get('h2').should('contain.text', 'Sign Up')
cy.get('#userName').type(userName);
cy.get('#email').type(email);
cy.get('#password').type(password);
cy.get('#passwordConfirm').type(password);
cy.get('.form > .btn').click();
cy.url().should('equal', Cypress.config().baseUrl)
});
```

login.page.spec.cy.js

```
import { signUpTestData } from './01-signup.page.spec.cy';
```

```
describe('Login page', () => {
  beforeEach(() => {
  cy.visit('/login/redirect=&');
})

  it('should be allow to login', () => {
    const { userName, password } = signUpTestData;
    cy.get('h2').should('contain.text', 'Login')
    cy.get('#userName').type(userName);
    cy.get('#password').type(password);
    cy.get('.form > .btn').click()
    cy.get('#flash').should('contain.text', 'Welcome ' + userName)
    });
  it.skip('should not be allow to login', () => {
    cy.get('h2').should('contain.text', 'Login')
    cy.get('#userName').type(userName + 'wrong_user');
    cy.get('#password').type(password);
    cy.get('.form > .btn').click()

    cy.get('#flash').should('contain.text', 'user not found')

  });
})
```

create.post.page.spec.cy.js

```
import { signUpTestData } from './01-signup.page.spec.cy';
describe('Create Post', () => {
  beforeEach(() => {
```

```
  cy.visit('/login/redirect=&');
})
  it('should create post', () => {
    const { userName, password } = signUpTestData;
    cy.get('#userName').type(userName);
    cy.get('#password').type(password);
    cy.get('.form > .btn').click()
    cy.get('[href="/"] > .btn').should('contain.text', 'flaskBlog');
    cy.get('[href="/createpost"] > .btn').should('contain.text', 'New Post').click()
    cy.get('[placeholder="post title"]').as('post title');
    cy.get('#postTitle').type('Good morning');
    cy.get('[placeholder="tags"]').as('post title');
    cy.get('#postTags').type('#test');
    cy.get('.ck-editor__editable').then(($editable) => {
      const editor = $editable[0].ckeditorInstance;
      editor.setData('Hello, this is a test test text.');
    });
    it('should show flash', () => {
    cy.get('.form > .btn').click();
    cy.get('#flash').should('contain.text', 'You earned 20 points by posting');
```

create.delete.comment.spec.cy.js

```
import { signUpTestData } from './01-signup.page.spec.cy';

describe('Create Post', () => {
  beforeEach(() => {
  cy.visit('/login/redirect=&');
})
```

```
  it('should create comment', () => {
    const { userName, password } = signUpTestData;
    cy.get('#userName').type(userName);
    cy.get('#password').type(password);
    cy.get('.form > .btn').click()
    cy.get('[href="/"] > .btn').should('contain.text', 'flaskBlog');
    cy.get('.post').should('have.length', 2);
    cy.get(':nth-child(7) > .title').click()
    cy.get('#comment').type('Have a nice day!')
    cy.get('.btnSubmit').click();
    cy.get('[href="/deletecomment/5/redirect=post&1"]').click()
  })
});
```

home.page.spec.cy.js

```
describe('My Home page', () => {
  beforeEach(() => {
    cy.visit('/');
  })
  it('should have main parts', () => {
    cy.get('h1')
    .should('contain.text', 'flaskBlog');
    cy.contains('a', 'Cool blog')
    .should('exist');
    cy.contains('p', 'Blah blah blah')
    .should('exist');
    cy.contains('a', 'read more')
    .should('exist');
```

```
    cy.contains('button', '🌞')
    .should('exist');
  })
    it('should click on Login', () => {
      cy.contains('a', 'Login')
      .should('exist').click();
      cy.url().should('include', '/login')
      cy.get('h1').should('contain.text', 'flaskBlog')
      cy.contains('button', '🌞').click();
    });
  it('should click on Sign Up', () => {
    cy.contains('a', 'Sign Up')
    .should('exist').click()
    cy.url().should('include', '/signup')
    cy.get('h1').should('contain.text', 'flaskBlog')
  });
})
```

search.spec.cy.js

```
import { signUpTestData } from './01-signup.page.spec.cy';

describe('Forgor password page', () => {
  beforeEach(() => {
  cy.visit('/login/redirect=&');
})
  it('should be able to search specified post', () => {
    const { userName, password } = signUpTestData;
    cy.get('h2').should('contain.text', 'Login')
    cy.get('#userName').type(userName);
```

```
    cy.get('#password').type(password);
    cy.get('.form > .btn').click()
    cy.get('#searchInput')
      .should('have.attr', 'placeholder')
      .should('eq', 'search')
      cy.get('#searchInput').type('Cool')
      cy.get('.searchBar > .btn').click()


    });
  })
```

change.password.spec.cy.js

```
import { signUpTestData } from './01-signup.page.spec.cy';
const { userName, password } = signUpTestData;
const newPassword = password + 'updated';
describe('Login page', () => {
  beforeEach(() => {
  cy.visit('/login/redirect=&');
})
  it('should be able to change password', () => {
    cy.get('h2').should('contain.text', 'Login')
    cy.get('#userName').type(userName);
    cy.get('#password').type(password);
    cy.get('.form > .btn').click()
    cy.get('#flash').should('contain.text', 'Welcome ' + userName)
    cy.get('.btnPrimary').click()
    cy.get('[href="/accountsettings"]').click()
    cy.get(':nth-child(2) > .toPanel').click()
    cy.get('#oldPassword').type(password)
```

```
    cy.get('#password').type(newPassword)
    cy.get('#passwordConfirm').type(newPassword)
    cy.get('.form > .btn').click()
    cy.get('#flash').should('contain.text', 'you need login with new password')


  })
```

change.username.spec.cy.js

```
describe('Login page', () => {
  beforeEach(() => {
  cy.visit('/login/redirect=&');
})
  it('user should be able to changed username', () => {
    cy.get('h2').should('contain.text', 'Login')
    cy.get('#userName').type(userName);
    cy.get('#password').type(newPassword);
    cy.get('.form > .btn').click()
    cy.get('#flash').should('contain.text', 'Welcome ' + userName)
    cy.get('.btnPrimary').click()
    cy.get('[href="/accountsettings"]').click()
    cy.get(':nth-child(1) > .toPanel').click()
    cy.get('#newUserName').type(newUserName)
    cy.get('.form > .btn').click()
    cy.get('#flash').should('contain.text', 'user name changed')
  })
  })
```

logout.page.spec.cy.js

```
describe('Log out', () => {
```

```
  beforeEach(() => {
  cy.visit('/login/redirect=&');
})
  it('should log out successfully', () => {
    const { newUserName, newPassword } = signUpTestData2;

    cy.get('h2').should('contain.text', 'Login')
    cy.get('#userName').type(newUserName);
    cy.get('#password').type(newPassword);
    cy.get('.form > .btn').click()
    cy.get('.btns > [href="/logout"]').click();
    cy.get('[href="/login/redirect=&"] > .btn').should('contain.text', 'Login');
    cy.get('[href="/signup"] > .btn').should('contain.text', 'Sign Up');
    });
  })
```

delete.page.spec.cy.js

```
import { signUpTestData2 } from './08-change.username.spec.cy.js';

describe('Login page', () => {
  beforeEach(() => {
  cy.visit('/login/redirect=&');
})

  it('should be able to delete account', () => {
    const { newUserName, newPassword } = signUpTestData2;

    cy.get('h2').should('contain.text', 'Login')
    cy.get('#userName').type(newUserName);
```

```
    cy.get('#password').type(newPassword);
    cy.get('.form > .btn').click()
    cy.get('.btnPrimary').click()
    cy.get('[href="/accountsettings"]').click()
    cy.get(':nth-child(3) > .toPanel').click()
    cy.get('.btn > .textPrimary').click()
    });
  })
```

settings.page.spec.cy.js

```
import { signUpTestData2 } from './08-change.username.spec.cy.js';
describe('Login page', () => {
  beforeEach(() => {
  cy.visit('/login/redirect=&');
})
  it('should have access to settings', () => {
    const { newUserName, newPassword } = signUpTestData2;
    cy.get('h2').should('contain.text', 'Login')
    cy.get('#userName').type(newUserName);
    cy.get('#password').type(newPassword);
    cy.get('.form > .btn').click()
    cy.visit(`/user/${newUserName}`);
    cy.get('[href="/accountsettings"]').click()
    cy.get(':nth-child(2) > .toPanel').click()
  });
  })
```