Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота

за освітнім ступенем «бакалавр»

на тему:

«Автоматизоване тестування вебдодатків із використанням хмарних технологій та контейнеризації»

Виконав:
здобувач IV курсу
групи ІПЗ-41
спеціальності 121 «Інженерія
програмного забезпечення»
Яроцький Михайло Віталійович

Науковий керівник:
к.т.н., доцент Сяський В.А.

Рівне – 2026

ЗМІСТ

ВСТУП

Питання **актуальності дослідження** починається з місця, де тестування перетворюється на невід'ємну частину процесу створення програмного забезпечення. Не менш важливою за проєктування чи розробку воно стало завдяки постійним появах нових функцій у будь-якому програмному продукті. Сьогодні навіть, з першого погляду, найпростіший вебзастосунок може мати внутрішню логіку складніше, ніж умовний банківський інтерфейс кілька років тому.

Таким чином, масштаб не завжди є основним - головне, що система повинна працювати без збоїв. Оскільки оновлення випускаються часто, а старі підходи до перевірок просто не встигають за темпом - нова реальність вимагає швидших реакцій, при цьому надійність тестів не може втрачатися. Коли помилка потрапляє у фінальне середовище, наслідки можуть бути серйозними - особливо для критичних систем. Отже, пошук способів забезпечити оперативність разом із точністю стає ключовим напрямком у сучасному тестуванні.

Через великі витрати часу й зусиль ручне тестування інколи просто не спрацьовує. Замість нього поволі, але наполегливо пробивається автоматизація.

Сьогодні важко уявити перевірку вебзастосунків без автотестів користувацього, які імітують дії реальної людини. Крім цього, паралельно з автоматизацією, набирають обертів рішення на основі контейнерів та **процеси безперервної інтеграції й доставки (CI/CD)**. Через такий підхід тести працюють у чистому оточенні, не залежно від стану машини. Натомість кожна правка в коді може запустити перевірки самостійно, що робить контроль якості природньою частиною розробки.

Метою дослідження, у цій роботі, можна виділити розробку та дослідження системи автоматизованого UI (user interface) тестування вебзастосунку. При цьому в роботі буде використовуватись набір популярних інструментів, кожен з яких виконуватиме свою особисту функцію, як ось запуск тестів, керування

браузерами, пакування програм у контейнери, тощо. Кожен компонент має свою роль.

Завданням дослідження можна виділити наступні пункти:

- проаналізувати основні підходи до автоматизованого тестування вебдодатків;
- дослідити особливості UI і E2E тестування;
- розглянути принципи роботи контейнеризації та CI/CD;
- створення тестового фреймворку з Python, Pytest та Playwright;
- реалізувати контейнеризоване середовище на базі Docker;
- виконати автоматичний запуск тестів через GitHub Actions;
- провести порівняння часу виконання тестів у різних середовищах.

Об’єктом дослідження можна назвати сам процес сучасного тестування вебзастосунків за допомогою автоматизованих систем.

Предметом дослідження є принципи, підходи та методи розробки автоматизованої системи тестування, а також порівняння роботи різних середовищ для запуску автотестів, з виділенням слабких та сильних сторін для кожного з них.

Серед **методів дослідження** можна виділити наступні:

- теоретичний: аналіз та систематизація інформації з наукової та технічної літератур;
- емпіричний: адаптація існуючих методів автоматизованого тестування вебдодатків для виявлення сильних та слабких сторін кожного з наведених способів.

Практичне значення дослідження - у розробці системи для автоперевірки інтерфейсів сайтів. Ця основа стане прикладом реальних методів налаштування перевірок через контейнери й безперервну поставку коду. Буде показано, як працюють тести на локальному комп’ютері, у середовищах типу Docker чи в хмарі. Варто наголосити, що результати цього дослідження дозволять продемонструвати практичне використання Docker і GitHub Actions у процесі забезпечення якості програмного забезпечення.

Апробація та впровадження результатів дослідження.

Основні теоретичні та практичні результати дослідження були представлені на І Міжнародній науково-практичній конференції «Сучасні інформаційні технології: від теорії до практики», (м. Луцьк, 22–23 травня 2026 р.).

Структура кваліфікаційної роботи скдадається з вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. Обсяг основної частини курсової роботи становить 43 сторінки.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ВЕБДОДАТКІВ

1.1. Основні поняття процесів тестування

Коли пишуть програми, треба переконатися, чи все працює так, як було задумано. Кожен сайт чи сервіс може зламатись через дрібну помилку всередині. Через це необхідно перевіряти кожне нововведення вчасно та регулярно.

Перевіряти програмне забезпечення – значить шукати, де воно ламається, може зламатись потенційно, та чи робить воно те, для чого його створили. Сьогодні майже жоден проект не обходиться без такого контролю, бо воно допомагає уникнути хаосу після запуску та підтримки. Хоча часто гадають, ніби головне це знайти якомога більше помилок, справжня мета принципово інша. Цю мету можна сформулювати наступним чином: переконатися, що система робить саме те, що має робити згідно з планом.

Є дві головні стратегії: одна базується на живому контролі, інша – на скриптах та командах. Особа може просто сидіти перед екраном, клікати кнопки й спостерігати за реакціями сайту. Уявіть: хтось вводить логін, шукає продукт, клацає «купити». Цей шлях добре працює там, де треба зробити пару перевірок без зайвого нагромадження техніки. Але коли проект розростається чи починає міняти функції кожен тиждень, все стає важче. Тут потрібно багато рук, багато очей, а помилки можуть з'явитись навіть у найпростіших операціях.

Замість людей у тестах інколи краще працюють машини, їх налаштовують раз, далі вони самі все роблять. Програма може запуститися навіть коли ніхто не дивиться, і все одно виконає перевірку точно. Швидкість стає набагато вищою, якщо потрібно повторювати одне й те ж тисячі разів. Хоча й не всі продукти такі складні, де треба автоматика для кожного кроку, проте там, де оновлення виходять регулярно, без автотестів практично неможливо обійтись - онлайн-сервіси тримаються саме на таких інструментах.

Після того як хтось змінює код, важливо перевірити чи те, що працювало раніше, досі на місці. Замість цього просто запускають старі тести, аби подивитися, чи нічого не зламалося. Без таких, здавалося б елементарних перевірок, можна швидко потрапити у пастку помилок. Іноді все виглядає добре, допоки не спрацює саме той кут, який забули протестувати. Ось чому кожне оновлення йде слідом за перевірками, які були ще з минулого разу. Навіть маленька правка може потягнути за собою проблеми там, де ніхто не очікував. Тому система тестів постійно повертається до головного: що вже працювало - те має працювати й надалі.

Серед способів автоматичного тестування особливо часто зустрічаються перевірки через інтерфейс. Замість людини все виконує програма - клікає, пише, листає сторінки. Це нагадує те, як справжній користувач працює у браузері кожен день. Наприклад, хтось може заповнювати форми або переходити з одного екрана на інший. Деякі сценарії повторюють покупку товару чи реєстрацію на сайті. Усе це допомагає побачити, чи працює вебзастосунок без помилок.

Часто UI-перевірки йдуть поряд з тестванням усього процесу.

Перевірка «від початку до кінця», що називають **E2E (End-to-End) тестуванням**, охоплює цілковитий шлях роботи системи. Уявімо: людина заходить у свій акаунт, обирає продукт, кладе його в кошик - далі слідує оплата. Коли кожен етап пройшов без перешкод, стає ясно: частини програми спілкуються так, як треба і тест можна вважати успішно пройденим.

Особливість автоматичних перевірок - самостійність. Кожна перевірка має працювати без підстраховки інших. Помилка в одній не повинна тягнути збої решти, особливо коли перша - це вхід до системи, а всі наступні спираються на цей вхід. Щоб такого не сталося, перед кожним тестом запускають нову сесію браузера

Одне з ключових правил – це надійність тестів. Кожен автоматичний запуск має закінчуватися тим самим результатом, якщо навколишні умови не мінялися. Іноді перевірка працює правильно, потім раптово падає - хоча код залишився

колишнім, такий тест називають *flaky test*. Такі моменти плутають розробників, руйнуючи впевненість у точності перевірок та справності продукту.

У розробці також важливу роль відіграє відтворюваність середовища виконання тестів. Один і той самий тест повинен однаково працювати на різних комп'юторах та в різних середовищах. Саме для цього використовуються технології контейнеризації та CI/CD, які дозволяють запускати тести у стандартизованому середовищі, типу *Docker* чи *GitHub Actions*, які будуть описані далі (рис. 1.1).

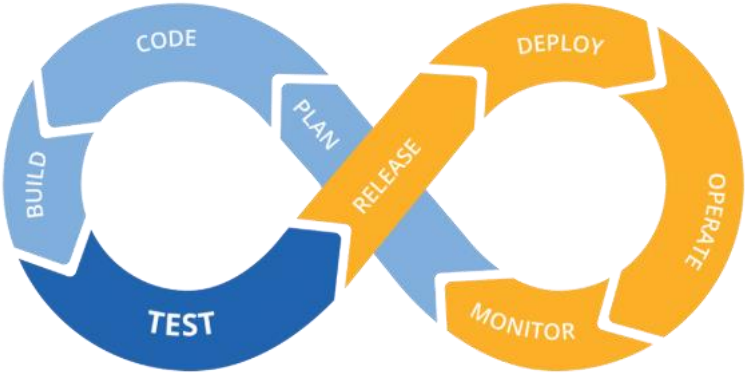


Рисунок 1.1. Умовна візуалізація процесу CI/CD

Підсумовуючи, автоматизоване тестування вже є невід'ємною частиною процесу розробки програмного забезпечення. Воно дозволяє швидше знаходити помилки, підвищує стабільність системи та спрощує підтримку програм після оновлень. Особливо важливим автоматизоване тестування є для сучасних вебсистем, де зміни у функціоналі відбуваються регулярно, а швидкість перевірки напряму впливає на процес розробки чи оновлення версій.

1.2. Інструменти автоматизації тестування

Створити автоматичну перевірку можливо десятками різних засобів. Без урахування людської практики - технічний вибір грунтується на специфіці проекту, архітектурі, мові коду й очікуваннях від тестування. Сьогодні, коли мова заходить про веб, найчастіше обирають те, що прискорює написання UI-

перевірок, дає запускати тести одночасно й спрямовано входить у процеси доставки коду.

Автоматизація не обходиться без тестового фреймворку - ось що точно має бути. Він забезпечує запуск перевірок, аналізує, що вийшло, збирає підсумки, дбає про порядок у коді. У цьому проекті було обрано *Pytest* фреймворк. Зараз це частий вибір серед інструментів для перевірок у *Python*. Однією з головних причин є відносна простота використання цього інструменту. Сценарій складається наступним чином: пишеться функція з контрольними точками, далі відбувається запуск - і система сама перевіряє описані в коді модулі.

Другою важливою рисою Pytest стає його здатність легко запускати тести у різних середовищах. Одночасно варто зауважити, що підтримка плагінів дозволяє значно розширити можливості тестування без зайвих ускладнень.

Коли потрібно перевірити щось у системі - часто доводиться робити одні й ті ж кроки. Один із способів це автоматизувати - використовувати те, що називають *fixture*. Уявімо: кожного разу треба зайти під обліковим записом. Цей процес можна виділити окремо. Замість того щоб повторювати логінування в кожному тесті - просто підключаєш готовий блок. Те саме з відкриттям сторінки в браузері. Так код стає чистішим. Повторень майже не залишається. Структура схожа на шаблон, але без надбудов. Тестовий фреймворк сам подбає про черговість, а результататом - менше помилок через людський фактор

Також вважаю, що надзвичайно важливо буде виділити ібліотеку **Playwright** у складі Pytest.

Ця бібліотека створена для перевірки інтерфейсів через код. Працює не просто в одному браузері, а трьох основних системах, на яких найкраз-таки будуються сучасні браузери: Chromium, Firefox, WebKit. Тест можна запустити без самого вікна браузера, приховано. Такий режим називають *headless*. Виглядає це як процес у терміналі, без анімації чи стрибаючих вікон. Результат той самий, але швидше набагато. Якщо порівняти з *Selenium* та методом запуску тестів через *webDriver* - різниця буде очевидною. Не треба чекати завантаження всього інтерфейсу кожного разу. Кожна секунда має значення, коли проганяєш сотні

тестів, тож відносно цього, автоматизація стає набагато швидшою та ефективнішою.

Ще один спосіб - це працювати з вебсторінками завдяки Playwright. Відкриття потрібної адреси, заповнення форм, натискання на кнопки, пошук елементів, тощо. Усе, що робить людина в браузері, тепер може робити код. Крім того, серед усіх таких інструментів - цей виділяється надійністю. Особливо в порівнянні з іншими, старішими інструментами, коли тести могли ламатись через те, що сам фреймворк давав збій через внутрішні помилки.

Замість того щоб чекати проходження кожного тесту по черзі - запускають тести паралельно, через *pytest-xdist*. Ця фукнція поділяє перевірки між декількома процесами. Через це весь прогін займає менше часу. Завдяки такому підходу комп'ютер працює повніше, а система перевіряється швидше.

Запуск тестів у різних місцях - справа складна через розбіжність конфігурацій, підтримки елементів використованою системою та внутрішніх налаштувань. Однак *Docker* може полегшувати цей процес значно. Усередині контейнера все потрібне вже налаштоване: бібліотеки, інструменти, конфігурація. Коли середовище однакове всюди, помилки, які виникають через локальні особливості, просто зникають. Кожна машина отримує ту саму основу - незалежно від того, хто її вмикає. Тестування стає передбачуванішим та стабільнішим, а не додатковою проблемою.

Коли хтось щось міняє в сховищі, тести самі запускаються - завдяки *GitHub Actions*. Сервіс цей знаходиться прямо всередині GitHub, не треба нічого додавати окремо. Як тільки код потрапляє через *push* чи *pull* request, процес починається без участі людини. Кожна нова правка в коді тепер проходить таку саму перевірку автоматично. Перевага в тому, що помилки можна побачити раніше, ніж код потрапить кудись далі.

1.3. Контейнеризація та CI/CD у тестування

Як було визначено раніше, сьогодні перевірку коду рідко обмежують однією машиною. Саме тому потрібна надійність - будь-яка система має працювати однаково скрізь. Тому з’являються спеціальні середовища в малих пакетах. Вони допомагають долучитися до безперервних процесів збірки й контролю якості.

Запуск тестового середовища в ізоляції стає можливим завдяки контейнерам. Кожна зміна в коді потрапляє на перевірку миттєво. Коли елементи коректно працюють один з одним, процес контролю за якістю стає легшим для команди розробки. Результат - менше помилок, більше чіткості.

Окреме *середовище*, у яке запаковують програму разом з тим, що їй потрібно, - ось що таке **контейнер**. Цю справу найчастіше роблять за допомогою **Docker**.

Можна уявити, що все оточення для перевірок прописане в одному файлі - *Dockerfile*. Потім цей запис стає основою для запуску скрізь, де потрібно. Налаштовувати знову нічого не доведеться, а робота продовжується там, де закінчилася раніше.

Ось що допомагає уникнути класичної проблеми з тестами: пройшли на одному комп’ютері, а на іншому - ні. Часто причина десь у версіях бібліотек, залежностях чи самій ОС (операційній системі). Різниця між середовищами рано чи пізно може дати про себе знати, а чим більше та комплексніше стає покриття програми тестами - тим більший шанс на невдачу. Цей спосіб просто забирає таку невизначеність.

У контексті автоматизованого тестування контейнеризація дає такі переваги:

- ізольоване середовище виконання;
- запуск кожного тесту відбувається за однаковими правилами;
- спрощене відтворення помилок;
- можливість запуску без встановлення додаткового ПЗ на комп’ютері;
- зручність інтеграції з CI/CD.

Тут беруть образ із уже вбудованим *Playwright* - на його базі пускають контейнер. Так не треба витрачати багато ресурсів із довгим налаштовуванням браузерів чи ще якихось модулів.

У *Dockerfile* записано кроки, що ведуть до налаштування оточення, а саме:

- копіювання проєкту;
- встановлення залежностей;
- запуск тестів.

Отже, контейнер - це наче закрита пробірка з усім потрібним всередині, що дозволяє без зайвих турбот запустити все раз за разом за лічені хвилини.

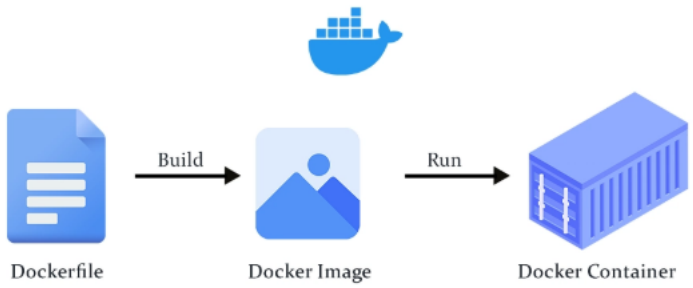


Рисунок 1.2. Візуалізація процесу контейнеризації

Уявімо собі, як образ перетворюється на те, що працює всередині. З нього роблять копію, яка вже може функціонує самостійно. Ця версія потрапляє туди, де перевіряють, чи все діє правильно.

Слід пам’ятати: тестування не прискориться саме через контейнеризацію. Насправді, ізольований шар часто трохи гальмує процес. Але коли мова заходить про стабільність проходження системи тестами - перевага очевидна.

CI CD і тестування.

Кожна зміна коду одразу проходить перевірку завдяки **CI/CD**. Цей метод розробки дозволяє швидко ловити помилки при завантаженні змін у коді у репозиторій. Автоматизація тут працює без участі людини, а в результаті процес стає стабільнішим із кожною операцією.

Коли мова заходить про перевірку коду, автоматизація дозволяє запускати тести самостійно - скажімо, одразу після того, як хтось щось змінив у сховищі.

Основними цілями CI/CD у тестуванні можна виділити:

- автоматична перевірка коректності змін;
- швидке виявлення помилок;
- забезпечення стабільності проєкту.

Тепер не треба щоразу самому запускати тести - процес став автономним та може виконуватись на хмарі.

Етапами роботи є:

- збір коду з локального середовища;
- налаштування хмарного середовища;
- встановлення залежностей;
- запуск тестів;
- формування звітів.

У проєкті CI/CD працює завдяки GitHub Actions. Коли щось змінюється в репозиторії, процеси стартують самі собою. Так відбувається реакція на новий код чи оновлення гілки.

У процесі автоматизації все починається з *workflow* - це просто файл налаштувань, де написано, які кроки слід пройти. Кожна дія тут має свою чергу через такий документ, а для роботи встановлюється Playwright разом із потрібними браузерами

Отже, при кожному старті workflow насправді створюється свіжа копія середовища - перевірка проходить заново. Саме тому кожен раз система стартує з нуля - минулі спроби не мають жодного значення. Завдяки цьому похибка не нагромаджується, а те саме можна повторити без зайвих складнощів.

Замість звичайного прискорення роботи, контейнеризація разом із CI/CD дає змогу отримати однаковий та стабільний результат кожного разу. Через це тестування стає передбачуваним, бо навколишнє середовище майже не впливає. Така система менше реагує на хаос довкола, адже все попередньо запечатане. Врешті-решт важливіше не швидкість, а надійність кожного кроку.

Тут ці методи працюють по-простому, без важкої технічної бази. Зате можна глибше подивитися, як різні умови впливають на хід перевірок. Такий підхід показує чітку картину того, що дає кожне середовище при тестуванні.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА ОРГАНІЗАЦІЯ ТЕСТОВОГО ФРЕЙМВОРКУ

2.1. Архітектура тестового фреймворку

У межах даної роботи тестовий фреймворк було спроєктовано з урахуванням простоти, зрозумілості та можливості стабільного виконання тестів у різних середовищах. Тести мають чітко працювати скрізь: на комп'ютері розробника, у Docker або під час автоматизації. Однаковий запуск без правок у коді - обов'язкова умова для всіх сценаріїв. Архітектура спроєктована так, щоб забезпечити цю стабільність завжди.

Сам фреймворк зроблено на базі Python та Pytest – саме вони задають каркас для проєкту та логіку розташування перевірок. Він хоч і має чітку будову, але не перенавантажений складними шарами абстракцій, бо спрямований на живий приклад автотестів, а не на створення універсального інструментарію.

У проєкті теки розподілені так, щоб кожна вирішувала свою задачу - нічого зайвого. Розділення дозволяє не плутатися, коли коду багато. Одні файли тут через функції, інші - бо відповідають за взаємодію. Кожна папка має чітке призначення - так легше шукати помилки, адже все на своїх місцях.

Структура проєкту організована у вигляді окремих директорій:

- папка *tests/* - має в собі сценарії для перевірок;
- папка *pages/* - зберігає файли, які описують кожну сторінку сайту, залучених до тестування. Тут можна знайти код, який формує інтерфейс. Кожен елемент працює незалежно, а структура допомагає організувати навігацію. Файли, в свою чергу, взаємодіють з маршрутизацією автоматично;
- файл *conftest.py* використовується для зберігання загальних налаштувань та *fixtures*;
- директорія *.github/workflows/* містить конфігурацію CI/CD;
- файл *Dockerfile* описує контейнеризоване середовище;
- файл *requirements.txt* містить список залежностей.

Ось так можна точно визначити, хто за що відповідає у різних частинах проєкту, а це полегшує технічне обслуговування. Можливості для помилок зменшуються через те, що кожен елемент окремий.

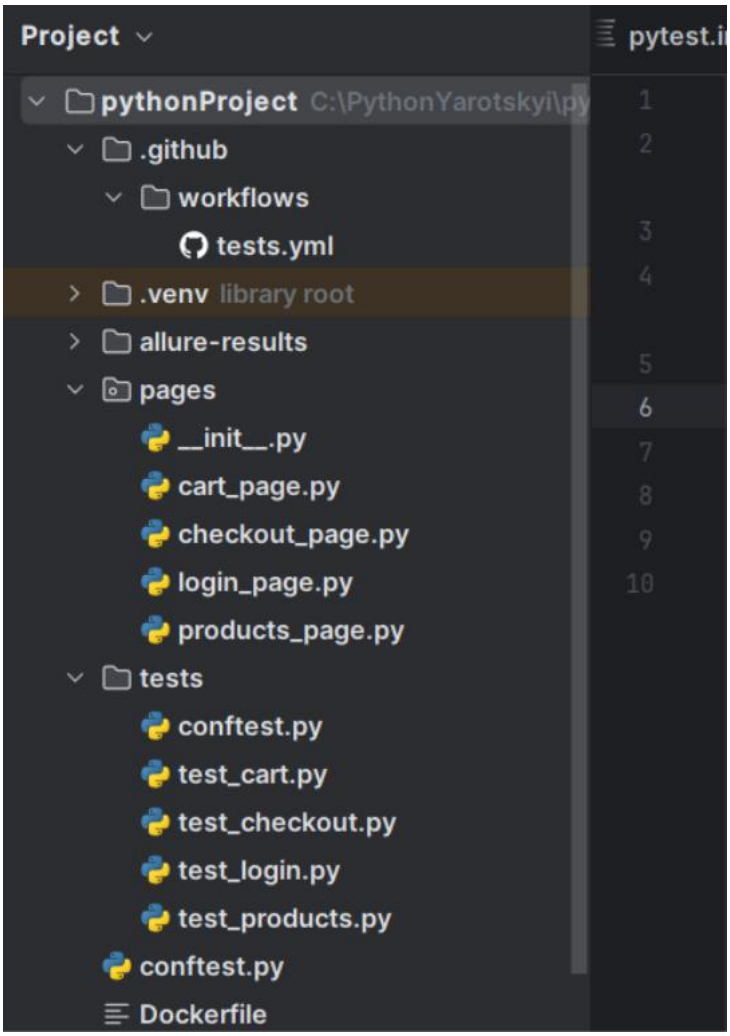


Рисунок 2.1. Структура проєкту з основними директоріями

Особливість будови - перевірки працюють однаково скрізь. Налаштовують процес інструменти типу Pytest чи зовнішні файли. Docker або CI/CD беруть участь у підготовці. Той самий набір перевірок стартує в різних місцях без правок. Умови не мають значення для коду.

Один з ключових моментів у створенні - окремий запуск перевірок. Жодна перевірка не чекає на іншу, працює самостійно. Так можна швидше знайти проблему, коли щось піде не так, що в свою чергу дозволяє:

- уникнути впливу на порядок виконання тестів;
- підвищити стабільність;
- спростити пошук помилок;

- ефективно використовувати паралельне виконання.

Окрема сесія в браузері запускається для кожного тесту - і зникає одразу після його закінчення. Через це наступний тест нічого не знає про те, що було до. Стан додатку не має шансу потрапити з одного перевірочного етапу на інший.

Масштабувати систему можна легко - дизайн це дозволяє. Скажімо, коли потрібно вставити ще одну перевірку, старий код лишається незайманим, головне - триматися заданого шаблону. Саме тому тут немає заплутаних схем чи надбудов - жодних окремих блоків для логіки чи обслуговування процесів, як це буває в масштабних рішеннях.

Запуск тесту починається з ініціалізації у системі. Потім запит потрапляє до браузера. Браузер взаємодіє з вебдодатком. Вебдодаток обробляє дані. Результат повертається назад через ланцюжок. Перевірка завершується фіксуванням виходу.

Важливим, що б я виділив в цій системі - це простота та простір для дій. Через це кожна дія функціонує окремо, не зачипляючи інші. Працювати можна там, де знадобиться: на одному комп'ютері чи кількох разом.

2.2. Використання Page Object Model

Працюючи над тестовим фреймворком для вебдодатка, було обрано модель **Page Object Model**. Завдяки цьому спосібу код стає зручнішим - перевірки не плутаються з тим, як шукати кнопки чи поля. Один блок відповідає за сторінку, інший - за те, що з нею роблять у тестах.

Такий поділ допомагає менше помилятися під час оновлення інтерфейсу.

Якщо змінюється верстка, правити доведеться лише один файл замість десятків тестів. Це прискорює супровід і робить перевірки стабільнішими без зайвих складнощів.

Ось ідея: кожна сторінка сайту стає своїм класом у коді. Всередині - те, що на ній є, плюс те, що з цим можна робити. Перевірки, в свою чергу, не зачіпають селектори прямо, замість цього вони покладаються на функції таких класів.

Так можна перестати копіювати один і той самий код. Коли потрібно оновити те, як знайти елемент на сторінці - правлять лише одну частину, замість усіх тестів разом. Замість того щоб редагувати по кожному файлу окремо, просто змінюють центральне правило. Якщо верстка змінилась чи перемістилась певна кнопка - необхідно буде змінити один блок, а решта залишилася цілою. Порядок тримається навіть якщо уявити, що хтось новий приходить у проект. Нема потреби повторювати одне й те ж у десяти місцях одночасно. Щоразу, як щось змінюється - система не розпадається. Простіше бачити помилки, бо все зібране в одному місці, а не розкидане різних директоріях.

Тож, було створено окремі класи сторінок для перевірки вебдодатку *SauceDemo*. Зокрема, саме так підходили до автоматизації:

- *products_page/;*
- *cart_page/;*
- *checkout_page/;*
- *login_page/;*

Один за одним - усі тут беруть на себе певну частину роботи в середині програми.

А класи сторінок у вебдодатку з класами у патерні Page Object містять:

- локатори елементів (кнопки, поля, списки);
- способи спілкуватися із такими елементами;
- допоміжні перевірки (наприклад, чи відображається певний елемент).

Сторінка з товарами часто має функції - покласти у кошик, впорядкувати чи просто показати список. Перевірка запускає одну з них, не цікавлячись, як саме все влаштовано всередині.

Таким чином перевірки стають зрозумілішими. Всі дії можна порівняти з тим, що робить людина крок за кроком, використовуючі подібні ресурси:

- відкрити сторінку;
- додати товар;
- перейти до кошика;
- перевірити результат.

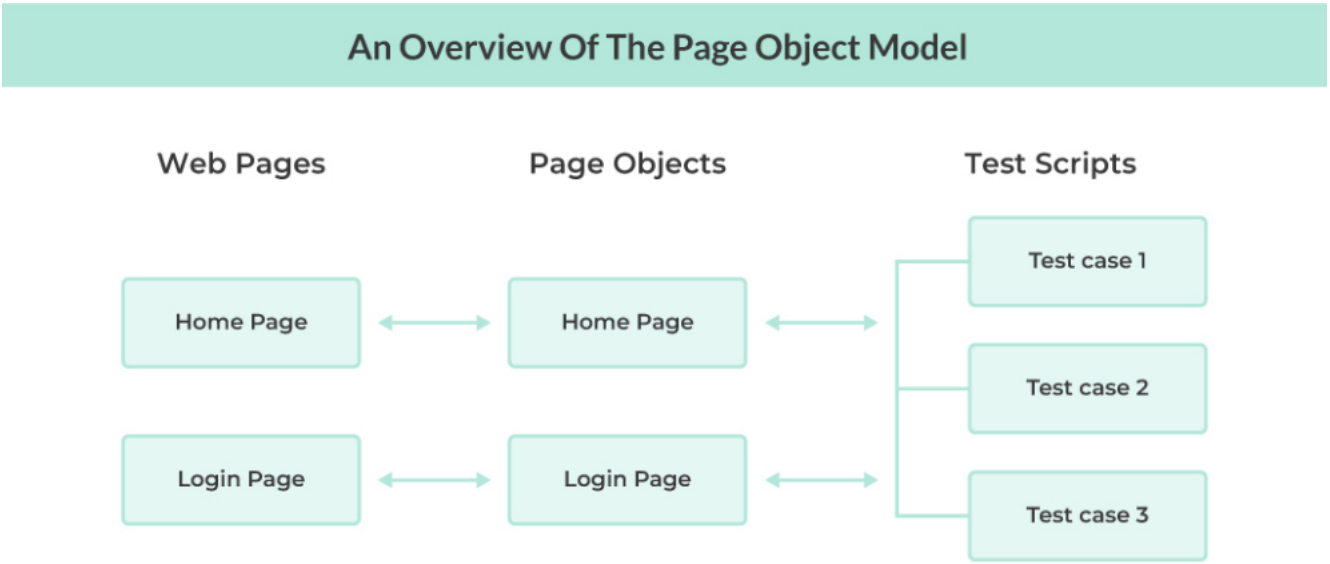


Рисунок 2.2. Структура роботи з елементами за допомогою Page Object Model

Завдяки POM тестування стає незалежнішим від інтерфейсу. Через це зміни в HTML-структурі ламають тести значно рідше.

Окрема папка pages/ - де знаходяться класи сторінок у цьому проєкті. Вони там не просто так: розділення допомагає тримати порядок між кодом і тестами. Пересування між файлами та робота з ними стали при цьому набагато зручнішими.

Варто звернути увагу, що модель Page Object тут реалізовано простіше, ніж можна було очікувати. Додаткових шарів абстракції немає, як і складних прийомів на кшталт фабрик сторінок чи впровадження залежностей. Було вирішено обійтись без цього, аби не перетворювати все на лабіринт, залишивши код доступним для розуміння.

Отже, завдяки моделі Page Object стало можливим:

- підвищити читабельність тестів;
- зменшити дублювання коду;
- спростити підтримку;

Тепер перевірка коду виглядає чіткіше, логічніше та є простішою в доповненні.

2.3. Організація тестових сценаріїв та fixtures

При організації процесу була закладена ідея, аби кожен тест функціонував самостійно, а через продуману структуру було легко збагнути, як влаштований сценарій. Файли для налагодження готувалися так, щоб працювати скрізь однаково надійно. Такий підхід дозволяє не загубитися при перевірках.

У папці *tests/* лежать сценарії перевірок - так зазвичай роблять із *Pytest*. Файл кожен охоплює те, що стосується одного шматка коду вебпроекту.

Такий підхід дозволяє:

- швидко знаходити потрібні тести;
- розділяти сценарії за змістом;
- не збирати всі тести в одному файлі. Замість цього - розподіляти їх логічно. Так буде простіше шукати потенційні помилки.

Тестування будують так, аби кожен сценарій мав свій шлях. Жоден крок одного перевірочного процесу не чекає на успішне завершення іншого. Якщо один упаде - решта продовжать роботи без затримок. Немає ланцюгів, немає черг.

Інколи тести починаються з будь-якого етапу. Порядок не обов’язково має бути строгим. Ідея в тому, що чергування не має заважати процесу. Кожен раз система має працювати незалежно - що було раніше на результат не впливає. У такому випадку значно легше знайти, звідки взялися проблеми.

Один тест - одна ситуація: як людина входить у кабінет, кладе щось у кошик чи оплачує замовлення. Хоч ці тексти не розповсюджуються на десятки кроків, кожен тримається за окремий етап. Замість накопичення дій, тут важливий акцент на простоті. Невеличкий об’єм допомагає швидше знаходити, де саме зламалося. Жодних масштабних ланцюжків - лише те, що потрібно перевірити зараз. Фокус тримається там, де ми очікуємо логічний фактичний результат. Спрощений підхід дає можливість не губитися серед зайвого.

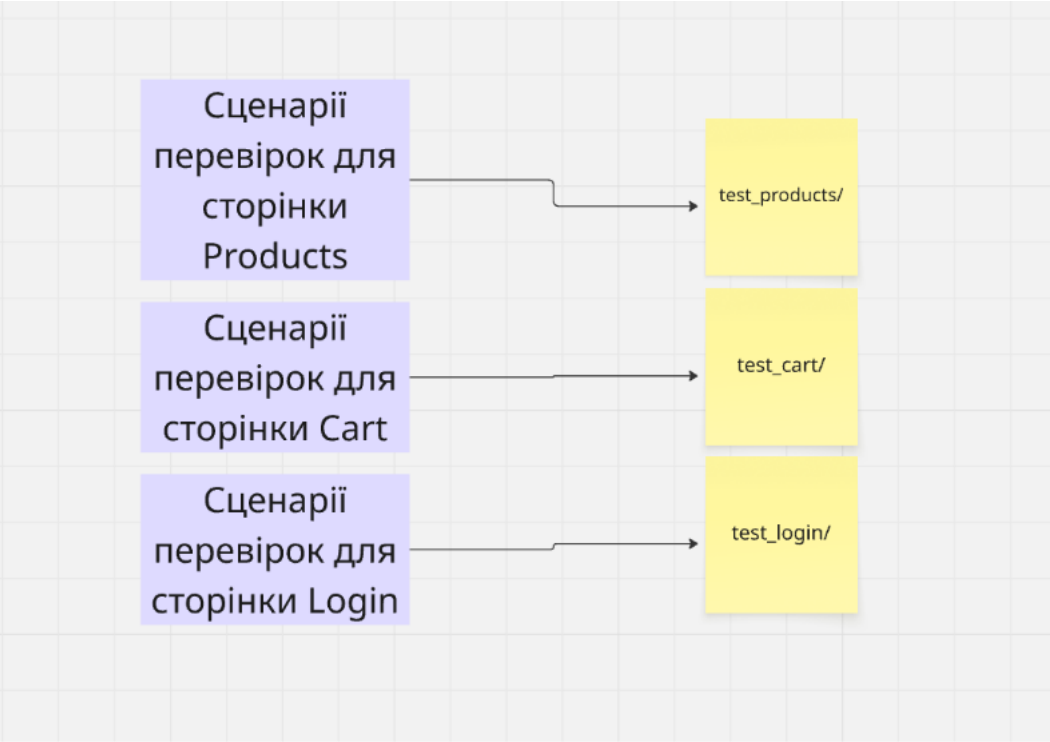
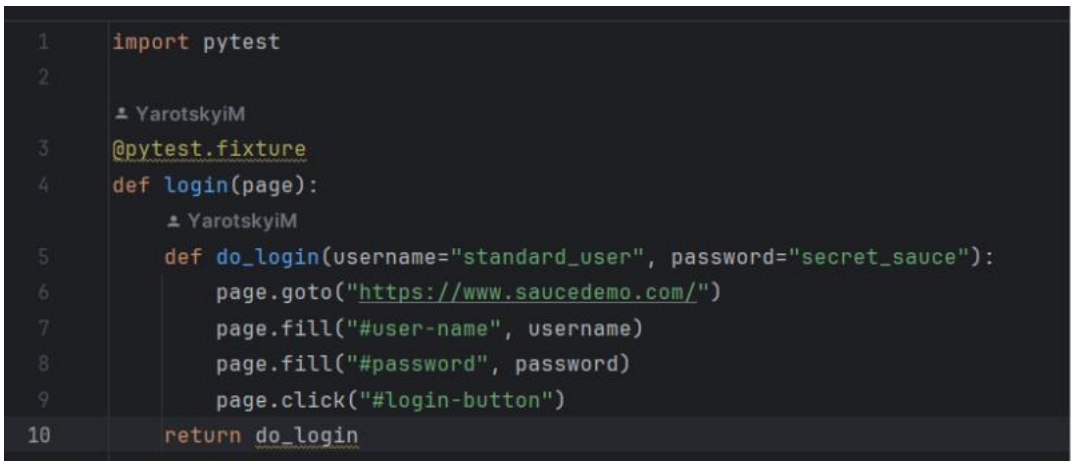


Рисунок 2.3. Приклад структури директорії *tests/*

Серед елементів фреймворку виділяються fixtures. Перш ніж запустити будь-який тест - саме вони налаштовують оточення. Їхнє завдання: ділитися однаковою логікою між окремими перевірками. Без них кожен сценарій довелося б готувати з нуля. У проєкті фікстури мають знаходитись у файлі conftest.py - тому до них можна дістатися будь-де серед тестів, навіть якщо явно не підключаєш. Код стає простішим просто завдяки цьому розташуванню.

Тести влаштовані так, аби було легко зрозуміти, що вони роблять. Назви функцій підказують - про що кожен окремий тест. Це допомагає не губитись у деталях. Сценарії об’єднуються разом, коли стосуються одного блоку коду. Так простіше тримати все під контролем.

Один із моментів - це скорочення повторень. Там, де операції часто повертаються, їх не пишуть кожного разу всередині перевірок, замість цього виносять на окремий рівень логіки. Через такий підхід сценарії стають простішими для сприйняття й займають менше рядків.



```python
import pytest

YarotskyiM
@pytest.fixture
def login(page):
    YarotskyiM
    def do_login(username="standard_user", password="secret_sauce"):
        page.goto("https://www.saucedemo.com/")
        page.fill("#user-name", username)
        page.fill("#password", password)
        page.click("#login-button")
    return do_login
```

Рисунок 2.4. Приклад наповнення файлу *conftest.py*

Отже, перевірка сценаріїв і підготовлених даних тут влаштована так, щоб:

- чітку структуру тестів;
- незалежність виконання;
- повторюваність результатів;
- зручність підтримки та можливість розширення.

В своїй сумі це створює основу для подальшого дослідження ефективності виконання тестів у різних середовищах.

2.4. Паралельне виконання тестів

Запуск кількох тестів одночасно допомагає швидше отримати результати. Хоча процес може ускладнюватися, саме такий спосіб часто лежить в основі продуктивного тестування.

Одразу кілька перевірок можна запустити разом - так працює **паралельне виконання**. Замість черги кожен процес йде окремим шляхом. Навіть якщо один затримається, інші продовжать далі. Таке розширення стає можливим через додаткові потоки чи фонові завдання.

Іноді запускаються різні сессії у браузері паралельно - кожна перевіряє свою частину інтерфейсу. Так працюють тести: не одна, а багато сторінок водночас, незалежно одна від одної. У результаті, кожен екземпляр живе окремо,

хоч і на одному пристрої, що може здатися складним, проте всередині прості дії - один клік, одна перевірка, тощо.

Безліч тестів працює одночасно лише тоді, коли кожен із них самостійний. У випадках коли одна перевірка зв’язана з іншою через загальні дані - починаються помилки під час запуску.

Отже, створюючи цей фреймворк, було звернуто увагу на кілька важливих правил:

- у кожному тесті - своя сесія браузера;
- відсутній спільний змінний стан між тестами;
- тестування можна проходити в будь-якому порядку, важливо лише зробити всі кроки;
- для кожного тесту стан готують окремо.

Безпечне виконання у кілька потоків стає можливим через слідування простим правилам. Неочікувані помилки у тестах тоді просто не з’являються, однак треба врахувати, що паралельні тести навантажують комп'ютер. Кожен із них працює у власному браузері, тому разом - це серйозне навантаження для процесора й пам’яті. Коли їх забагато, або пристрій на якому вони працюють заслабкий - система може почати гальмувати.

Отже, одночасна робота не обов’язково пропорційно скорочує час. Коли процесів стає забагато, користь спадає - інколи до мінусу.

У порівнянні, всередині Docker-контейнера працює своя логіка - тут усе тримається на межах цього самого контейнера, а у межах CI/CD-середовища, все працює завдяки можливостям від платформи. Тобто угазальнюючи, оптимальна кількість паралельних процесів залежить від:

- у локальному середовищі вона обмежена ресурсами комп’ютера;
- у Docker контейнері - обмеженнями контейнера;
- у CI/CD середовищі - ресурсами, які надає платформа.

Працюючи над цим проєктом, було вирішено врахувати швидкість обробки через кілька потоків. Такий підхід дає змогу побачити, як кожна платформа тримається під навантаженням у вигляді багатьох тестів зараз.

Це однаково працює у будь-якому оточенні - тому результати можна порівнювати чесно. Випадкові зміни навколо майже не трапляються, бо все запускається синхронно. Так виходить стабільніше дивитися на різницю. Умови не будуть змінюватись, тож ефект видно краще.

Виконання у паралелі прив'язується до таких речей, як можливість повторити результат чи передбачуваність поведінки. Коли тести правильно побудовано, ця практика не псує їх - навпаки, все лишається надійним, тільки час проходження зменшується.

Отже, завдяки паралельному запуску операцій у цьому дослідженні стає можливим:

- скоротити час виконання тестів;
- підвищити ефективність тестового процесу;
- з’ясувати, як умови навколо впливають на результативність;
- створюватимуть справжніше середовище для перевірок.

Важливість цього методу зростає разом із потребою перевіряти дані, бо без нього аналіз просто не запрацює.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ СЕРЕДОВИЩ ВИКОНАННЯ ТЕСТІВ

3.1. Локальне середовище виконання тестів

Локальне середовище для запуску тестів є найбазовішим способом автоматизованого тестування. Запуск фреймворку локально: це звичний спосіб будувати й налагоджувати перевірки. Саме це можна рахувати за основну точку розробки та налаштувань системи.

Це можна вважати за першу лінію перевірки робоспроможності коду, так як в цьому середовищі зручніше всього ловити помилки до моменту потрапляння в хмарну автоматизацію чи контейнери. Логіка така, що спочатку це відбувається тут, потім - у складніших умовах, коли впевненість уже є.

Тести на комп’ютері запускаються через Pytest - інструмент, що працює швидше завдяки функції pytest-xdist, за допомогою якої паралельне виконання допомагає уникнути довгих очікувань, хоч самі перевірки залишаються незмінними.

Щоб почати тести у локальному середовищі використовуються спеціальні команди, які вже є частиною тестового фреймворку по замовчуванню. Наприклад, **основними командами** для запуску тестів у локальному середовищі є:

- *pytest*
- *pytest -n auto*

Замість ручного встановлення кількості потоків, опція auto підбирає їхню кількість самостійно - враховуючи можливості пристрою. Так випробування стартують одночасно, адаптуючись до продуктивності апаратури.

Запуск тестів на робочому комп’ютері починається з інсталяції пакетів з файла **requirements.txt**. Браузери для *Playwright* додаються окремо - потрібні вони для перевірок інтерфейсу *SauceDemo*.

Без зайвих налаштувань - локальна машина сама впорається з тестами. Результати з'являються одразу, бо процес йде прямо у розробника, тож немає

потреби в спеціальному сервері так як все працює прямо на місці. Швидке виявлення помилок можливе завдяки запуску власноруч. Тестування, в свою чергу, проходить там само де пишеться код.

Кожна перевірка породжує свою власну сесію в браузері - так працює ізоляція. Через це один сценарій не чіпає інший. Паралельний запуск стає можливим саме завдяки такому розподілу.

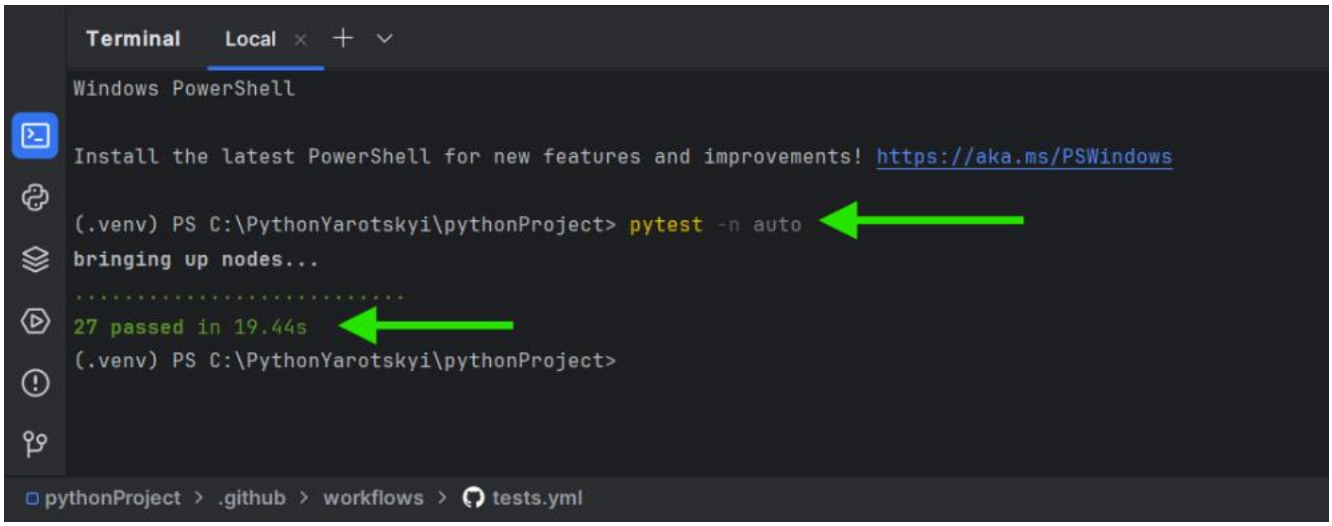


Рисунок 3.1. Демонстрація запуску тестів через pytest –n auto

При потребі зібрати більш детальну інформацію проходження тестів, у проєкті продемонстрована робота Allure Report. У власному оточенні створити звіт можна через додаткову команди:

- *pytest --alluredir=allure-results*
- *allure serve allure-results*

Отримані дані можна бачити як зручний звіт у форматі HTML - там є статус кожного тесту, послідовність дій під час перевірки й описано всі збої, це буде продемонстровано у подальшому звіті.

Як було зазначено раніше - найважливішою особливістю запуску автотестів на локальному середовищі є залежність від конфігурації конкретного пристрою. Тобто на швидкість та успішне відтворення можуть впливати такі чинники як:

- характеристики процесора;
- обсяг оперативної пам’яті;
- загальне навантаження системи.

Отже, тут локальна система дозволяє швидше перевіряти код. Хоча точність повторення роботи залежить від інших факторів - про це далі.

3.2. Контейнеризація тестового середовища за допомогою Docker

Частина цього дослідження покладається на Docker, щоб упакувати середовище для автоматичних перевірок. Замість того щоб запускати все прямо на машині - застосунки й бібліотеки можуть бути розміщені окремо в контейнерах. Такий підхід знімає проблеми сумісності: працює однаково на будь-якому комп’ютері. Навіть якщо система має свої особливості - результат не змінюється, так як вона застосовується тільки для запуску самого Docker-файлу, а не тестового середовища. Контейнери фіксують стан програмного оточення раз і назавжди. Все, що потрібне для тестів, вже є всередині та немає жодних зайвих кроків перед початком, а різні платформи, в свою чергу, перестають бути перешкодою. Один образ має усі необхідні версії на своїх місцях.

У проєкті застосовується готовий базовий образ:

http://mcr.microsoft.com/playwright/python:v1.59.0-jammy

У цьому шаблоні вже є Python, інструменти перегляду сторінок та все потрібне для запуску Playwright - ніяких додаткових кроків не знадобиться. Завдяки готовому оточенню конфігурувати Docker стає значно легше, адже нема чого встановлювати вручну.

В самому проєкті *Dockerfile* зроблений просто - в нього закладено мінімальну кількість команд, щоб відпрацьовував саме той функціонал, який закладався у дослідження:

- визначення базового образу;
- встановлення робочої директорії;
- у середині контейнера створено копію проєкту;
- встановлення Python-залежностей;
- запуск тестів.

Варто зазначити, що інколи задіюють *docker-compose*, щоб швидше підняти оточення. Хоча в цьому проекті його використано тільки як додатковий елемент. Він зовсім не формує структуру перевірочного каркасу. Замість основного механізму - просто помічник.

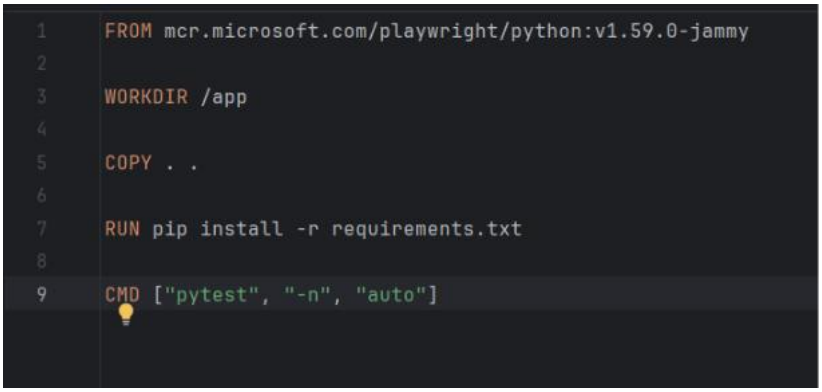


Рисунок 3.2. Налаштування *Dockerfile*

Запуск автоматизованої системи тестування можливий завдяки створенню й запуску контейнера, якщо спочатку скомпілювати образ, а потім дати йому команду про старт тестів:

- *docker build -t ui-tests .* - компіляція образу;
- *docker run ui-tests* - запуск автотестів всередині контейнера.

У Docker тести стартують завжди однаково - навколо них немає нічого зайвого. Тобто, можна це описати як один раз налаштовано - далі працює без додаткових змін. Середовище не плутається з іншим кодом через чистий контейнер. Контроль над версіями пакетів тримається міцно, бо все вже є всередині. Нема потреби вручну готувати систему кожного разу перед прогоном. Залежності конфліктують один з одним помітно рідше, або ж можуть не конфліктувати взагалі. Таке оточення повторюване, система не забруднена кешом чи іншими файлами від минулих запусків, адже починається заново. Результати тестів стабільніші просто тому, що фон постійний. Випадкові фактори мають менше шансів впливати на проходження.

Тож, з основних переваг, які забезпечують стабільність виконання та які можна виділити при використанні контейнерезації є:

- на роботу не впливає місцева система управління комп'ютером;

- кожна залежність залишається незмінною;
- версії бібліотек однакові для кожного запуску;
- випробування показують менше залежності від умов, де вони проводяться.

Коли мова заходить про перевірку інтерфейсу, різниця в умовах стає критичною - малі зміни середовища тут же позначаються на реакції браузера чи темпах прогону тестів.

Хоча Docker і допомагає, запуск тестів може ставити трохи або навіть значно повільнішим, в залежності від кількості тестів та складності реалізації сценаріїв. Справа у тому, що цей шар між кодом і системою, який гарантує нам більшу стабільність, в той же момент забирає час. Проте можна сказати, що так працює архітектура і часто приходиться миритись з сильними та слабкими сторонами різних систем.

Ця частина дослідження дає змогу співставити результати - через практичні перевірки. Порівняння виходить після тестувань у реальних умовах. Так ми можемо побачити різницю без зайвих припущень. Дані накопичуються поступово, протягом кількох етапів. Кожен крок має значення для загального висновку, а саме:

- швидкість локального виконання;
- швидкість виконання всередині контейнера;
- вплив ізоляції на стабільність результатів.

Очевидно можна виділити, що Docker тут не для масштабних схем чи керування контейнерами. Навпаки, він працює просто як інструмент, що забезпечує однакові умови запуску тестів та цим надає стабільність. Усе тому, що головне - порівняти результати при різних налаштуваннях, а не створювати потужну систему доставки коду, яку безперечно можна реалізувати на справжньому проєкті.

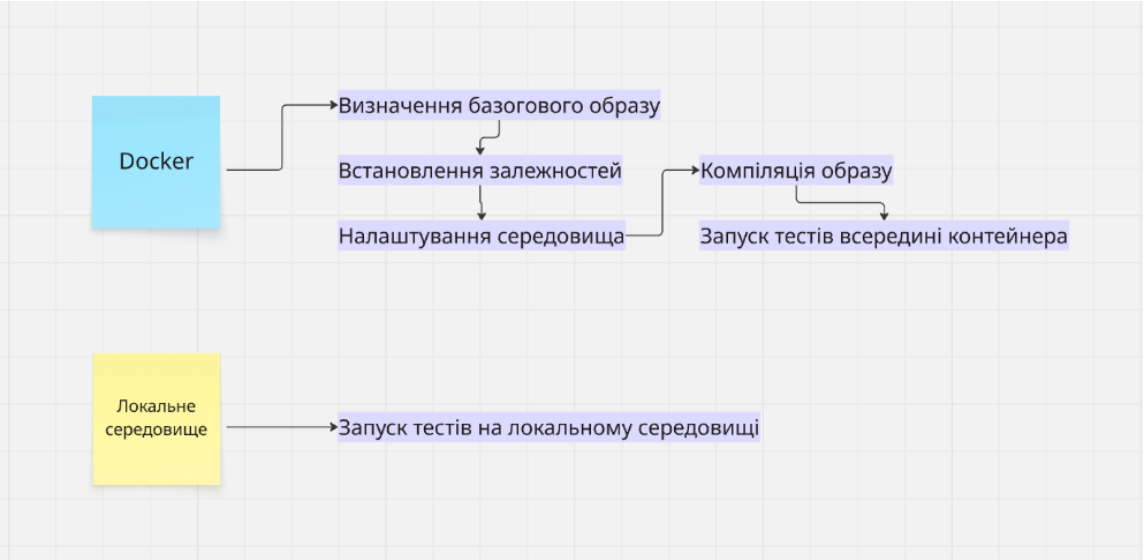


Рисунок 3.3. Порівняння шарів виконання запуску тестів у контейнері та у локальному середовищі

Отже, завдяки контейнерам у рамках проєкту результати залишаються однаковими навіть при різних умовах. Через це працювати з даними простіше - все прогнозовано. Порівнювати підходи до тестування тепер реально без зайвих поправок.

3.3. Реалізація CI/CD середовища у GitHub Actions

Щоб тести стартували самостійно в іншій системі - у цьому проекті задіяний GitHub Actions. Кожна нова версія коду в сховищі запускає перевірки без будь-яких додаткових дій чи налаштувань.

Кожен раз, коли хтось надсилає код на репозиторій - автоматично вмикається процес перевірки. Запуск відбувається після push або pull request, ніби система сама починає слідкувати за змінами. Тести проганяються одразу, щоб уникнути несподіванок далі під час розробки.

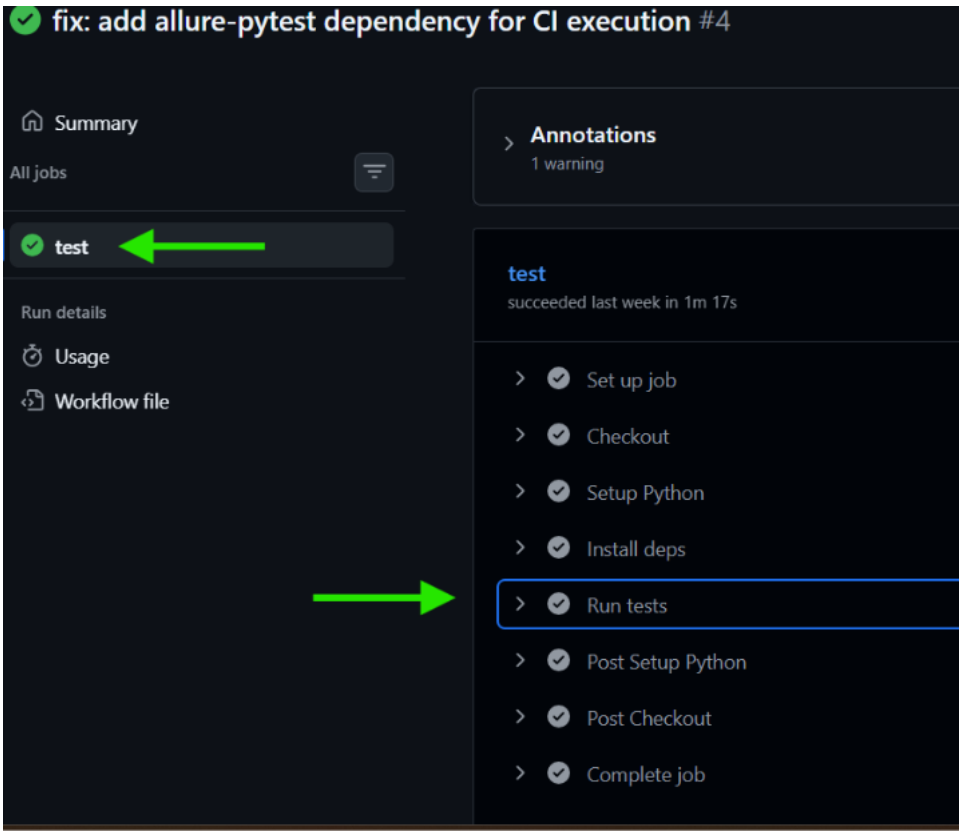


Рисунок 3.4. Демонстрація пройденого тесту на GitHub Actions

У файлі *tests.yml* задані налаштування - там чітко прописані всі ключові кроки роботи:

- отримання коду із репозиторію;
- налаштування середовища виконання;
- встановлення залежностей;
- запуск тестів.

Спочатку запускається action - це копіює репозиторій і забезпечує найновішу збірку коду. Потрапляння до файлів стає можливим одразу після клонування.

Для цього необхідно встановити потрібну версію Python та виконати його налаштування:

name: setup Python

uses: actions/setup-python@v5

with:

python-version: '3.12'

Далі системі необхнідно підтягнути потрібні модулі - браузери для роботи з Playwright:

name: Install deps

run: |

pip install -r requirements.txt

playwright install --with-deps

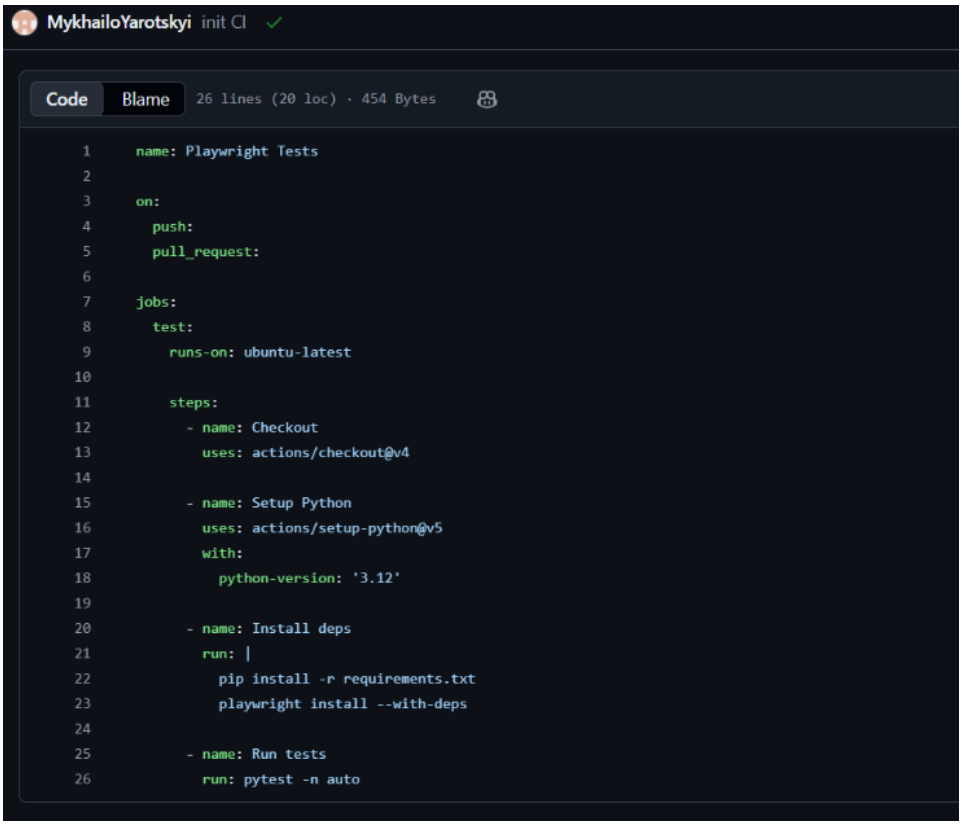


Рисунок 3.5. Ось виглядає файл *tests.yml* у сховищі

Крок має важливе значення, так як середовище GitHub Actions тимчасове, його перезбирають на кожному прогоні пайплайну. Щоб все працювало, потрібно встановлювати компоненти прямо під час роботи процесу.

Тести ж, в свою чергу, запускаються ось цією командою:

- name: Run tests run: pytest -n auto

У разі хмарного запуску тести виконуються паралельно - це завантажує потужності віртуальної машини повніше. Так само, як на локальному комп’ютері, процеси не чекають один одного.

Рисунок 3.6. Етап виконання тестів у хмарному середовищі

Один раз запустивши процес, отримується нове середовище з нуля. Кожен крок починається знову - без слідів минулих дій. Так працює система: немає зайвих даних від попередніх спроб. Навіть якщо щось зламалося раніше - тепер цього немає:

- після кожного старту нічого не лишається від минулих сеансів;
- кожна залежність налаштовується спочатку та без готових рішень;
- виконання дає той самий результат кожного разу - минулі тестування системи не мають значення.

Це допомагає знайти неполадки, що не завжди видно на локальній машині - іноді це пропущені бібліотеки чи невірні налаштування. Іншим разом виникають складнощі через різницю між системами. Таке трапляється часто, коли код переїжджає з одного комп’ютера на інший. Певні помилки починають з’являтись лише там, де всі частини збираються разом.

Ця робота працює з CI/CD середовищем, наче це окрема платформа для тестів - все запускається само, нічого не треба налаштовувати вручну. Результати приходять автоматично, бо процес побудовано так, щоб не чекати людини. Система діє самостійно, коли починається перевірка. Кожен крок виконується за заздалегідь заданим сценарієм, немає потреби втручатися. Такий підхід економить час інженерам, адже все відбувається у фоні. Тести проходять одразу після змін, завдяки сталому потоку опрацювання коду.

Завдяки цьому запуск тестів через GitHub Actions стає зручнішим - перевірка відбувається без участі людини. Процес працює стабільніше, коли його не потрібно організовувати вручну кожного разу. Таке рішення приживається всередині системи проєкту майже непомітно.

3.3. Автоматизація запуску тестів

Однією з головних думок, яка була закладена в цю роботу - запуск тестів має працювати однаково скрізь, хоч на комп’ютері, хоч у хмарі. Щодо самої організації: ключове - не переписувати перевірки кожен раз заново при зміні середовища.

Один і той самий рядок для запуску діє завжди, бо всередині структура залишається сталою. Неважливо чи це локальний прогін, контейнер чи збірка у віддаленій системі, важливо зрозуміти - усе це лише обгортка навколо спільного ядра. Різниця в оточенні, проте сценарій живе окремо, його не чіпають технічні деталі запуску. Іншими словами - те, що пише людина, не повинно залежати від того, де воно виконається. Тестова логіка стоїть осторонь конфігурацій, хоч їх багато. Не важливо, куди потрапляє команда - результат очікується один і той самий.

Запустити тести в проєкті можна такою командою:

pytest

Вона застосовується скрізь, де потрібні опції на кшталт одночасного прогону чи створення звітів. Без повторень у коді для запуску, підтримка тестування стає значно легшою.

Код проєкту містить усю конфігурацію - тому автоматизація працює без зайвих дій. Замість ручної підготовки, оточення налаштовується саме, як і fixtures з тестами. Наприклад, один раз створений логін-сетап прихований у fixtures-файлі означає, що його не потрібно описувати де небудь ще.

Інколи архітектура перевірок разом із шаблоном Page Object дає змогу прогнати один і той самий сценарій на різних середовищах майже без коригувань. Важко переоцінити, наскільки це допомагає тримати результати однаковими.

Також важливу роль відіграє саме налаштування параметрів запуску. Залежно від різних середовищ можуть з'являютися свої особливості, скажімо:

- паралельне виконання тестів;
- генерація звітів;

- починається всередині контейнера або у CI-оточенні.

Тим часом тести не міняються - різниця лише в тому, як до них звертаються та де саме відбувається запуск.

Рисунок 3.7. Єдина точка запуску, проте різні причини та способи виконання

Один із ключових моментів - це скорочення роботи вручну. Через звичайні команди запускаються всі процеси, без зайвого налаштування:

- локальний запуск;
- запуск у Docker;
- Запуск відбувається сам, коли починається процес у пайплайні.

Це працює однаково добре на етапі створення коду, а потім - коли його додають до сховища. Запуск через автоматизацію дає змогу експерементувати, так як усі тести проходять приблизно в тих самих обставинах, адже людина майже не бере участі.

Отже, ця робота побудована на тому, що команди для запуску тестів стали однаковими. Логіка прихована всередині спеціального шару - фреймворку, завдяки чому все працює стабільно. Запускати перевірки тепер можна скрізь: навіть якщо місце змінюється, код не треба правити. Кожен отримує те саме, коли повторює дії, а кожне виконання передбачуване. Немає зайвого - лише чіткий процес, який просто брати й використовувати.

РОЗДІЛ 4
АНАЛІЗ РЕЗУЛЬТАТІВ ТЕСТУВАННЯ ТА ЕКСПЕРЕМИНТАЛЬНЕ ДОСЛІДЖЕННЯ

4.1. Методологія проведення бенчмаркінгу та порівняння часу виконання

У межах експерименту ця робота повинна перевірити, як умови оточення впливають на швидкість автотестів для сайтів. Час проходження всього комплекту інтерфейсних тестів став головною мірою - він чітко показує, що працює швидше.

Дослідження охоплює три середовища виконання:

- локальне середовище розробника;
- контейнеризоване середовище (Docker);
- CI/CD середовище (GitHub Actions).

Всі застосовані тести - однакові та мають ідентичні сценарії роботи. Зміни у коді чи архітектурі фреймворку не торкаються жодного тестового сценарію. Так працює контроль над точністю порівнянь. Кожен замір тримається за одні й ті самі правила.

У кожному середовищі було виконано по п’ять тестувань для наглядності. Це допомагає знизити вплив від сторонніх подій - наприклад, коли система зайнята своїми справами або процесор перегрівається через інші процеси. Також варто зазначити, що важливою є затримка в мережі, особливо якщо все це відбувається у CI/CD.

Як було описано в минулих розлідах - кожен запуск проходив за однією й тією ж інструкцією, в не залежності від середовище, тобто за допомогою команди:

pytest –n auto

Усі тести працюють паралельно під час запуску - так порівнювати платформи стає точніше. Різниця видна одразу, так як нічого зайвого не заважає. Кожне середовище випробовується окремо, для отримання точніших результатів.

Порядок запуску не змінює дані, тож можна побачити справжню поведінку систем.

Отримані результати.

На скільки швидко пройде перевірка - залежить від обсягу завдань та особливостей роботи кожного з середовищ.

У локальному режимі час виконання наступний:

1. 7.30 секунд - початковий результат;
2. 8.06 секунд - **найповільніший результат;**
3. 6.57 секунд - **найшвидший результат;**
4. 7.63 секунд;
5. 7.91 секунд.

Час виконання для Docker:

1. 9.21 секунд - початковий результат;
2. 8.31 секунд;
3. 10.43 секунд - **найповільніший результат;**
4. 7.19 секунд - **найшвидший результат;**
5. 9.25 секунд.

Час виконання для GitHub Actions:

1. 1.17 хвилин - початковий результат;
2. 1.13 хвилин;
3. 2.01 хвилин - **найповільніший результат;**
4. 1.18 хвилин;
5. 1.10 хвилин - **найшвидший результат.**

Рисунок 4.1. Час виконання тестів у GitHub Actions з метриками від платформи

Щоб краще зрозуміти різницю - варто обчислити середнє значення:

- локальне середовище: приблизно 7,49 секунд;
- запуск у Docker середовищі: приблизно 8,89 секунд;
- біля 1 хвилини та 30 секунд триває процес у GitHub Actions.

Аналіз результатів.

Видно, що результати суттєво змінюються залежно від умов. Там, де працювало одне - інше давало не той ефект. Різні особливості середовищ призводили до видимого відриву в показниках. Цифри чітко вказували: середовище має значення. Помітили це ще на початкових кроках перевірки.

Найменший час виконання - результат роботи у локальному середовищі, яке використовує потужність апаратного забезпечення. Через це перевагу часто віддають саме локальному запуску під час створення чи поліпшення тестів.

Хоча Docker трохи уповільнює виконання - причина в тому, що тести працюють у межах контейнера. Саме це середовище створює зайвий шар ізоляції. Проте розрив у продуктивності ледь помітний. І взагалі не заважає тестам проходити нормально. Це можна рахувати чудовим вибіром між локальним та хмарним середовищами, так як контейнеризація додатково забезпечує стабільність. При цьому, варто зазначити, що на практиці реальних проєктів, часова різниця між Docker-середовищем та локальним буде збільшуватись на користь останнього, в залежності від кількості тестів.

Найдовше триває виконання через GitHub Actions. У хмарі доводиться чекати, поки стартує віртуальна машина - потім йде черга на завантаження необхідних компонентів і налаштування всього для роботи. Але так як запуск тестів на хмарному середовищі часто є фінальним етапом етапу CI/CD - довший час виконання аж ніяк не зменшує користь, яку приносить цей процес.

Стабільність виконання.

Важливо сказати: у кожному із трьох випробувань помилок не виникало. Тобто, перевірки пройшли чітко і без збоїв всі рази. Це означає, що тестовий набір є стабільним, а різниця результатів пов’язана виключно з особливостями кожного з середовищ.

Висновок експерименту

Загалом, отримані результати підтверджують закономірність.

Швидше за все працює локальне середовище - хоча результат тримається на потужності комп’ютера, який використовувався для запуску.

Docker трохи уповільнює роботу, він окремо тримає процеси. Проте завдяки цьому програми не ламають одна одну. Навіть якщо контейнер зламається, решта лишиться цілими. Крім того, запускати оновлення стає простіше через таку схему.

Хоча GitHub Actions працює довше за інших, усе ж він сам робить і збирання, і тестування, і розгортання. В той же момент цим він закриває важливий процес доставки та перевірки коду, що не може ігноруватись в комерційних проєктах.

Отже, при виборі середовища для запуску - важлива не тільки швидкість, але й те, навіщо потрібен саме цей тест. Швидке середовище може допомогти коли потрібно перевірити новий функціонал, але команда не має великого часового ресурсу. З іншої сторони, повільніші середовища можуть програвати у часі виконання, проте в свою чергу приносять стабільність та надійність.

4.2. Аналіз стабільності та відтворюваності тестів

Працездатність автоматичних перевірок часто свідчить про те, наскільки добре влаштована система тестування. У цьому дослідженні надійність означає,

що тести мають пройти ще раз так само, навіть коли їх запустити інакше або на інших середовищах.

Під час експерименту тестовий набір запускався багаторазово і кожен раз у нових умовах. Локальна машина, контейнери й процес інтеграції - усе пройшло успішно. Отже, робота тестового каркаса передбачувана - наслідки запуску не залежні від зовнішніх обставинами чи внутрішніх особливостей системи. Це стало можливим завдяки злиттю декількох архітектурних підходів:

- кожен тест працює самостійно у своїй браузерній сесії;
- відсутність залежності між тестами;
- використання Page Object Model для уніфікації взаємодії з UI;
- централізована підготовка стану через fixtures;
- одне й те саме тестове середовище задіюється протягом окремого запуску.

Кожен тест працює в стабільному оточенні - завдяки чому результат проходження не спирається ні на які додаткові особливості, а вплив інших перевірок виключено саме через такі методики.

Значна увага приділяється здатності отримувати той самий результат після чергового прогону перевірок. Якщо говорити про це дослідження - мається на увазі, що за однакових обставин тести мають давати ідентичні відповіді. Це виходить з того, що тестування завжди дає однакові підсумки - причина в тому, що фреймворк уникав сторонніх ненадійних елементів. Немає залучення сторонніх API чи дані на кшталт «випадковостей». Через це кожне запускання тримається за той самий шаблон без жодних винятків.

Отже, коли застосовують контейнери разом із налаштуваннями безперервної інтеграції, результат стає стабільнішим - усе тому, що середовище не міняється залежно від пристрою чи людини.

Проте, зазвичай у справжніх проєктах тести інтерфейсу можуть ламатись доволі часто. Це може ставатись через несправний тестовий фреймворк, проте набагато частіше можна виділити наступні причини:

- залежність від часу завантаження сторінки;
- нестабільні локатори UI елементів;

- асинхронна поведінка вебдодатку;
- залежність від попередніх тестових сценаріїв;
- різниця у середовищах для тестування.

Інколи тести, які працювали нормально, раптово починають збоїти - особливо коли доводиться мати справу з нестабільним оточенням. Таке руйнує процес CI/CD.

У таких випадках замість того щоб старатись зробити новий функціонал та покрити тестами дедалі більшу частину проєкту, команді варто зосередитися на стабільності перевірок. Через поділ логіки тестів від елементів сторінки помилки стали рідшими. Кожен сценарій працює самостійно - навіть якщо один упаде, решта продовжать працювати. Такий підхід може пробрати нестачу повторюваності через затримки чи чергу запуску.

4.3. Використання Allure Reports для візуалізації результатів тестування

Як згадувалось раніше - в роботі також реалізовано один з методів візуалізації результатів тестування, а саме Allure Reports. Це система, яку можна інтегрувати до локального середовища розробки, та за допомогою якої кожен крок перевірки потрапляє у звіт, де його можна оглянути окремо.

Важливо те, як організована інформація: не просто список, а логічний запис подій. Такий підхід дозволяє швидше розібратися в помилках, що значно легше робити коли є логування.

Після того як тести були запущені, дані осідають у файлах всередині папки allure-results. Звіт у форматі HTML створюється не паралельно із перевірками - вже потім, коли вся інформація доступна. Іншими словами, спочатку накопичуються результати, а лише пізніше перетворюються на сторінки, які можна дивитись у браузері. Цей порядок забезпечує точність: те, що показано, базується на завершеному циклі тестування.

Для цього використовуються наступна команда:

pytest --alluredir=allure-results

Далі команда для відкриття сторінки зі звітом:

allure serve allure-results

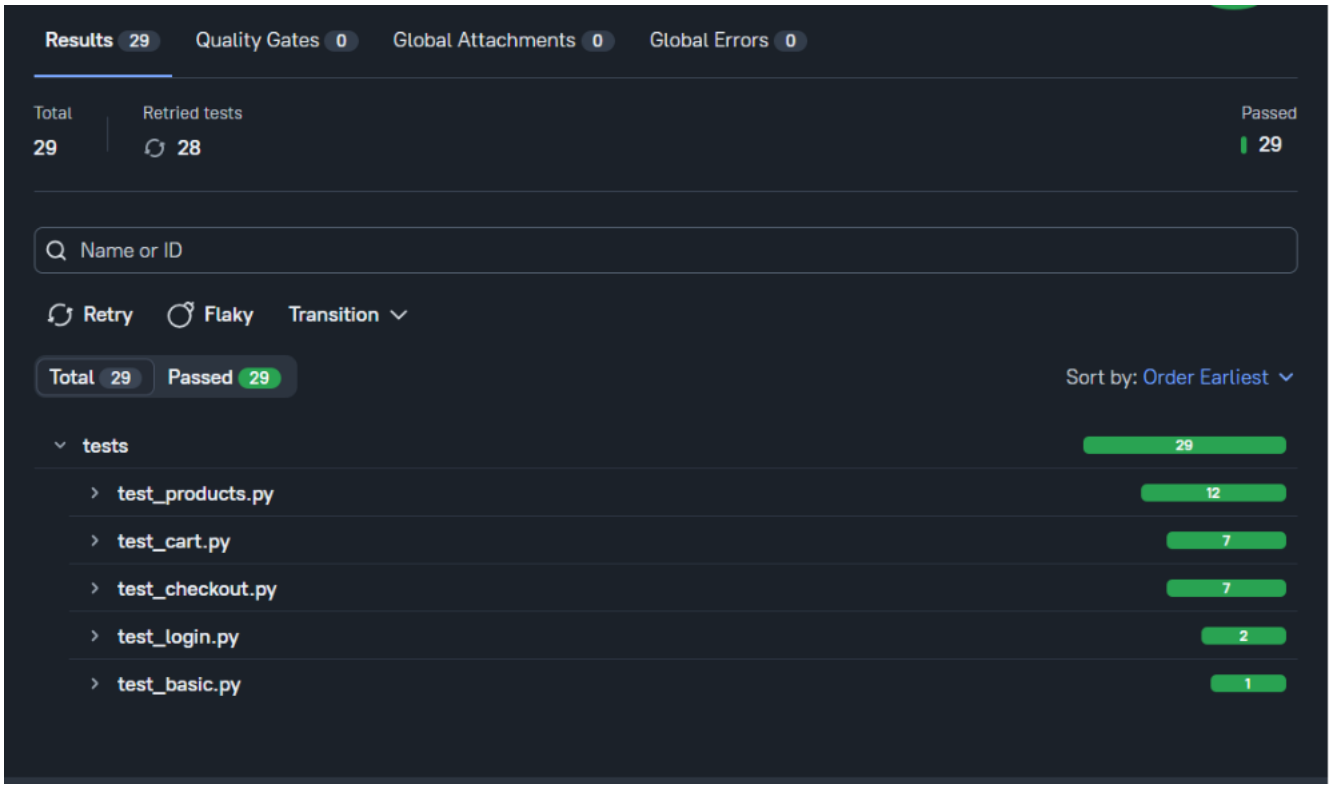


Рисунок 4.2. Головна сторінка Allure report з результатами запуску

Кожний тест описано детально у звіті та розділено по папках, у яких вони були запущенні. Іншими словами - тут є дані про всі перевірки, які були додані у сам код тестового фреймворку.

Загалом звіт містить у собі інформацію про всі важливі моменти:

- статус виконання тесту: passed / failed / skipped;
- тривалість виконання;
- кроки тесту (якщо були додані);
- додаткові метадані.

У цьому проєкті були також реалізовані Allure-анотації, щоб звіт був зручнішим. Вони допомагають краще організувати матеріал, роблячи його читабельним. Це спроба візуалізувати логічну послідовність у тестовій звітності. Ці анотації можна виділити як:

- feature - для групування тестів за функціональністю;

- severity - важливість тесту для системи;
- story - для деталізації сценаріїв;
- title - має пояснювати суть перевірки;
- step - для відокремлення дій, які виконує сценарій.

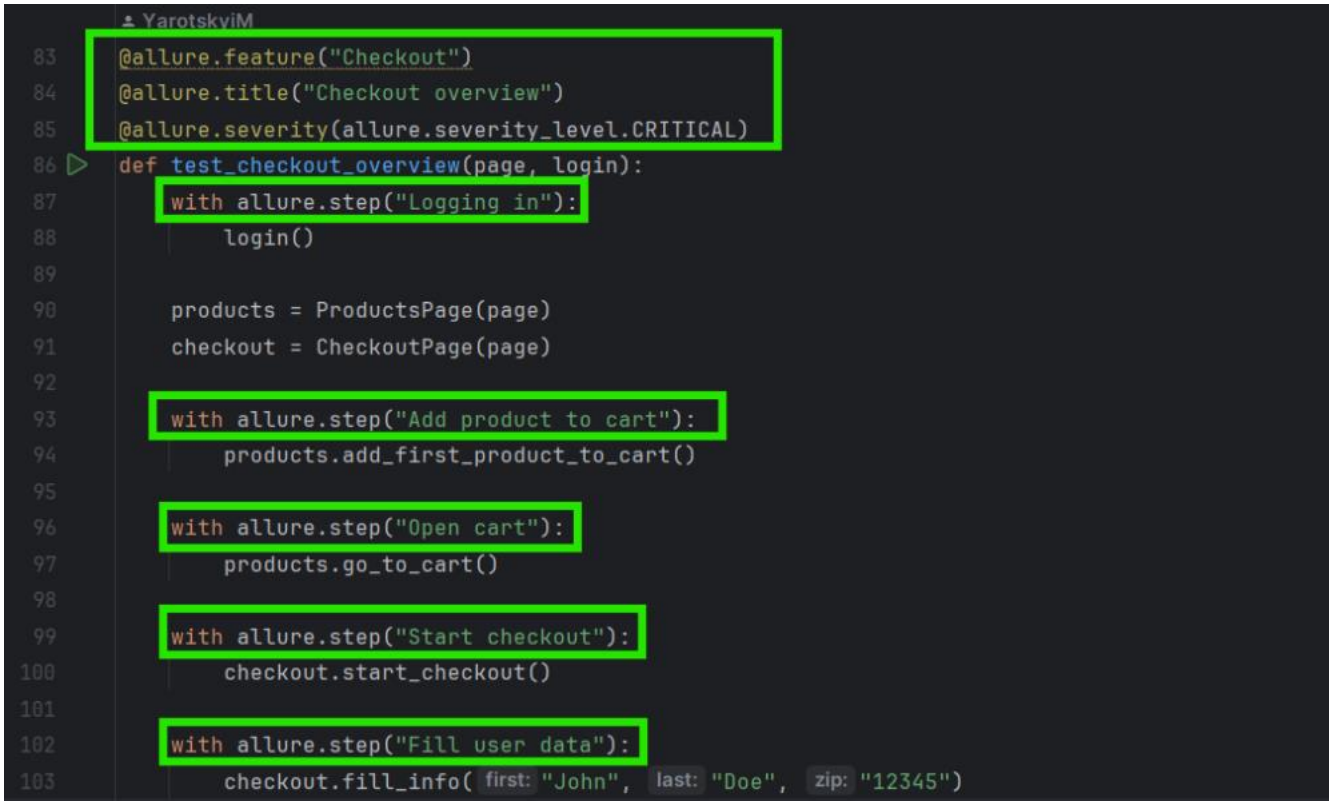


Рисунок 4.3. Приклади анотацій, які були додані у тестові сценарії

За допомогою цього перевірка стає простішою, її підсумки легко сприймаються не тільки програмістом, а й рештою команди. Завдяки цьому кожен, хто задіяний у створенні продукту, швидше вловлює суть помилок. Хтось оцінює логіку, інший – технічну точність, але всі мають доступ до однаково зрозумілої картини. Це працює як міст – пояснює стан системи без зайвих термінів. Тому можна вважати підхід візуалізації результатів тестування корисним для роботи та співпраці різних членів команди.

Проте, важливо зазначити, що Allure Reports використовується у даній роботі виключно для візуалізації результатів та можливостей, які надає цей інструмент спеціалістам, але він не є частиною тестування чи аналізу.

Завдяки ньому можна побачити:

- які тести пройшли успішно;

- які сценарії виконувались найдовше та найшвидше;
- на якому кроці виникли помилки (у випадку їх появи);
- загальний стан тестового набору.

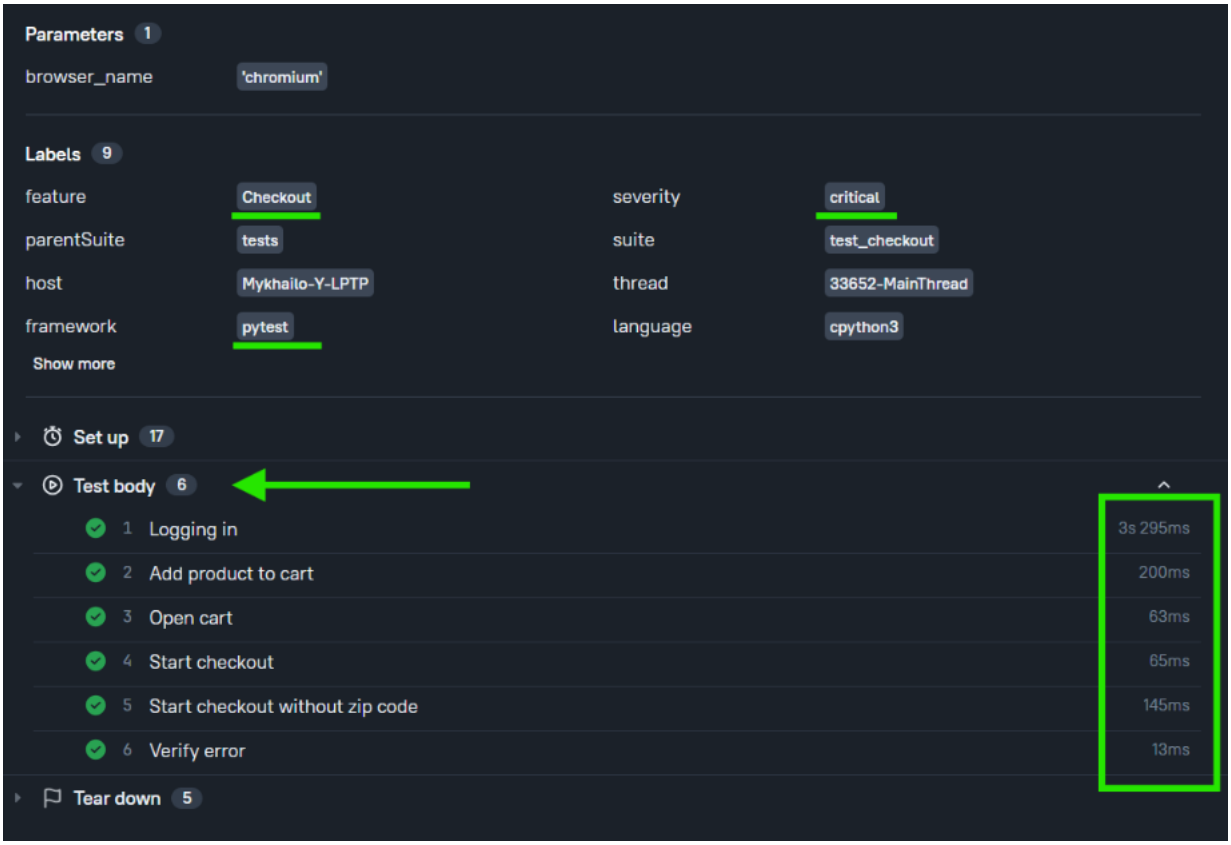


Рисунок 4.4. Приклад візуалізації одного з пройдених тестів

Отже, завдяки Allure Reports результати перевірок стають видимими для всіх учасників, це спрощує розбір помилок, адже все лежить на поверхні. Замість потенційної плутанини з’являється порядок, де кожен крок має місце та пояснення.

ВИСНОВКИ

У цій роботі була виконана спроба автоматизувати перевірку вебзастосунків, використовуючи новітні засоби разом із різними умовами запуску тестів. Крім того, було розглянуто те, як все це працює в реальних умовах. Деталі аналізу базуються на практичному застосуванні інструментів, а основна увага приділяється тому, що відбувається під час виконання. Процес показує, як технології взаємодіють між собою. Результатом став чіткий погляд на механіки тестування.

З погляду теорії - увага приділена ключовим ідеям перевірки вебзастосунків за допомогою автоматизованого тестування. Серед них: тестування через інтерфейс, повноцінні сценарії E2E, окреме запускання кожного тесту, а також результати при повторних запусках на різних середовищах.

Зупинка на головних засобах для створення середовищ тестування виявилася неминучою. У числі таких: Pytest як основа та Playwright для емуляції дій користувача, Docker задля контролю над оточенням, GitHub Actions для запуску процесів без участі людини.

Через декілька кроків з’явився робочий каркас для перевірки сайту SauceDemo - основою стали шаблони сторінок і підготовлені блоки в Pytest. Кожен сценарій тепер існує окремо, не зачіпаючи інших, може працювати паралельно з рештою, та має один стиль запуску, хоч у хмарі, хоч на локальному комп’ютері.

Спеціально для перевірок було підготовано три варіанти оточення - локальне, на базі Docker та через GitHub Actions. У кожному було запущено одну й ту саму групу тестів, щоб порівняти швидкість. Час фіксувався чітко, без зайвих ускладнень. Результати показали, як працює система в різних умовах. Тести давали стабільні дані незалежно від платформи.

Виявилося під час тестування - отримані цифри вказують наступне:

Тестувати на власному середовищі - швидше за все. Просто тому що це локально. Навколо немає затримок мережі, запуск відбувається миттєво, бо нічого

не треба чекати, а результат отримується відразу після запуску. Так, у даному прикладі всі тести проходили успішно, але на комплексних та комерційних проєктах локальне середовище не рекомендується для запуску тестів з метою отримання фінальних результатів, через його потенційну неточність та залежність від апаратного забезпечення.

Завдяки Docker'у середовище залишається стабільним і легко відтворюваним, хоча й з невеликим перевантаженням. Майже завжди присутній додатковий шар - це плата за узгодженість між системами. Так само важко уникнути затримок, проте результатом є передбачувана робота коду, що є ключовим плюсом.

Хоча GitHub Actions працює довше за інших, усе ж він сам робить перевірку коду без зайвих дій з боку спеціалістів, а кожна зміна у коді репозиторія супроводжується запуском тестів, що є одним з найважливіших етапів підходу CI/CD.

У процесі також були застосовані звіти Allure Reports - це допомогло краще бачити результати перевірок. Візуалізація спростила розуміння пройдених тестів, адже інформація стала доступнішою для перегляду.

Загалом можна сказати, що вийшло те, що закладалось як ціль з самого спочатку: з'явився функціонуючий тестовий фреймворк, який готовий для розширення та адаптації на реальні проєкти. Він же в свою чергу дає змогу перевіряти, як UI тести себе поводять у різних умовах. Порівняння показує, як навколишнє середовище впливає на швидкість. Ще один аспект - стабільність процесу.

Ці дані стануть у нагоді для реалізації автотестування через контейнеризацію разом із безперервною інтеграцією - особливо в учбових чи дослідницьких роботах. Такий підхід допоможе побачити, як техніки працюють на практиці, не залишаючись лише теорією. У процесі можна спостерігати, як елементи взаємодіють між собою під час збірки й перевірок. При цьому структура лишається гнучкою завдяки модульному оформленню середовищ. Кожен крок виконується послідовно, хоча порядок операцій легко адаптувати. Практика

показує: такі схеми швидко входять до повсякденного використання там, де потрібна точність. Навіть невеликий проєкт отримує користь від передбачуваності таких систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Мова програмування Python для інженерів і науковців: навчальний посібник. Ivano–Frankivsk, Ukraine : ІФНТУНГ, 2019. 275 с. (дата звернення: 07.05.2025).
2. Van Veenendaal E., Black R., Graham D. Foundations of Software Testing: ISTQB Certification. 5th Edition. Andover: Cengage Learning EMEA, 2024. 288 p.
3. Poulton N. Docker Deep Dive: Zero to Docker in a Single Book. Stratford Living Publishing, 2020. 250 p.
4. Obuse E., Erigha E.D., Okare B.P., Uzoka A.C., Owoade S., Ayanbode N. Integrating Continuous Integration Pipelines Using GitHub Actions, Jenkins, and End-to-End Test Automation Frameworks. International Journal of Multidisciplinary Evolutionary Research. 2021. Vol. 2, Issue 1. P. 63–75.
5. Авраменко А.С., Авраменко В.С., Косенюк Г.В. Тестування програмного забезпечення: ЧНУ імені Богдана Хмельницького, 2017 р. URL: https://eprints.cdu.edu.ua/1482/1/testyvan.pdf
6. Python Official Documentation: v.3.14.5, URL: https://docs.python.org/3/
7. Смагіна О.О., Якість програмного забезпечення та тестування: ЛНУ імені Тараса Шевченка, 2021 р.
8. Pytest Documentation: URL: https://docs.pytest.org/en/stable/
9. Playwright Python Documentation: URL: https://playwright.dev/python/docs/intro
10. Соммервілль І., Інженерія програмного забезпечення. 10-те вид.: Київ: Видавнича група BHV, 2019 р.
11. Martin Fowler, Page Object Model: URL: https://martinfowler.com/bliki/PageObject.html
12. Гамма Е., Хелм Р., Джонсон Р., Вліссідес Дж., Патерни проєктування: 2020 р.
13. Ramalho L., Fluent Python. 2nd Edition.: Sebastopol: O’Reilly Media, 2022 p.

14. Трофименко О. Г., Дика А. І., Тестування та забезпечення якості програмних систем: Національний університет «Одеська юридичнр академія», Одеса, 2024 р.
15. Crispin L., Gregory J., Agile Testing: A Practical Guide for Testers and Agile Teams: Addison-Wesley, 2009 p.
16. Allure Framework Documentation: URL: https://allurereport.org/docs/