ПОЛТАВСЬКИЙ ДЕРЖАВНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
Навчально-науковий інститут економіки, управління, права та інформаційних технологій
Кафедра інформаційних систем та технологій

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти магістр

на тему: **«Оптимізація процесу автоматизації тестування вебдодатків на основі фреймворку Playwright»**

Виконав: здобувач вищої освіти
за освітньою програмою
Інформаційні управляючі системи та технології
спеціальності 126 Інформаційні системи та технології
ступеня вищої освіти магістр
групи 126ІСТ_мд_2024
Даценко Назар Ігорович
Керівник: Флегантов Леонід Олексійович
Рецензент: Ковальчук Станіслав Богданович

Полтава – 2025 року

ПОЛТАВСЬКИЙ ДЕРЖАВНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
Навчально-науковий інститут економіки, управління, права та інформаційних технологій
Кафедра інформаційних систем та технологій

Освітня програма Інформаційні управляючі системи та технології
Спеціальність 126 Інформаційні системи та технології
Рівень вищої освіти другий (магістерський)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Юрій УТКІН

«08» листопада 2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА ВИЩОЇ ОСВІТИ

Даценко Назара Ігоровича

1. Тема роботи:
«Оптимізація процесу автоматизації тестування вебдодатків на основі фреймворку Playwright».
Керівник роботи к.ф.-м.н., доцент, професор кафедри інформаційних систем та технологій Флегантов Леонід Олексійович.
Затверджено наказом закладу вищої освіти від «31» жовтня 2025 року № 1332-ст

2. Строк подання здобувачем вищої освіти роботи «09» грудня 2025 р.

3. Вихідні дані до роботи: науково-технічна література, наукові статті та публікації з автоматизації тестування вебдодатків, архітектурних патернів і технічних рішень у сфері QA; матеріали з наукових баз даних Google Scholar, IEEE Xplore, SpringerLink, ScienceDirect, Scopus, ResearchGate; вітчизняні та міжнародні стандарти і методології тестування програмного забезпечення (зокрема ISTQB, ISO/IEC/IEEE 29119); офіційна документація та ресурси фреймворку Playwright і суміжних інструментів автоматизації; джерела з веброзробки та мов програмування (JavaScript, TypeScript, Python); аналітичні інтернет-ресурси, блоги, статті та технічні звіти експертів у галузях QA, DevOps та тестової автоматизації.

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Розділ 1. Теоретико-методологічні основи автоматизації тестування вебдодатків
Розділ 2. Фреймворк Playwright як технічне рішення для автоматизації тестування
Розділ 3. Практична реалізація оптимізованого процесу автоматизації тестування вебдодатків на основі Playwright

5. Перелік графічного матеріалу: схеми, рисунки, діаграми за темою та об’єктом дослідження.

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання отримав
Оцінювання економічної ефективності результатів дослідження	Калініченко О. В., к. е. н., доцент, доцент кафедри економіки та публічного управління	24.11.2025	04.12.2025

7. Дата видачі завдання «07» листопада 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1.	Вибір і затвердження теми роботи	29.10.2024 р.	
2.	Складання і затвердження розгорнутого плану та завдання на кваліфікаційну роботу	30.10.2024 р. – 08.11.2024 р.	
3.	Опрацювання джерел інформації	11.11.2024 р. – 27.12.2024 р.	
4.	Збір, вивчення і обробка інформації, необхідної для виконання роботи	30.12.2024 р.– 19.01.2025 р.	
5.	Виконання теоретико-методологічного розділу роботи	17.02.2025 р.– 16.05.2025 р.	
6.	Виконання дослідницько-аналітичного розділу роботи	02.06.2025 р.– 13.07.2025 р.	
7.	Виконання проєктно-рекомендаційного розділу роботи	08.09.2025 р.– 14.11.2025 р.	
8.	Оцінювання економічної ефективності результатів дослідження	24.11.2025 р.– 04.12.2025 р.	
9.	Оформлення тексту роботи	05.12.2025 р.– 08.12.2025 р.	
10.	Попередній захист роботи на кафедрі	09.12.2025 р.	
11.	Доопрацювання роботи з урахуванням зауважень і пропозицій	10.12.2025 р.- 14.12.2025 р.	
12.	Нормоконтроль	15.12.2025 р. – 16.12.2025 р.	
13.	Захист кваліфікаційної роботи	18.12.2025 р.	

Здобувач вищої освіти **Назар ДАЦЕНКО**

Керівник роботи **Леонід ФЛЕГАНТОВ**

ПОЛТАВСЬКИЙ ДЕРЖАВНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ЕКОНОМІКИ, УПРАВЛІННЯ, ПРАВА ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

ДАЦЕНКО НАЗАР ІГОРОВИЧ

«ОПТИМІЗАЦІЯ ПРОЦЕСУ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ВЕБДОДАТКІВ НА ОСНОВІ ФРЕЙМВОРКУ PLAYWRIGHT»

Освітньо-професійна програма
Інформаційні управляючі системи та технології
Спеціальність 126 Інформаційні системи та технології
Ступінь вищої освіти Магістр

РЕФЕРАТ
кваліфікаційної роботи на здобуття кваліфікації –
магістр з інформаційних систем та технологій

Полтава – 2025 року

Кваліфікаційна робота складається зі вступу, 3 розділів, висновків, списку використаних джерел (50 найменувань), додатків. Робота містить 15 рисунків, 10 таблиць, викладена на 70 сторінках.

Основний зміст роботи

Теоретична частина роботи включає аналіз теоретико-методологічних основ автоматизації тестування вебдодатків. Визначено роль автоматизації в CI/CD процесах та розглянуто сучасні архітектурні патерни (Page Object Model, Data-Driven Testing, BDD). Проведено порівняльний аналіз інструментів Selenium, Cypress, TestCafe та Playwright, за результатами якого обґрунтовано вибір Playwright як найбільш ефективного засобу для масштабованого кросбраузерного тестування завдяки його архітектурі на основі WebSocket.

Практична частина роботи присвячена розробці та програмній реалізації оптимізованого тестового фреймворку на базі стеку Python + Pytest + Playwright. Реалізовано архітектуру з використанням патернів проєктування POM та фікстур для управління залежностями. Впроваджено методику гібридного тестування, що поєднує API-запити для підготовки стану системи з UI-перевірками, а також налаштовано паралельне виконання тестів та їх запуск у Docker-контейнерах.

Проєктно-рекомендаційний розділ містить результати експериментальної оцінки ефективності розробленого рішення. Порівняння з традиційним підходом продемонструвало скорочення часу виконання тестів в 11 разів та підвищення їх стабільності до 99.5%. Виконано техніко-економічне обґрунтування, яке показало, що впровадження методики дозволяє заощадити $683.62 щомісяця, з терміном окупності 2.34 місяці та річним ROI 412.7%.

Заключні висновки підсумовують результати дослідження, підтверджуючи, що запропонована методика архітектурної оптимізації дозволяє суттєво підвищити швидкість та надійність процесу тестування, забезпечуючи високу економічну ефективність для IT-проєктів.

Висновки

Метою дослідження було обґрунтування та розробка методики архітектурної оптимізації фреймворку автоматизації тестування на базі Playwright, а також експериментальне визначення впливу цієї оптимізації на швидкість, стабільність та масштабованість процесу тестування.

Аналіз теоретичних основ та сучасних інструментів показав, що в умовах гнучких методологій розробки традиційні підходи до автоматизації (наприклад, на базі Selenium) часто стають вузьким місцем через низьку швидкість та нестабільність («flaky tests»). Обґрунтовано вибір фреймворку Playwright, архітектура якого на базі WebSocket та ізольованих контекстів вирішує більшість цих проблем.

Головним результатом роботи є розробка та практична реалізація гібридного підходу до тестування. Цей метод поєднує використання API-запитів для миттєвої підготовки стану системи (автентифікації) з UI-тестами для перевірки бізнес-логіки. Такий підхід дозволив усунути найбільш нестабільні етапи тестування.

Практична реалізація оптимізованої архітектури здійснена на стеку Python + Pytest із використанням патернів Page Object Model та Data-Driven Testing. Впроваджено механізм фікстур для управління залежностями, що забезпечило високу модульність та легкість підтримки коду.

Оцінка ефективності запропонованого рішення показала значне покращення показників у порівнянні з традиційним підходом: час виконання повного набору тестів скоротився в 11,2 рази, а стабільність тестів зросла до 99,5%. Техніко-економічне обґрунтування підтвердило доцільність впровадження: щомісячна економія операційних витрат склала $683,62, термін окупності інвестицій − 2,34 місяці, а річний ROI − 412,7%.

Наукова новизна та практична значущість роботи полягають у створенні універсального, економічно ефективного шаблону для автоматизації тестування вебдодатків, який може бути легко адаптований та масштабований для потреб сучасних ІТ-проєктів.

Список публікацій здобувача

1. Даценко Н. Архітектурна оптимізація процесу автоматизації тестування вебдодатків. *Матеріали науково-практичної конференції за підсумками проходження виробничих практик здобувачів вищої освіти спеціальності 126 Інформаційні системи та технології*, Полтава: ПДАУ, 2025. С. 54-56.
2. Даценко Н. Архітектура та принципи роботи фреймворку Playwright. *Матеріали XXII щорічного міждисциплінарного семінару «Студентські роботи за науковою тематикою кафедри інформаційних систем та технологій ННІ ЕУП та ІТ ПДАУ»*, Полтава, 2025.
3. Флегантов Л. Даценко Н. Архітектурні патерни у побудові фреймворків автоматизації тестування вебдодатків. *Innovative Approaches in Modern Science and Technology: Collection of Scientific Papers with Proceedings of the 3rd International Scientific and Practical Conference. Lisbon, Portugal*, 2025. P. 253-260.

АНОТАЦІЯ

Даценко Н.І. «Оптимізація процесу автоматизації тестування вебдодатків на основі фреймворку Playwright». Кваліфікаційна робота на правах рукопису.

Кваліфікаційна робота на здобуття ступеня вищої освіти магістр за освітньо-професійною програмою Інформаційні управляючі системи та технології спеціальності 126 Інформаційні системи та технології. Полтавський державний аграрний університет, Полтава, 2025.

Робота присвячена вирішенню актуальної задачі оптимізації процесу автоматизації тестування сучасних вебдодатків. Досліджено архітектурні особливості фреймворку Playwright та обґрунтовано його переваги над аналогами. Робота складається з трьох розділів, що охоплюють теоретичні основи, аналіз інструментарію та практичну реалізацію оптимізованої методики.

У першому розділі проаналізовано принципи побудови тестових фреймворків та виконано порівняння існуючих інструментів. Другий розділ розкриває технічні можливості Playwright, включаючи роботу з WebSocket та ізоляцію контекстів. Третій розділ містить опис розробленої архітектури на базі Python і Pytest, реалізацію гібридного підходу (API+UI) та розрахунок економічної ефективності, що підтверджує доцільність впровадження запропонованих рішень.

Результати дослідження мають практичну цінність для QA-команд, дозволяючи зменшити витрати на інфраструктуру та підтримку тестів при підвищенні їх стабільності.

Ключові слова: автоматизація тестування, Playwright, Python, Pytest, Page Object Model, гібридне тестування, CI/CD, оптимізація.

ANNOTATION

Datsenko N.I. «Optimization of the web application test automation process based on the Playwright framework». Master’s thesis manuscript.

The master’s thesis for obtaining a Master’s degree in the educational and professional program of Information Management Systems and Technologies, specialty 126 Information Systems and Technologies. Poltava State Agrarian University, Poltava, 2025.

The work is devoted to solving the urgent problem of optimizing the automation process of testing modern web applications. The architectural features of the Playwright framework are investigated, and its advantages over analogues are substantiated. The work consists of three sections covering theoretical foundations, analysis of tools, and practical implementation of the optimized methodology.

The first section analyzes the principles of building test frameworks and compares existing tools. The second section reveals the technical capabilities of Playwright, including WebSocket support and context isolation. The third section describes the developed architecture based on Python and Pytest, the implementation of a hybrid approach (API+UI), and the calculation of economic efficiency, which confirms the feasibility of implementing the proposed solutions.

The research results have practical value for QA teams, allowing to reduce infrastructure and test maintenance costs while increasing their stability.

Keywords: test automation, Playwright, Python, Pytest, Page Object Model, hybrid testing, CI/CD, optimization.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК 6
ВСТУП 8
РОЗДІЛ 1. ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ОСНОВИ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ВЕБДОДАТКІВ 12
1.1 Мета і завдання автоматизації тестування в розробці програмного забезпечення 12
1.2 Принципи, рівні та типи автоматизованого тестування вебдодатків 19
1.3 Архітектурні патерни у побудові фреймворків автоматизації тестування 23
1.4 Огляд сучасних інструментів автоматизації вебтестування 31
Висновки до розділу 1 36
РОЗДІЛ 2. ФРЕЙМВОРК PLAYWRIGHT ЯК ТЕХНІЧНЕ РІШЕННЯ ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ 38
2.1 Архітектура та принципи роботи фреймворку Playwright 38
2.2 Можливості Playwright у кросбраузерному, мобільному та інтеграційному тестуванні 43
2.3 Інтеграція Playwright із сторонніми сервісами 46
2.4 Розробка структури тестових сценаріїв і використання мов програмування у Playwright 50
Висновки до розділу 2 53
РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ОПТИМІЗОВАНОГО ПРОЦЕСУ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ВЕБДОДАТКІВ НА ОСНОВІ PLAYWRIGHT 54
3.1 Розробка архітектури тестового фреймворку на базі Playwright 54
3.2 Реалізація тестових сценаріїв для вебдодатку 55
3.3 Впровадження архітектурних патернів для підвищення масштабованості та зручності підтримки 60
3.4 Оцінка економічної ефективності оптимізації процесу автоматизації тестування вебдодатків на основі фреймворку Playwright 63
Висновки до розділу 3 67
ВИСНОВКИ 69
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 71
ДОДАТКИ 76

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	Application Programming Interface	Програмний інтерфейс прикладного програмування
AWS	Amazon Web Services	Хмарна платформа Amazon Web Services
AWS EC2	Amazon Elastic Compute Cloud	Хмарний сервіс обчислень Amazon EC2
BrowserContext	Browser Context (Playwright concept)	Контекст браузера (ізольований контекст у Playwright)
CI/CD	Continuous Integration / Continuous Delivery	Безперервна інтеграція / безперервне доставлення
Chromium	Chromium (browser engine)	Chromium (рушій браузера)
conftest.py	Pytest configuration/fixtures file	Файл конфігурації/фікстур Pytest (conftest.py)
Cypress	Cypress (testing framework)	Cypress (фреймворк для тестування)
Docker	Docker Container Platform	Платформа контейнеризації Docker
DOM	Document Object Model	Об’єктна модель документа
E2E	End-to-End	Наскрізне тестування (End-to-End)
Firefox	Firefox (browser engine)	Firefox (рушій браузера)
GitHub Actions	GitHub Actions (CI/CD)	Сервіс CI/CD GitHub Actions
GitLab CI	GitLab Continuous Integration	Система безперервної інтеграції GitLab CI
GUI	Graphical User Interface	Графічний користувацький інтерфейс
Headless	Headless mode (browser)	Безголовий режим браузера (без UI)
Jenkins	Jenkins Automation Server	Сервер автоматизації Jenkins
Jest	Jest (test runner)	Jest (тестовий ранер)
JSON	JavaScript Object Notation	Формат обміну даними (JSON)
Mocha	Mocha (test runner)	Mocha (тестовий ранер)
Mocking	Mocking (test technique)	Імітація/мокування компонентів в тестах

Playwright	Playwright (testing framework)	Playwright – фреймворк автоматизації вебтестів
Postman	Postman (API testing tool)	Postman – інструмент для тестування API
Puppeteer	Puppeteer (automation library)	Puppeteer – бібліотека автоматизації браузера
Pytest	Pytest (Python testing framework)	Pytest – фреймворк тестування на Python
POM	Page Object Model	Модель об’єкта сторінки (POM)
DDT	Data-Driven Testing	Тестування, кероване даними (DDT)
RestAssured	RestAssured (API test library)	RestAssured – бібліотека для тестування API
ROI	Return on Investment	Повернення інвестицій
Selenium	Selenium (WebDriver)	Selenium – інструмент автоматизації (WebDriver)
SPA	Single Page Application	Односторінковий вебдодаток (SPA)
TestCafe	TestCafe (testing tool)	TestCafe – інструмент для тестування
Trace Viewer	Trace Viewer (Playwright tracing tool)	Trace Viewer – інструмент трасування Playwright
WebKit	WebKit (browser engine)	WebKit (рушій браузера)
WebSocket	WebSocket (protocol)	Протокол WebSocket
Unit-test(s)	Unit test(s)	Юніт-тести (модульні тести)

ВСТУП

Актуальність теми даної роботи визначається тенденціями у розробці програмного забезпечення. Сучасні вебдодатки характеризуються високою складністю, динамічністю функціоналу та необхідністю роботи у різних браузерних середовищах, а застосування гнучких методологій розробки Agile та DevOps вимагає частого випуску оновлень (релізів), що скорочує час на перевірку якості. У таких умовах ручне тестування може стати менш ефективним, оскільки воно не здатне забезпечити необхідну швидкість, повноту покриття та регулярність регресійних перевірок. Автоматизація тестування є необхідною складовою успішного CI/CD процесу, і безпосередньо впливає на конкурентоспроможність бізнесу та якість кінцевого продукту.

Для вирішення цих завдань застосовують сучасні інструменти, зокрема фреймворк Playwright. Цей інструмент позиціонує себе як рішення для кросбраузерної автоматизації, яке забезпечує швидкості виконання, підвищеній стабільності тестів та вбудованій підтримці основних браузерних рушіїв (Chromium, Firefox, WebKit). Однак, ефективне впровадження Playwright у великих, проєктах потребує не лише його базового використання, але й глибокої архітектурної оптимізації – створення інженерних рішень та дизайн-патернів, які перетворюють базові можливості Playwright на повноцінний, ефективний тестовий фреймворк, здатний працювати з тисячами тестів у великих, динамічних проєктах, використовуючи оптимальні практики для масштабування тестового покриття, забезпечення ефективної паралелізації та зниження витрат на підтримку тестової бази.

Відтак, дослідження способів оптимізації автоматизації тестування за допомогою фреймворку Playwright є актуальним.

Воно має на меті систематизувати знання та розробити конкретні методики, які дозволять IT-компаніям максимально реалізувати потенціал цього інструменту, підвищити продуктивність QA-інженерів та забезпечити стабільну якість вебдодатків при частих оновленнях.

Зв'язок роботи з науковими програмами, планами, темами. Магістерська кваліфікаційна робота відповідає дослідженням в межах науково-дослідної ініціативної тематики «Організаційно-методологічні аспекти впровадження інформаційно-комунікаційних систем і технологій в управлінні діяльністю сучасних організацій та підприємств за умов переходу до цифрової економіки» (ДРН 0123U105060, 2023-2028 рр.), що реалізується на кафедрі інформаційних систем та технологій, тематиці досліджень навчально-дослідної лабораторії інтелектуальних систем, комп'ютерних мереж інтернет речей кафедри інформаційних систем та технологій Полтавського державного аграрного університету.

Метою роботи є обґрунтування та розробка методики архітектурної оптимізації фреймворку автоматизації тестування на базі Playwright, а також експериментальне визначення кількісного впливу цієї оптимізації на головні метрики якості процесу тестування (швидкість, стабільність та масштабованість).

Завдання роботи:

- проаналізувати теоретичні основи процесу автоматизації тестування вебдодатків, визначити його роль у забезпеченні якості програмного забезпечення та сучасні архітектурні патерни для створення тестових фреймворків;

- провести порівняльний аналіз сучасних інструментів автоматизації тестування з метою обґрунтування вибору Playwright як оптимального засобу для масштабованого кросбраузерного тестування;

- розробити та впровадити архітектуру тестового фреймворку на основі Playwright, застосовуючи дизайн-патерни та методики паралелізації для забезпечення високої швидкості та зниження витрат на підтримку;

- провести експериментальну оцінку впровадженого оптимізованого рішення, порівнявши його ефективність (за метриками швидкості виконання та стабільності тестів) з базовим підходом, та сформулювати практичні рекомендації щодо подальшого вдосконалення процесу тестування.

Об'єкт дослідження: процес автоматизації тестування вебдодатків.

Предмет дослідження: методи та засоби оптимізації процесу автоматизації тестування з використанням фреймворку Playwright.

Методи дослідження:

- огляд літератури, наукових статей;

- аналіз існуючих фреймворків для автоматизації тестування

- дослідження можливостей фреймворку Playwright, його архітектури, інструментів та особливостей реалізації тестів.

- практична реалізація автоматизованих тестів для вебдодатку з використанням Playwright, аналіз ефективності, стабільності та зручності роботи створеного рішення

Інформаційна база дослідження. Науково-технічна література, навчальні посібники та публікації, присвячені автоматизації тестування вебдодатків і забезпеченню якості програмного забезпечення; наукові статті та матеріали з баз даних Google Scholar, IEEE Xplore, SpringerLink, ScienceDirect, Scopus, ResearchGate, що стосуються сучасних підходів до тестування, CI/CD та DevOps-практик; офіційна документація фреймворку Playwright, а також суміжних інструментів автоматизації (Selenium, Cypress, Puppeteer); вітчизняні та зарубіжні стандарти, методології та рекомендації з розробки та тестування програмного забезпечення; матеріали з відкритих інтернет-джерел, блогів і ресурсів експертів у сфері QA, тест-автоматизації та оптимізації процесів розробки.

Елементи наукової новизни. У роботі проведено порівняльний аналіз сучасних фреймворків для автоматизації тестування вебдодатків (зокрема Playwright, Selenium, Cypress, Puppeteer), що дозволило визначити їх сильні та слабкі сторони у контексті ефективності, зручності та стабільності роботи.

Розроблено оптимізований підхід до побудови процесу автоматизації тестування із використанням фреймворку Playwright, який передбачає ефективну структуру тестів, паралельне виконання та інтеграцію з CI/CD.

Практична реалізація запропонованого рішення дала змогу оцінити вплив оптимізації на швидкість, надійність і підтримуваність тестів, що підвищує загальну ефективність тестування у процесі розробки вебдодатків.

Практична значущість: Результати роботи надають конкретні рекомендації щодо організації та оптимізації процесу автоматизації тестування вебдодатків, зокрема щодо побудови структури тестів, їх паралельного виконання та інтеграції з CI/CD-процесами.

Розроблений приклад оптимізованого процесу автоматизації за допомогою Playwright демонструє практичне застосування інструменту для підвищення швидкості та надійності тестування вебдодатків.

Отримані результати можуть бути використані QA-командами для покращення ефективності тестування, зменшення часу на підтримку тестів та підвищення стабільності продукту при частих оновленнях вебзастосунків.

Апробація результатів дослідження. За результатами проведеного дослідження опубліковано тези доповідей: «Архітектурна оптимізація процесу автоматизації тестування вебдодатків». Матер. наук.-практ. конф. за підсумками проходження виробничих практик здобувачів вищої освіти спеціальності 126 Інформаційні системи та технології, кафедра інформаційних систем та технологій Полтавського державного аграрного університету, 22 жовтня 2025 р. Вип. XI.; «Архітектура та принципи роботи фреймворку Playwright». Матер. XXII щорічного міждисциплінарного семінару «Студентські роботи за науковою тематикою кафедри інформаційних систем та технологій ННІ ЕУП та ІТ ПДАУ», 25 листопада 2025 року, м. Полтава; Архітектурні патерни у побудові фреймворків автоматизації тестування вебдодатків. Innovative Approaches in Modern Science and Technology: Collection of Scientific Papers with Proceedings of the 3rd International Scientific and Practical Conference. International Scientific Unity. November 12-14, 2025. Lisbon, Portugal.

Структура та обсяг кваліфікаційної роботи. Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. Основний текст роботи викладений на 69 сторінках, містить 15 рисунків і 10 таблиць. Список використаних джерел налічує 46 найменувань.

РОЗДІЛ 1
ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ОСНОВИ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ВЕБДОДАТКІВ

1.1 Мета і завдання автоматизації тестування в розробці програмного забезпечення

В умовах стрімкого розвитку інформаційних технологій та ускладнення архітектури програмних систем, забезпечення якості стає одним з головних приоритетів у життєвому циклі розробки програмного забезпечення. У цьому контексті автоматизація тестування виступає як методологія, що передбачає використання спеціалізованих програмних засобів для виконання тестових сценаріїв з метою верифікації та валідації функціональних і нефункціональних вимог до продукту.

Автоматизація тестування полягає у створенні та використанні скриптів, які імітують взаємодію користувача з додатком або виконують запити до API для порівняння фактичного результату роботи системи з очікуваним результатом. На відміну від ручного тестування, яке покладається на людське втручання, автоматизований підхід забезпечує високий рівень повторюваності, точності та масштабованості тестових процедур за допомогою скриптів [1].

Головна мета автоматизації тестування полягає не у повній заміні ручних методів, скільки в оптимізації процесу забезпечення якості шляхом його інтеграції в методологію безперервної інтеграції та безперервної доставки, основна ціль якої – створення системи швидкого зворотного зв’язку для команди розробки, що дозволяє ідентифікувати дефекти, які виникають внаслідок модифікації коду. Таким чином, автоматизація тестування є інструментом мінімізації ризиків, пов’язаних із впровадженням нового функціоналу, і сприяє підвищенню загальної стабільності та надійності програмного продукту на кожному етапі його еволюції.

Досягнення цієї мети розподіляється на низку конкретних завдань, кожне з яких спрямоване на вирішення певного аспекту контролю якості програмного

забезпечення. Ці завдання формують основу комплексної стратегії автоматизованого тестування, що представлено у таблиці 1.1.

Таблиця 1.1 – Класифікація завдань автоматизованого тестування [2]

Напрямок тестування	Функціональне призначення та завдання в рамках стратегії
Димове тестування (Smoke Testing)	Проведення поверхневої перевірки базової працездатності основних функціональних блоків системи після розгортання нової збірки. Завданням є швидке прийняття рішення про доцільність подальшого, більш глибокого тестування.
Регресійне тестування (Regression Testing)	Верифікація того, що внесені зміни в одному модулі системи не спричинили порушення працездатності існуючого функціоналу в інших модулях. Головне завдання якого забезпечити стабільність продукту в процесі ітеративної розробки.
Кросбраузерне та кросплатформне тестування	Забезпечення консистентної поведінки та коректного візуального представлення вебдодатку в різних браузерних рушіях та операційних системах.
Тестування програмних інтерфейсів (API Testing)	Валідація бізнес-логіки на рівні сервісів, минаючи графічний інтерфейс користувача. Це дозволяє ізолювати та тестувати логіку застосунку, забезпечуючи вищу швидкість та стабільність тестів.
Тестування продуктивності (Performance Testing)	Оцінка поведінки системи під певним навантаженням (наприклад, одночасна кількість користувачів, обсяг даних). Завдання включають визначення часу відгуку, пропускної здатності та стабільності системи.

Для ефективного впровадження автоматизації тестування необхідно диференціювати її від ручного тестування. Вибір того чи іншого підходу є стратегічним і залежить від цілей тестування, стадії життєвого циклу продукту, економічної доцільності тощо. Критерії для порівняння видів представлені у таблиці 1.2.

Таблиця 1.2 – Порівняння видів тестування

Критерій	Ручне тестування (Manual Testing)	Автоматизоване тестування (Automated Testing)
Основна мета	Виявлення непередбачуваних дефектів, перевірка зручності та користувацького досвіду.	Швидке та багаторазове виконання визначених сценаріїв, перевірка регресій.
Головна перевага	Гнучкість, людська інтуїція, суб’єктивна оцінка якості.	Швидкість, точність, можливість паралельного виконання, ефективність у повторюваних завданнях.
Початкові витрати	Низькі. Потрібен лише кваліфікований тестувальник.	Високі. Потрібні інструменти, фреймворки та інженери з автоматизації.

Продовження таблиці 1.2

Довгострокові витрати / ROI	Високі. Час тестувальника потрібен на кожне повторне виконання. Низький ROI для повторюваних завдань.	Низькі. Високий ROI (повернення інвестицій) з часом, оскільки тестування є швидким і дешевим у повторенні.
Рівень повторюваності	Низький/Складний. Схильність до людських помилок при багаторазовому повторенні.	Високий. Сценарії виконуються ідентично щоразу.
Вимоги до функціоналу	Функціонал може бути нестабільним або мати часті зміни.	Функціонал має бути стабільним та детермінованим для ефективної автоматизації.

Ручне тестування є незамінним у сферах, що вимагають людської розумової гнучкості, інтуїції та суб’єктивної оцінки. До таких сфер належать дослідницьке тестування(exploratory testing), тестування зручності використання (usability testing), перевірка нових, ще не стабілізованих функціональних можливостей, де тестові сценарії формалізувати неможливо. Автоматизоване тестування є максимально ефективним у детермінованих, повторюваних та великомасштабних завданнях [3].

Економічний ефект від впровадження автоматизації розраховується через показник повернення інвестицій, який враховує високі початкові витрати на розробку та підтримку тестових скриптів у порівнянні з довгостроковим скороченням витрат на регресійне тестування. Тому, обґрунтованість застосування автоматизації прямо пропорційна частоті виконання тестів та стабільності функціоналу, що тестується.

Отже, ручний та автоматизований підходи до тестування є не антагоністичними, а двома взаємодоповнюючими інструментами у практиці QA-інженера. Роль автоматизації фундаментально змінилася: від ізольованої фінальної фази у класичній каскадній моделі вона трансформувалася в компонент DevOps і концепції «Shift Left» Testing. Відтак тести інтегровані на ранні етапи розробки та виконуються автоматично після кожної зміни у кодовій базі, забезпечуючи розробникам зворотний зв’язок щодо якості [5].

Рішення про автоматизацію тестового сценарію є стратегічною інвестицією, що вимагає комплексного аналізу технічних та економічних факторів. Кожен

автоматизований тест є активом, цінність якого визначається сукупною економією ресурсів протягом життєвого циклу продукту, що вимагає початкових капіталовкладень у його створення. Найкращими для автоматизації вважаються сценарії, що характеризуються високою частотою виконання, стабільністю бізнес-логіки та критичною важливістю для системи (наприклад, аутентифікація чи платіжні транзакції). Інвестиції в такі процеси окупаються багаторазово завдяки їх постійному включенню в регресійний набір [4]. Надмірна автоматизація може створити хибне відчуття безпеки, оскільки успішне виконання автоматизованих тестів підтверджує лише коректність заздалегідь відомих послідовностей, але не гарантує якості користувацького досвіду або поведінки в непередбачуваних умовах, які вимагають людської оцінки.

Автоматизовані тести також є програмними артефактами зі своїм життєвим циклом, що вимагають постійної підтримки, рефакторингу та адаптації. Ігнорування вартості їх підтримки є поширеною помилкою.

Для об’єктивної оцінки ефективності впровадженої стратегії автоматизації використовуються кількісні та якісні метрики. Вони дозволяють оцінити не лише технічний стан тестового набору, але і його вплив на бізнес-процеси.

До головних метрик належить тестове покриття, яке може вимірюватися як на рівні коду (відсоток кодових рядків, що виконуються тестами), так і на рівні вимог (відсоток бізнес-вимог, що перевіряються). Іншою важливою метрикою є стабільність тестів, що показує відсоток тестів, які дають різний результат без змін у коді продукту. Високий рівень «крихкості» свідчить про проблеми в архітектурі тестів.

Нестабільність не лише збільшує час на аналіз результатів прогонів, але й може підірвати довіру команди до автоматизації загалом. Окрім того, важливим показником є час виконання повного набору тестів, який безпосередньо впливає на швидкість циклу розробки та частоту релізів. Комплексний моніторинг цих індикаторів є необхідною умовою для підтримки життєдіяльності проєкту.

Метрики оцінки ефективності тестування вебдодатків представлені у таблиці 1.3.

Таблиця 1.3 – Метрики оцінки ефективності тестування вебдодатків [6]

Метрика	Тип метрики	Визначення та спосіб вимірювання	Вплив на бізнес та технічний стан
Тестове покриття (Test Coverage)	Кількісна	1. Покриття коду (Code Coverage) – відсоток рядків коду, функцій або гілок, що виконуються автоматизованими тестами. 2. Покриття вимог (Requirements Coverage) – відсоток бізнес-вимог, на які розроблено автоматизовані тести.	Технічний стан: визначає, наскільки повно тестовий набір охоплює продукт. Високе покриття знижує ризик пропуску дефектів.
Стабільність тестів (Test Stability / Flakiness)	Якісна / кількісна	Формула: (1 – К-ть нестабільних прогонів/ Загальна к-ть прогонів) * 100%. Нестабільний тест («flaky test») – це тест, який дає різний результат (успіх/провал) без змін у коді продукту.	Архітектура тестів: висока нестабільність (flakiness) вказує на «крихкість» тестової архітектури, проблеми з синхронізацією або погане управління даними/середовищем. Знижує довіру до автоматизації.
Час виконання тестового набору (Test Suite Execution Time)	Кількісна	Загальний час, необхідний для виконання всього набору автоматизованих регресійних тестів.	Швидкість розробки (CI/CD): визначає, наскільки швидко команда може отримувати зворотний зв’язок. Швидкий прогін критично важливий для ефективного конвеєра безперервної інтеграції та розгортання (CI/CD).
Коефіцієнт пропущених дефектів (Defect Escape Ratio)	Якісна / кількісна	Формула: К-ть дефектів, знайдених клієнтами / Загальна к-ть дефектів * 100%. Кількість помилок, що «просочилися» у продакшн, оминаючи тестування.	Якість продукту та задоволеність користувачів – головний показник успішності автоматизації. Ефективний тестовий набір мінімізує цей показник, прямо впливаючи на репутацію компанії та бізнес-результати.

Головним показником успішності тестування є коефіцієнт пропущених дефекті, що показує кількість помилок, знайдених кінцевими користувачами, а не на етапі тестування. Ефективна система автоматизації повинна мінімізувати цей показник, тим самим безпосередньо впливаючи на якість кінцевого продукту та задоволеність користувачів.

Цінність тестового набору визначається не кількістю тестів, а їх якістю та релевантністю. В інженерній практиці сформульовано низку фундаментальних атрибутів якісного автоматизованого тесту, які є запорукою стабільності та надійності всього процесу.

По-перше, тест має бути надійним та повторюваним. Це означає, що за незмінних умов у програмному продукті тест повинен завжди давати однаковий результат.

По-друге, тест повинен бути незалежним. Його виконання не повинно залежати від результатів інших тестів або від порядку їх запуску. Ця властивість є критично важливою для реалізації паралельного виконання тестів, що є ключовим фактором у досягненні високої швидкості зворотного зв’язку.

По-третє, якісний тест повинен містити чіткі критерії успіху чи невдачі і не вимагати від людини додаткової ручної перевірки чи інтерпретації результатів. Нарешті, тест має бути максимально швидким, оскільки сукупний час виконання всього тестового набору безпосередньо впливає на тривалість CI/CD циклу [7].

Важливо розмежовувати два фундаментальні поняття забезпечення якості: верифікацію та валідацію. Успішний QA-процес вимагає як верифікації (для запобігання помилкам на ранніх стадіях), так і валідації (для підтвердження, що кінцевий продукт є цінним і корисним). У той час як автоматизація підходить для верифікації (перевірки відповідності специфікації), ручне тестування (зокрема, дослідницьке) є інструментом для валідації (перевірки відповідності потребам користувача). Автоматизовані скрипти не призначені для валідації суб'єктивних аспектів користувацького досвіду (UX), таких як зручність інтерфейсу чи логічність навігації. Ці завдання залишаються у сфері ручного дослідницького тестування (exploratory testing) [8].

Стратегія автоматизації тестування не є статичною та універсальною: вона має бути адаптивною та гнучкою, оскільки її ефективність напряму залежить від специфічних характеристик програмного продукту, команди та бізнес-цілей. Спроба застосувати однаковий підхід до автоматизації тестування короткострокового промо-сайту та довгострокового проєкту enterprise-рівня

призведе до неефективного розподілу ресурсів. Відтак, перед початком робіт з автоматизації необхідно провести аналіз факторів, що визначають її обсяг та пріоритетність.

Для систематизації цього аналізу можна виділити кілька основних факторів, які комплексно впливають на прийняття рішення (таблиця 1.4).

Таблиця 1.4 – Фактори, що впливають на стратегію та обсяг автоматизації тестування [9]

Фактор	Сценарій з низьким пріоритетом для автоматизації	Сценарій з високим пріоритетом для автоматизації
Тривалість та життєвий цикл проєкту	Короткострокові проєкти (наприклад, сайти для заходів, промо-сторінки), де початкові інвестиції в автоматизацію не встигнуть окупитися.	Довгострокові продукти, що постійно розвиваються (SaaS-платформи, інтернет-магазини), де регресійне тестування є регулярним та масштабним.
Стабільність вимог та функціоналу	Проєкти на ранній стадії (прототипи), де вимоги та користувацький інтерфейс постійно й значно змінюються.	Зрілі продукти зі стабільним ядром функціоналу, який не зазнає частих архітектурних змін.
Технологічна складність та обсяг даних	Прості, переважно статичні вебсайти з мінімальною бізнес-логікою.	Складні вебдодатки (Single Page Applications) з великою кількістю інтеграцій, API та сценаріїв
Критичність функціоналу для бізнесу	Функціонал з низьким впливом на дохід або репутацію компанії	Критично важливі бізнес-процеси, де помилка може призвести до фінансових або репутаційних втрат (наприклад, платіжні шлюзи, процеси реєстрації).

Парадигма автоматизованого тестування пройшла значний еволюційний шлях. Ця трансформація відображає перехід до «культури якості», в рамках якої відповідальність за стабільність та надійність продукту розподіляється між усіма учасниками процесу розробки. Автоматизація в цьому контексті виступає як інструмент, що надає швидкий та об’єктивний зворотний зв’язок.

Метою автоматизації є не повна заміна ручного тестування, а оптимізація процесу. Вона дозволяє перенести фокус QA-інженерів з виконання повторюваних

регресійних сценаріїв на неформалізовані дослідницькі завдання (exploratory testing), де людська оцінка є ключовою [9].

1.2 Принципи, рівні та типи автоматизованого тестування вебдодатків

В основу сучасної стратегії автоматизації тестування програмного забезпечення покладено декілька основних принципів.

Автоматизація не є самоціллю, а лише засобом досягнення якості. Головний принцип полягає в тому, що автоматизувати слід лише ті процеси, де це приносить реальну цінність. Прагнення до 100% тестового покриття за допомогою автоматизації є поширеною помилкою, оскільки воно не гарантує відсутності дефектів і призводить до створення надлишкових, дорогих у підтримці тестів. Набагато ефективніше зосередитися на критично важливих для бізнесу сценаріях та функціоналі з високим ризиком виникнення помилок.

Автоматизовані тести повинні бути повністю детермінованими та незалежними. Це означає, що тест має давати стабільний результат за однакових умов і не залежати від результатів виконання інших тестів. Цей принцип є важливим для забезпечення стабільності системи тестування та для реалізації паралельного запуску тестів, що є умовою для отримання зворотного зв’язку в CI/CD.

Необхідно враховувати «парадокс пестициду». Ця концепція, запозичена з агрономії, стверджує, що якщо постійно використовувати один і той самий набір тестів, з часом він втратить свою ефективність у виявленні нових дефектів. Програмний продукт еволюціонує, і тестовий набір повинен еволюціонувати разом з ним. Тому автоматизовані тести потребують регулярного аудиту, рефакторингу та доповнення новими перевірками [10].

Щоб систематизувати цей процес, для структурної організації тестового покриття використовується концептуальна модель піраміди тестування. Вона розподіляє тести на три основні рівні, визначаючи їх ідеальне співвідношення за

кількістю, швидкістю та вартістю підтримки [11]. Таблиця 1.5 узагальнює відомості про модель піраміди автоматизованого тестування.

Таблиця 1.5 – Піраміда автоматизованого тестування

Характеристика	Unit-тести (основа піраміди)	Інтеграційні тести (середина піраміди)	Наскрізні/E2E-тести (вершина піраміди)
Ціль	Локалізувати та виявити помилки в логіці коду.	Перевірити коректність обміну даними та стиків модулів.	Підтвердити, що критичні бізнес-сценарії працюють.
Область перевірки	Ізольовані частини коду (функції, методи, класи).	Взаємодія між компонентами (API, база даних).	Повний шлях користувача через графічний інтерфейс (GUI).
Кількість тестів	Найвища (більшість)	Середня	Найнижча (мінімальна)
Хто пише	Розробники (Developers).	Розробники та QA-інженери.	QA-інженери-автоматизатори.
Швидкість виконання	Найшвидша (мілісекунди).	Середня (секунди).	Найповільніша (десятки секунд / хвилини).
Вартість підтримки	Найнижча (висока стабільність).	Середня.	Найвища (висока «крихкість»).
Типові інструменти	Mocking, фреймворки (Pytest, JUnit).	Інструменти для API-тестування (Postman, RestAssured).	Фреймворки для GUI (Playwright, Selenium, Cypress).

Основою піраміди тестування є unit-тести – найнижчий і найширший рівень. Юніт-тести перевіряють найменші ізольовані частини коду (функції, методи, класи) в ізоляції від решти системи.Вони створюються розробниками, та дозволяють локалізувати помилку в логіці. Саме вони повинні складати більшість тестового набору.

Середній шар піраміди тестування – інтеграційні тести. На цьому рівні перевіряється взаємодія між декількома компонентами системи.

Для вебдодатків це найчастіше тестування API, яке валідує коректність обміну даними між клієнтською частиною, сервером та базою даних. Ці тести дозволяють виявити дефекти на стику різних модулів, які неможливо знайти на нижчому рівні.

Вершина піраміди тестування – наскрізні (End-to-End) тести. Цей рівень є найближчим до реального користувача.

Тести на цьому рівні імітують повні користувацькі сценарії, взаємодіючи з додатком через його графічний інтерфейс (GUI).

Порушення цих пропорцій призводить до формування антипатерну, де більшість тестів є повільними GUI-тестами, що визначає всю систему тестування неефективною та ненадійною.

Окрім рівнів, автоматизовані тести класифікуються за функціональним призначенням на дві великі категорії: функціональні та нефункціональні.

Функціональне тестування, відповідає на питання: «Чи робить система те, що вона повинна робити?». Його мета – верифікація бізнес-логіки додатку на відповідність функціональним вимогам, що закладені в технічних специфікаціях. Функціональні тести імітують дії користувача для перевірки функцій. Вони фокусуються на вхідних даних та очікуваних вихідних результатах, перевіряючи коректність поведінки системи. Цей тип тестування найчастіше є основним вибором для реалізації автоматизації тестування.

На противагу цьому, нефункціональне тестування відповідає на питання: «Наскільки добре система це робить?». Даний тип тестування досліджує атрибути якості та експлуатаційні характеристики системи. До цієї категорії тестів належить перевірка того, наскільки швидко завантажується сторінка (продуктивність), чи здатна система витримати наплив великої кількості користувачів (тестування навантаження), наскільки вона захищена від зовнішніх загроз (тестування безпеки), та чи є вона зручною для людей з обмеженими можливостями (тестування доступності) [11].

Ігнорування будь-якої з цих категорій призводить до формування неповного уявлення про якість продукту та підвищує ризики виходу продукту в реліз із критичними дефектами. Це дозволить мінімізувати післярелізні дефекти та підвищити рівень задоволеності кінцевих користувачів якістю програмного продукту.

Зважаючи на те, що кожен із цих глобальних класів охоплює широкий спектр специфічних перевірок, які застосовуються на різних етапах життєвого циклу розробки, важливо чітко розмежовувати їх за цільовим призначенням (таблиця 1.6).

Таблиця 1.6 – Основні типи автоматизованого тестування вебдодатків [12]

Категорія	Тип тестування	Призначення та роль в автоматизації
Функціональне тестування	Димове тестування (Smoke Testing)	Швидка перевірка базової працездатності основних функцій після розгортання нової версії. Зазвичай це перший етап автоматичного тестування в CI/CD.
	Регресійне тестування (Regression Testing)	Масштабна перевірка того, що нові зміни не порушили існуючий функціонал. Це основна область застосування автоматизації.
	Кросбраузерне тестування (Cross-browser Testing)	Забезпечення коректної роботи та візуального відображення додатку в різних веббраузерах (Chrome, Firefox, Safari) та на різних платформах.
Нефункціональне тестування	Тестування продуктивності (Performance Testing)	Оцінка поведінки системи під навантаженням. Автоматизовані скрипти симулюють дії тисяч користувачів для вимірювання часу відгуку та стабільності.

Зазначена класифікація типів тестування є універсальною, однак вебдодатки мають специфічні технічні риси, що породжують низку унікальних викликів для стратегії автоматизації [12].

Ключовою проблемою є динамічна природа DOM (Document Object Model). У сучасних односторінкових додатках (SPA), побудованих на фреймворках (React, Angular, Vue), оновлення компонентів інтерфейсу відбувається без перезавантаження сторінки . Це призводить до високої динамічності стану DOM-дерева, роблячи статичні селектори ненадійними. Тестові інструменти повинні містити механізми для очікування появи елементів та їх готовності до взаємодії (actionability).

Другим викликом є асинхронність операцій. Завантаження даних з API, фонові процеси та анімації виконуються асинхронно. Лінійний підхід до виконання тестових скриптів призводить до збоїв (race conditions), коли тест намагається взаємодіяти з елементом до його повного завантаження чи рендерингу . Тому надійні фреймворки вебтестування повинні мати вбудовані механізми інтелектуального очікування (smart waits) для автоматичної синхронізації тестів з життєвим циклом додатка.

Третьою проблемою є фрагментація браузерного середовища. Вебдодатки мають коректно функціонувати в різних комбінаціях браузерів та операційних

систем, а кожен браузерний рушій (Chromium, WebKit, Gecko) має відмінності у рендерингу . Забезпечення кросбраузерної сумісності є ключовим завданням, яке вирішується переважно засобами автоматизації.

Четвертим аспектом є адаптивний дизайн (Responsive Design). Додатки повинні коректно відображатися та функціонувати на пристроях з різними розмірами екрану (viewports), що вимагає тестування верстки та функціоналу в різних конфігураціях, які ефективно симулюються за допомогою інструментів автоматизації.

Таким чином, ефективне автоматизоване тестування вебдодатків вимагає застосування спеціалізованих інструментів та підходів, здатних вирішувати вищезазначені проблеми динамічності та гетерогенності середовища [12].

1.3 Архітектурні патерни у побудові фреймворків автоматизації тестування

Однією з проблем автоматизації тестування є висока вартість підтримки тестів. Без чіткої та продуманої архітектури тестовий проєкт швидко втрачає структуру, перетворюючись на набір тестів, що є складними для підтримки. Для вирішення цієї проблеми в інженерній практиці використовуються архітектурні патерни.

Патерни в автоматизації тестування визначають структуру та взаємозв’язки між компонентами: вони задають структуру, визначають взаємозв’язки між компонентами та забезпечують довгострокову стійкість тестових наборів. Їх головна мета підвищити читабельність (readability), легкість підтримки (maintainability) та повторного коду (reusability).

Найбільш поширеним та фундаментальним патерном у тестуванні вебдодатків є об’єктна модель сторінки (Page Object Model, POM). Його головна ідея полягає у розділенні відповідальностей: логіка самого тесту (що перевіряється)

має бути повністю відокремлена від технічної реалізації взаємодії зі сторінкою (як це робиться) [13-15].

Згідно з цим патерном, кожна вебсторінка або значущий UI-компонент (наприклад, форма логіну, хедер сайту) представляється у вигляді окремого класу в програмному коді «об'єкта сторінки». Цей клас інкапсулює в собі дві ключові речі: елементи сторінки – локатори (селектори) для всіх інтерактивних елементів, що знаходяться на цій сторінці (кнопки, поля вводу, посилання); методи для взаємодії – публічні методи, що імітують дії користувача з цими елементами (наприклад, login(username, password), clickSearchButton(), getProductTitle()) (рисунок 1.1).

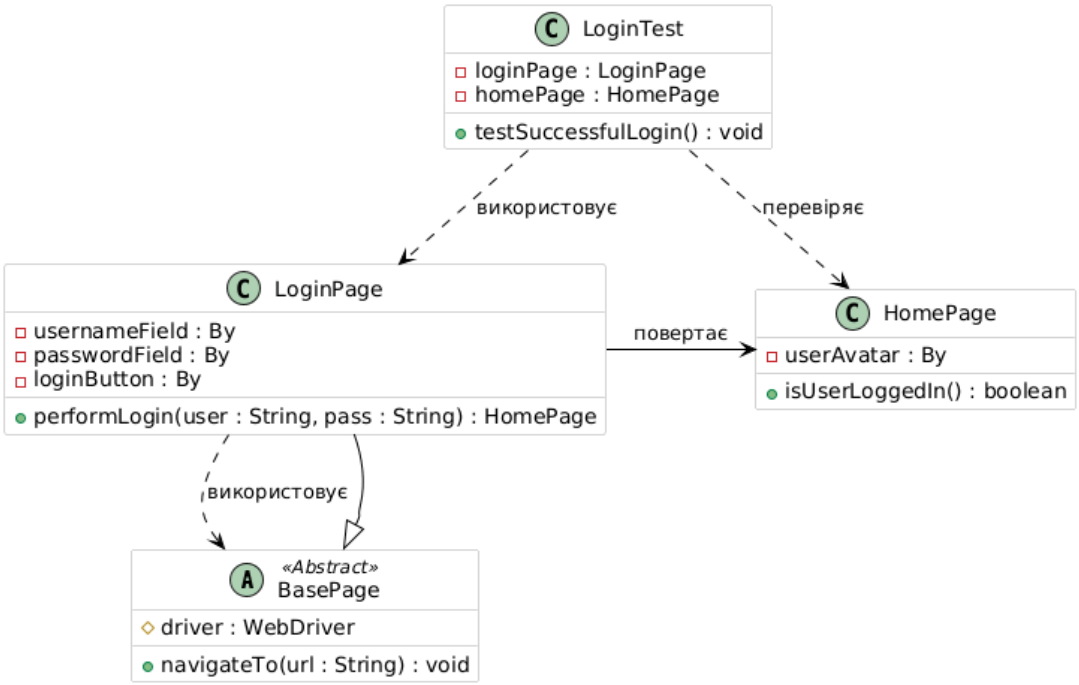


Рисунок 1.1 – Принцип роботи POM

Таким чином, завдяки POM, сам тест-кейс оперує не технічними деталями, подібними до driver.findElement(By.cssSelector(«#login-btn»)), а викликає високорі́вневі, зрозумілі методи об'єкта сторінки: loginPage.performLogin(«user», «pass»). Головна перевага такого підходу: якщо в майбутньому розробники змінять, наприклад, селектор кнопки входу, то зміни потрібно буде внести лише у

відповідному класі Page Object, а не в десятках тестів, які використовують цю кнопку. Це робить тестовий набір надзвичайно стійким до змін в інтерфейсі [16].

В основі Screenplay покладено декілька головних абстракцій. Центральною фігурою тут є Actor (дійова особа), що представляє користувача, наділеного певними Abilities (здібностями), наприклад, можливістю взаємодіяти з веббраузером. Дійова особа виконує високорівневі Tasks (завдання), що складаються з кількох логічних кроків, наприклад: «Авторизуватися в системі». Кожне таке завдання, у свою чергу, декомпозується на послідовність низькорівневих, атомарних Interactions (взаємодій), таких як «клікнути на кнопку» або «ввести текст» (рисунок 1.2).

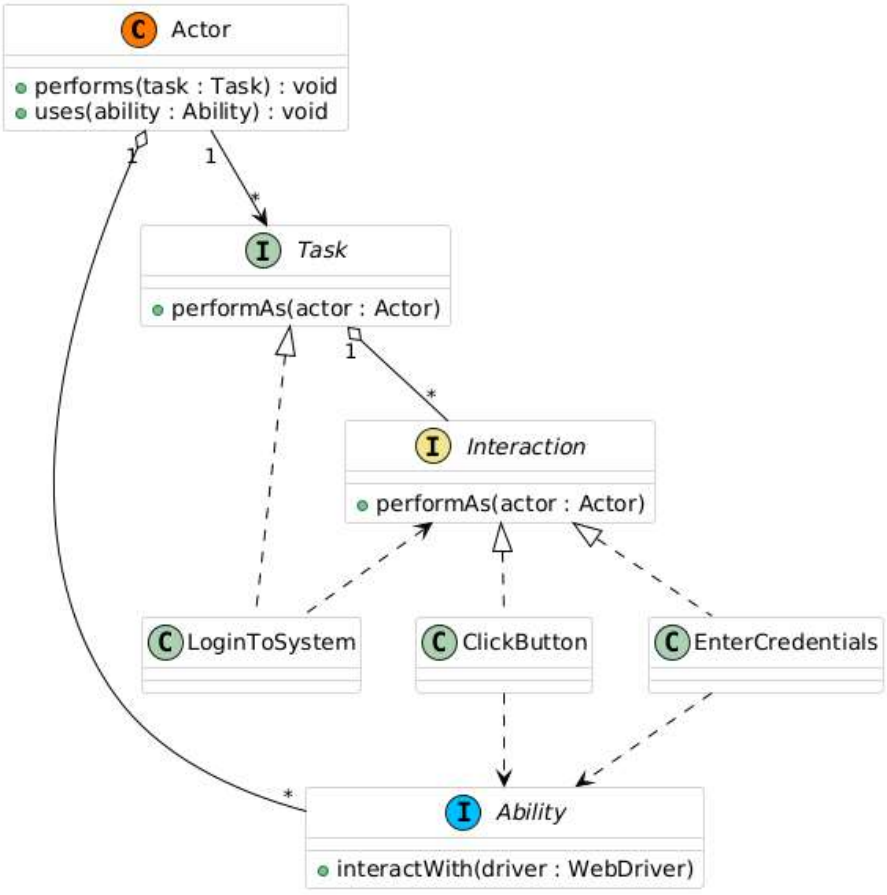


Рисунок 1.2 – Принцип роботи Screenplay Pattern

Окрім очевидної переваги у підтримці, використання POM, покращує ще два важливі аспекти – читабельність та повторне використання коду. Завдяки цьому тестові сценарії стають лаконічними, оскільки вони те, що має відбуватись, на

бізнес рівні, а не так, як це реалізовано у DOM-елементах. І це робить логіку процесу більш зрозумілою не лише для QA, а й іншим членам команди розробки. Водночас, методи, що використовуються у програмі тестування, перетворюються на багаторазові блоки, які можна комбінувати у різних тест-кейсах, що значною мірою зменшує дублювання коду та прискорює написання нових тестів.

Screenplay Pattern є більш сучасною та просунутою еволюцією ідеї POM. Якщо POM фокусується на структурі вебсторінок, то Screenplay концентрується на намірах та ролях користувача. Цей патерн прагне зробити тести ще більш читабельними та масштабованими, організовуючи їх навколо дійових осіб [16].

У побудові фреймворків, окрім специфічних, широко застосовують класичні патерни Factory та Singleton. Патерн Factory (рисунок 1.3), що приховує логіку створення об’єктів, в автоматизації зазвичай використовують для ініціалізації вебдрайвера. Спеціальний клас WebDriverFactory самостійно обирає драйвер відповідно до конфігурації, дозволяючи легко змінювати браузери без редагування коду тестів [17].

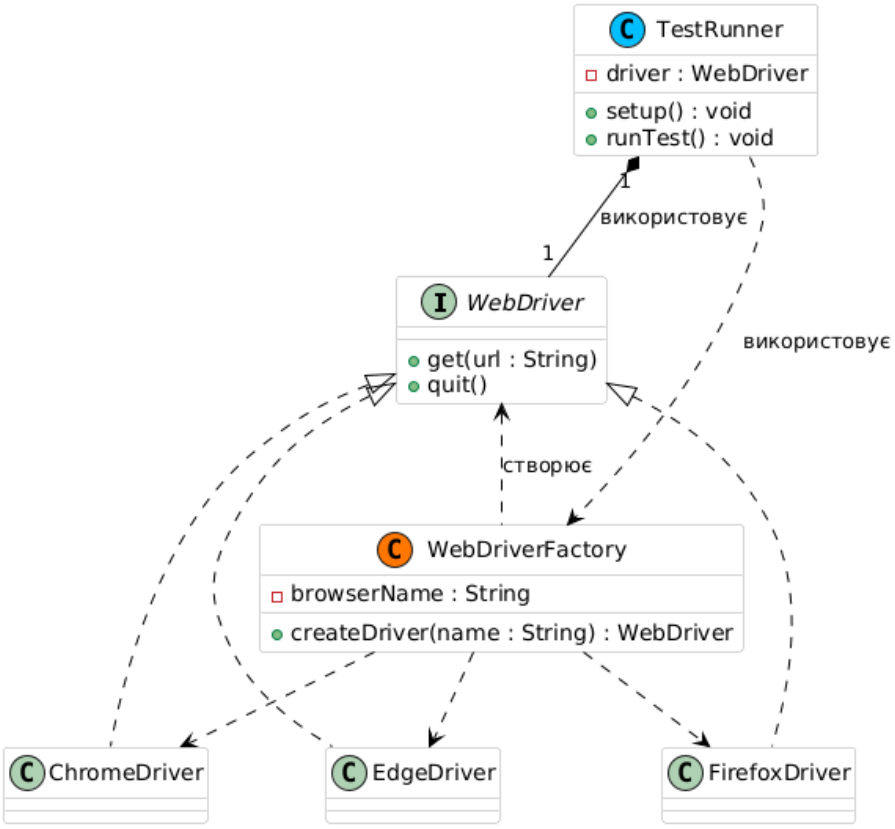


Рисунок 1.3 – Принцип роботи паттерну Factory

Singleton це патерн, що гарантує існування лише одного екземпляра певного класу та надає глобальну точку доступу до нього (рисунок 1.4). У фреймворках автоматизації він часто застосовується для керування екземпляром драйвера, щоб уникнути створення нового вікна браузера для кожного тесту, що значно економить ресурси. Також його використовують для класів, що читають конфігураційні файли, дані з них потрібно завантажити лише один раз на початку виконання всіх тестів [18].

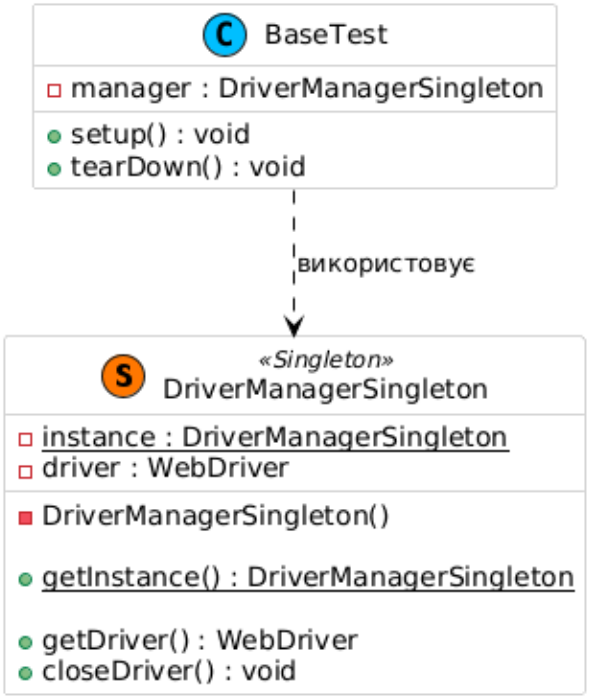


Рисунок 1.4 – Принцип роботи паттерну Singleton

Поєднання цих архітектурних патернів дозволяє створювати потужні, гнучкі та легкі у підтримці фреймворки, які служать надійною основою для всього процесу автоматизованого тестування.

Хоча розглянуті патерни, такі як POM та Screenplay, визначають фундаментальну структуру та спосіб взаємодії з додатком, ефективний тестовий фреймворк повинен вирішувати ще два важливих завдання: керування тестовими даними та забезпечення максимальної прозорості й читабельності тестів для всіх учасників проєкту. Для вирішення цих завдань застосовуються додаткові архітектурні підходи.

Одним із найбільш потужних підходів до організації тестування є DDT – Data-Driven Testing (тестування, кероване даними). Його головна ідея полягає в повному відокремленні логіки тестового сценарію від тестових даних, що використовуються в цьому сценарії. Замість того, щоб жорстко «зашивати» вхідні значення (логіни, паролі, пошукові запити) безпосередньо в код тесту, ці дані виносяться у зовнішні джерела [19].

Як приклад, розглянемо тест для перевірки форми входу (рисунок 1.5). Без використання DDT, щоб перевірити 10 різних комбінацій логіна та пароля (валідних, невалідних, порожніх), довелося б написати 10 практично ідентичних тестів або функцій. Це призводить до масивного дублювання коду та ускладнює підтримку.

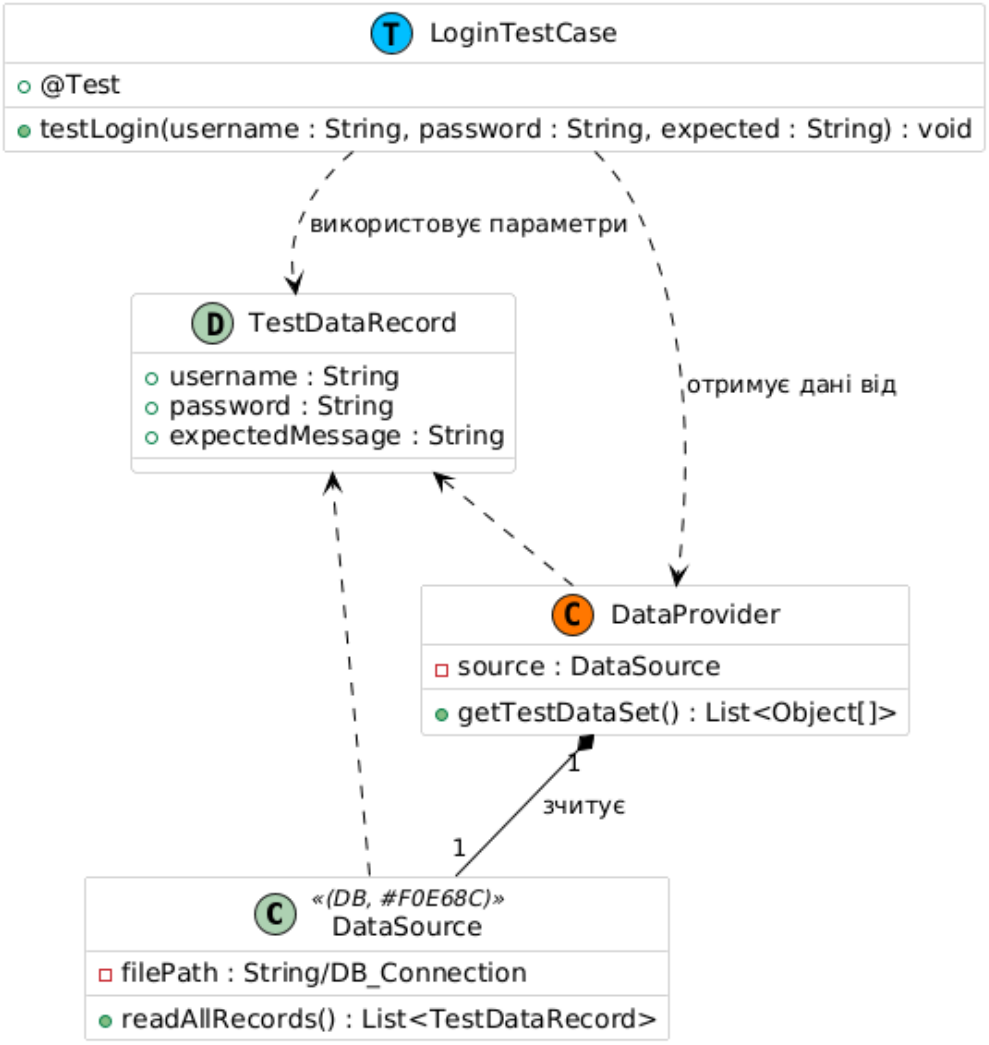


Рисунок 1.5 – Принцип роботи паттерну Data-Driven Testing

Паттерн DDT вирішує цю проблему. Створюється єдиний тестовий скрипт, який є своєрідним шаблоном. А всі набори даних зберігаються окремо, наприклад, у файлах формату CSV, Excel, JSON, YAML або навіть у базі даних. Фреймворк автоматично зчитує цей файл рядок за рядком і виконує один і той самий тест для кожного набору даних, передаючи їх у якості параметрів.

Ще одним важливим підходом, що удосконалює архітектуру тестування, є Behavior-Driven Development (BDD) (рисунок 6). Це не просто патерн, а ціла методологія, що виросла з ідей Test-Driven Development (TDD), і має за мету налагодити ефективну комунікацію між бізнесом та командою розробки [19].

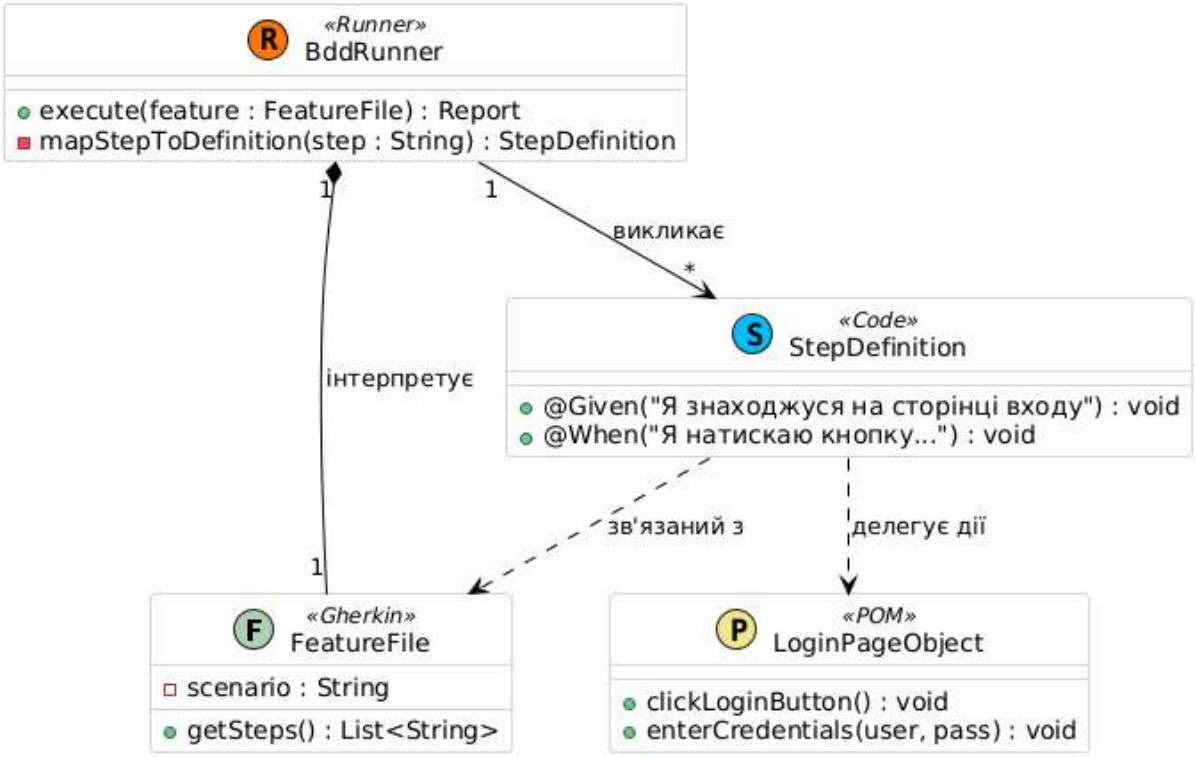


Рисунок 1.6 – Концептуальна структура паттерну Behavior-Driven Development

В основі BDD лежить концепція створення єдиної, спільної мови для опису поведінки системи, яка була б зрозумілою для всіх учасників процесу. Для цього використовується спеціальний синтаксис, наприклад Gherkin, який дозволяє описувати тестові сценарії у форматі, наближеному до природної мови. Кожен сценарій структурується за допомогою головних слів, що описують передумови (Given), дії користувача (When) та очікувані результати (Then). Такий текстовий

опис поведінки системи стає «живою документацією» єдиним джерелом правди, яке одночасно є і специфікацією, і виконуваним тестом. З технічної точки зору, кожен крок тестового сценарію пов’язується з конкретним методом у програмному коді, який реалізує дану дію. Таким чином, BDD виступає як додатковий семантичний шар над такими патернами, як POM, що робить автоматизовані тести максимально прозорими та бізнес-орієнтованими [20].

Для підвищення читабельності самого програмного коду на рівні виклику тестових методів часто застосовується патерн Fluent Interface (рисунок 1.7). Його головна ідея полягає в такому проєктуванні методів, що використовуються у коді тестового сценарію, щоб їх можна було викликати послідовним ланцюжком, який візуально та семантично нагадує зв’язне речення. Це досягається за рахунок того, що кожен метод у ланцюжку повертає екземпляр того ж самого об’єкта, що дозволяє одразу ж викликати наступний метод [20].

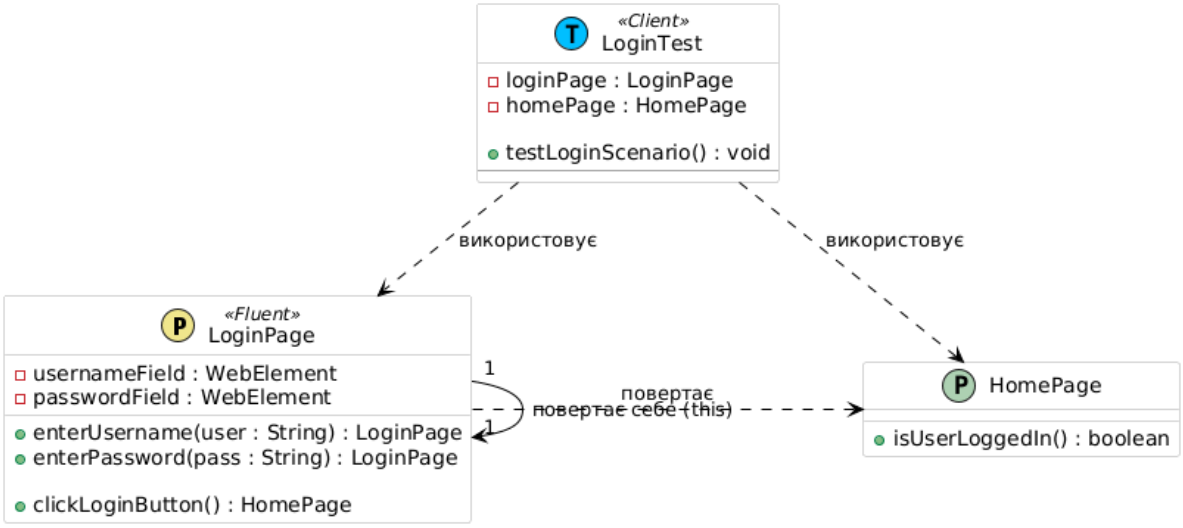


Рисунок 1.7 – Принцип роботи паттерну Fluent Interface

Застосування патерну Fluent Interface перетворює послідовність окремих команд на єдиний, логічно завершений вираз, що значно покращує розуміння логіки тесту. Такий код не лише виглядає елегантніше, але й краще відображає послідовність дій користувача, ніби розповідаючи історію його взаємодії з додатком. Патерн Builder є більш структурованою та формалізованою версією

цього підходу і є особливо корисним при конструюванні складних тестових об’єктів або підготовці тестового оточення з багатьма параметрами.

Комплексне застосування цих патернів та підходів – POM для структурування взаємодії з UI, DDT для керування даними, BDD для бізнес-орієнтованого опису сценаріїв та Fluent Interface для підвищення читабельності коду, дозволяє створювати не просто гнучкий набір тестів, а повноцінну, масштабовану та легку в підтримці екосистему автоматизації тестування вебдодатків.

1.4 Огляд сучасних інструментів автоматизації вебтестування

Вибір інструменту для автоматизації тестування є стратегічним рішенням, яке впливає на ефективність, стабільність та вартість всього процесу забезпечення якості. Ринок пропонує низку фреймворків, кожен з яких має власну архітектуру, та набір можливостей. Для обґрунтованого вибору необхідно провести глибокий порівняльний аналіз головних рішень на ринку: Selenium, Cypress, TestCafe та Playwright.

Selenium є інструментом, що тривалий час виступав у ролі галузевого стандарту в автоматизації вебтестування. В основі його архітектури лежить протокол WebDriver, стандартизований консорціумом W3C, який функціонує як універсальний інтерфейс для віддаленого керування браузерами. Процес взаємодії побудований на клієнт-серверній моделі, де тестовий скрипт надсилає команди у форматі JSON через HTTP на проміжний сервер драйвер, який транслює їх у нативні виклики API браузера.

Однією з переваг такої архітектури є її універсальність. Selenium підтримує більшість існуючих браузерів, включаючи версії, які вийшли з підтримки розробників, та операційні системи. Завдяки доступності мов програмування команди можуть використовувати Java, C#, Python, JavaScript чи Ruby для розробки автоматизованого фреймворку [21].

Ця архітектура також є джерелом основних недоліків Selenium. Послідовні HTTP-запити для кожної команди створюють мережеві затримки, що робить виконання тестів відносно повільним. Однією з поширених проблем є відсутність вбудованих механізмів автоматичного очікування. Selenium не підтримує моніторинг внутрішнього стану вебдодатку, тому інженерам доводиться вручну писати складну логіку синхронізації.

Cypress був спроєктований з фокусом на швидкість та досвід розробника. Архітектура Cypress полягає у виконанні тестів в тому самому циклі подій браузера, що і сам вебдодаток. Це надає тестам прямий, безпосередній доступ до DOM та всіх браузерних API, усуваючи мережеві затримки. Cypress має вбудовані механізми автоматичного очікування. Він автоматично чекає на появу елементів та їх готовність до взаємодії. Його інтерактивний Test Runner з детальними логами та скріншотами помилок надає інструменти для відлагодження, що значно прискорює процес розробки [22].

Інтеграція Cypress в браузер накладає архітектурні обмеження. Робота в рамках одного iframe ускладнює тестування сценаріїв з декількома вкладками, вікнами чи переходами між різними доменами. Cypress підтримує лише JavaScript/TypeScript і не має вбудованої підтримки рушія WebKit, на якому працює Safari. Це робить його нерелевантним рішенням для проєктів, де потрібне повне крос-браузерне покриття, що є критичним для більшості комерційних продуктів.

TestCafe використовує третій архітектурний підхід, що базується на проксіюванні URL. Під час виконання тесту він створює проксі-сервер, який модифікує трафік, впроваджуючи у код вебсторінки власні скрипти для автоматизації. Перевага цього підходу – простота налаштування. Оскільки TestCafe не потребує жодних зовнішніх драйверів, початок роботи зводиться до роботи в терміналі. Він забезпечує крос-браузерну підтримку і, як і Cypress, має вбудовані механізми очікування.

Однак, сам механізм проксіювання, на якому побудована функціональність TestCafe, має недоліки. Процес перехоплення та модифікації трафіку може сповільнювати завантаження сторінок та конфліктувати зі складними політиками

безпеки контенту (CSP), що призводить до розбіжностей у поведінці додатку під час тестування та в реальному використанні. Відлагодження проблем, пов’язаних з роботою проксі-сервера, може бути складним завданням. Цей інструмент, як і Cypress обмежений екосистемою JavaScript/TypeScript [23].

Архітектура Playwright базується на встановленні прямого, постійного з’єднання з браузером через протокол WebSocket. Цей протокол забезпечує асинхронний та двонаправлений канал зв’язку, що дозволяє надсилати команди та отримувати події від браузера.

Важливою архітектурною відмінністю Playwright є те, що він автоматизує не браузери, а безпосередньо їх браузерні рушії: Chromium, WebKit та Firefox. Цей низькорівневий підхід забезпечує крос-браузерне тестування. Playwright має механізми авто-очікування, які аналізують сторінку і чекають, доки елементи стануть повністю готовими до взаємодії. Концепція ізольованих браузерних контекстів дозволяє виконувати тести паралельно з повною ізоляцією в рамках одного екземпляра браузера, що є перевагою для швидкості.

Крім того, Playwright має додатковий функціонал, такий як перехоплення та модифікація мережевих запитів, що дозволяє легко тестувати поведінку додатку при помилках сервера. Його інструмент Trace Viewer є покращенням для відлагодження, оскільки він записує повний слід виконання тесту з DOM-снапшотами, що дозволяє бачити, що саме пішло не так. Підтримка кількох мов робить його доступним для широкого кола команд. Відносним недоліком Playwright можна вважати менш зрілу спільноту порівняно з Selenium, проте вона зростає надзвичайно швидко.

Як показує проведений аналіз, кожен з розглянутих фреймворків представляє собою певний еволюційний етап у розвитку інструментів для автоматизації вебтестування. Їх фундаментальні архітектурні відмінності – віддалене керування через протокол WebDriver, внутрішньо-браузерне виконання, проксіювання трафіку та пряме з’єднання з рушієм через WebSocket, – зумовлюють розбіжності у їх практичних можливостях. Ці розбіжності проявляються у метриках: швидкості виконання тестового набору, стабільності та надійності тестів, широті крос-

браузерного покриття, а також у складності налаштування та подальшої підтримки фреймворку. Вибір оптимального рішення вимагає зваженої оцінки цих компромісів у контексті конкретних бізнес-вимог та технічних обмежень проєкту [24].

Окрім технічних аспектів, важливо враховувати наявність документації, активність спільноти та поріг входження для команди, оскільки ці фактори визначають швидкість інтеграції інструменту в процес розробки. Тільки комплексний підхід до аналізу дозволить обрати інструмент, що забезпечить довгострокову ефективність QA-процесів.

Для систематизації вищесказаного, проведемо порівняльний аналіз головних характеристик розглянутих інструментів у таблиці 1.7.

Таблиця 1.7 – Порівняльна характеристика фреймворків для автоматизації тестування

Характеристика	Selenium	CyPress	TestCafe	Playwright
Архітектура	WebDriver (HTTP)	Внутрішньо-браузерна	URL-проксі	WebSocket до рушія
Крос-браузерність	Найширша	Обмежена	Широка	Широка (Chromium, WebKit, Firefox)
Швидкість	Повільна	Дуже висока	Середня	Дуже висока
Автоочікування	Відсутні	Вбудовані	Вбудовані	Вбудовані та надійні
Підтримка мов	Найширша	JS/TS	JS/TS	JS/TS, Python, Java, .NET
Паралельне виконання	Складне (Grid)	Платне/складне	Вбудоване	Вбудоване та ефективне

Дані, представлені у таблиці, ілюструють, що Playwright поєднує широку крос-браузерну підтримку з високою швидкістю виконання та вбудованими механізмами авто-очікування. Ці характеристики є наслідком його архітектури, яка відрізняється від підходу WebDriver. Playwright працює з браузерами на нижчому рівні, використовуючи нативні протоколи браузерів. Такий підхід дозволяє виконувати команди асинхронно та з мінімальними затримками, усуваючи необхідність у проміжних драйверах [25]. Порівняння традиційного архітектурного підходу до автоматизації тестування за допомогою Selenium, та новітнього підходу за допомогою фреймворку Playwright представлене на рисунку 1.8.

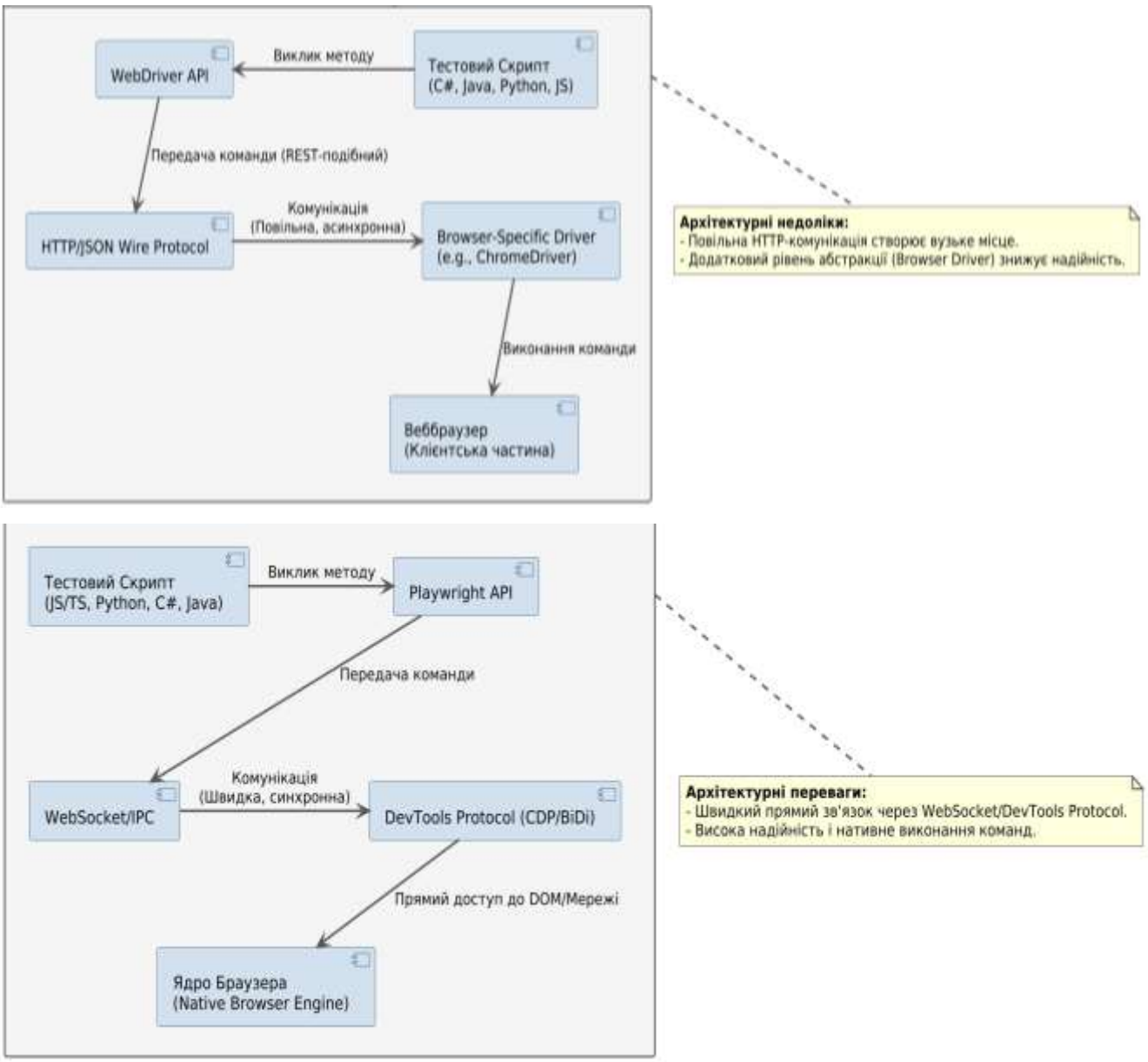


Рисунок 1.8 – Порівняння архітектур автоматизації тестування

Підхід (Selenium WebDriver) вказує на необхідність проходження через кілька проміжних шарів: WebDriver API → HTTP/JSON Wire Protocol → Browser-Specific Driver. Одним з недоліків такого підходу є використання HTTP/JSON Wire Protocol, що робить комунікацію більш повільною та асинхронною, збільшуючи ризик нестабільності. Відсутність вбудованих очікувань може ускладнити підтримку коду, і є причиною появи крихких тестів що падають не через реальний дефект а через те, що елемент завантажився на 100мс довше ніж скрипт очікував.

Підхід Playwright показує інший шлях: Playwright API → WebSocket/IPC → DevTools Protocol (CDP/BiDi). DevTools Protocol дозволяє Playwright напряму

спілкуватися з ядром браузера, обходячи більшість проміжних шарів. Це забезпечує безпосереднє виконання команд, високу швидкість і надійність (усуває необхідність у постійних очікуваннях) (рисунок 1.9).

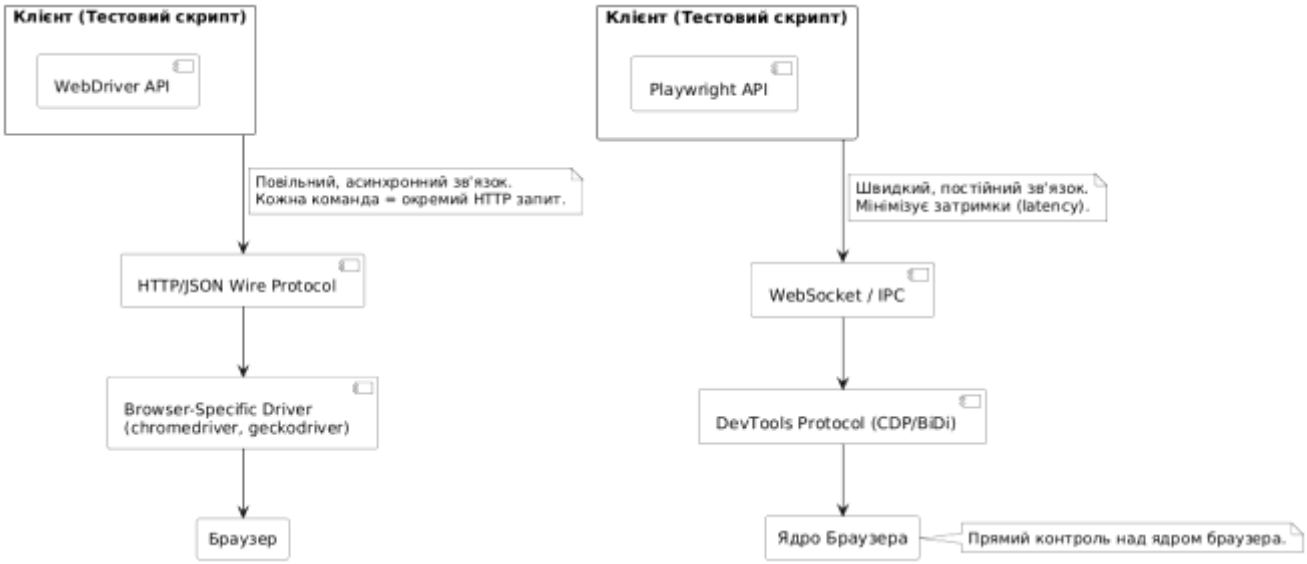


Рисунок 1.9 – Діаграма діяльності підходів тестування

Playwright архітектурно усуває проблему «крихкості» тестів шляхом впровадження механізмів автоматичного очікування. Завдяки цьому механізму, тест падає лише тоді, коли елемент дійсно відсутній або нефункціональний [25].

Висновки до розділу 1

У першому розділі проведено теоретичний аналіз процесу автоматизації тестування вебдодатків. Встановлено, що в умовах гнучких методологій розробки автоматизація є інтегрованим інженерним процесом, головною метою якого є створення системи швидкого зворотного зв’язку в межах CI/CD. Успішність процесу визначається зваженою стратегією, що враховує економічну доцільність, ризики та синергію з ручними методами тестування.

Обґрунтовано, що стабільна система тестування має базуватися на принципах піраміди тестування, поєднуючи unit та інтеграційні тести з мінімально

необхідною кількістю наскрізних GUI-тестів. Також проаналізовано специфічні виклики вебтестування, зокрема динамічну природу DOM та асинхронність операцій, що висувають підвищені вимоги до інструментів.

Проведено аналіз архітектурних патернів (Page Object Model, Data-Driven Testing, BDD), який показав, що їх застосування є важливим для забезпечення довгострокової підтримки, масштабованості та читабельності тестового коду.

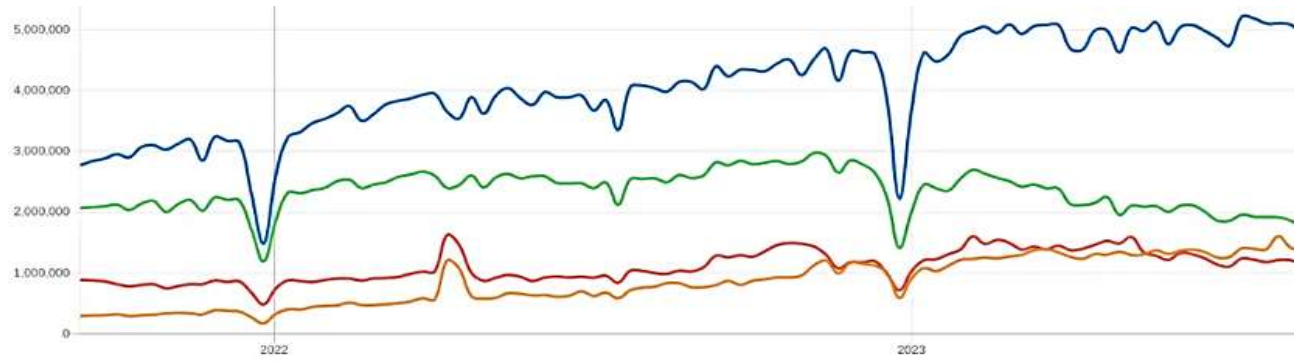
Виконано порівняльний аналіз архітектур провідних інструментів автоматизації (Selenium, Cypress, TestCafe, Playwright). Аналіз виявив значні відмінності у підходах до взаємодії з браузером, що безпосередньо впливає на швидкість, надійність та функціональні можливості. За результатами аналізу, архітектура Playwright, що базується на прямому з’єднанні з браузерними рушіями, була обрана як основа для подальшої практичної реалізації оптимізованого фреймворку.

РОЗДІЛ 2
ФРЕЙМВОРК PLAYWRIGHT ЯК ТЕХНІЧНЕ РІШЕННЯ ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

2.1 Архітектура та принципи роботи фреймворку Playwright

Playwright – це фреймворк автоматизації від Microsoft, для створення end-to-end тестів вебдодатків, а також для вебскрапінгу. Його універсальність є фундаментальною, оскільки він має офіційні API для JavaScript/TypeScript, Python, Java та C#/.NET, а також забезпечує мультибраузерну підтримку (Chromium, Firefox, WebKit/Safari) через єдиний API, використовуючи протокол DevTools для безпосередньої взаємодії з браузерними рушіями [23-25].

Playwright є одним із найбільш поширених фреймворків для автоматизації, це open-source рішення на базі Node.js, яке підтримує крос-платформність (Windows, Linux, macOS), інтегрується в CI/CD процеси (наприклад, Jenkins) та має офіційні API для основних мов програмування. Статистика використання автоматизованих фреймворків для тестування вебдодатків представлена на рисунку 2.1.



Stats

	Stars	Issues	Version	Updated	Created	Size
cypress	44,923	1,387	13.2.0	5 days ago	10 years ago	bundlephobia 429
playwright	54,889	688	1.38.0	3 days ago	9 years ago	bundlephobia 500
selenium-webdriver	27,718	227	4.12.0	16 days ago	11 years ago	bundlephobia 520
webdriverio	8,287	159	8.16.10	2 days ago	9 years ago	bundlephobia 429

Рисунок 2.1 – Статистика використання автоматизованих фреймворків

Playwright має відмінну, позапроцесну архітектуру, де тестовий скрипт та браузерний рушій існують як абсолютно незалежні процеси, що взаємодіють через протоколи віддаленого керування (наприклад, CDP для Chromium), а не шляхом безпосереднього втручання в сторінку (рисунок 2.2).

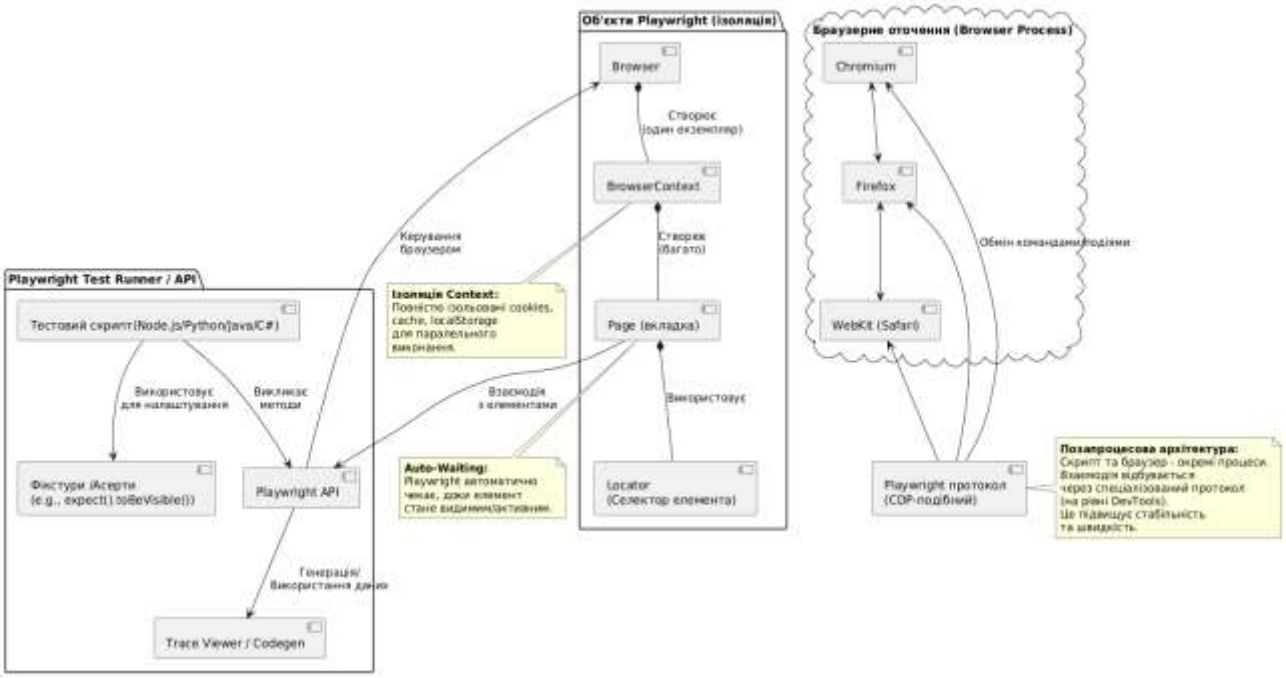


Рисунок 2.2 – Архітектура тестування Playwright

Таким чином, актуальність Playwright підкріплена його архітектурою, яка вирішує проблему нестабільності тестів: фреймворк реалізує механізми автоматичного очікування (Smart Waiting та Actionsability Checks), що гарантують взаємодію лише після того, як елемент стає видимим, стабільним та активним. Як повноцінний фреймворк, Playwright надає комплексне середовище з вбудованим тестовим ранером та інструментами для трасування (Trace Viewer), що дозволяє записувати та відтворювати повний лог виконання тесту; крім того, він підтримує виконання кожного тесту у повністю ізольованому «браузерному контексті» для забезпечення чистоти даних (cookies, сховище) та забезпечує паралельний запуск, що прискорює прогони в крос-платформних CI/CD конвеєрах [26]. Завдяки такому поєднанню архітектурної надійності та розвинутого інструментарію, впровадження Playwright дозволяє суттєво знизити витрати на підтримку

автоматизації та мінімізувати ризики пропуску дефектів у динамічних вебзастосунках.

Окрім архітектури, однією з головних причин швидкого впровадження Playwright є його орієнтація на досвід розробника (Developer Experience). Фреймворк поставляється з повним набором вбудованих інструментів. Сюди входить Playwright Codegen – утиліта, яка автоматично генерує код тесту, записуючи дії користувача безпосередньо в браузері. Це дозволяє інженерам створювати основу для нових тестів та прискорює поріг входження для нових членів команди.

Цей підхід забезпечує стійкість тестів та знімає обмеження безпеки браузера, надаючи повний зовнішній контроль, включаючи керування кількома вкладками, фреймами та мережевою активністю сторінки (що дозволяє QA-інженеру мокувати (mock) відповіді бекенду або блокувати запити). В основі тестової ізоляції лежить концепція BrowserContext – це легковаговий, ізольований сеанс (подібний до режиму інкогніто), який має власний набір Cookies, LocalStorage та кешу, повністю усуваючи необхідність ручного менеджменту браузерних драйверів, як це було у Selenium. Саме ця ізоляція є технічною основою для інтегрованої у @playwright/test функції паралельного виконання тестів, де кожен тест запускається у власному чистому контексті, скорочуючи загальний час прогону. Перед виконанням будь-якої дії він запускає цілий набір Actionsability Checks (перевірок на видимість, активність, відсутність перекриття), усуваючи потребу в ручних командах очікування, а функціонал Web-First Assertions розширює це очікування на команди expect() (наприклад, toBeVisible()), повторюючи перевірку, доки умова не буде виконана. Додатково, тестовий ранер запроваджує патерн фікстур, що спрощує налаштування та очищення тестового середовища (наприклад, автоматизація логіну адміністратора), полегшуючи підтримку тестового коду [27].

Playwright використовується для імітації дій користувача. Розглянемо його застосування на прикладі простого сценарію тестування: перевірити, чи коректно працює пошук на сайті документації Playwright. Цей сценарій реалізується через таку послідовність дій:

- відкрити вебсторінку https://playwright.dev/;
- натиснути на кнопку пошуку, щоб активувати поле вводу;
- ввести текст «install» у поле пошуку;
- перевірити, що в результатах пошуку з’явився пункт «Installation».

Ось як цей сценарій реалізується на трьох найбільш поширених мовах програмування, які підтримує Playwright: TypeScript (Node.js), Python та Java.

TypeScript – це найпоширеніший спосіб використання Playwright, оскільки він має вбудований тестовий фреймворк Playwright Test Runner. Нижче подано приклад використання Playwright на мові TypeScript:

```
// example.spec.ts
import { test, expect } from ‘@playwright/test’;

test(‘Пошук документації Playwright’, async ({ page }) => {
 // 1. Відкрити сторінку
 await page.goto(‘https://playwright.dev/’);

 // 2. Натиснути кнопку пошуку
 // Шукаємо кнопку з текстом "Search" або натискаємо Ctrl+K
 // (якщо це desktop)
 await page.getByRole(‘button’, { name: ‘Search’ }).click();

 // 3. Ввести текст "install" у поле пошуку
 await page.getByPlaceholder(‘Search docs’).fill(‘install’);
 // 4. Перевірити, що в результатах з’явився пункт "Installation"
 const searchResult = page.locator(‘.DocSearch-Hits .DocSearch-Hit-
title:has-text("Installation")’);
 await expect(searchResult).toBeVisible();
});
```

У Python Playwright часто використовується з фреймворком pytest. У наступному прикладі використовується чистий Playwright API в асинхронному режимі:

```
# test_search.py
import asyncio
```

```
from playwright.async_api import async_playwright, expect
async def test_playwright_documentation_search():
  async with async_playwright() as p:
    # Запускаємо Chromium
    browser = await p.chromium.launch()
    page = await browser.new_page()
    # 1. Відкрити сторінку
    await page.goto("https://playwright.dev/")
    # 2. Натиснути кнопку пошуку
    await page.get_by_role("button", name="Search").click()
    # 3. Ввести текст "install" у поле пошуку
    await page.get_by_placeholder("Search docs").fill("install")
    # 4. Перевірити, що в результатах з’явився пункт "Installation"
    search_result = page.locator('.DocSearch-Hits .DocSearch-Hit-
title:has-text("Installation")')
    await expect(search_result).to_be_visible()
    await browser.close()
# Запуск тесту
if __name__ == "__main__":
  asyncio.run(test_playwright_documentation_search())
```

У Java Playwright часто використовується разом із JUnit або TestNG:

```
// PlaywrightSearchTest.java
import com.microsoft.playwright.*;
import com.microsoft.playwright.options.*;
import org.junit.jupiter.api.*;
import static
com.microsoft.playwright.assertions.PlaywrightAssertions.assertThat;
public class PlaywrightSearchTest {
  @Test
  void playwrightDocumentationSearch() {
    try (Playwright playwright = Playwright.create()) {
      Browser browser = playwright.chromium().launch();
      BrowserContext context = browser.newContext();
      Page page = context.newPage();
      // 1. Відкрити сторінку
```

```
        page.navigate("https://playwright.dev/");
        // 2. Натиснути кнопку пошуку
        page.getByRole(AriaRole.BUTTON, new
Page.GetByRoleOptions().setName("Search")).click();
        // 3. Ввести текст "install" у поле пошуку
        page.getByPlaceholder("Search docs").fill("install");
        // 4. Перевірити, що в результатах з’явився пункт
"Installation"
        Locator searchResult = page.locator(".DocSearch-Hits
.DocSearch-Hit-title:has-text(‘Installation’)");
        assertThat(searchResult).isVisible();
        browser.close();
    }}}
```

Незважаючи на різницю в синтаксисі мов, структура та методи API Playwright (page.goto(), page.getByRole(), page.locator(), expect()) залишаються практично ідентичними, що підкреслює універсальність фреймворку.

2.2 Можливості Playwright у кросбраузерному, мобільному та інтеграційному тестуванні

Playwright надає можливість проводити як E2E так і інтеграційне тестування за допомогою вбудованих функцій. Ціль інтеграційного тестування – підтвердити, що всі модулі, API, бази даних та інші сервіси злагоджено працюють разом.

Playwright може окремо проводити API запити та працювати з базами даних, що робить його інструментом для інтеграційного тестуваня, він надає розробникам можливість тестувати відповіді від API та валідувати їх за потреби, тобто дозволяє досить інтегруватись у бекенд процеси не прибігаючи до E2E тестів. Також Playwright дозволяє верифікувати правильність бази даних за допомогою функцій які надають можливість маніпулювати даними всередині баз даних. Однією з переваг даного фреймворку, що він дає можливість комбінувати різні рівні тестування в рамках одного тестового сценарію та єдиного інструменту. Замість

того, щоб використати окремий інструмент для API , окрему бібліотеку для роботи з БД, і сам фреймворк для тестування інтерфейсу, QA інженер може побудувати цілісний та ефективний гібридний тест, Що дозволяє перевірити повний життєвий цикл функції.

Такий підхід мінімізує кількість взаємодій з інтерфейсом для більшої стабільності і більш детальної перевірки функцій. Можливість вільно маніпулювати станом системи через бекенд і одразу ж валідувати результат у браузері підкреслює універсальність фреймворку.

Крім того, частиною сучасного інтеграційного тестування є здатність ізолювати компоненти та симулювати різні стани системи. Playwright надає для цього механізм перехоплення запитів за допомогою вбудованої функції. Це дозволяє тестувати інтеграцію між фронтендом та бекендом не покладаючись на реальну відповідь сервера або тестувати інтеграцію сервісу зі сторонніми рішенням без виконання реальних запитів до них. Тобто замість налаштування тестових середовищ або штучної модифікації стану серверної частини для відтворення помилок, фреймворк дозволяє імітувати відповіді, що спрощує процес тестування.

Таким чином, гнучкість фреймворку робить його інструментом для інтеграційного тестування, він дозволяє обирати не тільки один тип тесту а комбінувати підходи для досягнення максимальної ефективності у тестуванні додатку.

Здатність створювати гібридні сценарії де стан може готуватись через API, а стан перевірятись в UI і результат валідується на рівні даних вирішує проблему E2E тестів.

Водночас вбудовані можливості мокування надають контроль та ізоляцію, що дозволяє покрити тестами складні сценарії та граничні випадки які практично неможливо відтворити у реальному середовищі.

Playwright надає можливості для тестування мобільних версій веб додатків без використання мобільних девайсів, хоча запуск тестів на емуляторі Android є можливим. Натомість, Playwright дає можливість інженерам емулювати мобільний девайс у браузері, що конфігурується за допомогою програмного коду. Перед

запуском тесту, є можливість налаштувати роздільну здатність екрану, спосіб взаємодії та device scale factor [28].

Playwright може емулювати нестабільне інтернет з’єднання, що може бути корисним не тільки для мобільного тестування, але так як більшість користується смартфонами за допомогою мобільного інтернету, що може значно сповільнити завантаження сайту, на даний тестовий випадок у Playwright є функція яка дозволяє емулювати повільний інтернет, щоб зафіксувати як себе поводить додаток у різних умовах.

Playwright може емулювати сенсорний екран смартфону, що дозволяє робити приближення, свайпи і натискання як на екрані девайсу. Також є можливість відслітковувати performance метрики що може бути корисно у тестуванні і розумінні як додаток працює на мобільних девайсах і як швидко він відповідає на ті чи інші запити [29].

Варто додати, що для максимального спрощення цього процесу Playwright має вбудовану бібліотеку пристроїв. Замість того, щоб вручну підбирати конфігурації, для кожного тесту є можливість імпортувати конфігурацію певної моделі смартфону чи планшету, що імітують саме цей пристрій гарантуючи, що сервер вважатиме запит мобільним трафіком. Окрім фізичних характеристих та мережевих налаштувань емуляція поширюється на інші сервіси притаманні смартфонам. Фреймворк надає простий інтерфейс для емуляції геолокації, дозволяючи підмінити координати пристрою. Це може знадобитись при тестуванні функціоналу який залежить від місцезнаходження пристрою, наприклад карт або для регіональних пропозицій.

Також можна керувати дозволами такими як наприклад використання камери чи мікрофону у тестовому сценарії, або перевірити як додаток реагує на відмову користувача. Емуляція яку надає саме playwright відбувається на рівні браузеру, а не на рівні самої мобільної операційної системи, це означає що він не може взаємодіяти з нативними елементами ОС, і що важливіше не відтворить специфічні помилки рендерингу чи недоліки JS, притаманні лише прошивці конкретного смартфону чи виробника.

Таким чином емуляція є швидким, економічно вигідним та ефективним рішенням для тестування мобільних веб інтерфейсві та адаптивної верстки, щоб покрити значну частину функціоналу на ранніх етапах, проте для повної впевненості у якості продукту вона не скасовує потребу фінальної валідації на реальних пристроях, чи повноцінних емуляторах.

Playwright надає можливості кросбраузерного тестування за допомогою вбудованих функцій, фреймворк дає можливість використовувати найпопулярніші браузерні рушії, які підтягуються автоматично і завжди мають найновіші версії що виключити помилки коли тест пройшов у старій версії браузеру, але нове оновлення змінило функціонал і це пройшло непомітно.

Кросбраузерне тестування є одним з найважливіших аспектів у тестування веб додатків через те, що зараз існує безліч веб браузерів які працюють за різними принципами, тому в одному браузері додаток може працювати нормально, в іншому мати певні похибки, через те, що той самий код може по різному відображатись у різних рушіях. Саме для цього потрібно проводити та автоматизовувати даний тип тестування, з чим ефективно справляється Playwright [30].

Налаштування кросбраузерності відбувається на рівні конфігураційних файлів, де Playwright використовує концепцію проєктів, де кожен проєкт це опис одного тестового середовища. Коли QA інженер визначає параметри для кросбраузерного тестування, через паралелізацію тестів вони можуть одночасно запускатись у декількох браузерах для економії часу.

2.3 Інтеграція Playwright із сторонніми сервісами

Playwright має можливостеі інтеграції із різними сервісами і іншими фреймворками наприклад для CI/CD, генерації репортів, та контейнеризації, доцільно ретельно розглянути кожен аспект для повного розуміння можливостей фреймворку [31].

Інтеграція з системами CI/CD є однією з головних архітектурних переваг фреймворку, від самого початку він проєктувався з розрахунком на те, що тести будуть виконуватись автоматично в повністю ізольованих середовищах забезпечуючи зворотній зв’язок для команди розробки, що повністю відповідає сучасній методології «Shift Left», де тестування інтегрується на якомога раніші етапи розробки [32].

Тестові репорти визначено одним з найважливіших елементів в тестуванні програмного забезпечення тому, що він допомагає як QA так і розробникам визначати чи правильно працює функціонал, і на основі репортів приймати рішення, наприклад, про удосконалення продукту.

Playwright має змогу генерувати репорти самостійно без запобігання до сторонніх фреймворків чи сервісів за допомогою вбудованих функцій, він дає змогу побачити які тести пройшли добре, які впали, та були пропущені [33].

У репорті згенерованому фреймворком, є наступні компоненти:

- статус тесту – компонент відображає інформацію про пройдені, провалені, плаваючі, та пропущені тести;
- деталі помилок – компонент показує типи помилок та їх позиції у додатку;
- час виконання – показує скільки часу витрачено на те, щоб пройти тест, допомагає розкрити повільні тести, та проблеми з продуктивністю;
- скріншоти – для провалених тестів фреймворк автоматично робить скріншот на точці провалу тесту для надання візуального контексту;
- відео – фреймворк має змогу записувати відео всього тестового прогону для надання динамічної інформації про помилку;
- логи – детальні логи для відлагодження функціоналу та/або тестів;
- покриття тестів – показує кількість тестів які покриті поточним тестовим прогоном.

Тестові репорти у Playwright спроєктовані, щоб бути інтерактивними з опціями згортання і розгортання, фільтрування тестів та навігації по детальним логам помилок, що дає QA та розробникам детальну інформацію про контекст тесту. Приклад звіту згенерованого у Playwright наведено на рисунку 2.2.

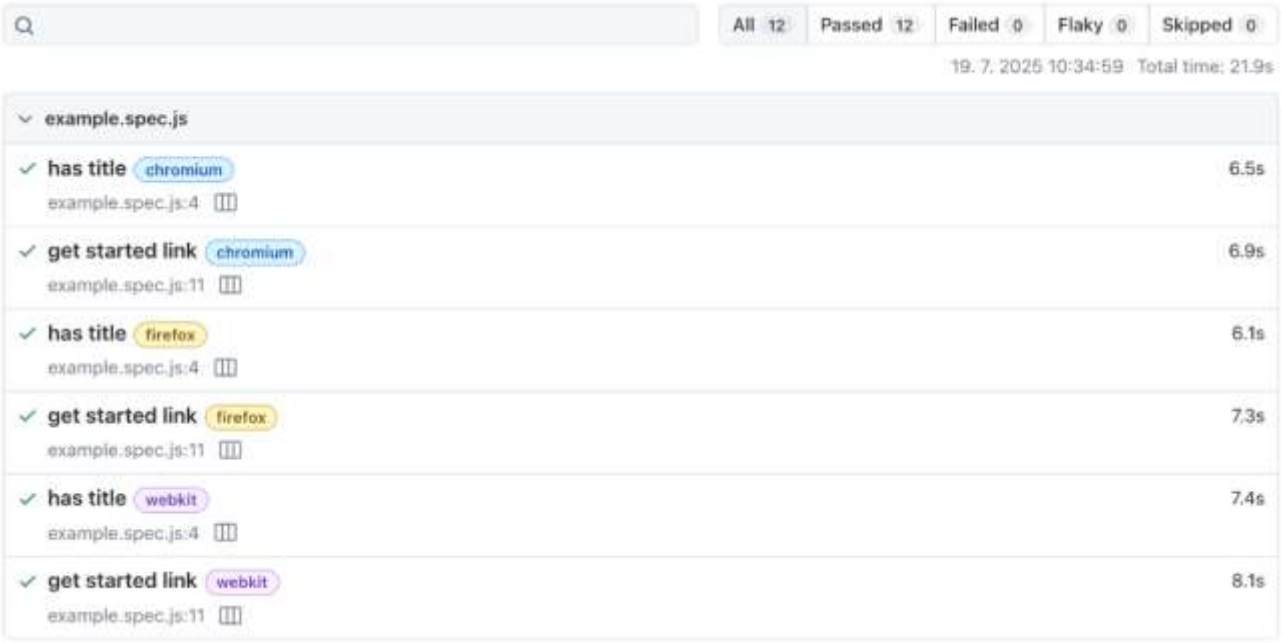


Рисунок 2.2 – Приклад HTM-репорту у Playwright

У Playwright вбудовані декілька варіантів репортів такі як Junit репорт, Json репорт, та Dot репорт і інші варіанти репортів які в більшості доступні через консоль [34].

Проте справжня гнучкість фреймворку полягає не лише у вбудованих звітах але й у його здатності інтегруватися зі сторонніми, більш потужними системами звітності, хоч і вбудований звіт є прийнятним варіантом для локального аналізу, у командних середовищах часто потрібні складніші аналітичні рішення. Фреймворк дозволяє підключати репортери що відкриває можливість для інтеграції з платформами наприклад як Allure Report. Allure не просто створює звіт про один прогін, а повноцінні інтерактивні таблиці з історією, графіками, можливістю класифікувати тести та в динаміці аналізувати сам стан продукту.

Іншою перевагою яку надають сторонні сервіси це глибока інтеграція із системами управління тестуванням, такими як Testomat.io, Report Portal та інші. На відміну від статичного звіту який може генеруватись тільки після завершення тестового прогону, такі інтеграції працюють у режимі реального часу. Вони надсилають результати на централізований сервер миттєво під час виконання прогонів, що дозволяє командам спостерігати стан тестів не чекаючи повного

завершення, це також дозволяє пов’язувати результати авто-тестів з ручними тестами і вимогами в TMS.

Варто зазначити, що фреймворк дозволяє налаштовувати декілька репортерів одночасно, що налаштовується програмно у конфігураційному файлі playwright.config.ts. Наприклад команда може одночасно генерувати локальний репорт, репорт junit для інтеграції з Jenkins та надсилати дані в Allure для загального аналізу. Ця гнучкість робить систему звітності надзвичайно адаптивною до будь-яких потреб проєкту [35].

Для досягнення максимального рівня надійності, відтворюваності та ізоляції тестового середовища у конвеєрах, використовується технологія контейнеризації, лідером якої є такий сервіс як Docker. Запуск тестів у контейнерах вирішує одну з найпоширеніших проблем автоматизації, коли на локальних машинах розробників або тестувальників результат може відрізнятись. Причини цього можуть бути неочевидними, накшталт дрібні відмінності в операційних системах, відсутність специфічних бібліотек чи шрифтів, необхідних для коректного завантаження сторінки, або відмінності у налаштуваннях мережі.

Контейнери фундаментально вирішують цю проблему, бо вони загортають тестовий код, та всі залежності у єдиний легкий і портативний образ, що гарантує, що тестове середовище буде ідентичним підчас кожного запуску незалежно від того, він запускався з локального ПК розробника у Полтаві, чи на кластері серверів у Франкфурті.

Команда розробників Playwright офіційно підтримує та публікує власні налаштовані образи у публічних репозиторіях.

Візуалізація того, як контейнеризація вбудовується в загальну архітектуру, представлена на діаграмі (рисунок 2.3), де відображено внутрішню структуру Docker-контейнера, підготовленого для інтеграції в процеси CI/CD. Такий підхід дозволяє інкапсулювати все необхідне оточення в єдиний портативний артефакт, завдяки чому гарантується повна ідентичність виконання тестів як на локальній машині розробника, так і на будь-якому агенті CI-системи, незалежно від його конфігурації.

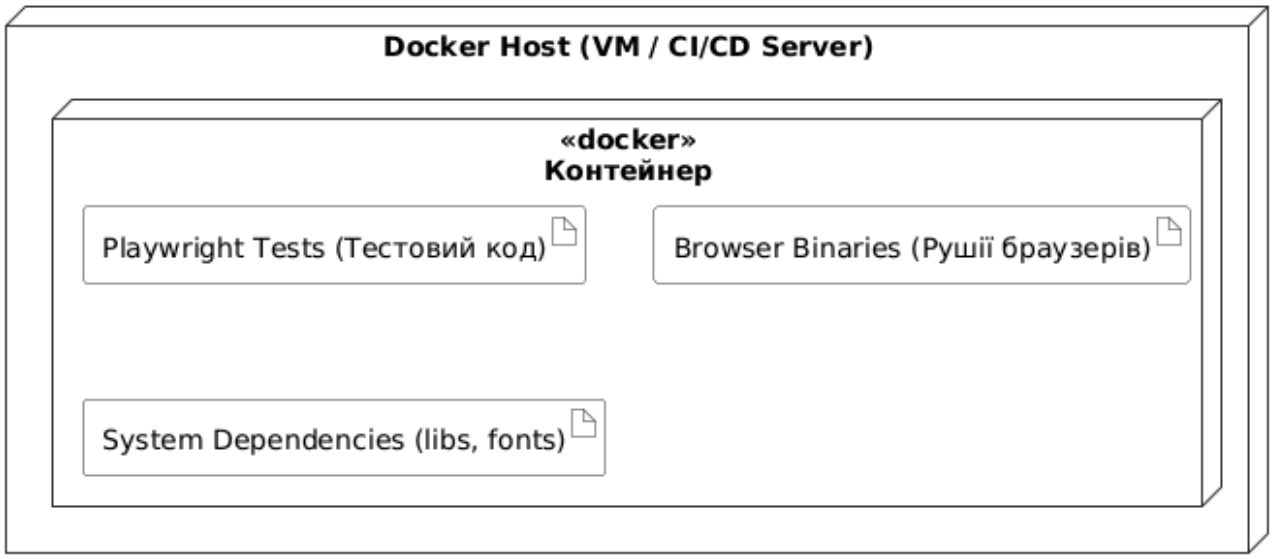


Рисунок 2.3 – Діаграма розгортання процесу тестування в контейнеризованому CI/CD

Процес починається на сервері, наприклад Jenkins, який виступає як керуючий вузол. Він не виконує тести безпосередньо, замість цього він віддає команду хосту Docker , а той, отримавши команду створює один або декілька ізольованих контейнерів на основі офіційного образу Playwright. Кожен контейнер є абсолютно ідентичним середовищем що містить браузерні рушії та тестовий код. Контейнери виконують тести паралельно а потім надсилають тестові артефакти назад для аналізу. Такий підхід оптимізує процес тестування [36].

2.4 Розробка структури тестових сценаріїв і використання мов програмування у Playwright

Вибір мови програмування та продумана архітектура проєкту є факторами, що безпосередньо впливають на ефективність, масштабованість та довгострокову вартість підтримки системи автоматизованого тестування

Продумана архітектура забезпечує чітке розділення відповідальностей, наприклад, ізолюючи логіку бізнес-сценаріїв від технічної реалізації взаємодії з інтерфейсом, як це пропонує Page Object Model. Без спланованої структури проєкт

перетворюється на хаотичний набір тестів. У такому проєкті будь-яка незначна зміна у вебдодатку призводить до каскадного падіння десятків тестів [37-38].

Вибір мови в свою чергу визначає, наскільки легко та ефективно ця архітектура може бути реалізована. Інтеграція Playwright з ранером Pytest надає інженерам доступ до механізму фікстур, саме ці фікстури стають інструментом для реалізації архітектурних патернів, управління станом браузера та підготовки даних що дозволяє створювати підтримуваний масштабований код. Це є глибокою інтеграцією, модель фікстур Pytest є потужною реалізацію патерну впровадження залежностей. Тестова функція оголошує свої потреби вказуючи назви необхідних фікстур у своїх аргументах, Pytest в свою чергу бере на себе всю роботу з пошуку, ініціалізації та, що важливо очищення ресурсів після завершення тесту щоб завадити забруденню тестових даних. Такий підхід дозволяє повністю відокремити логіку підготовки середовища від логіки самого тестового сценарію [39].

Такий механізм є цінним для практичної реалізаці патерну Page Object Model який був детально розглянутий у Розділі 1. Замість того щоб кожен тест окремо вручну створював екземпляри класів сторінок створюється спеціальна фікстура яка інкапсулює логіку ініціалізації, де тестова функція отримує готовий до роботи об’єкт сторінки.

Використання Python, Pytest та патерну автоматизації Page Object Model, як описано вище це необхідна умова для створення оптимізованого фреймворку. Дані інструменти та патерни природним чином диктують чітку модульну та масштабовану структуру файлів. Така архітектура проєкту є фактором оптимізації оскільки вона реалізує фундаментальні поняття проєктування ПЗ. Це в свою чергу впливає на вартість підтримки фреймворку в довгостроковій перспективі. Ефективний проєкт автоматизації що слідує цим принципам зазвичай чітко організований розділенням відповідальностей. У корені самого проєкту створюється директорія tests/ яка містить файли з тестовими сценаріями які згруповані за модулями. Код у цих файлах описує бізнес-логіку кроків та перевірок. Він залишається декларативним і читабельним оскільки не містить жодних технічних деталей реалізації типу селекторів чи XPath [40].

Всі технічні деталі інкапсулюються в окремій директорії pages/ де розміщуються класи сторінок. Вони містять локатори для всіх інтерактивних елементів сторінки та публічні методи, які імітують дії користувача. Таким чином досягається головна мета патерну POM [41].

Основою фреймворку є файл conftest.py, що є спеціальним конфігураційним файлом який Pytest автоматично розпізнає і завантажує перед виконанням тестів. В ньому розміщуються всі загальні фікстури описані в попередньому розділі. Це можуть бути фікстури для налаштування та запуску браузера, фікстури для ініціалізації об’єктів сторінок, а також для підготовки тестових даних. Така структура гарантує що при заміні локатора умовної кнопки QA інженеру треба буде замінити лише один файл, зміна якого автоматично без будь-якого втручання застосується до десятків тестів які знаходяться у директорії tests і використовують цю сторінку [42].

Також продумана структура проєкту передбачає окрему директорію для даних data/ яка слугує сховищем для зовнішніх тестових даних, що дозволяє реалізувати ще один архітектурний патерн Data-Driven Teting, або тестування кероване даними. Ідея цього патерну полягає в повному відокремленні логіки тесту від даних, на яких цей тест виконується. Замість кодування вхідних значень безпосередньо у тілі тестової функції ці дані завантажуються з зовнішніх легко читомих джерел таких як файли форматів CSV, JSON, YAML [43].

Проблема яку вирішує DDT це масивне дублювання коду, якщо уявити умовний сценарій перевірки форми для логіну користувача, без DDT для перевірки 5 різних комбінацій QA інженеру довелося б написати 5 практично ідентичних функцій, що відрізнялися б лише вхідними даними та очікуваним результатом. Цей підхід є вкрай неефективним та значно ускладнює підтримку.

Екосистема Pytest надає для вирішення цієї проблеми яким є вбудований та потужний механізм – декоратор. Цей декоратор дозволяє параметризувати тестову функцію, тобто вказати що її необхідно запустити декілька разів, послідовно підставляючи декілька наборів даних у її аргументи. Дані можуть бути завантажені з файлів у директорії data/ за допомогою невеликої допоміжної функції. Таким

чином інженер пише єдину тестову функці, а Pytest автоматично проганяє її стільки разів, скільки потребує для заданого тесту. Це є оптимізацією процесу масштабування тестового покриття. Розширення тестування зводиться до простого додавання нових рядків у файли з даними не вимагаючи писати нові ряди коду [44].

Висновки до розділу 2

У другому розділі проаналізовано позапроцесну архітектуру Playwright на базі WebSocket, що забезпечує пряму комунікацію з браузером. Це гарантує надійне автоматичне очікування (Auto-Waiting), синхронізуючи команди з DOM без ручних затримок, та надає абстракцію BrowserContext для повної ізоляції даних і нативного паралелізму. Також архітектура дає повний контроль над мережевими запитами для їх аналізу та мокування.

Встановлено ефективність фреймворку для гібридних інтеграційних сценаріїв, що поєднують API-запити та UI-валідацію. Для мобільного тестування доступна емуляція в'юпорта та сенсорів, а кросбраузерність реалізовано підтримкою рушіїв Chromium, WebKit і Firefox з автоматичним керуванням бінарними файлами.

Досліджено інтеграцію з CI/CD (Jenkins, GitHub Actions), Allure та Docker для відтворюваності середовища. Обґрунтовано вибір стеку Python + Pytest, де модель фікстур дозволяє гнучко реалізувати патерни Page Object Model та Data-Driven Testing, що стало фундаментом для побудови оптимізованого фреймворку, деталізованого в наступному розділі.

РОЗДІЛ 3
ПРАКТИЧНА РЕАЛІЗАЦІЯ ОПТИМІЗОВАНОГО ПРОЦЕСУ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ВЕБДОДАТКІВ НА ОСНОВІ PLAYWRIGHT

3.1 Розробка архітектури тестового фреймворку на базі Playwright

Для валідації та подальшого якісного аналізу, як об’єкт тестування було обрано демонстраційний вебдодаток «Rahul Shetty Academy». Даний додаток є симулятором стандартного сайту електронної комерції, що містить ключові бізнес-процеси: автентифікацію користувача, перегляд каталогу товарів, сортування, додавання товарів у кошик та оформлення замовлення. Ця функціональність є властивою для більшості сучасних вебдодатків, що робить її ідеальним полігоном для апробації розробленого фреймворку.

Архітектура оптимізованого тестового фреймворку (надалі – «Підхід Б») базується на чіткому розділенні відповідальностей та реалізації головних патернів проєктування, розглянутих у розділі1. Візуалізація архітектури цього підходу наведена на рисунку 3.1.

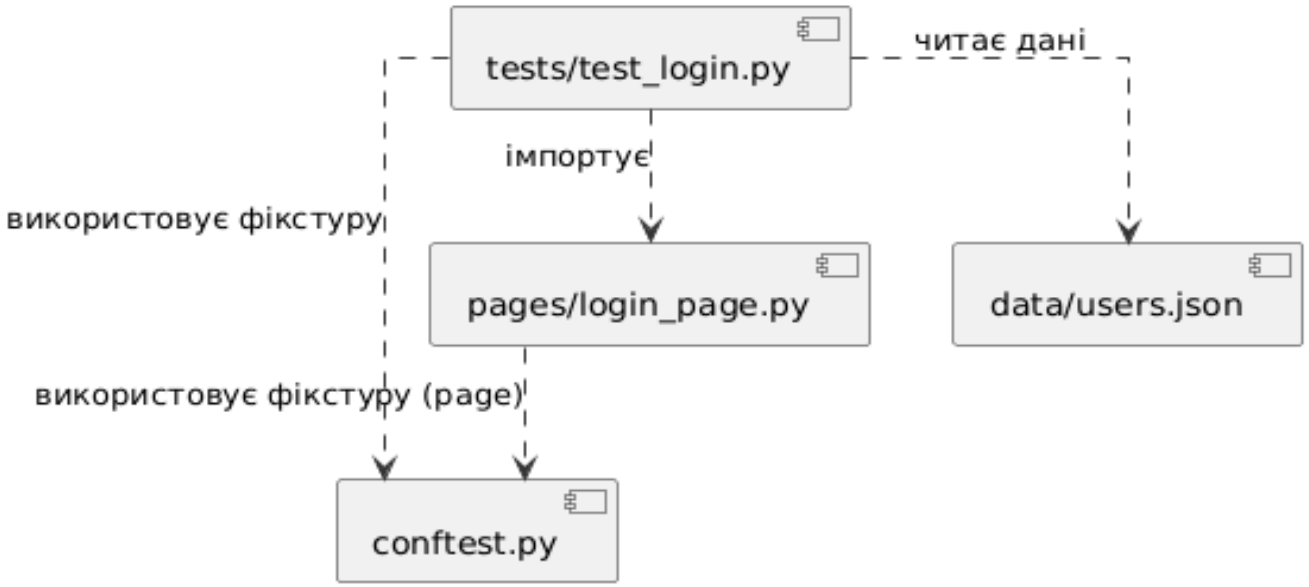


Рисунок 3.1 – Структурна схема оптимізованого фреймворку

З метою отримання об’єктивних та верифікованих порівняльних даних для подальшого аналізу в підрозділі 3.4, паралельно розробляється альтернативний програмний проєкт (надалі – «Підхід А» – традиційний). Ця реалізація імітує поширений «наївний» підхід до автоматизації, що характеризується лінійним написанням коду без застосування архітектурних абстракцій та принципу розділення відповідальності. У рамках цього рішення тестові сценарії акумулюються в єдиному файлі, ідентифікатори елементів (селектори) та вхідні дані жорстко кодуються (hard-coded) безпосередньо в тілі тестових функцій, а структурні патерни, такі як Page Object Model (POM) та Data-Driven Testing (DDT), повністю ігноруються. Створення такого проєкту формує необхідну базову лінію (baseline) для експерименту, що дозволить наочно продемонструвати критичну різницю між підходами в часі виконання, рівні стабільності тестів та трудомісткості підтримки кодової баз.

3.2 Реалізація тестових сценаріїв для вебдодатку

Після визначення архітектури фреймворку в підрозділі 3.1, даний етап присвячено її практичній реалізації шляхом розробки тестових сценаріїв для вебдодатку Rahul Shetty Academy. Метою цього етапу є верифікація гнучкості розробленого рішення на прикладі трьох взаємопов’язаних типів тестування: класичного тестування інтерфейсу, прямого API-тестування та оптимізованого гібридного тестування. Кожен приклад демонструє практичне застосування архітектурних рішень, обґрунтованих у попередніх розділах, для підвищення ефективності та надійності тестів. Програмний код для класів сторінок та тестових сценаріїв представлено у Додатку А.

На першому етапі реалізовано класичний наскрізний сценарій, що повністю імітує дії користувача. Для демонстрації обрано базовий бізнес-процес автентифікації користувача. Цей сценарій ілюструє застосування патерну Page

Object, обґрунтованого у підрозділі 1.3, оскільки він є важливим для функціоналу та часто виступає передумовою для інших тестів.

Відповідно до архітектури, описаної в 3.1, логіку взаємодії зі сторінкою входу інкапсульовано в окремому класі pages/login_page.py.Цей клас містить локатори елементів сторінки та надає високорівневий метод, що приховує деталі реалізації:

```
class loginPage:
  def __init__(self, page: Page):
    self.page = page
    self.login_url = INDEX_URL

  def open_login_url(self):
    self.page.goto(self.login_url)

  def fill_username(self, username):
    self.page.locator("#userEmail").fill(username)

  def fill_password(self, password):
    self.page.locator("#userPassword").fill(password)
```

Завдяки такому підходу тестовий сценарій у файлі tests/test_login.py стає декларативним та структурованим. Він не містить селекторів або XPath, а оперує виключно бізнес-сутнастями. Тест використовує фікстури, які автоматично створюються та передаються засобами Pytest:

```
class TestLogin:
  @pytest.mark.usefixtures("browser_setup")
  def test_login_user(self, browser_setup):
    login_page_obj = LoginPage(browser_setup)
    login_page_obj.open_login_url()

login_page_obj.enter_username(config["users"]["username"])

login_page_obj.enter_password(config["users"]["password"])
    login_page_obj.click_login_button()
```

Наведений фрагмент демонструє, що файл test_login.py містить читабельні тести без жорстко закодованих значень (локаторів, облікових даних), які зберігаються в окремому YAML-файлі. Це підвищує ефективність підтримки тестів, оскільки дана сторінка та її методи можуть бути багаторазово використані в межах фреймворку.

Важливо відзначити технічну особливість, що ілюструє перевагу обраного інструментарію. Реалізація класу loginPage не лише інкапсулює локатори, але й використовує механізм автоматичного очікування. При виклику методу fill_username виконання команди відбувається не миттєво, а після проходження внутрішніх перевірок фреймворку. Цей механізм вирішує проблему нестабільності тестів («flaky tests»), розглянуту у Розділі 1. Традиційні підходи («Підхід А») вимагають ручного додавання явних очікувань або використання ненадійних команд на кшталт time.sleep(), що сповільнює тести. Playwright нівелює цю проблему на рівні архітектури.

Аналіз файлу тесту демонструє, як реалізація патернів Pytest сприяє оптимізації. Використання декоратора та передача об’єктів як аргументів функції є прикладом впровадження залежностей (Dependency Injection), переваги якого обґрунтовано у підрозділі 2.4. Тестова функція не виконує створення чи конфігурацію браузера, а лише запитує необхідне середовище, тоді як Pytest забезпечує логіку підготовки та очищення ресурсів.

Як зазначено в розділі 2.2, фреймворк надає інструменти для інтеграційного тестування, що виходять за межі UI-взаємодії.

Якщо підхід, описаний у 3.2, стосується верхнього рівня піраміди тестування, то для комплексної оптимізації доцільно перемістити частину перевірок на нижчі рівні.

Для цього у фреймворк інтегровано інструмент APIRequestContext, який дозволяє виконувати та валідувати запити до бекенду, оминаючи браузер. Це уможливлює реалізацію API-тестів у межах єдиного технологічного стеку.

Для демонстрації підходу реалізовано тест, що перевіряє аналогічний бізнес-процес (автентифікацію) на рівні API. Це дозволяє ізольовано перевірити логіку

бекенду. Тест використовує спільний конфігураційний файл YAML, що демонструє перевикористання компонентів (файл test_api_login_success):

```
def test_api_login_success():
  with sync_playwright() as p:
    request_context = p.request.new_context(
      base_url=config["urls"]["base_url"]
    )
    response = requests_context.post(
      config["urls"]["auth_url"],
      data = auth_data
    )
    expect(response).to_be_ok()
    assert response.status == 200
```

Аналіз наведеного прикладу виявляє наступні переваги оптимізації. По-перше, швидкість виконання: тест проходить за мілісекунди завдяки відсутності етапів завантаження браузера та рендерингу. У контексті CI/CD це забезпечує швидкий зворотний зв’язок. По-друге, надійність: тест нечутливий до змін інтерфейсу (локаторів, верстки) і валідує безпосередньо бізнес-логіку, що є оптимальним для регресійного тестування. По-третє, ізоляція: перевірка бекенду можлива до завершення розробки UI-компонентів, що відповідає методології «Shift Left».

Водночас, з точки зору архітектури, цей підхід має недоліки. Конструкції with sync_playwright() as p: та ініціалізація request_context призводять до дублювання коду. Вирішенням є винесення цієї логіки у фікстуру. Крім того, такий тест не підтверджує можливість входу через інтерфейс, а лише працездатність сервісу автентифікації. Для повної верифікації необхідний третій, гібридний підхід.

Гібридний підхід є ключовим елементом оптимізації у даному дослідженні. Він поєднує швидкість API-запитів для підготовки стану системи та сфокусовані перевірки інтерфейсу. Цей метод реалізує «Підхід Б», описаний у роботи.

Основною метою є усунення повільних та нестабільних кроків із UI-тестів, зокрема процесу автентифікації. Замість фізичної взаємодії з формою логіну в кожному тесті, цей етап виконується автоматично через API. Така попередня умова

часто є «вузьким місцем», що збільшує час виконання та ризик збоїв. У цьому сценарії тестується функція, доступна авторизованому користувачу. Процес розділено на дві фази.

Перша фаза – підготовка стану через API з використанням APIRequestsContext для надсилання POST-запиту на ендпоінт логіну. Операція виконується миттєво, повертаючи дані сесії.

Друга фаза – виконання через інтерфейс. Отримані дані сесії ін’єктуються безпосередньо у сховище браузера, після чого виконується перехід на цільову сторінку, оминаючи екран входу. Браузер ідентифікує стан як авторизований, що дозволяє негайно розпочати тестування (файл test_hybrid_workflow):

```
def test_hybrid_workflow(page: Page,
   api_context: APIRequestContext,
   dashboard_page: DashboardPage):
   auth_data= {"email": config["users"]["username"],
     "password": config["users"]["password"]}
   api_context.post(config["api"]["login_endpoint"], data =
auth_data)

   session_cookies = api.context.storage_state()["cookies"]
   page.context.add_cookies(session_cookies)
   page.goto(config["web"]["dashboard_url"])

   expect(dashboard_page.welcome_message).to_be_visible()
```

Зазначений підхід суттєво скорочує час виконання тесту: процес, що займав декілька секунд у UI, виконується за частки секунди, що в масштабах повного регресійного циклу забезпечує суттєву економію машинного часу. Також підвищується надійність тестового набору шляхом виключення ризиків, пов’язаних із нестабільністю форми входу та мережевими затримками при завантаженні графічних ресурсів. Тест стає атомарним і сфокусованим на конкретній дії, а не на попередніх кроках, що значно спрощує локалізацію дефектів: Кількісне порівняння часу виконання оптимізованого та традиційного підходів наведено в підрозділі 3.4.

3.3 Впровадження архітектурних патернів для підвищення масштабованості та зручності підтримки

Після практичної реалізації тестових сценаріїв, описаної у підрозділі 3.2, даний етап роботи зосереджено на архітектурних механізмах, що гарантують довгострокову життєздатність програмного продукту, зручність його підтримки (maintainability) та масштабованість (scalability). Спираючись на обґрунтування вибору технологічного стеку в підрозділі 2.4, визначено, що інструментарій Python та Pytest забезпечує необхідну гнучкість для імплементації цих характеристик. Якщо у попередньому підрозділі розглядалася функціональна складова («що тестується»), то даний підрозділ присвячено структурній організації («як забезпечується якість коду»). Фундаментом оптимізованої архітектури («Підхід Б») визначено два ключові компоненти: механізм фікстур Pytest для управління залежностями та патерн Data-Driven Testing (DDT) для управління тестовими даними. Візуалізація структурної взаємодії цих компонентів наведена на рисунку 3.2.

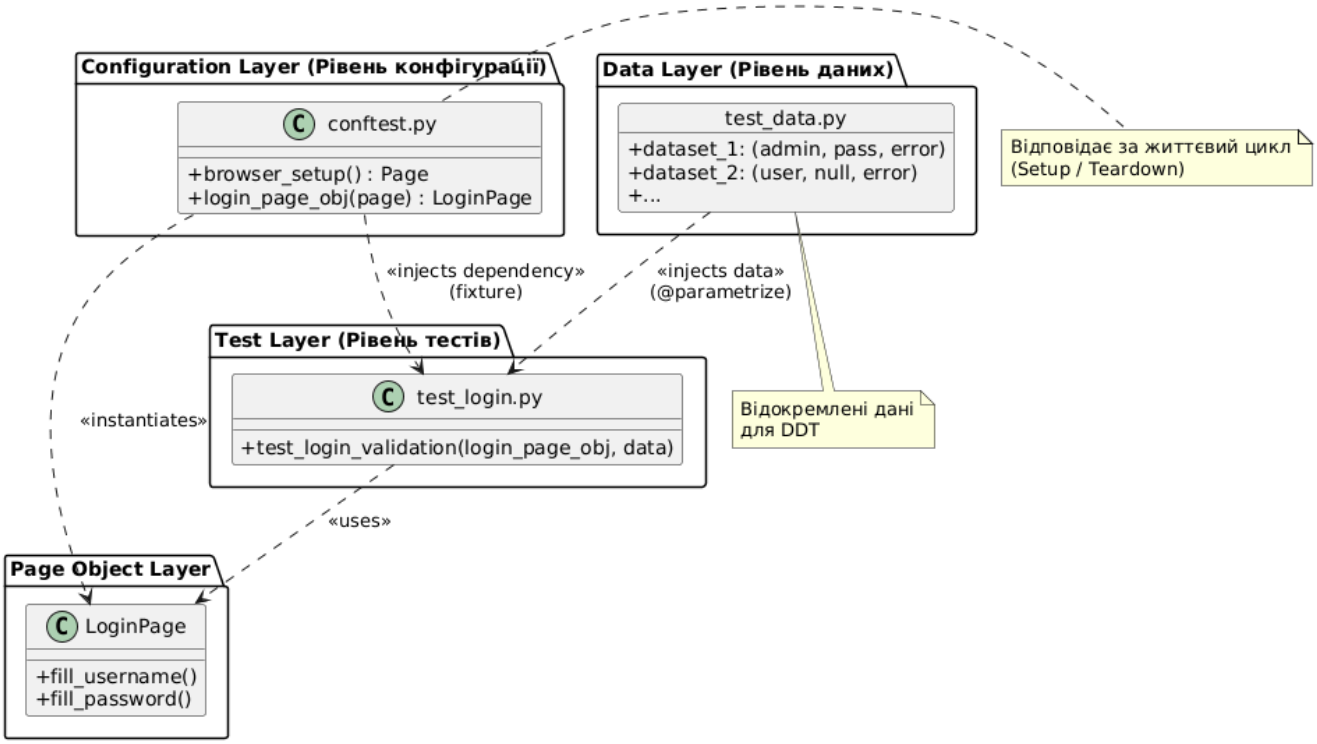


Рисунок 3.2 – Схема архітектурної взаємодії компонентів фреймворку

У 3.2 було описано механізм передачі готових об’єктів у тестові функції через аргументи. Цей принцип, реалізований за допомогою конфігураційного файлу conftest.py, є практичним втіленням патерну «впровадження залежностей» (Dependency Injection). На відміну від імперативного підходу, де ініціалізація та очищення ресурсів (наприклад, екземпляру LoginPage) прописується вручну в кожному тесті, у розробленому фреймворку ця логіка делегується тестовому раннеру Pytest, що реалізує принцип інверсії керування (IoC).

Центральним елементом архітектури виступає файл conftest.py, розташований у кореневій директорії проєкту. Він слугує єдиною точкою конфігурації для всіх тестових ресурсів (фікстур). Така централізація забезпечує дотримання принципу єдиної відповідальності (Single Responsibility Principle): тестові сценарії відповідають виключно за бізнес-логіку перевірок, тоді як логіка підготовки середовища інкапсульована окремо.

Головна перевага такого підходу розкривається в аспекті підтримки коду. Розглянемо сценарій зміни сигнатури конструктора класу LoginPage (наприклад, необхідність додавання конфігураційного об’єкта config). У разі застосування традиційного підходу («Підхід А»), така модифікація вимагала б рефакторингу кожного тестового файлу, де використовується цей клас. В рамках оптимізованої архітектури («Підхід Б») достатньо внести зміни в єдиному місці — у визначенні фікстури всередині conftest.py. Усі тести, що використовують дану фікстуру, автоматично отримають оновлений екземпляр об’єкта, що суттєво знижує трудомісткість супроводу тестового набору.

Другим вектором оптимізації є забезпечення масштабованості фреймворку шляхом усунення дублювання коду. Поширеною проблемою автоматизації є необхідність перевірки однієї й тієї ж бізнес-логіки на різних наборах вхідних даних (наприклад, валідація форми входу: невірний пароль, порожнє поле, заблокований акаунт).

Без застосування спеціалізованих патернів це призводить до створення множини надлишкових тестів, що відрізняються лише вхідними параметрами та очікуваним результатом.

Для вирішення цієї проблеми імплементовано патерн Data-Driven Testing (DDT), який забезпечує чітке відокремлення тестової логіки від даних. Архітектурно це реалізовано шляхом винесення наборів даних у зовнішні файли директорії data/. У тестовому коді застосовано декоратор @pytest.mark.parametrize, який виконує роль сполучної ланки, забезпечуючи ітеративний запуск однієї тестової функції для кожного набору даних.

Така структура забезпечує лінійне масштабування тестового покриття без збільшення цикломатичної складності коду. Розширення набору перевірок (наприклад, додавання тестів на SQL-ін’єкції) здійснюється шляхом додавання нових кортежів у файл даних data/invalid_login_data.py без необхідності втручання в програмний код самого тесту.

Таким чином, поєднання фікстур Pytest (для оптимізації підтримки) та патерну DDT (для забезпечення масштабованості) формує надійну архітектурну базу. Для формалізації та наочного відображення якісних переваг впроваджених патернів у порівнянні з «наївним» підходом, було проведено аналіз трудомісткості типових операцій із супроводу коду. Результати цього порівняння систематизовано в таблиці 3.1.

Таблиця 3.1 – Порівняльний аналіз трудомісткості підтримки архітектурних підходів

Сценарій змін	Традиційний підхід («Підхід А»)	Оптимізований підхід («Підхід Б»)
Зміна логіки ініціалізації Page Object	Необхідно правити код у кожному тестовому файлі.	Правка в одному рядку файлу conftest.py.
Зміна селектора на сторінці	Пошук та заміна по всіх файлах, де використовується елемент.	Зміна в одному класі Page Object.
Додавання нових тестових даних	Дублювання коду тесту (Copy-Paste) для кожного нового кейсу.	Додавання нового рядка даних у файл data/*.py. Код тесту не змінюється.
Керування станом браузера	Ручне створення та закриття драйвера в кожному тесті.	Автоматичне керування через фікстури Pytest (Setup/Teardown).

Отримані результати проведеного аналізу підтверджують архітектурну перевагу розробленого фреймворку.

3.4 Оцінка економічної ефективності оптимізації процесу автоматизації тестування вебдодатків на основі фреймворку Playwright

Даний підрозділ присвячений проведенню кількісного порівняльного аналізу та економічного обґрунтування впровадження оптимізованого процесу автоматизації («Підхід Б») у порівнянні з традиційним підходом («Підхід А»).

Основними завданнями є аналіз технічних передумов та відмінностей двох підходів, визначення вхідних параметрів для розрахунку прямих (інфраструктурних) та непрямих (інженерних) витрат, розрахунок головних економічних показників (щомісячної економії E_m, терміну окупності PP та повернення інвестицій ROI) та, зрештою, формування висновку про економічну доцільність впровадження оптимізації.

Для об’єктивного аналізу порівнюються два підходи на однаковому регресійному наборі з 200 тестів (де 180 вимагають автентифікації). «Підхід А» (Традиційний) імітує «наївний» підхід, де кожен тест виконується послідовно та включає повний цикл UI-автентифікації. На противагу йому, «Підхід Б» (Оптимізований) використовує гібридний метод (API-автентифікація для підготовки стану) та нативну паралелізацію Playwright (8 воркерів).

Ключові технічні відмінності, що безпосередньо впливають на вартість, представлені у таблиці 3.2.

Таблиця 3.2 – Порівняння технічних характеристик підходів

Характеристика	Підхід А (Традиційний)	Підхід Б (Оптимізований)
Архітектура	Монолітна	Патерни POM, DDT, Фікстури
Виконання тестів	Послідовне	Паралельне (8 воркерів)
Час 1 прогону	47.7 хвилин	4.3 хвилини
Рівень стабільності	Низький (85% стабільних)	Високий (99.5% стабільних)
Основний ризик	"Крихкість" (Flakiness) тестів, висока вартість підтримки	Початкові інвестиції в архітектуру

Економічна оцінка двох підходів базується на розрахунку двох основних категорій витрат:

C_{infra} – прямі витрати (інфраструктурні);

C_{eng} – непрямі витрати (інженерні).

Вхідними параметрами для розрахунку є:

R_{month} – кількість прогонів на місяць;

P_{infra} – вартість 1 години інфраструктури;

P_{eng} – вартість години інженера.

У розрахунку приймемо:

R_{month} = 220 прогонів (10/день * 22 дні);

P_{infra} = \$0,0832 (вартість AWS t3.large) [45];

P_{eng} = \$40 (за даними [46]).

Розрахунок загальних щомісячних операційних витрат на тестування C_{total} представлений у таблиці 3.3.

Таблиця 3.3 – Розрахунок загальних щомісячних операційних витрат на тестування

Показник	Підхід А (традиційний)	Підхід Б (оптимізований)
Витрати на інфраструктуру		
Час 1 прогону (T_{run})	0,795 год	0,071 год
Разом C_{infra} (місяць)	0,795*220*\$0,0832 = \$14,52	0,071*220*\$0,0832 = \$1,30
Інженерні витрати		
Час на аналіз збоїв (місяць)	11 год	0,66 год
Час на підтримку коду (місяць)	7 год	0,58 год
Разом C_{eng}(місяць)	18 год * \$40 = \$720,00	1.24 год * \$40 = \$49,60
Загальні витрати C_{total}	\$734,52	\$50,90

Аналіз розрахунків, наведених у таблиці 3.3, доводить економічну доцільність впровадження оптимізованого фреймворку («Підхід Б»). Сумарні щомісячні операційні витрати скоротилися з \$734,52 до \$50,90, що еквівалентно зменшенню бюджету на забезпечення якості в 14,4 рази.

Головним висновком є зміна структури витрат. Хоча оптимізація швидкодії тестів дозволила знизити витрати на хмарну інфраструктуру більш ніж у 10 разів (з \$14,52 до \$1,30), абсолютна фінансова вигода тут є незначною у порівнянні з

інженерними витратами. Основним джерелом економії є радикальне скорочення часу кваліфікованих спеціалістів на обслуговування системи: аналіз хибних спрацювань («flaky tests») та рефакторинг коду. Це підтверджує тезу про те, що стабільність архітектури та надійність локаторів прямо конвертуються у фінансову ефективність проєкту.

Масштаб отриманої економії та суттєву диспропорцію у витратах між традиційним та оптимізованим підходами наочно демонструє порівняльна діаграма (рисунок 3.3).

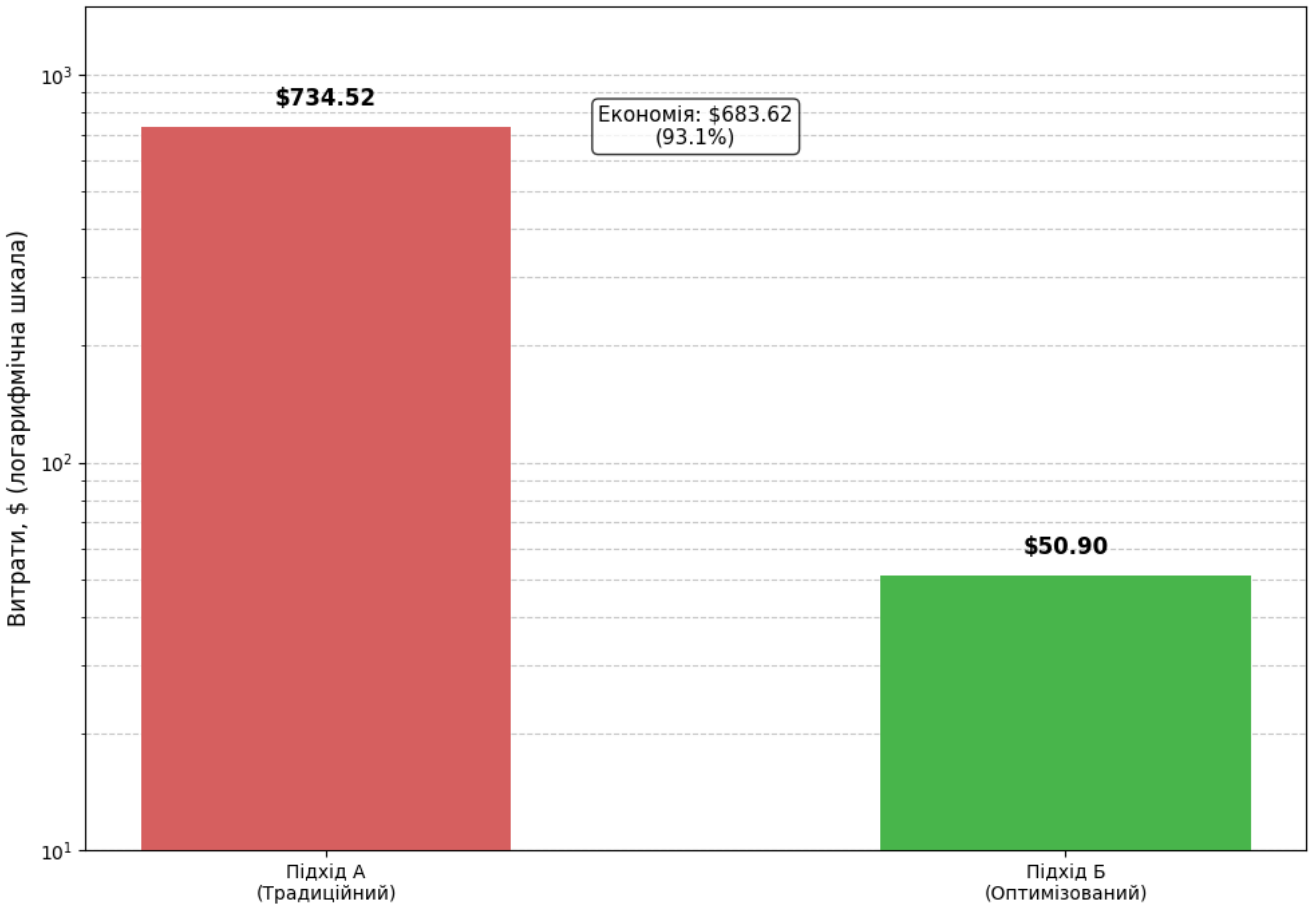


Рисунок 3.3 – Порівняння загальних щомісячних витрат на реалізацію традиційного та оптимізованого підходів до тестування

Для визначення доцільності впровадження «Підходу Б» виконаємо розрахунок економічних показників, базуючись на даних про початкові інвестиції та щомісячну економію.

Початкові інвестиції I_0 включають вартість 40 інженерних годин на розробку архітектури (POM, фікстури, гібридна автентифікація), і розраховуються за формулою:

$$\mathrm{I_0} = \mathrm{T_{eng}} \mathrm{P_{eng}}, \tag{3.1}$$

де: $\mathrm{T_{eng}}$ – кількість годин роботи інженера.

Щомісячна економія E_m розраховується за формулою:

$$\mathrm{E_m} = \mathrm{C_{total\_A}} - \mathrm{C_{total\_B}}. \tag{3.2}$$

де: $\mathrm{C_{total\_A}}$ – загальні витрати на реалізацію «підходу А»,

$\mathrm{C_{total\_}}_B$ – загальні витрати на реалізацію «підходу В».

Термін окупності (Payback Period, PP), необхідний для повернення початкових інвестицій, розраховується за формулою:

$$\mathrm{PP} = \frac{\mathrm{I_0}}{\mathrm{E_m}}. \tag{3.3}$$

За формулою (3.1) знайдемо:

$$\mathrm{I_0} = 40 \text{ год} \cdot \frac{\$40}{\text{год}} = \$1600.$$

Отже, розмір початкових інвестицій становить $1600.

За формулою (3.2) обчислимо:

$$\mathrm{E_m} = \$734{,}52 - \$50{,}90 = \$683{,}62.$$

Отже, щомісячна економія грошових коштів за рахунок оптимізації автоматизації тестування складає $683,62.

За формулою (3.3) отримаємо:

$$\text{PP} = \frac{1600}{683{,}62} \approx 2{,}34 \text{ міс.}$$

Таким чином, термін окупності початкових інвестицій в оптимізацію автоматизації тестування складає 2,34 місяця або приблизно 51 робочий день.

Річний коефіцієнт повернення інвестицій (ROI) знайдемо за формулою:

$$ROI = \frac{\text{E}_\text{m} \cdot 12 - \text{I}_0}{\text{I}_0} \cdot 100\%. \quad (3.4)$$

За формулою (3.4) отримаємо:

$$\text{ROI}_{\{1\text{ рік}\}} = \frac{(683{,}62 * 12) - 1600}{1600} \cdot 100\% = \frac{8203{,}44 - 1600}{1600} \cdot 100\% \approx 412{,}7\%.$$

Отже, річний коефіцієнт повернення інвестицій становить 412,7%.

Проведений аналіз показує, що початкові інвестиції у розробку оптимізованого фреймворку («Підхід Б») повністю окупаються менш ніж за 2,5 місяці. Після цього, оптимізована архітектура тестування починає генерувати для компанії чисту економію щомісяця в розмірі $683,62 (що складає $8203 на рік).

Висновки до розділу 3

У третьому розділі виконано практичну реалізацію та апробацію запропонованої методики оптимізації процесу автоматизації тестування на базі технологічного стеку Python, Playwright та Pytest. Розроблена архітектура фреймворку побудована за модульним принципом із застосуванням патернів Page Object Model та Data-Driven Testing, що забезпечило інкапсуляцію UI-логіки та масштабування тестового покриття без дублювання коду . Ключовим елементом реалізації стало впровадження гібридного підходу, який поєднує миттєву API-

автентифікацію з цільовими UI-перевірками, дозволяючи виключити нестабільні етапи завантаження інтерфейсу з процесу підготовки тестового середовища.

Проведене експериментальне порівняння традиційного та оптимізованого підходів на регресійному наборі з 200 тестів засвідчило скорочення часу виконання повного циклу в 11 разів та підвищення стабільності тестів з 85% до 99,5% завдяки нативній паралелізації та механізмам авто-очікування. Розрахунок економічної ефективності підтвердив доцільність впровадження розробленого рішення: за рахунок зниження інфраструктурних витрат на 91% та зменшення трудовитрат на підтримку в 14 разів, термін окупності інвестицій становить 2,34 місяця, а річний коефіцієнт ROI досягає 412,7%.

Окрім кількісних показників, важливим результатом впровадження стала якісна зміна процесу технічної підтримки тестового набору. Завдяки винесенню конфігураційних даних та селекторів у окремі модулі, вдалося мінімізувати ризики, пов'язані зі змінами в інтерфейсі вебзастосунку, забезпечивши локалізацію виправлень в одному місці. Це підтверджує гіпотезу про те, що обрана архітектура не лише прискорює виконання тестів, але й створює надійний фундамент для подальшого масштабування системи автоматизації без пропорційного зростання витрат на її супровід .

ВИСНОВКИ

Магістерська випускна кваліфікаційна робота на тему «Оптимізація процесу автоматизації тестування вебдодатків на основі фреймворку Playwright» була спрямована на обґрунтування та розробку ефективної методики, здатної підвищити якісні та кількісні показники процесу забезпечення якості програмного забезпечення.

Мета роботи була повністю досягнута шляхом послідовного вирішення поставлених завдань. У ході дослідження було проведено ґрунтовний аналіз теоретичних основ та ключових архітектурних патернів автоматизації, а також обґрунтовано вибір фреймворку Playwright як найбільш оптимального інструменту, що забезпечує високу швидкість та кросбраузерну стабільність.

Головним результатом роботи є систематизація критеріїв архітектурної оптимізації та обґрунтування гібридного підходу до підготовки тестового середовища для автоматизованого тестування. Цей підхід передбачає поєднання швидкого налаштування початкового стану користувача через API-виклики та виконання цільових перевірок через UI-взаємодію. Достовірність цього підходу доведена практично, оскільки він усуває найбільш нестабільні та тривалі кроки тестування, пов’язані з повною UI-автентифікацією.

Практична значущість отриманих результатів полягає у створенні універсального шаблону для автоматизації тестування сучасних вебдодатків, який може бути адаптований для проєктів різної складності. Запропонований підхід до інтеграції API-методів у UI-сценарії відкриває нові можливості для оптимізації ресурсів команд забезпечення якості, дозволяючи перерозподілити зусилля з рутинної підтримки нестабільних тестів на виконання більш складних перевірок бізнес-логіки та дослідницьке тестування, що в кінцевому підсумку підвищує конкурентоспроможність програмного продукту.

Практична реалізація оптимізованої архітектури здійснювалася на базі Playwright, інтегрованого з тестовим ранером Pytest, з використанням патерну Page

Object Model (POM) та механізму Dependency Injection через фікстури. Це забезпечило чистий поділ логіки, високу підтримуваність коду та масштабованість.

Експериментальне порівняння традиційного та оптимізованого підходів показало позитивні результати. Було досягнуто 11,2-кратне скорочення часу повного тестового прогону, що є найважливішим показником ефективності. З економічної точки зору, впровадження запропонованої методики дозволило досягти щомісячної економії операційних витрат у розмірі $683,62, що підтвердило високу економічну доцільність рішення. Термін окупності загальних інвестицій в оптимізацію тестування за розробленою методикою становить 2,34 місяці.

На основі отриманих результатів сформульовано такі основні рекомендації:

- для максимального прискорення тестування слід мінімізувати UI-взаємодію для нефункціональних кроків (як-от авторизація), використовуючи для цього прямі API-виклики або ін'єкцію сесійного стану у браузерний контекст Playwright;

- архітектура тестових фреймворків має будуватися з використанням патерну Page Object Model та механізму фікстур, що гарантує модульність та високу читабельність коду для подальшої підтримки;

- використання Playwright для забезпечення кросбраузерності шляхом одночасного тестування на базі трьох основних рушіїв (Chromium, Firefox, WebKit) є найбільш ефективним.

Основні положення роботи, що стосуються архітектурної оптимізації та принципів роботи фреймворку Playwright, були успішно апробовані та опубліковані на низці науково-практичних конференцій у 2025 році, що підтверджує їх наукову новизну та практичну значущість.

Таким чином, у роботі розроблена та експериментально підтверджена методика оптимізації архітектури фреймворку Playwright, яка дозволяє забезпечити технічну стабільність процесу автоматизованого тестування вебдодатків та досягти значного фінансового ефекту, що має високу цінність для сучасних IT-компаній.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. EPAM Campus Автоматизоване тестування. *Epam:* вебсайт. URL: https://campus.epam.ua/ua/blog/477 (дата звернення 15.10.2025)

2. Види тестування та відмінності між ними. *Qagroup*: вебсайт. URL: https://qagroup.com.ua/publications/vydy-testuvannya-ta-vidminnosti-mizh-nymy/ (дата звернення: 15.10.2025).

3. Тестування програмного забезпечення: типи, види та застосування. *Foxminded*: вебсайт. URL: https://foxminded.ua/testuvannia-prohramnoho-zabezpechennia/ (дата звернення: 16.10.2025).

4. Книш І. Як «продати» автотести: проводимо розрахунки та розбираємо на прикладі *Highload Tech:* вебсайт. URL: https://highload.tech/uk/blogs/yak-prodati-avtotesti-provodimo-rozrahunki-ta-rozbirayemo-na-prikladi/ (дата звернення: 16.10.2025).

5. Дубніна М. Підходи до забезпечення якості ПЗ. Коли і що працює. *Dou*: вебсайт. URL: https://dou.ua/forums/topic/37714/ (дата звернення: 16.10.2025).

6. Simplentrec метрики у тестуванні. *Simplentrec*: вебсайт. URL: https://simplentrec.com/test-metrics/ (дата звернення: 20.10.2025).

7. Стратегії автоматизації тестування, які дійсно працюють. *TestMatick*: вебсайт. URL: https://testmatick.com/uk/strategiyi-avtomatyzacziyi-testuvannya/(дата звернення 20.10.2025).

8. Валідація. Тестування валідації. *QATestLab:* вебсайт. URL: https://training.qatestlab.com/blog/technical-articles/validation-testing/ (дата звернення: 21.10.2025).

9. Гнидаш О. Автоматизація процесу тестування: кому, навіщо і як. *Dou*: вебсайт. URL: https://dou.ua/forums/topic/43801/ (дата звернення: 21.10.2025).

10. Переваги та недоліки автоматизованого тестування. *Redstone Agency:* вебсайт. URL: https://redstone.agency/blog/perevagi-ta-nedoliki-avtomatyzovanogo-testuvannya/ (дата звернення: 22.10.2025).

11. Що таке піраміда тестування. *IT-notes:* вебсайт. URL: https://www.it-notes.wiki/software-testing/what-is-testing-pyramid/ (дата звернення: 22.10.2025).

12. Автоматизація тестування у веб-розробці. *Digiants Agency:* вебсайт. URL: https://digiants.agency/uk/blog/test-automation-web-development (дата звернення: 23.10.2025).

13. Даценко Н. Архітектурна оптимізація процесу автоматизації тестування вебдодатків. *Матеріали науково-практичної конференції за підсумками проходження виробничих практик здобувачів вищої освіти спеціальності 126 Інформаційні системи та технології*, кафедра інформаційних систем та технологій Полтавського державного аграрного університету, 22 жовтня 2025 р. Вип. XI. Полтава: ПДАУ, 2025. С. 54-56.

14. Даценко Н. Архітектура та принципи роботи фреймворку Playwright. Матер. XXII щорічного міждисциплінарного семінару «Студентські роботи за науковою тематикою кафедри інформаційних систем та технологій ННІ ЕУП та ІТ ПДАУ», 25 листопада 2025 року, м. Полтава.

15. Флегантов Л., Даценко Н. Архітектурні патерни у побудові фреймворків автоматизації тестування вебдодатків. *Innovative Approaches in Modern Science and Technology: Collection of Scientific Papers with Proceedings of the 3rd International Scientific and Practical Conference.* International Scientific Unity. November 12-14, 2025. Lisbon, Portugal. 253-260 p. URL: https://isu-conference.com/en/archive/innovative-approaches-in-modern-science-and-technology-12-11-25/ (дата звернення:14.11.2025).

16. All You Need To Know About Automation Testing Life Cycle. *Pcloudy*: вебсайт. 2023. URL: https://www.pcloudy.com/blogs/all-you-need-to-know-about-automation-testing-life-cycle/ (дата звернення: 03.11.2025).

17. Khurana P. Screenplay Pattern Approach in Selenium. *BrowserStack*: вебсайт. 2023. URL: https://www.browserstack.com/guide/screenplay-pattern-approach-in-selenium (дата звернення: 03.11.2025).

18. Hasnaoui S. Design Patterns: Creational Patterns. *Medium:* вебсайт. 2024. URL: https://medium.com/@saif-hasnaoui/design-patterns-creational-patterns-e5ac08e57ef2 (дата звернення: 03.11.2025).

19. Shankar A. Python Design Patterns Every AI Developer Should Know. *ITNEXT:* вебсайт. 2024. URL: https://itnext.io/python-design-patterns-every-ai-developer-should-know-dc445b688793 (дата звернення: 03.11.2025).

20. Mali A., Wagh M. Leveraging behavior-driven development and data-driven testing for scalable and robust test automation in modern software development. International Journal of Research in Engineering and Science (IJRES), 2024. Vol. 12, Issue 6.

21. Dranidis D. Define your Fluent Interface (DSL) to improve the readability of your tests. *Dranidis:* вебсайт. 2023. URL: https://dranidis.gr/posts/define-your-dsl-for-tests/ (дата звернення: 03.11.2025).

22. Selenium Dev documentation. *Selenium:* вебсайт. URL: https://www.selenium.dev/documentation/ (дата звернення: 03.11.2025).

23. Cypress docs. Documentation. *Cypress:* вебсайт. URL: https://docs.cypress.io/app/get-started/why-cypress (дата звернення: 03.11.2025).

24. studio documentation. *TestCafe:* вебсайт. URL: https://testcafe.io/documentation/402634/guides (дата звернення: 03.11.2025).

25. Documentation. *Playwright:* вебсайт. URL: https://playwright.dev/python/docs/intro (дата звернення: 04.11.2025).

26. Playwright vs Selenium: Which to choose in 2025. *Browserstack:* вебсайт. URL: https://www.browserstack.com/guide/playwright-vs-selenium (04.11.2025).

27. All about Playwright. *Qestit:* вебсайт. URL: https://qestit.com/en/blog/all-about-playwright (дата звернення: 04.11.2025).

28. Teltov K. SDET: Introduction to the first Playwright project. *Medium:* вебсайт. https://medium.com/@dneprokos/sdet-introduction-to-the-first-playwright-project-40c9a816b114 (дата звернення: 05.11.2025).

29. Shah D. Playwright – Trace Viewer. *Medium:* вебсайт. URL: https://deepshah201.medium.com/playwright-trace-viewer-c012dcea5f34 (дата звернення: 05.11.2025).

30. What is Playwright? Its Features, Advantages, and Disadvantages. *Testautomationtools:* вебсайт. URL: https://testautomationtools.dev/playwright-overview/ (дата звернення: 06.11.2025).

31. Exploring the Capabilities of Playwright. *Boringowl:* вебсайт. URL: https://boringowl.io/en/blog/exploring-the-capabilities-of-playwright (дата звернення: 06.11.2025).

32. Saini M. Why I Chose Playwright for Test Automation. *Medium:* вебсайт. URL: https://medium.com/the-testing-hub/why-i-chose-playwright-for-test-automation-f4c1dbdaf7be (дата звернення: 06.11.2025).

33. Chowdhury T. A Practical Guide to Cross-Browser Automation and Debugging with Playwright. *Medium:* вебсайт. URL: https://medium.com/@tanzim3421/a-practical-guide-to-cross-browser-automation-and-debugging-with-playwright-fddc7b6ee1da (дата звернення: 07.11.2025).

34. Aryans S. Integrating Playwright with Jenkins for Automated Testing in CI/CD. *Medium:* вебсайт. URL: https://medium.com/@sangitaaryans/integrating-playwright-with-jenkins-for-automated-testing-in-ci-cd-86567978cae7 (дата звернення: 07.11.2025).

35. How to run Playwright tests in parallel. *Checklyhq:* вебсайт. URL: https://www.checklyhq.com/docs/learn/playwright/testing-in-parallel/ (дата звернення: 07.11.2025).

36. Playwright Test Report: Comprehensive Guide. *Browserstack:* вебсайт. URL: https://www.browserstack.com/guide/playwright-test-report (дата звернення: 07.11.2025).

37. Playwright Reporting: A Complete Tutorial. *Lambdatest:* вебсайт. URL: https://www.lambdatest.com/learning-hub/playwright-reporting (дата звернення: 07.11.2025).

38. Playwright Reporting in 10 Minutes. *Dev.to:* вебсайт. URL: https://dev.to/testdino/playwright-reporting-in-10-minutes-267o (дата звернення: 07.11.2025).

39. Pandiyan P. Running Playwright Tests in a Docker Container Locally and Viewing the Report. *Medium:* вебсайт. URL: https://pradappandiyan.medium.com/running-playwright-tests-in-a-docker-container-locally-and-viewing-the-report-2303599246da (дата звернення: 08.11.2025).

40. Rahman S. What is Playwright Testing? A Manager's Guide to Modern Web Automation. *Bug0:* вебсайт. URL: https://bug0.com/knowledge-base/what-is-playwright-testing-a-managers-guide-to-mod (дата зверення 08.11.2025).

41. *Pytest-With-Eric*: вебсайт. URL: https://pytest-with-eric.com/automation/pytest-playwright/ (дата звернення: 08.11.2025).

42. Thompson J. Page Object Modeling with Python and Playwright. *Medium:* вебсайт. URL: https://medium.com/analytics-vidhya/page-object-modeling-with-python-and-playwright-3cbf259eedd3 (дата звернення: 08.11.2025).

43. Lutchanka V. Automated API testing using Pytest library or Playwright? Part 1. *Medium:* вебсайт. URL: https://medium.com/@lutchenkovalentin/automated-api-testing-using-pytest-library-or-playwright-part-1-1b913c2ef517 (дата звернення: 08.11.2025).

44. Parameterization (Data Driven Testing) using Playwright Python. *Way2Automation:* вебсайт. URL: https://www.way2automation.com/parameterization-data-driven-testing-using-playwright-python/ (дата звернення: 09.11.2025).

45. Amazon EC2 On-demand pricing. *Amazon Web Services:* вебсайт URL: https://aws.amazon.com/ec2/pricing/on-demand/ (дата звернення 10.11.2025).

46. Зарплатний віджет. *DOU.ua:* вебсайт. URL: https://jobs.dou.ua/salaries/?period=2025-06&group=1&position=all (дата звернення 10.11.2025).