Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента *Ларикової Марії Володимирівни*
(ПІБ)

академічної групи *122м-22-2*
(шифр)

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

освітньої програми *«122 Комп'ютерні науки»*
(назва освітньої програми)

на тему: *«Дослідження ефективності процесу наскрізного тестування веб-застосунків сучасними засобами автоматизації у багатопоточному середовищі»*

__________________ *М.В. Ларикова*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
кваліфікаційної роботи	*Проф. Мороз Б.І.*			

Рецензент	*Доц. Желдак Т.А.*			

Нормоконтролер	*Доц. Гуліна І.Г.*			

Дніпро
2023

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)
_______________ М.О. Алексєєв
(підпис) (прізвище, ініціали)
«___» ___________ 2023 року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

студенту *122м-22-2* *Лариковій Марії Володимирівні*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *«Дослідження ефективності процесу наскрізного тестування веб-застосунків сучасними засобами автоматизації у багатопоточному середовищі»*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від __.__.2023 р. № _______

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт дослідження – процес наскрізного тестування у багатопоточному середовищі.

Предмет дослідження – ефективність процесу наскрізного тестування веб-застосунків сучасними засобами автоматизації у багатопоточному середовищі.

Мета НДР – визначення чинників для підвищення ефективності процесу наскрізного тестування у багатопоточному середовищі.

Вихідні дані для проведення роботи – теоретичні та експериментальні дослідження, дані отримані під час використання сучасних засобів автоматизації на реальних проєктах.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень та очікуваних результатів полягає у дослідженні ефективності процесу наскрізного тестування у багатопоточному середовищі за різних умов і визначенні параметрів для прискорення цього процесу.

Практична цінність отриманих результатів полягає в виявленні найбільш оптимального інструменту для проведення наскрізного тестування в умовах паралельного запуску тестів та установленні умов для прискорення процесу тестування.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень повинні бути представлені у формі, що дозволяє їх безпосереднє використання в реальних проєктах, без необхідності додаткового аналізу інструментів та визначення чинників для оптимізації швидкості виконання процесу наскрізного тестування за допомогою сучасних засобів автоматизації.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	Строки виконання робіт (початок – кінець)
Розробка проєкту з наскрізного тестування з використанням Playwright для автоматизації тестування в багатопоточному середовищі	01.09.2023-01.10.2023
Розробка проєкту з наскрізного тестування з використанням Selenium для автоматизації тестування в багатопоточному середовищі	01.10.2021-14.11.2023
Аналіз отриманих результатів внаслідок зміни параметрів запуску розроблених проєктів	15.11.2021-10.12.2023

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації результатів роботи очікується позитивним завдяки скороченню часу проведення процесу наскрізного тестування. Це надає конкурентну перевагу тестованому продукту, прискорюючи постачання його нових версій кінцевому користувачеві з меншим ризиком виявлення дефектів на етапі використання. Ці покращення суттєво знижують фінансові витрати на тестування та, відповідно, на процес розробки нового функціоналу. Крім того, вони сприяють зменшенню ризику дефектів, які можуть призвести до блокування продукту або відмови користувачів від його використання.

Соціальний ефект від реалізації результатів роботи очікується позитивним завдяки підвищенню швидкості проведення наскрізного тестування. Зменшення часу на виконання тих самих завдань працівниками призведе до підвищення продуктивності праці на підприємстві. Крім того, це може призвести до покращення умов праці, оскільки працівникам не доведеться виконувати велику кількість завдань протягом обмеженого часового періоду.

Завдання видав	______________	*Алексєєв М.О.*
	(підпис)	(прізвище, ініціали)
Завдання прийняв до виконання	______________	*Ларикова М.В.*
	(підпис)	(прізвище, ініціали)

Дата видачі завдання: *01.09.2023 р.*

Термін подання кваліфікаційної роботи до ЕК *11.12.2023*

РЕФЕРАТ

Пояснювальна записка: 90 с., 2 додатки, 11 джерел.

Об’єкт досліджень: процес наскрізного тестування у багатопоточному середовищі.

Предмет досліджень: ефективність процесу наскрізного тестування веб-застосунків сучасними засобами автоматизації у багатопотоковому середовищі.

Мета роботи: визначення чинників для підвищення ефективності процесу наскрізного тестування у багатопотоковому середовищі.

Методи дослідження: реалізація тестових проєктів за допомогою різних інструментів, вимірювання часу та аналіз результатів для обох інструментів.

Наукова новизна: вивчення ефективності наскрізного тестування в багатопотоковому середовищі та визначенні параметрів для прискорення.

Практична цінність: вибір найбільш оптимального інструменту та умов для наскрізного тестування у багатопоточному середовищі.

Область застосування: визначення інструментів для автоматизації тестування реальних продуктів з метою підвищення ефективності процесу наскрізного тестування.

Економічний ефект: скорочення часу проведення процесу наскрізного тестування, конкурентна перевага продукту у прискоренні постачання нових версій кінцевому користувачеві з меншим ризиком виявлення дефектів.

Значення роботи та висновки: проведені дослідження відображають, що використання фреймворку Playwright значно зменшує час, необхідний для проведення процесу наскрізного тестування.

Прогнозні припущення про розвиток досліджень: перспективним напрямком розвитку цієї роботи може стати порівняння інших фреймворків.

Список ключових слів: ТЕСТИ, НАСКРІЗНЕ ТЕСТУВАННЯ, АВТОМАТИЗАЦІЯ, БАГАТОПОТОЧНЕ СЕРЕДОВИЩЕ, SELENIUM, PLAYWRIGHT.

ABSTRACT

Explanatory note: 90 pp., 2 appendix, 11 sources.

Research Object: cross-browser testing process in a multi-threaded environment.

Research Subject: the effectiveness of cross-browser testing of web applications using modern automation tools in a multi-threaded environment.

Research Objective: identification of factors to enhance the efficiency of cross-browser testing in a multi-threaded environment.

Research Methods: implementation of test projects using various tools, measurement of time, and analysis of results for each tool.

Scientific Novelty: investigation into the effectiveness of cross-browser testing in a multi-threaded environment and identification of parameters for acceleration.

Practical Value: selection of the most optimal tool and conditions for cross-browser testing in a multi-threaded environment.

Application Area: determining tools for automating testing of real products to increase the efficiency of the cross-browser testing process.

Economic Impact: reduction of time spent on cross-browser testing, providing a competitive advantage by accelerating the delivery of new versions to end-users with a lower risk of defect detection.

Significance and Conclusions: the conducted research demonstrates that the use of the Playwright framework significantly reduces the time required for cross-browser testing.

Future Research Assumptions: a prospective direction for future research could involve comparing other frameworks.

Keywords: TESTING, CROSS-BROWSER TESTING, AUTOMATION, MULTI-THREADED ENVIRONMENT, SELENIUM, PLAYWRIGHT.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

E2E tests – end-to-end tests (наскрізне тестування);

CI/CD – continuous integration and continuous delivery (безперервне розгортання та доставка);

ОС – операційна система;

ШІ – штучний інтелект.

ЗМІСТ

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАННЯ............ 12
1.1 Тенденції розвитку процесу тестування 12
1.2 Стан розвитку автоматизованого тестування 14
1.3 Вибір інструментів для автоматизованого тестування 16
1.4 Визначення стратегії автоматизації процесу наскрізного тестування 18
1.5 Аналіз багатопоточного виконання тестування 19
1.5.1 Переваги 19
1.5.2 Недоліки 19
1.5.3 Визначення стратегії для засобів автоматизації 20
1.6 Висновки першого розділу 21
2 РОЗРОБКА ПРОГРАМНОГО КОДУ ДЛЯ ТЕСТУВАННЯ У БАГАТОПОТОЧНОМУ СЕРЕДОВИЩІ 22
2.1 Структура та архітектурні рішення проєктів 22
2.2 Опис обраних тест-кейсів 24
2.3 Розробка скриптів для паралельного виконання тестів 27
2.4 Висновки другого розділу 30
3 ДОСЛІДНИЦЬКА ЧАСТИНА 31
3.1 Результати виконання тестів у багатопоточному середовищі 31
3.1.1 Виконання тестів зі збоєм у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Windows 31
3.1.2 Виконання тестів зі збоєм у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Mac 34
3.1.3 Виконання тестів без збою у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Windows 37
3.1.4 Виконання тестів без збою у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Mac 39
3.1.5 Виконання тестів зі збоєм у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Windows 40

3.1.6 Виконання тестів зі збоєм у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Mac 42
3.1.7 Виконання тестів без збою у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Windows 44
3.1.8 Виконання тестів без збою у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Mac 46
3.1.9 Виконання тестів зі збоєм у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Windows 48
3.1.10 Виконання тестів зі збоєм у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Mac 51
3.1.11 Виконання тестів без збою у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Windows 53
3.1.12 Виконання тестів без збою у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Mac 55
3.1.13 Виконання тестів зі збоєм у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Windows 58
3.1.14 Виконання тестів зі збоєм у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Mac 60
3.1.15 Виконання тестів без збою у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Windows 62
3.1.16 Виконання тестів без збою у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Mac 64
3.2 Аналіз результатів виконання наскрізного тестування у багатопоточному середовищі 66
ВИСНОВКИ 73
ПЕРЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 74
ДОДАТОК А 76
ДОДАТОК Б 89

ВСТУП

Актуальність теми. Наразі у світі найбільш високий рівень активності у розробці різноманітних програмних застосунків та загострена конкуренція на ринку ІТ. У цьому контексті виникає необхідність вдосконалення процесу контроля якості продуктів для забезпечення їхньої стійкої позиції серед конкурентів. Забезпечення якості великих продуктів із значним обсягом функціональностей за участю значної кількості людських ресурсів не завжди призводить до очікуваного результату.

Для підтримки високого рівня якості продукту виникає потреба у втручанні комп'ютерних засобів, оскільки комп'ютери характеризуються мінімальним рівнем помилок, відсутністю відхилень від плану та чітким виконанням вказівок. Таким чином, розробка програмного забезпечення, яке спрямоване на тестування інших програм, стала основою автоматизованого тестування, широко використовуваного великими корпораціями для забезпечення якості їхніх продуктів. Проте виникає проблема вибору найшвидшого та найоптимальнішого методу автоматизації тестування, яка залишається актуальною на ринку, оскільки швидкість тестування визначає швидкість поставки продукту кінцевому користувачеві.

Об'єкт досліджень: процес наскрізного тестування у багатопоточному середовищі.

Предмет досліджень: ефективність процесу наскрізного тестування веб-застосунків сучасними засобами автоматизації у багатопотоковому середовищі.

Мета дослідження: реалізація тестових проєктів за допомогою різних інструментів, вимірювання часу та аналіз результатів для обох інструментів.

Методи дослідження. Для проведення дослідження ефективності процесу наскрізного тестування в багатопоточному середовищі використовуються методи експерименту. Ці методи включають реалізацію тестових проектів за допомогою різних інструментів, вимірювання часу, витраченого на тестування, та аналіз результатів для обох інструментів. Для

визначення впливу різних інструментів на ефективність тестування застосовуються методи візуалізації, такі як побудова діаграм та графіків.

Новизна отриманих результатів полягає у дослідженні ефективності процесу наскрізного тестування у багатопоточному середовищі за різних умов і визначенні параметрів для прискорення цього процесу.

Практичне значення роботи полягає в виявленні найбільш оптимального інструменту для проведення наскрізного тестування в умовах паралельного запуску тестів та установленні умов для прискорення процесу тестування.

Особистий внесок автора полягає в реалізації проєктів автоматизованого тестування за допомогою інструментів Playwright та Selenium. Також було виконано процес наскрізного тестування у багатопотоковому середовищі за різних умов конфігурації виконання тестів.

Структура та обсяг дипломної роботи. Робота складається з вступу, трьох розділів і висновків. Містить 90 сторінок друкованого тексту, в тому числі 61 сторінку тексту основної частини, 29 рисунків, 36 таблиць, перелік використаних джерел, 2 додатки на 14 сторінках.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАННЯ

1.1 Тенденції розвитку процесу тестування

На сучасному етапі розвитку індустрії програмного забезпечення постає важливе питання забезпечення високої якості продуктів у контексті насиченості ринку та зростаючої конкуренції . Якість програмного продукту визначається не лише його функціональністю, але й критичністю для сфер, де передбачається його використання. Контроль якості стає суттєвим для продуктів, які несуть юридичні та фінансові відповідальності.

Виявлення критичних помилок може визначати не лише фінансові втрати, але й, у випадку медичних чи інших спеціалізованих систем, призводити до серйозних наслідків, включаючи загрозу життю. Нагляд за якістю продукту стає вирішальним у забезпеченні його надійності та безпеки, що стає особливо актуальним у високотехнологічних сферах, таких як медицина та фінанси.

Існує відомий епізод, де недолік у програмному забезпеченні, призначеному для оптимізації системи екстреної медичної допомоги, який виник під час реального експлуатаційного використання, мав надзвичайно серйозні наслідки, призводячи до загибелі багатьох осіб [1]. Зазначена програма, розроблена на замовлення держави, мала сприяти оптимальному вибору шляху для швидкої реакції на виклики шляхом аналізу місцезнаходження виклику та розташування доступних швидких. Під час періоду тестування було виявлено відсутність проблем із використанням системи у реальних сценаріях.

Однак, коли наступив момент реального навантаження та функціонування швидких служб, система несправно обробляла дані, що призвело до неефективного розподілу ресурсів та тривалих затримок у відгуку на виклики. Ця ситуація мала фатальні наслідки, забираючи життя численних осіб та створюючи значний соціальний та юридичний вплив. Намагання

держави компенсувати завдані збитки фінансовими виплатами не повернуло втрачених життів.

З того часу минуло багато років, але проблематика контролю якості в сфері розробки програмного забезпечення залишається належною та актуальною. Основним фактором, який визначає цю актуальність, є особливість людської природи, схильності до помилок. Тестування, визнане як повноцінна професійна сфера, пройшло значний шлях розвитку, у результаті чого виникли різні напрями та спеціалізації.

Початково тестування виконувалось виключно ручним підходом, що передбачало значну кількість людей, які вкладали зусилля за короткий період для перевірки якості програмного продукту перед його релізом. Забезпечувалась перевірка того, чи функціонал, що був присутній у попередніх версіях, залишається стабільним, і чи нововведення відповідають технічним вимогам.

Сфера тестування об'єднує декілька типів тестування [2], серед яких вирізняються:

- функціональне тестування – цей тип має на увазі тестування вхідних та вихідних даних, обробки подій, функцій та інших аспектів, пов'язаних із функціональністю програми.
- нефункціональне тестування – спрямоване на оцінку продуктивності, надійності, безпеки, сумісності та інші характеристики системи, що не пов’язані з функціоналом системи;
- структурне тестування – включає в себе тестування на рівні коду та визначення взаємодії між різними частинами програми;
- тестування змін – зосередження на перевірці впливу внесених змін на функціональність і продуктивність системи.

Унаслідок переходу з ручного тестування до автоматизованого усі ці типи тестування були згодом перекладені на програмне забезпечення. Створення спеціальних інструментів дозволило розробляти програми для автоматизованого тестування, що значно полегшило та прискорило виконання тестів.

Комп'ютери, які виконують програмний код, є менш схильними до помилок порівняно з людьми, а також здатні виконувати процеси швидше. Тим не менш, автоматизоване тестування не охоплює всю функціональність програми, наприклад, тести, які стосуються стилів, розташування елементів або взаємодії з іншими реальними сервісами. Проведення таких тестів вручну коштує менше, ніж їх автоматизація. Кожна процедура автоматизації, насправді, передбачає розробку програмного коду, що призводить до значних витрат часу на його створення, які, проте, в майбутньому компенсуються скороченням часу тестування в більш ніж в 10 разів після завершення написання коду.

1.2 Стан розвитку автоматизованого тестування

На поточний момент автоматизоване тестування перебуває на вершині свого розвитку та користується широкою популярністю. Спостерігається інтенсивний розвиток нових підходів, бібліотек і інструментів, в рамках якого проводяться численні конференції та презентації. У сучасному програмуванні неможливо уявити проєкт, який б функціонував без впроваджених автоматичних тестів у будь-якому їх вигляді.

Розвиток автоматизованого тестування просувається в різних напрямках, аналогічно розвитку програмного забезпечення, оскільки нові програми вимагають нових підходів до тестування [3]. Інтеграція з системами неперервної інтеграції та постачання (CI/CD) є одним із ключових напрямків розвитку, оскільки забезпечує ефективну та швидку поставку змін у продукцію. Автоматизоване виконання тестів у пайплайнах є критичним, оскільки при кожному запиті на злиття у головну гілку необхідно перевіряти функціональність коду та уникати порушень існуючого функціоналу. Автоматизація цього процесу виключає вплив людського фактору, що робить його надійним та ефективним.

Хмарні рішення відіграють важливу роль у процесі автоматизованого тестування, забезпечуючи стабільність продукту на різних пристроях та при

будь-яких конфігураціях. У хмарному середовищі можна налаштувати будь-який контейнер для виконання тестів, таким чином не важливо чи це мобільний пристрій, чи стара версія браузера, чи особливості ОС – усе це можна протестувати без необхідності мати екземпляр реального фізичного девайсу.

У 2022 році значно збільшилась роль штучного інтелекту в усіх сферах людської діяльності, включаючи тестування програмного забезпечення. Технології ШІ стали невід’ємною частиною розробки тестових сценаріїв, виявлення помилок у коді та аналізу результатів. Моніторингові сервіси широко використовують штучний інтелект для обробки великих обсягів даних, що робить можливим отримання висновків про якість продукту в області тестування.

Наприклад, якщо говорити про навантажувальні тести ШІ може допомогти проаналізувати, які сервіси не можуть впоратись з кількістю запитів. У випадку перевірки безпеки продуктів та захисту від зловмисників існують інструменти, які шукають слабкості у коді.

Зростає популярність реклами застосунків як "кишенькових помічників", оскільки майже кожна людина користується мобільним телефоном. Це призводить до збільшення попиту на підтримку мобільних версій та розробку мобільних додатків. Хоча написання автоматизованих тестів для мобільних додатків стало стандартною практикою, існують певні проблеми, такі як різні версії прошивок та розміри екранів.

Це призводить до необхідності запуску тестів на різних конфігураціях, що може спричинити різке збільшення використання хмарних ресурсів або часу виконання тестів. Також важливою проблемою є потреба у запуску тестів на реальних фізичних пристроях, оскільки хмарні ферми пристроїв та емулятори не можуть гарантувати повну достовірність результатів.

Зростання кількості хакерів та зловмисників створює необхідність не лише в перевірці загальної якості продукту, але і в тестуванні безпеки для виявлення потенційних вразливостей. Ця специфічна галузь IT вимагає

використання спеціалізованих інструментів для автоматизації тестування безпеки.

Популярність мікросервісів також ставить перед тестувальниками нові виклики. Необхідність тестування інтеграції між різними сервісами призводить до втрати ізоляції тестів і ускладнює сценарії тест-кейсів в декілька разів. Адаптація тестів до мікросервісного підходу часто включає розподіл тестів між різними репозиторіями.

У зв'язку з цим великі компанії активно використовують автоматизацію в усіх аспектах розробки, де це можливо, для забезпечення якості продукту та виявлення потенційних проблем.

1.3 Вибір інструментів для автоматизованого тестування

Існує велика кількість інструментів, фреймворків та бібліотек для автоматизації тестування для різних вимог та мов програмування [4], ось декілька з них:

- Java:
  - Junit – основний інструмент на Java для будь-якого рівня тестів;
  - TestNG – інструмент для запуску тестів вищого рівня на Java, походить від Junit;
  - Selenium – один з перших фреймворків для написання E2E тестів.
- Python:
  - Pytest – фреймворк для тестування, який забезпечує простий синтаксис та підтримку фікстур;
  - Selenium.
- JavaScript:
  - Cypress – фреймворк для тестування веб-додатків, який дозволяє писати тести відразу в браузері;

- Jest – фреймворк для тестування JavaScript, використовується для тестування React-додатків та інших проєктів;
- Selenium.

Кожен з фреймворків та бібліотек має певну популярність (Рис. 1.1) серед автотестувальників та їх вибір базується на потребах проєкту.

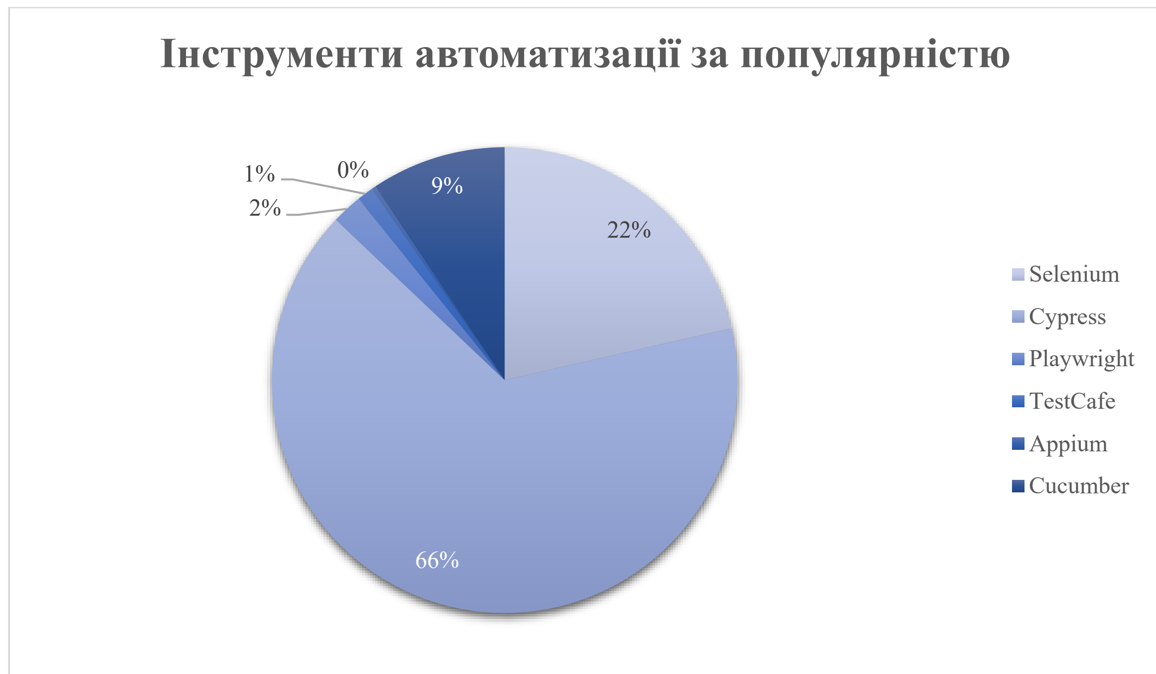


Рисунок 1.1 – Порівняння популярності різних інструментів для автоматизації тестування [5]

Щоб порівняти ефективність виконання тестів у багатопоточному середовищі використаємо Selenium [6] та Playwright [7]. Цей вибір базується на декількох факторах. По-перше, Selenium був «піонером» серед інструментів та один з перших довів усьому світу важливість автоматизації тестів на більш високих рівнях. По-друге, Selenium має велике спільнота, яке робить його досить розвинутим та підтримуваним. По-третє, у протиставлення йому є Playwright, який у силу свого відносно нещодавнього релізу, менш популярний та має маленьке суспільство.

1.4 Визначення стратегії автоматизації процесу наскрізного тестування

Процес наскрізного тестування дуже важливо автоматизувати для забезпечення ефективності та покращення якості продукту. Стратегія автоматизації складається з декількох пунктів, нижче наведено їх список:

1. Необхідно визначитись з інструментами, які підтримують наскрізне тестування та добре поєднуються із загальним стеком технологій проєкту.
2. Розробити тестові сценарії, які охоплюють важливий функціонал продукту. Для проєктування тестів важливо користуватись шаблонами, які полегшують підтримку та розширення тестового проєкту.
3. Обрати стратегії побудови тестових даних. Це може мати на увазі як генерацію даних для їх унікальності, так і збереження їх у заздалегідь підготованих файлах.
4. Уникати залежності між тестами та робити їх якомога ізольованими один від одного.
5. Важливо також інтегрувати інструменти для автоматично генерації звітів після кожного запуску тестів з необхідними для проєкту даними, такими як час, статус, дані про помилку, якщо така присутня тощо.
6. Взяти до уваги необхідність запуску тестів на різних платформах та пристроях. Адаптувати у разі необхідності тести для запуску при різних конфігураціях.
7. Розробити стратегію параллельного запуску.
8. Поділяти тести за їх розміром та швидкістю. Тести в одному наборі повинні мати приблизно однаковий час виконання, щоб покращити процес паралельного запуску.

1.5 Аналіз багатопоточного виконання тестування

Одним з ключових пунктів оптимізації тестового процесу є виконання тестів у багатопоточному середовищі. Паралельне виконання тестування визначається як одночасне виконання кількох тестових сценаріїв або тест-кейсів.

1.5.1 Переваги

Багатопоточне виконання тестів має багато переваг, які допомагають покращити процес автоматизованого тестування [8]. Нижче наведено перелік найбільш значних переваг паралельного виконання тестів.

- *Збільшення швидкості виконання тестів.* Вочевидь якщо запускати декілька тестів одночасно, то загальний час на запуск усіх тестів зменшиться. Тож заради цієї переваги більшість проєктів обирають стратегію багатопоточного виконання тестів.

- *Покращення ефективності процесу тестування.* Процес тестування займає багато часу, тож зменшення витрат на очікування завершення автоматизованого тестування значно оптимізує людські ресурси та підвищує продуктивність тестувальників.

- *Більше покриття тестами.* Менший час на прогін тестів також значить і можливість створення більшої кількості тестів. Процес тестування обмежений часом виконання тестів, адже кожна додаткова година дорівнює додатковій годині відкладеного релізу продукту. Це фінансові витрати, які компанія може заперечити та обмежити тестувальників часовими рамками.

1.5.2 Недоліки

Незважаючи на те, що багатопоточне виконання тестів дуже поліпшує швидкість тестів, важливо не забувати про його недоліки [8]. Ось деякі з них:

- *Синхронізація тестів.* Важливо не забувати, що тести можуть конфліктувати один з одним, особливо коли вони використовують спільні

ресурси. Тому необхідно формувати набори тестів таким чином, щоб вони не хибили тестові дані.

- *Специфічність тестів.* Не всі тести можна запускати паралельно, адже деякі з них мають залежність, або потребують постійного фокусу на їх потоці. Такі тести потрібно запускати окремо, або аналізувати можливі стратегії їх запуску у наборі з іншими тестами.
- *Складний процес конфігурації та більша кількість ресурсів.* Щоб запустити тести паралельно необхідно витрати багато часу на налаштування проєкту адаптацію його архітектури та виділення більшої кількості ресурсів,. Адже кількість потоків дорівнюватиме кількості можливих водночас працюючих процесів.
- *Ускладнений процес аналізу помилок.* Як вже було сказано вище, специфічність тестів може призводити до того, що в один потік вони проходять без жодних проблем, однак при багатопоточному запуску не зрозуміло, що саме перешкоджає успішному виконанню тестів.

1.5.3Визначення стратегії для засобів автоматизації

Завдяки тому, що Playwright може самостійно запускати тести у багатопоточному середовищі, нема необхідності у використанні додаткових інструментів для виконання цього процесу. Однак, слід зауважити, що Playwright бере на себе розділення на набори тестів в залежності від кількості потоків та створює окремий процес у системі для кожного потоку. У додаток до усього вище переліченого, він пропонує власний менеджер драйверів, який за допомогою однієї команди встановлює необхідні версії браузерів.

Щоб виконувати асинхронні операції та підвищити стабільність коду ще на етапі компіляції його, рекомендовано писати тести Playwright на мові TypeScript. Проте важливо пам’ятати про витрати часу на інтерпретацію коду, адже TypeScript – це типізований JavaScript.

Важливо уніфікувати мову програмування для прозорості отриманих результатів будемо використовувати Selenium на мові JavaScript теж. Хоча Selenium не може самостійно запускати тести у багатопотоковому середовищі, важливо або обрати інструмент, або самостійно розробити стратегію для паралельного тестування. Оскільки, JavaScript має не дуже багато сумісних інструментів для багатопоточного запуску Selenium, було прийнято рішення про розбиття заздалегідь тестів на набори для потоків та написання скриптів для командної строки системи, щоб створювати окремі процеси для кожного з них.

1.6 Висновки першого розділу

Було проведено аналіз інструментів та стратегії паралельного виконання тестів. Визначено інструменти (Playwright та Selenium), мову (TypeScript та JavaScript) та стратегію для розробки проєкту з автоматизованого тестування у багатопоточному середовищі.

2 РОЗРОБКА ПРОГРАМНОГО КОДУ ДЛЯ ТЕСТУВАННЯ У БАГАТОПОТОЧНОМУ СЕРЕДОВИЩІ

2.1 Структура та архітектурні рішення проєктів

Автоматизація тестів – це теж програмування, яке потребує архітектурних рішень та використання шаблонів. В обох проєктах буде використано структурний шаблон для автотестування під назвою POM (Page Object Model) [9]. Він каже про те, що кожна сторінка застосунку повинна бути описана як окремий елемент. Іншими словами будь які шляхи до елементів повинні бути інкапсульовані у об’єкт сторінки, можливі дії з цими елементами або набір дій, який використовується зазвичай, також описують у класі сторінки.

Наступний шаблон, який буде застосований до проєкту Selenium та Playwright, – Data Provider [10], який по суті каже про використання xml, json або інших файлів у якості даних. Такі файли заготуються заздалегідь та зчитуються у якості тестових даних.

Для обох тестів буде додано .env файли, які допоможуть в майбутньому у автоматизації процесу запуску цих тестів у пайплайнах. В таких файлах зберігаються змінні оточення, які існують тільки в контексті запуску програми.

Необхідно також додати файл .gitignore, щоб уникнути додавання мусору та чутливих файлів до віддаленого репозиторію. Однак, слід зазначити, щоб продемонструвати результати тестів, усі файли звітів не будуть виключені з проєкту.

Кожен з проєктів матиме файл run.bat та run.sh для швидкого запуску тестів на будь-якій системі. Усі звіти будуть складені до відповідних папок під назвою report.

Playwright має власні особливості архітектури, адже підтримує використання фікстур [11]. Фікстури – це функції, які виконуються до або після тестових функцій. У фікстурах у Playwright визначають об’єкти сторінок або передумови для тестів під час визначення головних елементів тесту, запиту та сторінки.

Усі тести будуть зберігатися у папці під назвою tests. Проте на відміну від Selenium, у проєкті Playwright буде достатньо одного файлу з тестами, бо він вміє самостійно поділяти на набори тестів та розділяти їх на потоки. Нижче наведена остаточна структура проєкту для інструменту Playwright (Рис. 2.1).

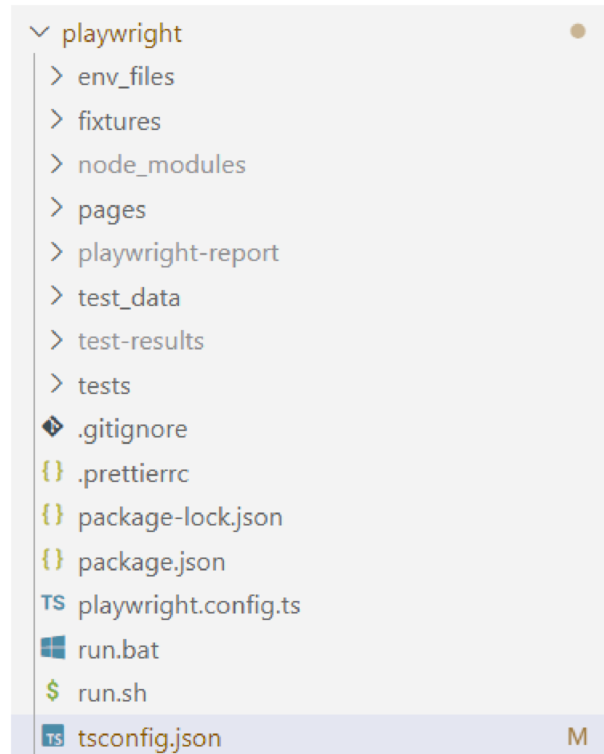


Рисунок 2.1 – Структура проєкту для Playwright

Selenium матиме схожу архітектуру, але в силу свої відмінностей у роботі потребуватиме додаткових змін. На відміну від Playwright, він не завантажує драйвери для запуску браузерів самостійно, тож необхідно або шукати додаткові бібліотеки, або заздалегідь їх завантажувати собі у проєкт. Також тут нема фікстур, тож необхідно додавати окремо ініціалізацію браузера з параметрами для запуску. Для цього у папці під назвою global буде створено додатковий файл з функціями, які будуть викликатися перед тестами один раз і повертати екземпляр браузеру. Нижче на Рисунку 2.2 наведено загальну структуру проєкту для Selenium.

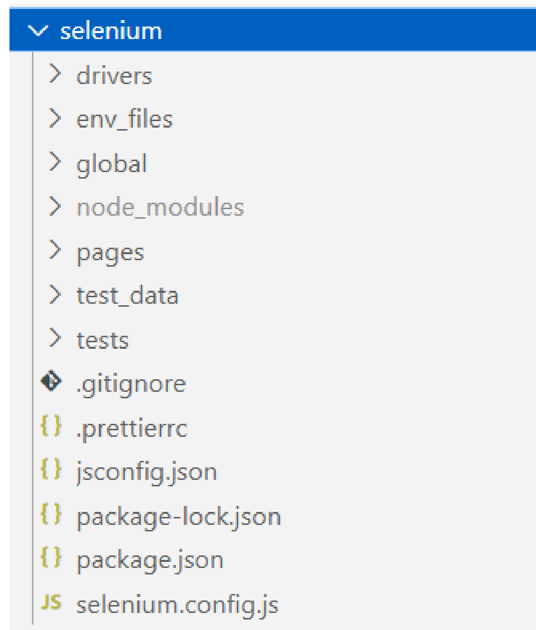


Рисунок 2.2 – Структура проєкту для Selenium

2.2 Опис обраних тест-кейсів

Наскрізні тести мають на увазі перевірку продукту у якості реального користувача. Тобто необхідно мислити «а як би зробив реальний користувач, який не знає систему». Першим кроком у будь-якому застосунку є вхід у систему, саме тому тест кейси будуть перевіряти цю функційність. Нижче наведено повний перелік тест-кейсів, які будуть автоматизовані (Таб. 2.1 – Таб.2.3).

Таблиця 2.1 – опис першого тестового сценарію для автоматизації

<table>
<tr><td colspan="2">ID тесту</td><td>1</td><td>Пріоритет тесту</td><td>Високий</td></tr>
<tr><td colspan="2">Назва тесту</td><td>Вхід у систему – перевірка елементів на сторінці</td><td>Передумови</td><td>Нема</td></tr>
<tr><td colspan="5">Кроки для виконання тесту:</td></tr>
<tr><td>№</td><td colspan="2">Дія</td><td>Вхідні дані</td><td>Очікуваний результат</td></tr>
<tr><td>1</td><td colspan="2">Відкрити сторінку застосунку 'Swag Labs', перейшовши за посиланням https://www.saucedemo.com/.</td><td></td><td>На сторінці присутні елементи:
- назва сайту;
- поле для введення імені</td></tr>
</table>

ID тесту	1	Пріоритет тесту	Високий
Назва тесту	Вхід у систему – перевірка елементів на сторінці	**Передумови**	Нема
Кроки для виконання тесту:			

№	Дія	Вхідні дані	Очікуваний результат
			користувача; - поле для введення імені користувача; - поле для введення паролю користувача; - кнопка логіну; Назва сайту повинна бути 'Swag Labs'. Поле для пароля повинно бути замаскованим.

Таблиця 2.2 – опис тестового сценарію для другого – четвертого автоматизованих тестів

ID тесту	2 - 4	Пріоритет тесту	Високий
Назва тесту	Вхід у систему – позитивний сценарій	**Передумови**	Відкрити сторінку застосунку
Кроки для виконання тесту:			

№	Дія	Вхідні дані	Очікуваний результат
1	Ввести «ім’я» у поле для імені користувача. Ввести стандартний пароль у поле для паролю користувача.	1. ім’я: "standard_user"; 2. ім’я: "problem_user"; 3. ім’я: "performance_glitch_user".	Відкрилась сторінка з продуктами. Елемент «корзина» відображається на сторінці.

Таблиця 2.3 – опис тестового сценарію для п’ятого – десятого автоматизованих тестів

ID тесту	5-10	**Пріоритет тесту**	Високий
Назва тесту	Вхід у систему – негативний сценарій	**Передумови**	Відкрити сторінку застосунку

Кроки для виконання тесту:

№	Дія	Вхідні дані	Очікуваний результат
1	Ввести «ім’я» у поле для імені користувача. Ввести «пароль» у поле для паролю користувача.	1. ім’я: "someone"; пароль: ""; текст помилки: "Epic sadface: Password is required"; 2. ім’я: ""; пароль: "someone"; текст помилки: "Epic sadface: Username is required"; 3. ім’я: "someone"; пароль: "someone"; текст помилки: "Epic sadface: Username and password do not match any user in this service"; 4. ім’я: "standard_user"; пароль: "someone"; текст помилки: "Epic sadface: Username and password do not match any user in this service"; 5. ім’я: "locked_out_user"; пароль: "secret_sauce"; текст помилки: "Epic sadface: Sorry, this user has been locked out."; 6. ім’я: "someone"; пароль: "secret_sauce"; текст помилки: "Epic sadface: Username and password do not match any user in this service";	На сторінці відобразилась помилка, яка містить у собі «текст помилки».

2.3 Розробка скриптів для паралельного виконання тестів

Щоб запустити Selenium у багатопоточному середовищі, йому необхідні додаткові інструменти, які мають змогу поділяти автоматично усі тести на набори в залежності від кількості потоків. Однак, щоб уникнути втручання інших бібліотек з їх недоліками або особливостями, буде створено окреме рішення по запуску тестів. Це рішення найпростіше і найоптимальніше в даній ситуації. Для цього необхідно заздалегідь поділити тести на набори для кожного з потоків. Наприклад, якщо запускатись буде у 4 потоків, необхідно створити 4 файли і в кожному з них буде по 2 тести, окрім перших двох – в них буде по 3 тести, адже загалом ми маємо 10 тестів.

Було додатково створено файл з опціями для Selenium (Рис 2.3), де вказано посилання на застосунок, параметри для драйверу браузера, стан відображення інтерфейсу браузера, шлях до драйверів та час максимального очікування у секундах.

```
export const options = {
  baseURL: 'https://www.saucedemo.com/',
  timeout: 10 * 10000,
  disableExtensions: false,
  windowSize: {
    maximized: true,
  },
  disableGpu: true, // required for Windows,
  headless: !true,
  browserDrivers: {
    name: 'firefox',
    path: '../drivers/geckodriver.exe',
    //   name: 'chrome',
    //   path: '../drivers/chrome.exe',
  },
};
```

Рисунок 2.3 – Структура файлу з опціями для Selenium

Скрипт для запуску тестів буде складатися з двох частин – головної та окремої. Для кожної кількості потоків створена окрема папка з розбиттям на набори тестів для потоків. Через це на кожен з таких наборів необхідно

розробити скрипт запуску з виведенням часу виконання. Такий скрипт буде запускати файл з набором тестів та транслювати отриманий результат у файл (Рис. 2.4).

```
:: Set environment variables
set user=standard_user
set password=secret_sauce

:: Change the drive and directory
D:
cd /d "D:\Uni\diploma\selenium"

:: Run Mocha with the specified test file
npx mocha -R json-stream ".\tests\parallel_1.spec.js" > "./tests/report/report" && exit
```

Рисунок 2.4 – Скрипт для Windows для окремих наборів тестів

Головний скрипт буде виконувати роль створення різних процесів та запускання разом окремих частин загального тестового набору (Рис. 2.5.). Таким чином буде досягнуто цілі запуску у багатопоточному середовищі тестів з використанням Selenium.

```

start .\run1.bat
start .\run2.bat

exit
```

Рисунок 2.5 – Скрипт для Windows для окремих наборів тестів

Для Playwright все набагато простіше, адже цей інструмент пропонує самостійне розділення на потоки без додаткових засобів паралелізації. Щоб розбити тести на необхідну кількість потоків, достатньо лише прописати в файлі конфігурації інструменту кількість workers і він самостійно розіб’є загальну кількість тестів на потоки та створить для них процеси запуску (Рис. 2.5).

```
import { defineConfig, devices } from '@playwright/test';

export default defineConfig({
  testDir: './tests',
  fullyParallel: true,
  forbidOnly: !!process.env.CI,
  retries: process.env.CI ? 2 : 0,
  workers: process.env.CI ? 1 : 1,
  reporter: 'html',
  use: {
    trace: 'on-first-retry',
    baseURL: 'https://www.saucedemo.com/',
    headless: !true,
  },
  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
    },
    {
      name: 'firefox',
      use: { ...devices['Desktop Firefox'] },
    },
  ],
});
```

Рисунок 2.6 – Файл конфігурації для Playwright

Проте щоб було простіше запускати тести з використанням Playwright, необхідно створити простий скрипт, який за допомогою команди цього засобу автоматизації запускатиме тести в усій папці, а точніше в єдиному файлі (Рис 2.7).

```

:: Set environment variables
set user=standard_user
set password=secret_sauce

:: Change the drive and directory
D:
cd /d "D:\Uni\diploma\playwright"

:: Run with the specified test file
npx playwright test "./tests"

```

Рисунок 2.7 – Скрипт для Windows для запуску тестів Playwright

2.4 Висновки другого розділу

Було створено десять тест-кейсів для прикладу процесу наскрізного тестування. Також розроблено два окремих проєкти для автоматичного тестування за шаблонами тестування з використанням Selenium та Playwright. Обидва проєкти було конфігуровано для запуску у багатопоточному середовищі. Додано усі необхідні скрипти для виконання програми на Mac OC та Windows OC.

3 ДОСЛІДНИЦЬКА ЧАСТИНА

3.1 Результати виконання тестів у багатопоточному середовищі

Дослідження проводилось на двох комп'ютерах з наступними системними параметрами:

- Перший пристрій:
  - ОС: Windows 11 Home;
  - Кількість ядер: 10;
  - Процесор: 12th Gen Intel(R) Core(TM) i7-1255U;
  - Кількість логічних процесорів: 12;
  - ОЗП: 16,0 ГБ.
- Другий пристрій:
  - ОС: Ventura 13.6;
  - Кількість ядер: 8 (4 продуктивних та 4 ефективних);
  - Процесор: Apple M1;
  - ОЗП:8ГБ.

Версії засобів для тестування:

- Playwright – 1.38.1.
- Selenium – 4.13.0.

Версії браузерів:

- Для Windows:
  - Chrome:119.0.6045.105;
  - Firefox: 117.0;
  - Chromium: 117.0.5938.62.
- Для Mac:
  - Chrome: 119.0.6045.105;
  - Firefox: 117.0;
  - Chromium: 117.0.5938.62.

3.1.1 Виконання тестів зі збоєм у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Windows

При запуску тестів на Windows ОС за допомогою двох інструментів при наявності збою у продуктивності з використанням браузеру Chrome було отримано відносно схожі результати (Таб. 3.1 – Таб. 3.2). Найменше значення було отримано для Playwright при використанні 5 потоків, натомість Selenium показав найкращій результат при 6 потоках.

Таблиця 3.1 – результати виконання тестів з Playwright зі збоєм у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Windows

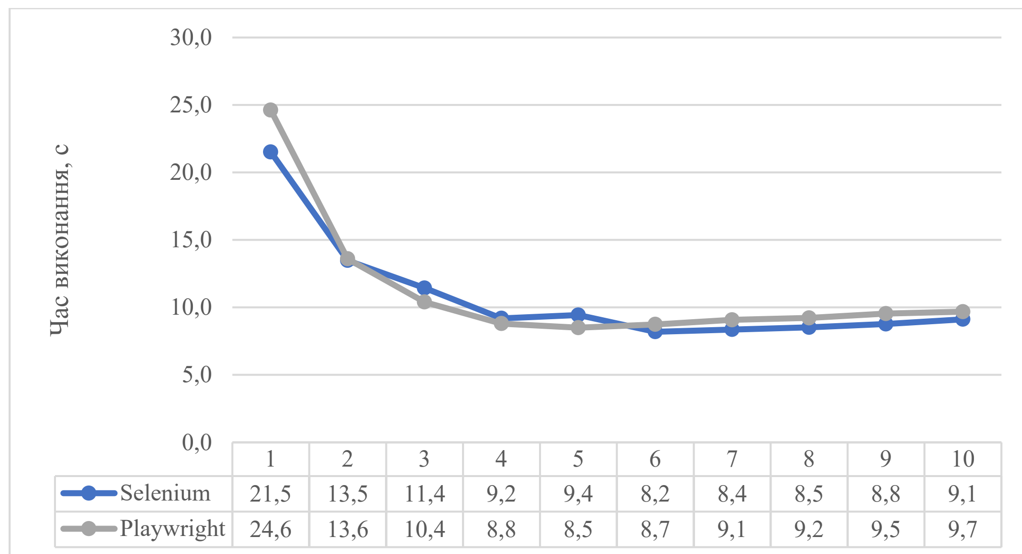
К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	24,8	25,0	24,5	24,4	24,4	24,6
2	13,7	13,4	13,7	13,6	13,6	13,6
3	10,3	10,2	10,4	10,2	10,9	10,4
4	8,4	8,7	8,3	9,4	9,2	8,8
5	8,3	8,5	8,5	8,7	8,5	8,5
6	8,8	8,6	8,7	8,9	8,7	8,7
7	8,9	9,3	8,9	9,1	9,2	9,1
8	9,1	9,1	9,1	9,3	9,5	9,2
9	9,6	9,3	9,5	9,5	9,8	9,5
10	9,5	9,8	9,8	9,5	9,8	9,7

По цим двом результатам можна побачити як іноді показники часу різко деградують через наявність збою у продуктивності, однак це не пік того як воно проявляється. Також наявна деградація часу з кількістю потоків, але цей все одно менший за час при використанні лише одного потоку (Таб. 3.1 – Таб. 3.2).

Таблиця 3.2 – результати виконання тестів з Selenium зі збоєм у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Windows

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	22,5	21,3	21,1	21,6	21,1	21,5
2	13,4	13,6	13,5	13,4	13,5	13,5
3	11,7	11,4	11,4	11,3	11,3	11,4
4	9,0	9,2	9,1	9,2	9,4	9,2
5	9,6	9,6	9,3	9,3	9,3	9,4
6	8,0	8,0	8,7	8,2	8,1	8,2
7	8,3	8,3	8,3	8,2	8,7	8,4
8	8,3	8,5	8,5	8,7	8,6	8,5
9	8,7	8,7	8,9	8,8	8,7	8,8
10	8,9	9,6	9,2	8,9	9,1	9,1

На Рис. 3.1 можна побачити як отримані результати підпорядковуються гіперболічній формі. Найменше значення для Playwright – це 8.5 секунд, а для Selenium – це 8.2 секунди, що на 0.3 менше. Якщо уявити, що тестів в 100 разів більше, то це буде перевага у 30 секунд, за умови, що усі тести займають однакову кількість часу.



	1	2	3	4	5	6	7	8	9	10
Selenium	21,5	13,5	11,4	9,2	9,4	8,2	8,4	8,5	8,8	9,1
Playwright	24,6	13,6	10,4	8,8	8,5	8,7	9,1	9,2	9,5	9,7

Рисунок 3.1 – Графік порівняння Selenium та Playwright при виконанні тестів зі збоєм у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Windows

3.1.2 Виконання тестів зі збоєм у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Mac

При використанні двох інструментів для запуску тестів на операційній системі Mac із зазначенням невдачі у продуктивності при використанні браузера Chrome, отримані результати були значно протилежні за даними у Таблиці 3.3 та Таблиці 3.4. Мінімальне значення було зафіксовано при використанні Playwright з використанням 5 потоків, тоді як Selenium показав найкращі результати при 4 потоках.

Таблиця 3.3 – результати виконання тестів з Playwright зі збоєм у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Mac

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	26,8	26,8	26,3	26,4	26,4	26,5
2	14,5	16,8	14,8	13,9	13,8	14,8
3	9,7	9,7	9,7	9,7	9,8	9,7
4	7,9	7,8	8,0	7,9	7,9	7,9
5	8,0	8,0	8,0	8,0	8,2	8,0
6	8,6	8,3	8,3	8,2	8,3	8,3
7	8,5	8,4	8,4	8,5	8,5	8,5
8	8,6	8,7	8,7	8,6	8,9	8,7
9	9,1	8,6	8,8	8,9	8,6	8,8
10	9,1	8,7	9,2	8,9	8,9	9,0

За цими двома результатами видно, як проблема актуальності версії браузера впливає на часові показники. На момент проведення дослідження (21.11.2023) не було доступного актуального драйверу Chrome для Selenium, що призвело до суттєвого погіршення результатів, ніж можна було б очікувати (Таб. 3.4).

Таблиця 3.4 – результати виконання тестів з Selenium зі збоєм у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Mac

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	60,7	61,4	67,9	70,0	67,7	65,5
2	42,5	48,1	49,6	45,3	44,2	45,9
3	31,8	38,3	38,2	35,0	31,7	35,0
4	26,7	27,0	27,8	31,8	29,7	28,6
5	29,1	31,0	30,6	30,6	33,5	31,0
6	38,4	37,2	36,8	35,8	36,5	36,9
7	28,2	37,7	33,7	35,0	33,5	33,6
8	34,8	36,1	33,8	34,6	35,0	34,9
9	28,8	34,7	27,6	30,8	30,8	30,5
10	29,4	34,6	27,6	33,2	33,0	31,6

На Рис. 3.2 видно, як отримані результати мають гіперболічний характер, хоч і не дуже явно виражений. Найменше значення для Playwright – 8.0 секунд, а для Selenium – 26.7 секунд, різниця в 18.7 секунд. Це дуже велика різниця, як може призвести до значної витрати ресурсів при більшої кількості тестів.

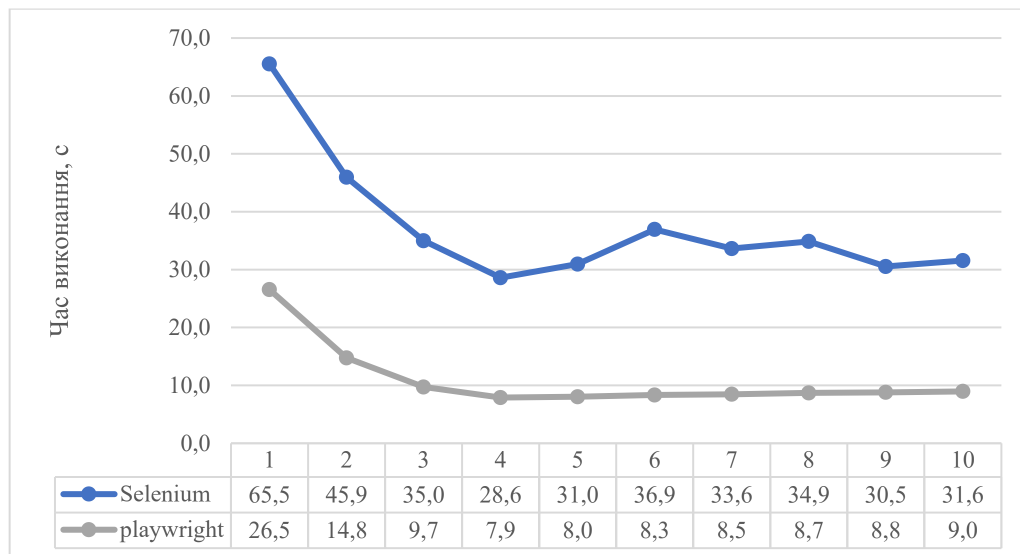


Рисунок 3.2 – Графік порівняння Selenium та Playwright при виконанні тестів зі збоєм у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Mac

3.1.3 Виконання тестів без збою у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Windows

Результати на операційній системі Windows при використанні браузера Chrome були неочікувано кращими в бік Selenium (Таб. 3.5 та Таб. 3.6). Найменший час для Playwright отримано з використанням 9 потоків, а для Selenium – при 7 потоках.

Таблиця 3.5 – результати виконання тестів з Playwright без збою у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Windows

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	20,4	18,2	19,5	18,2	17,5	18,8
2	10,3	10,4	10,3	10,3	10,3	10,3
3	7,8	6,8	8,0	7,8	7,7	7,6
4	7,9	8,0	7,9	7,9	7,9	7,9
5	6,3	6,3	6,1	6,3	6,3	6,3
6	5,7	5,7	5,6	5,6	5,9	5,7
7	6,0	6,0	6,0	6,0	6,0	6,0
8	5,9	6,1	5,8	6,1	6,1	6,0
9	4,7	5,6	5,0	5,6	4,8	5,1

З цих та попередніх результатів можна сказати, що результати Playwright не дуже значно відрізняються, адже цей набор без тесту, який має проблеми у роботі. Іншими словами щоб отримати час виконання 10 тестів необхідно додати до всіх результатів 1/10 частину часу. Selenium витратив меншу частину часу (Таб. 3.6) через те, що заздалегідь було поділено тести на набори тестів для кожного потоку.

Таблиця 3.6 – результати виконання тестів з Selenium без збою у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Windows

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	13,6	13,6	14,0	13,7	13,9	13,7
2	7,9	7,9	7,9	8,1	7,9	7,9
3	5,5	5,5	5,5	5,6	5,8	5,6
4	5,5	5,4	6,9	7,6	7,7	6,6
5	7,2	7,4	4,7	4,9	4,7	5,8
6	4,7	4,8	5,0	4,9	4,8	4,9
7	4,8	4,5	4,6	4,7	5,1	4,8
8	5,1	4,7	4,9	4,7	4,6	4,8
9	5,1	5,0	5,1	5,5	5,2	5,2

На Рис. 3.3 видно, як отримані результати різко спадають вже на 2 потоках, однак потім мають майже схожу кількість часу на виконання тестів. Найменше значення для Playwright та Selenium майже однакові, 4.7 та 4.5 секунд, відповідно.

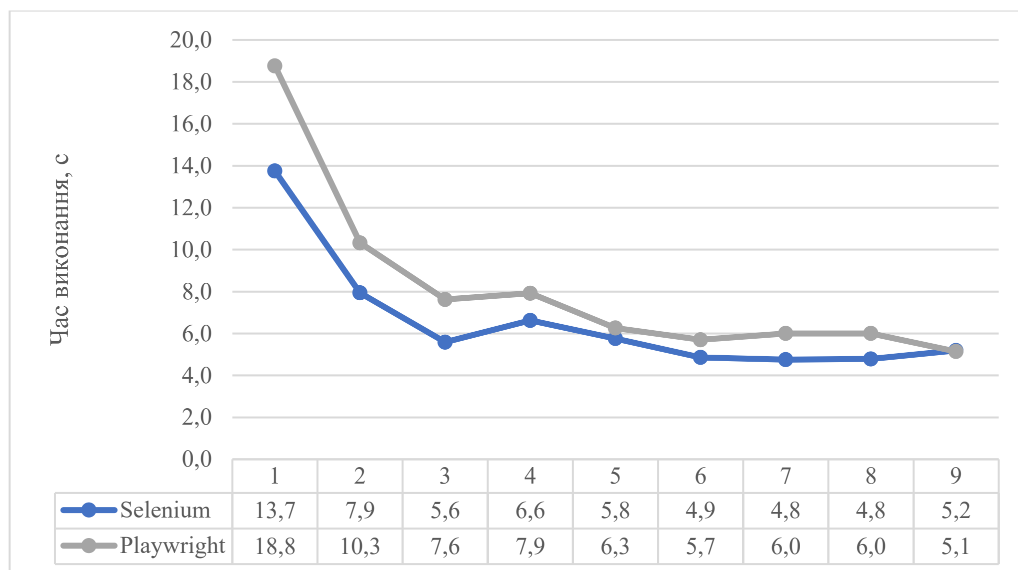


Рисунок 3.3 – Графік порівняння Selenium та Playwright при виконанні тестів без збою у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Windows

3.1.4 Виконання тестів без збою у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Mac

Як було раніше зазначено, результати часу з Selenium на Mac будуть гіршими порівняно з Windows. Мінімальний час 3.8 секунд було отримано з використанням 9 потоків Playwright (Таб. 3.7). Наразі це найменший час виконання тестів. Selenium використовуючи 4 потоки зміг провести тестування за 21.2 секунди (Таб. 3.8).

Таблиця 3.7 – результати виконання тестів з Playwright без збою у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Mac

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	19,4	19,5	19,6	19,3	19,4	19,4
2	10,9	11,0	10,9	10,8	11,1	10,9
3	6,9	7,0	7,0	7,1	7,0	7,0
4	7,1	7,1	7,1	7,1	7,1	7,1
5	5,2	5,2	5,3	5,2	5,4	5,3
6	5,3	5,5	5,4	5,3	5,4	5,4
7	5,4	5,5	5,7	5,5	5,6	5,5
8	5,6	6,3	5,6	5,6	5,7	5,8
9	4,0	3,8	3,9	3,8	3,8	3,9

Таблиця 3.8 – результати виконання тестів з Selenium без збою у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Mac

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	66,7	64,3	60,9	64,5	67,3	64,7
2	42,3	40,6	45,6	41,0	40,2	42,0
3	23,8	28,9	29,0	25,0	22,1	25,8
4	21,2	21,2	22,1	25,8	24,5	23,0
5	22,0	23,1	23,2	23,1	25,5	23,4

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
6	26,5	25,2	24,7	23,9	24,6	25,0
7	23,6	24,8	24,6	25,0	23,8	24,4
8	25,5	26,0	25,6	25,8	26,1	25,8
9	23,2	24,5	23,1	24,0	24,1	23,8

Графік, який зображено нижче (Рис. 3.4) відображає залежність часу від кількості потоків для обох засобів тестування. Через те, що різниця результатів велика, гіперболічний характер лінії для Playwright майже не помітно, хоча він і присутній.

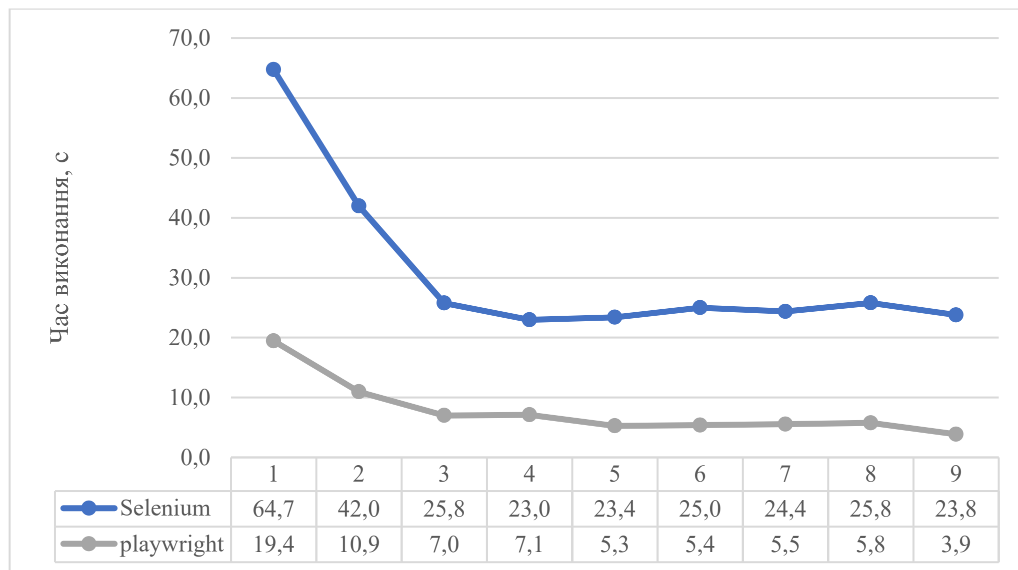


Рисунок 3.4 – Графік порівняння Selenium та Playwright при виконанні тестів без збою у продуктивності застосунку у Chrome без відображення інтерфейсу браузера на Mac

3.1.5 Виконання тестів зі збоєм у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Windows

Показники часу отримані для Firefox значно нижчі ніж для Chrome. Варто також зауважити, що Playwright використовує Chromium, натомість Selenium користується саме Chrome. Проте браузер Firefox для обох один і той самий. Selenium показав один раз найменший результат рівний 10.5 секунд, але в

середньому кращий результат у Playwright, який дорівнює 12.2 секунди (Таб. 3.9 та Таб. 3.10).

Таблиця 3.9 – результати виконання тестів з Playwright зі збоєм у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Windows

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	28,7	29,0	28,1	28,8	29,0	28,7
2	16,2	16,3	16,3	16,4	16,6	16,4
3	12,9	13,3	12,7	12,4	12,5	12,8
4	11,6	11,8	11,5	13,9	12,4	12,2
5	12,2	13,2	13,0	12,3	12,1	12,6
6	12,6	13,1	13,3	13,3	14,5	13,4
7	13,4	13,7	14,5	13,2	13,6	13,7
8	16,6	16,8	14,8	14,1	14,4	15,3
9	14,8	14,9	17,2	15,9	15,8	15,7
10	18,6	18,3	17,0	17,9	19,0	18,2

Таблиця 3.10 – результати виконання тестів з Selenium зі збоєм у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Windows

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	42,5	43,0	43,0	43,1	42,9	42,9
2	19,2	19,0	19,1	20,1	19,5	19,4
3	13,9	15,6	15,2	16,9	15,4	15,4
4	15,3	16,2	16,2	15,3	15,2	15,6
5	13,4	13,6	10,8	12,8	13,6	12,9
6	14,6	14,8	15,1	11,6	12,0	13,6
7	13,4	14,0	10,5	13,7	12,3	12,8
8	14,7	14,3	14,4	14,6	11,5	13,9
9	15,2	15,2	12,0	14,0	14,8	14,2
10	14,7	14,7	14,8	13,6	15,7	14,7

Також можна побачити деградацію часу з підвищенням кількості потоків для обох засобів тестування (Рис. 3.5). Найменший показник середнього часу помічено на 4 та 7 потоках для Playwright та Selenium, відповідно.

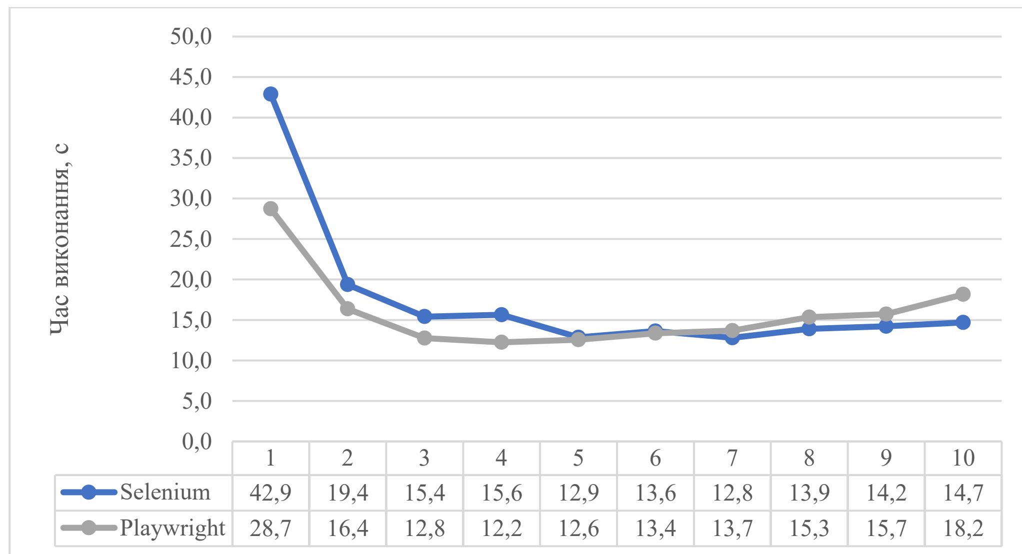


Рисунок 3.5 – Графік порівняння Selenium та Playwright при виконанні тестів зі збоєм у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Windows

3.1.6Виконання тестів зі збоєм у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Mac

Загалом можна зробити висновок, що на операційній системі Mac інструмент Selenium проявляє гіршу продуктивність, у порівнянні з Playwright, який демонструє переваги у часі виконання тестів. Навіть за умови відсутності проблем з версіями браузерів для Selenium, які впливають на ефективність у випадку використання Firefox, Playwright все ж здатний досягати кращих результатів (Таблиця 3.11 – Таблиця 3.12).

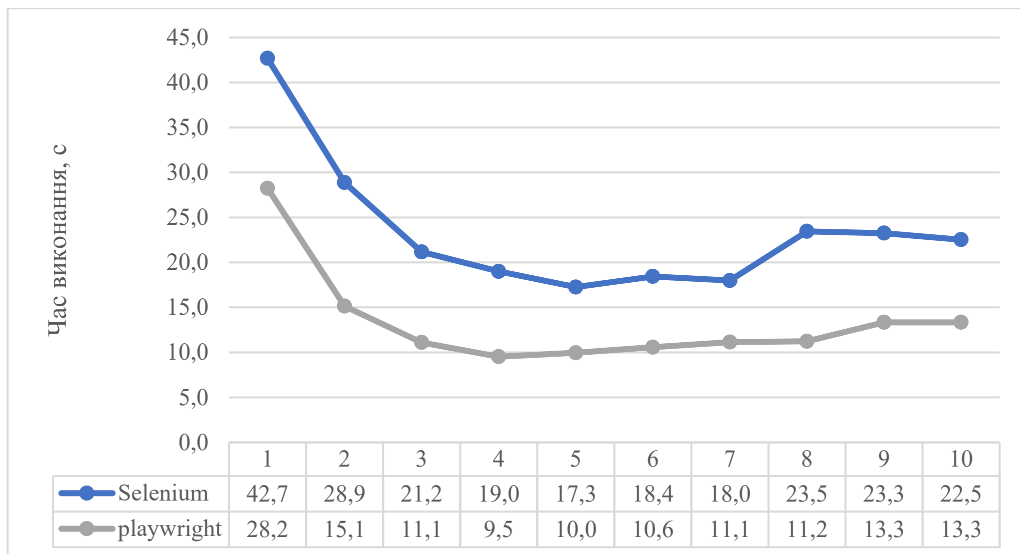
Таблиця 3.11 – результати виконання тестів з Playwright зі збоєм у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Mac

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	28,4	28,3	27,9	27,9	28,7	28,2
2	15,2	15,2	15,3	14,9	15,1	15,1
3	11,1	11,0	11,0	11,1	11,3	11,1
4	10,1	9,3	9,4	9,2	9,6	9,5
5	10,2	9,9	10,0	9,8	9,9	10,0
6	10,7	10,3	10,6	10,7	10,6	10,6
7	11,1	11,0	10,9	11,3	11,4	11,1
8	12,1	11,6	12,1	10,0	10,4	11,2
9	13,7	13,6	13,5	13,6	12,3	13,3
10	11,7	13,5	13,8	15,4	12,3	13,3

Таблиця 3.12 – результати виконання тестів з Selenium зі збоєм у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Mac

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	43,0	42,7	42,6	42,0	43,1	42,7
2	28,9	27,9	26,7	30,2	30,7	28,9
3	19,7	18,7	20,0	25,5	22,1	21,2
4	18,0	17,0	18,2	20,5	21,3	19,0
5	19,0	16,2	15,6	17,3	18,2	17,3
6	18,9	16,9	19,9	20,1	16,5	18,4
7	17,0	15,6	17,3	18,0	22,1	18,0
8	24,5	23,2	20,1	21,0	28,5	23,5
9	28,3	20,1	20,1	24,7	23,1	23,3
10	20,1	24,7	25,1	21,7	21,1	22,5

На Рис. 3.6 можна побачити різницю між часом виконання тестів цих обох інструментів. Playwright здобув середній найменший час під час використання 4 потоків, натомість Selenium – під час 5 потоків. Різниця між часом становить майже 8 секунд.



	1	2	3	4	5	6	7	8	9	10
Selenium	42,7	28,9	21,2	19,0	17,3	18,4	18,0	23,5	23,3	22,5
playwright	28,2	15,1	11,1	9,5	10,0	10,6	11,1	11,2	13,3	13,3

Рисунок 3.6 – Графік порівняння Selenium та Playwright при виконанні тестів зі збоєм у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Mac

3.1.7 Виконання тестів без збою у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Windows

Без збою у продуктивності результати завжди більш стабільні та займають менше часу. Результати, представлені у Таблицях 3.13 та 3.14, ілюструють кращу продуктивність на 2-7 секунд у порівнянні із попередніми даними, викладеними у Таблицях 3.10 та 3.9.

Важливо відзначити, що Playwright демонструє вищу ефективність при виконанні тестів на браузері Firefox у порівнянні з Selenium. Проте варто зауважити, що при збільшенні кількості потоків більше ніж на 5, відбувається помітна деградація часу виконання процесу наскрізного тестування у випадку Playwright.

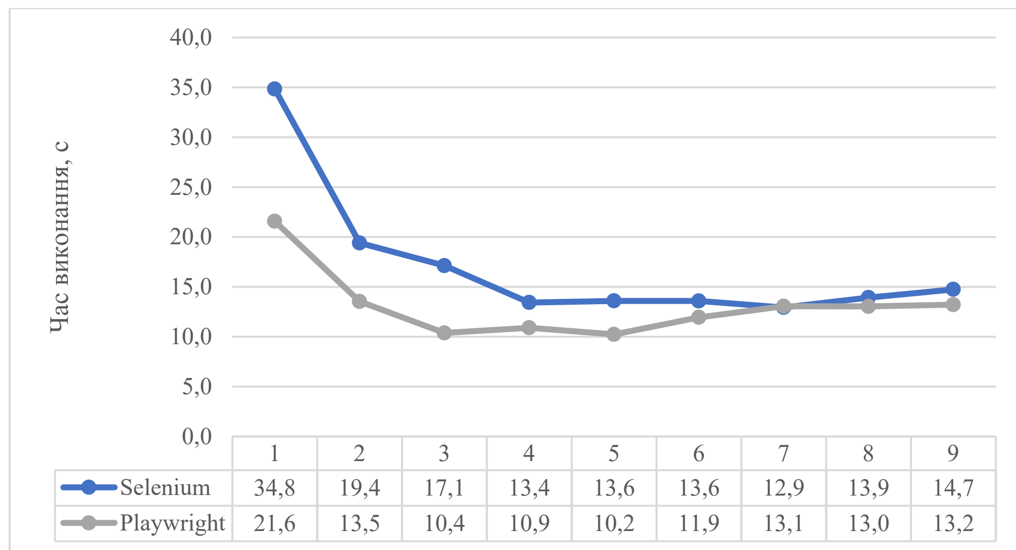
Таблиця 3.13 – результати виконання тестів з Playwright без збою у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Windows

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	21,2	22,0	21,6	21,7	21,4	21,6
2	13,6	13,3	13,8	13,4	13,6	13,5
3	10,7	10,1	10,2	10,2	10,7	10,4
4	11,1	10,9	10,5	10,8	11,2	10,9
5	10,8	10,7	10,2	9,8	9,7	10,2
6	11,0	11,9	13,6	11,9	11,3	11,9
7	10,7	14,6	13,5	14,5	12,0	13,1
8	12,1	12,9	11,7	13,8	14,7	13,0
9	13,5	14,3	14,6	11,3	12,4	13,2

Таблиця 3.14 – результати виконання тестів з Selenium без збою у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Windows

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	34,5	34,8	34,9	34,9	35,1	34,8
2	18,7	19,8	19,5	19,8	19,2	19,4
3	16,5	14,1	20,8	19,2	15,0	17,1
4	12,8	14,7	12,8	13,3	13,7	13,4
5	13,5	13,8	13,3	14,0	13,4	13,6
6	13,3	14,4	13,5	13,5	13,3	13,6
7	13,8	13,4	12,6	11,8	13,1	12,9
8	14,8	13,9	14,3	12,8	13,9	13,9
9	14,6	14,9	15,5	14,7	14,1	14,7

На графіку наведеному нижче (Рис. 3.7) можна відзначити різницю у часі виконання тестів. Playwright показав найкращий результат при використанні 5 потоків, тоді як Selenium продемонстрував оптимальну ефективність при 7 потоках. Різниця у часі виконання становить 2.7 секунд.



	1	2	3	4	5	6	7	8	9
Selenium	34,8	19,4	17,1	13,4	13,6	13,6	12,9	13,9	14,7
Playwright	21,6	13,5	10,4	10,9	10,2	11,9	13,1	13,0	13,2

Рисунок 3.7 – Графік порівняння Selenium та Playwright при виконанні тестів без збою у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Windows

3.1.8 Виконання тестів без збою у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Mac

При аналізі вихідних даних на операційній системі Mac в контексті використання браузера Firefox виявлено, що інструмент Playwright продемонстрував ефективність, перевершуючи свої показники порівняно з варіантом використання на платформі Windows (див. Таб. 3.13 і Таб. 3.15). З іншого боку, для Selenium спостерігається негативна динаміка, адже його результати виявилися менш задовільними в порівнянні з попередніми (див. Таблиця 3.14 і Таблиця 3.16). Ці відмінності у продуктивності можуть бути ключовим фактором при виборі оптимального інструменту для конкретного проекту, особливо при врахуванні особливостей операційної системи та браузера.

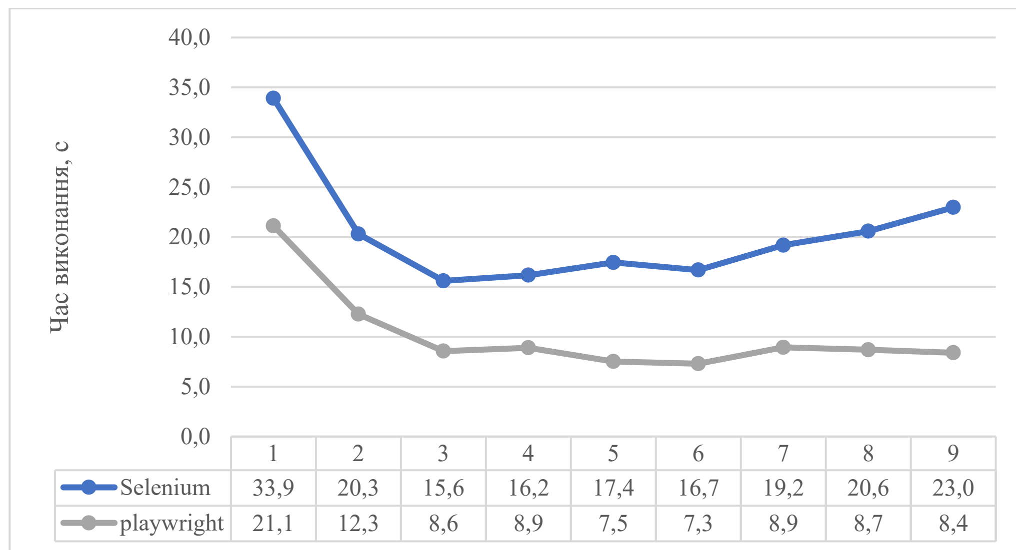
Таблиця 3.15 – результати виконання тестів з Playwright без збою у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Mac

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	21,3	21,1	21,0	21,1	21,1	21,1
2	12,2	12,3	12,3	12,4	12,1	12,3
3	8,7	8,4	8,8	8,6	8,3	8,6
4	8,7	8,7	8,9	9,5	8,7	8,9
5	7,6	7,6	7,3	7,5	7,6	7,5
6	7,3	7,3	7,3	7,3	7,3	7,3
7	9,6	9,0	8,6	8,7	8,8	8,9
8	9,8	9,9	8,2	7,6	8,0	8,7
9	8,6	8,8	8,2	8,2	8,2	8,4

Таблиця 3.16 – результати виконання тестів з Selenium без збою у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Mac

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	34,7	33,7	33,5	34,0	33,6	33,9
2	21,4	20,2	20,2	19,9	19,8	20,3
3	18,2	15,0	15,0	14,8	14,9	15,6
4	16,5	15,1	15,4	18,5	15,4	16,2
5	15,6	16,1	18,1	17,9	19,6	17,4
6	17,0	16,8	16,1	16,6	17,0	16,7
7	20,5	16,4	18,5	17,7	22,8	19,2
8	22,4	19,9	21,6	19,8	19,2	20,6
9	18,5	24,2	23,9	22,2	26,1	23,0

На представленому графіку (Рис. 3.8) спостерігається різниця у часі виконання тестів між інструментами Playwright і Selenium. Playwright досягнув найкращих результатів при використанні 6 потоків, в той час як Selenium проявив оптимальну ефективність при 3 потоках. Різниця в часі виконання становить 8.1 секунд, вказуючи на перевагу одного інструменту над іншим у конкретних умовах.



	1	2	3	4	5	6	7	8	9
Selenium	33,9	20,3	15,6	16,2	17,4	16,7	19,2	20,6	23,0
playwright	21,1	12,3	8,6	8,9	7,5	7,3	8,9	8,7	8,4

Рисунок 3.8 – Графік порівняння Selenium та Playwright при виконанні тестів без збою у продуктивності застосунку у Firefox без відображення інтерфейсу браузера на Mac

3.1.9 Виконання тестів зі збоєм у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Windows

У контексті виконання тестів з використанням Playwright у присутності збою у продуктивності та з відображенням інтерфейсу браузера на операційній системі Windows (Табл. 3.17), можливо виокремити певні висновки. Результати вказують на те, що в умовах деякого збою у продуктивності, час виконання тестів стає менш стабільним та збільшується. Зокрема, спостерігається, що мінімальний час досягає значення 9.3 секунди. Це свідчить про те, що фактори, пов'язані із збоєм у продуктивності та відображенням інтерфейсу, мають вплив на ефективність виконання тестових сценаріїв за допомогою Playwright.

Таблиця 3.17 – результати виконання тестів з Playwright зі збоєм у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Windows

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	26,0	33,8	26,7	25,8	26,0	27,7
2	14,5	14,5	14,7	14,8	14,4	14,6
3	12,0	10,7	12,7	10,7	12,1	11,6
4	9,3	10,4	9,3	9,4	10,4	9,8
5	9,9	9,6	9,9	10,0	10,2	9,9
6	10,3	11,0	9,6	9,8	10,3	10,2
7	10,4	10,1	11,6	10,5	11,2	10,8
8	10,8	10,8	11,5	11,4	13,0	11,5
9	12,1	11,1	13,5	11,4	12,7	12,2
10	14,1	12,1	11,4	12,4	11,8	12,4

У контексті виконання тестів за допомогою Selenium із відображенням інтерфейсу браузера (Табл. 3.18), варто відзначити, що Selenium демонструє трошки кращі результати, з відображенням мінімального часу в 8.8 секунди. Однак, схоже, що він також піддається деградації часу виконання зі зростанням кількості потоків. Ці спостереження підкреслюють, що відображення інтерфейсу браузера забирає більше ресурсів та може впливати на продуктивність тестових сценаріїв, незалежно від інструменту, і варто ретельно розглядати цей аспект під час планування та виконання тестів.

Таблиця 3.18 – результати виконання тестів з Selenium зі збоєм у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Windows

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	23,5	22,3	21,7	21,9	21,6	22,2
2	14,0	13,8	14,9	13,8	13,7	14,1
3	12,2	12,5	12,3	12,3	12,4	12,3
4	10,1	10,3	10,2	11,2	10,7	10,5
5	10,9	10,4	10,9	10,3	10,6	10,6
6	9,0	8,8	8,9	9,5	10,3	9,3
7	10,2	10,9	10,1	10,4	10,2	10,3
8	11,1	11,6	10,4	11,7	11,1	11,2
9	11,4	10,5	11,7	14,5	10,0	11,6
10	12,2	11,1	14,5	11,3	11,4	12,1

На графіку середнього часу виконання тестів для різної кількості потоків (Рис. 3.9) видно, що обидва інструменти демонструють збільшення часу виконання зі зростанням кількості потоків. Однак, не зважаючи на цей загальний тенденційний ріст, інструмент Selenium в середньому виявляється більш ефективним при 6 потоках, у порівнянні з Playwright, який виявляє свою оптимальну ефективність при 4 потоках. Це може свідчити про те, що відмінності в архітектурі та внутрішніх механізмах роботи цих інструментів впливають на їх продуктивність у відмінних умовах використання та завданнях.

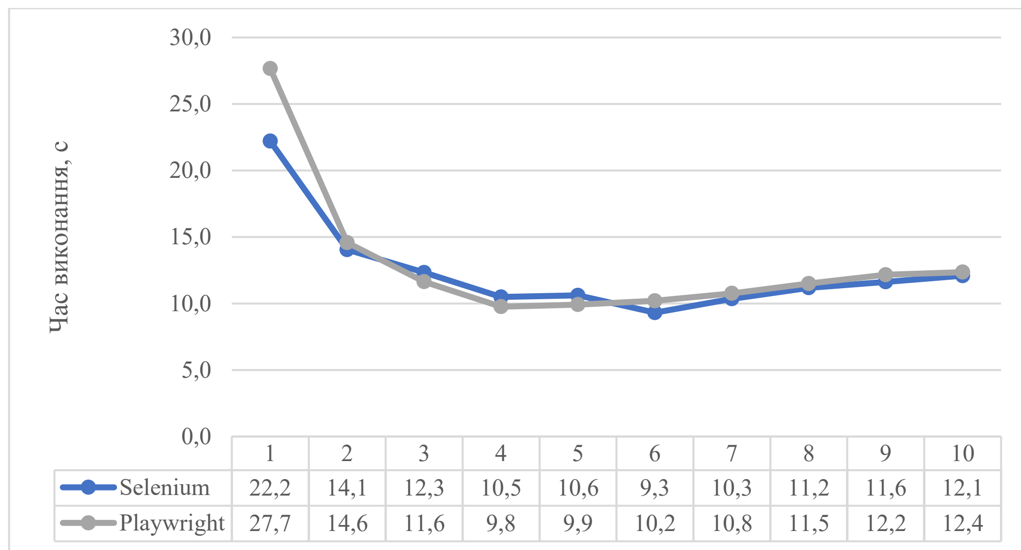


Рисунок 3.9 – Графік порівняння Selenium та Playwright при виконанні тестів зі збоєм у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Windows

3.1.10 Виконання тестів зі збоєм у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Mac

На платформі Mac результати виконання тестів за допомогою Playwright суттєво покращуються, досягаючи 8.9 секунд (Таблиця 3.19). Це може свідчити про більшу оптимізацію та високу продуктивність Playwright у середовищі Mac, що може бути пов'язано з оптимізацією для даної операційної системи або іншими факторами, специфічними для Mac-платформи.

Таблиця 3.19 – результати виконання тестів з Playwright зі збоєм у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Mac

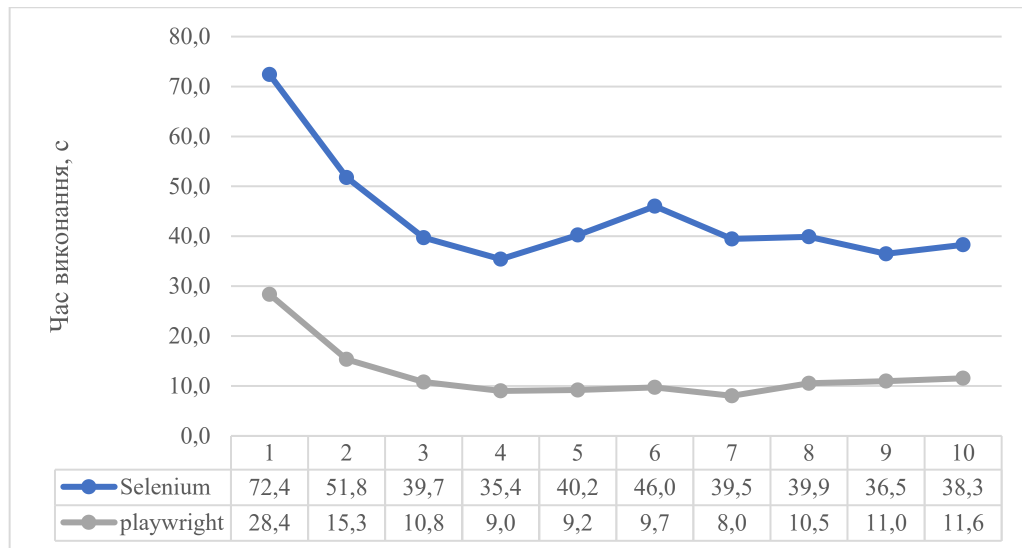
К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	28,5	28,0	28,4	28,4	28,5	28,4
2	15,0	15,2	15,3	15,2	16,0	15,3
3	10,8	11,0	10,8	10,8	10,6	10,8
4	9,1	8,9	8,9	8,9	9,3	9,0
5	9,4	9,3	9,0	9,0	9,3	9,2
6	9,7	9,8	9,4	10,0	9,7	9,7
7	10,4	9,9	9,9	10,0	9,8	10,0
8	10,2	10,6	10,3	10,7	10,9	10,5
9	11,0	10,8	10,9	11,1	11,1	11,0
10	11,4	11,7	11,2	11,7	11,8	11,6

Протилежно до Windows, взаємодія Selenium з Mac менше ефективна, що призводить до суттєвого погіршення результатів (Таб. 3.20). Тепер, при послідовному запуску тестів, кожен з них вимагає 7 секунд на виконання. Це значно перевищує прийнятний час для тесту, який лише заповнює поля та перевіряє наявність одного елемента.

Таблиця 3.20 – результати виконання тестів з Selenium зі збоєм у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Mac

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	65,4	68,4	75,3	78,2	74,6	72,4
2	47,2	54,9	55,6	51,3	49,9	51,8
3	36,3	42,8	42,7	40,0	36,8	39,7
4	32,5	33,3	34,5	39,4	37,5	35,4
5	37,2	40,4	40,1	40,1	43,4	40,2
6	47,5	46,4	45,9	44,8	45,5	46,0
7	33,9	43,4	39,6	41,0	39,4	39,5
8	40,2	41,4	38,6	39,4	40,0	39,9
9	34,8	40,5	32,8	37,1	37,1	36,5
10	37,4	42,5	32,6	39,9	39,0	38,3

Графік на Рис. 3.10 наочно демонструє значний розрив між двома інструментами. У найкращому випадку цей розрив становить 25 секунд, що практично дорівнює часу виконання тестів Playwright у послідовному порядку.



	1	2	3	4	5	6	7	8	9	10
Selenium	72,4	51,8	39,7	35,4	40,2	46,0	39,5	39,9	36,5	38,3
playwright	28,4	15,3	10,8	9,0	9,2	9,7	8,0	10,5	11,0	11,6

Рисунок 3.10 – Графік порівняння Selenium та Playwright при виконанні тестів зі збоєм у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Mac

3.1.11 Виконання тестів без збою у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Windows

Результати отримані для Playwright з відображенням інтерфейсу без збою у продуктивності наочно показують, що незважаючи на збільшення кількості потоків, час виконання тестів залишається стабільним або виявляє невелике зменшення (Таб. 3.21). Це може свідчити про те, що під час відображення інтерфейсу браузера вплив додаткових потоків на продуктивність менший або навіть компенсується системними оптимізаціями.

Таблиця 3.21 – результати виконання тестів з Playwright без збою у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Windows

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	19,1	18,8	21,0	18,9	18,9	19,3
2	11,3	11,5	11,3	11,3	12,0	11,5
3	9,0	9,6	9,2	9,6	9,3	9,3
4	8,6	8,5	9,9	8,7	8,7	8,9
5	8,6	8,8	7,5	8,3	7,8	8,2
6	8,1	7,7	7,8	7,6	7,9	7,8
7	8,2	7,9	7,7	7,7	8,0	7,9
8	9,7	8,4	8,1	8,3	9,3	8,8
9	9,8	9,5	8,7	9,3	9,4	9,3

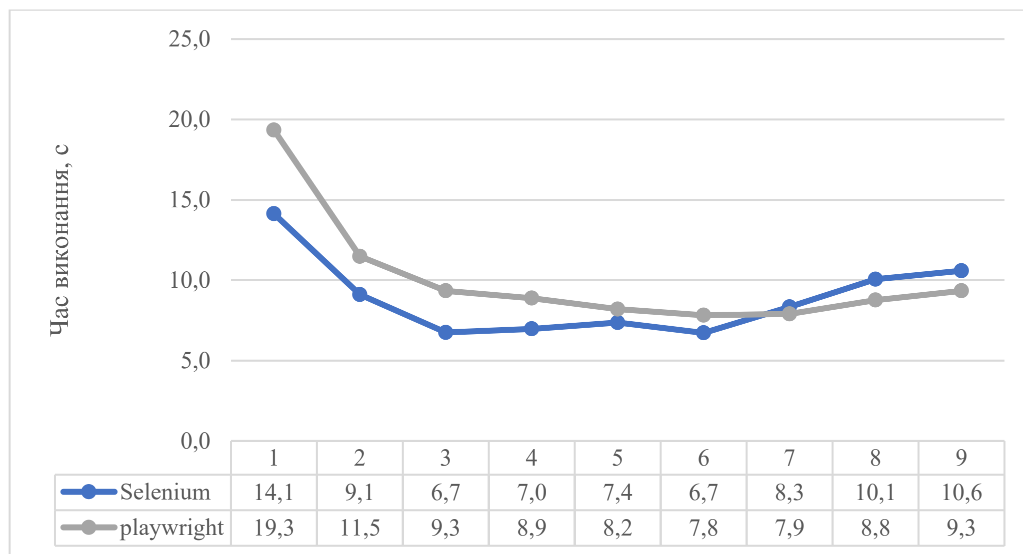
У випадку Selenium, визначений тренд деградації часу виконання тестів стає більш вираженим при збільшенні кількості потоків (Таб. 3.22). Навіть із досягненням мінімального часу в 6.3 секунди, величина цього часу стає значно вищою порівняно із тестами, які виконуються без відображення інтерфейсу. Такий виграш у часі не компенсує деградацію при збільшенні потоків, що може свідчити про те, що Selenium менше ефективний у випадках, коли тестові сценарії пов'язані із збільшеним завантаженням графічного інтерфейсу.

Таблиця 3.22 – результати виконання тестів з Selenium без збою у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Windows

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	14,0	14,5	14,2	14,0	13,9	14,1
2	9,8	8,4	8,5	8,9	9,9	9,1
3	6,8	6,6	7,4	6,5	6,5	6,7
4	7,3	7,1	6,5	6,5	7,5	7,0
5	7,3	6,8	7,1	7,5	8,0	7,4
6	7,5	6,6	6,6	6,3	6,5	6,7
7	9,5	9,2	8,7	6,8	7,5	8,3
8	10,2	10,8	9,7	9,7	9,9	10,1
9	10,6	10,8	10,5	10,2	10,9	10,6

На Рис. 3.11 чітко видно деградацію часу виконання у Selenium. Навіть при стабільних результатів у Playwright, ми можемо відзначити, що у Selenium найменший середній час спостерігається при використанні 3 та 6 потоків.

Це свідчить про значущі відмінності в реакції обох інструментів на збільшення кількості паралельно виконуваних тестів. Хоча Playwright може зберігати стабільність результатів, Selenium, здається, стикається зі збільшеною деградацією продуктивності. Це може вказувати на потребу оптимізації архітектури Selenium для забезпечення кращої ефективності у ситуаціях з великою кількістю одночасно запущених потоків.



	1	2	3	4	5	6	7	8	9
Selenium	14,1	9,1	6,7	7,0	7,4	6,7	8,3	10,1	10,6
playwright	19,3	11,5	9,3	8,9	8,2	7,8	7,9	8,8	9,3

Рисунок 3.11 – Графік порівняння Selenium та Playwright при виконанні тестів без збою у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Windows

3.1.12 Виконання тестів без збою у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Mac

У Таблиці 3.23 чітко видно, що результати, отримані за допомогою Playwright, не проявляють деградації часу виконання тестів, а мінімальний час складає 6 секунд. Порівняно із випадками зі збоєм, де можна спостерігати деградацію, результати без збою у продуктивності виявляються більш

стабільними та ефективними, з перевагою у часі на 2-3 секунди. Це підкреслює важливість обрання правильних стратегій тестування в залежності від умов і вимог проєкту для досягнення оптимальних результатів.

Таблиця 3.23 – результати виконання тестів з Playwright без збою у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Mac

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	21,1	20,7	20,7	20,9	20,6	20,8
2	12,4	12,2	12,2	12,3	12,3	12,3
3	8,2	8,2	8,3	8,2	8,2	8,2
4	8,4	8,4	8,4	8,2	8,5	8,4
5	7,1	6,9	6,9	6,8	7,3	7,0
6	7,6	6,9	7,1	7,7	7,1	7,3
7	7,2	7,8	7,2	7,4	7,4	7,4
8	7,5	7,5	7,8	8,0	7,7	7,7
9	6,2	6,1	6,4	6,0	6,5	6,2

Визначено, що проблема, пов'язана із версіями драйвера браузера, негативно впливає на оптимальність виконання тестів Selenium на операційній системі Mac, що відображено в Таблиці 3.24. Мінімальний час виконання становить 20.2 секунди. Це підкреслює важливість постійного моніторингу та оновлення драйверів для забезпечення ефективності тестового процесу та виключення факторів, що можуть спричиняти деградацію результатів. Для Playwright не існує необхідності у пошуку оптимальних версій, адже він має в собі автоматичне завантаження версій браузерів за допомогою команди *npm playwright install*.

Таблиця 3.24 – результати виконання тестів з Selenium без збою у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Mac

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	67,1	67,9	69,0	70,7	70,1	69,0
2	43,0	41,2	45,1	47,2	47,5	44,8
3	25,2	29,2	29,9	26,3	23,1	26,8
4	22,5	20,2	23,6	26,3	25,9	23,7
5	22,5	22,6	27,4	23,5	25,0	24,2
6	28,9	28,4	24,6	27,1	24,6	26,7
7	27,2	26,7	28,3	24,1	26,2	26,5
8	28,3	30,6	25,2	25,8	27,0	27,4
9	24,0	24,0	22,9	22,5	25,3	23,8

Виявлено, що Playwright досягає найменшого середнього часу виконання в розмірі 6.2 секунди при використанні 9 потоків (Рис. 3.12). У той час як Selenium показує свій найкращий результат, досягаючи часу в 23.7 секунди при використанні 4 потоків.

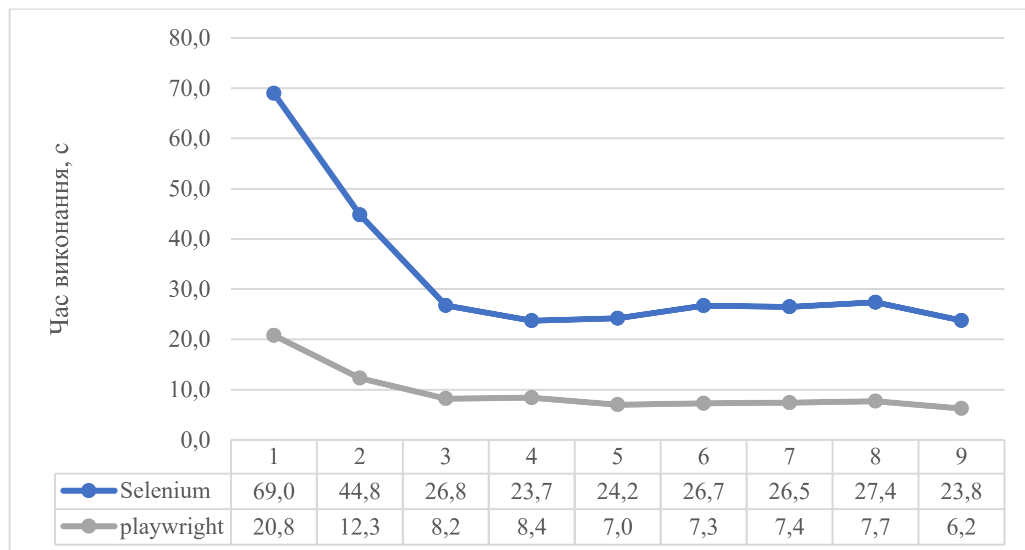


Рисунок 3.12 – Графік порівняння Selenium та Playwright при виконанні тестів без збою у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера на Mac

3.1.13 Виконання тестів зі збоєм у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Windows

Результати підтверджують, що Firefox, як браузер, виявляється менш продуктивним у порівнянні з Chrome. Найменший час виконання для Playwright складає 15.7 секунд, тоді як для Selenium він становить 17.7 секунд (Таблиця 3.25 - Таблиця 3.26).

Таблиця 3.25 – результати виконання тестів з Playwright зі збоєм у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Windows

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	31,3	30,5	31,2	30,4	31,3	30,9
2	25,8	21,0	19,8	20,1	22,5	21,8
3	21,4	19,9	19,5	17,3	19,5	19,5
4	20,3	19,6	19,8	18,1	15,5	18,7
5	15,7	15,9	18,2	20,3	24,7	19,0
6	22,3	22,3	23,8	22,6	20,8	22,4
7	23,3	23,8	23,7	25,9	25,2	24,4
8	24,7	24,0	29,8	24,6	25,2	25,7
9	24,0	24,7	22,5	28,6	24,4	24,8
10	23,9	24,7	27,1	22,5	21,2	23,9

Таблиця 3.26 – результати виконання тестів з Selenium зі збоєм у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Windows

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	43,1	42,2	43,9	42,9	43,8	43,2
2	23,8	24,2	22,2	22,0	23,2	23,1
3	19,1	19,7	23,0	18,2	21,2	20,2
4	24,1	18,3	22,9	14,0	22,8	20,4
5	22,5	22,7	22,3	22,4	16,9	21,3
6	22,1	22,5	20,8	20,9	17,7	20,8
7	20,8	22,4	21,5	22,4	23,4	22,1
8	21,8	21,0	22,9	19,6	20,2	21,1
9	20,5	18,5	19,7	20,6	21,1	20,1
10	22,3	18,8	19,0	17,7	18,3	19,2

На графіку нижче видно як результати мають схожі значення (Рис. 3.13). Найоптимальніший середній час був отриманий на 9 та 4 потоках, для Selenium та Playwright, відповідно.

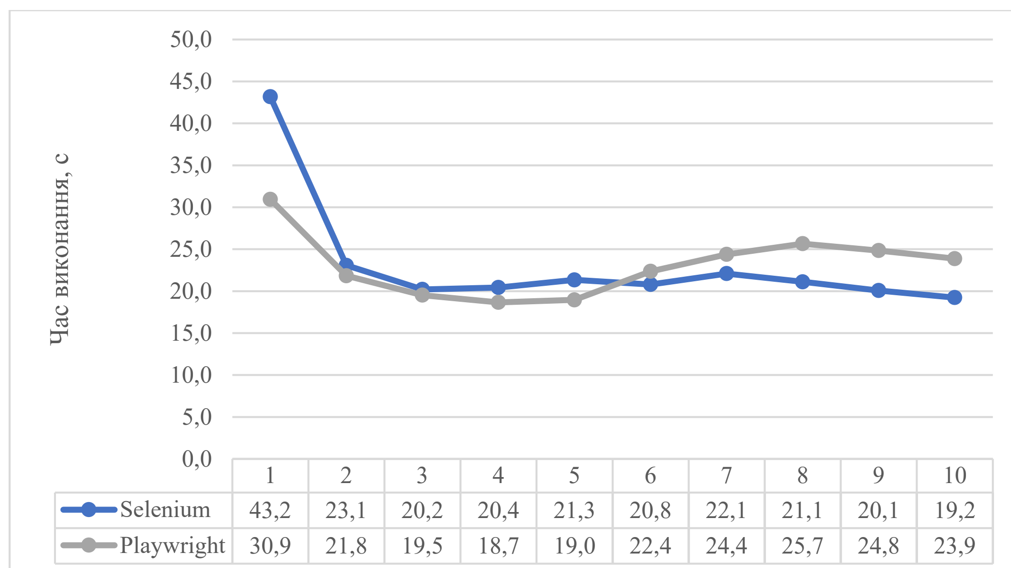


Рисунок 3.13 – Графік порівняння Selenium та Playwright при виконанні тестів зі збоєм у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Windows

3.1.14 Виконання тестів зі збоєм у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Mac

За результатами тестів видно, що на операційній системі Mac використання Playwright у Firefox демонструє трошки кращу продуктивність, порівняно із версією для Windows. Мінімальний час виконання склав 10.2 секунди (Таблиця 3.27), що є швидшим результатом, ніж 15.7 секунд на Windows.

Таблиця 3.27 – результати виконання тестів з Playwright зі збоєм у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Mac

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	29,7	29,5	30,0	29,6	29,7	29,7
2	16,6	16,0	15,7	16,2	15,9	16,1
3	11,9	11,9	11,8	11,7	12,2	11,9
4	10,6	10,2	10,2	10,6	10,4	10,4
5	11,1	10,7	11,2	10,8	10,8	10,9
6	11,7	11,6	11,7	11,6	11,1	11,5
7	12,6	10,2	12,8	12,0	12,7	12,1
8	13,6	13,2	11,6	12,3	13,4	12,8
9	14,6	19,4	14,4	14,7	14,6	15,5
10	19,3	18,5	22,6	16,2	17,0	18,7

Аналогічно Playwright, і для Selenium на операційній системі Mac для Firefox виявляється трошки кращий результат у порівнянні з версією для Windows. Мінімальний час виконання для Selenium на Mac становить 17 секунд, що є трошки швидшим, ніж на Windows на 700 мілісекунд (Таблиця 3.28).

Таблиця 3.28 – результати виконання тестів з Selenium зі збоєм у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Mac

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	48,5	46,9	46,7	46,6	46,1	47,0
2	28,2	27,9	27,4	30,4	30,6	28,9
3	25,1	24,5	27,0	27,6	24,3	25,7
4	21,2	23,4	24,9	21,5	20,3	22,3
5	17,4	17,6	19,5	19,6	17,0	18,2
6	19,7	19,5	19,9	17,5	17,0	18,7
7	20,6	20,8	19,4	18,7	19,6	19,8
8	21,9	20,6	22,0	20,9	21,0	21,3
9	24,3	20,6	27,2	27,8	24,3	24,8
10	24,2	26,7	20,9	24,2	27,3	24,6

Результати вказують на наявність деградації у часі виконання тестів на операційній системі Mac для обох інструментів автоматизації (Рисунок 3.14). Після 5 потоків спостерігається помітне погіршення часу виконання. Найкращий середній час досягається при використанні 4 та 5 потоків для Playwright та Selenium, відповідно.

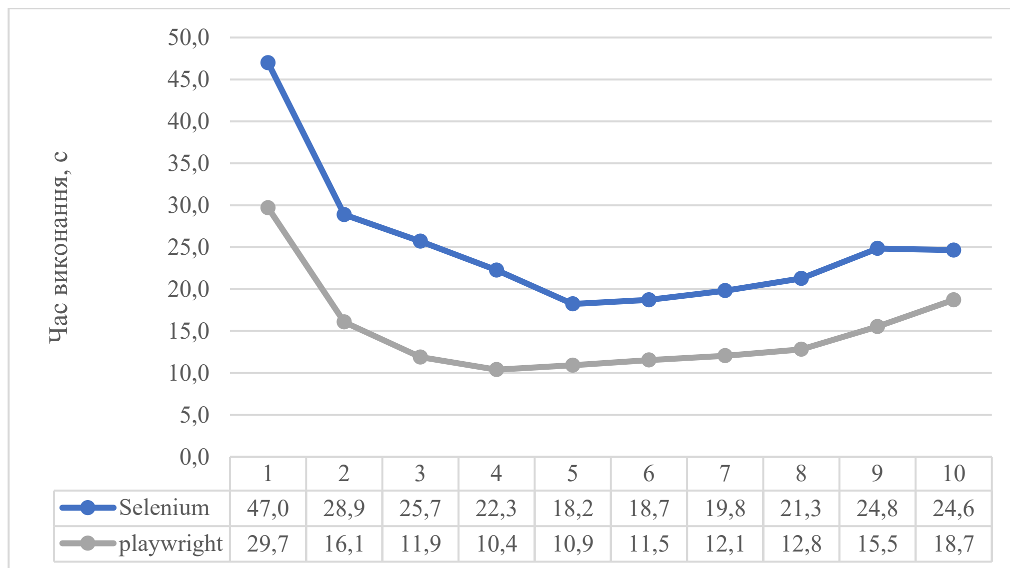


Рисунок 3.14 – Графік порівняння Selenium та Playwright при виконанні тестів зі збоєм у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Mac

3.1.15 Виконання тестів без збою у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Windows

Мінімальний час для Playwright на операційній системі Windows було зафіксовано при використанні 4 потоків і склав 13.3 секунди (Таблиця 3.29).

Таблиця 3.29 – результати виконання тестів з Playwright без збою у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Windows

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	23,7	23,6	23,5	23,2	23,5	23,5
2	15,2	16,9	18,4	19,7	15,5	17,1
3	13,8	16,4	15,8	16,1	16,9	15,8
4	15,2	15,1	15,3	14,6	13,3	14,7
5	14,9	14,9	15,7	16,4	15,6	15,5
6	16,0	18,4	17,2	15,8	15,6	16,6
7	18,0	19,1	16,2	18,8	17,4	17,9
8	17,9	17,6	19,7	20,8	21,1	19,4
9	21,9	21,3	21,2	16,1	18,0	19,7

Зафіксовано, що навіть при непоганих результатах, Selenium на операційній системі Mac не зміг досягти рівня продуктивності, який був досягнутий Playwright. Мінімальний час для Selenium на Mac склав 15 секунд (Таблиця 3.30).

Таблиця 3.30 – результати виконання тестів з Selenium без збою у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Windows

К-ість потоків	**Номер спроби**					**Ср. час**
	1	**2**	**3**	**4**	**5**	
1	33,7	33,6	34,5	34,2	33,9	34,0
2	23,4	20,8	19,9	23,2	22,7	22,0
3	17,8	17,8	15,7	19,1	19,1	17,9
4	17,8	17,4	16,5	17,2	18,2	17,4
5	15,0	16,3	17,8	15,9	16,8	16,4
6	17,4	15,7	16,3	17,0	17,5	16,8
7	17,6	17,6	16,9	16,0	17,5	17,1
8	16,2	18,6	18,5	18,9	19,1	18,3
9	16,9	18,0	16,7	18,6	18,9	17,8

На 6 потоках виявлено майже ідентичні результати для обох інструментів, хоча Playwright продемонстрував швидкість на 0.2 секунди вищу (Рис 3.15).

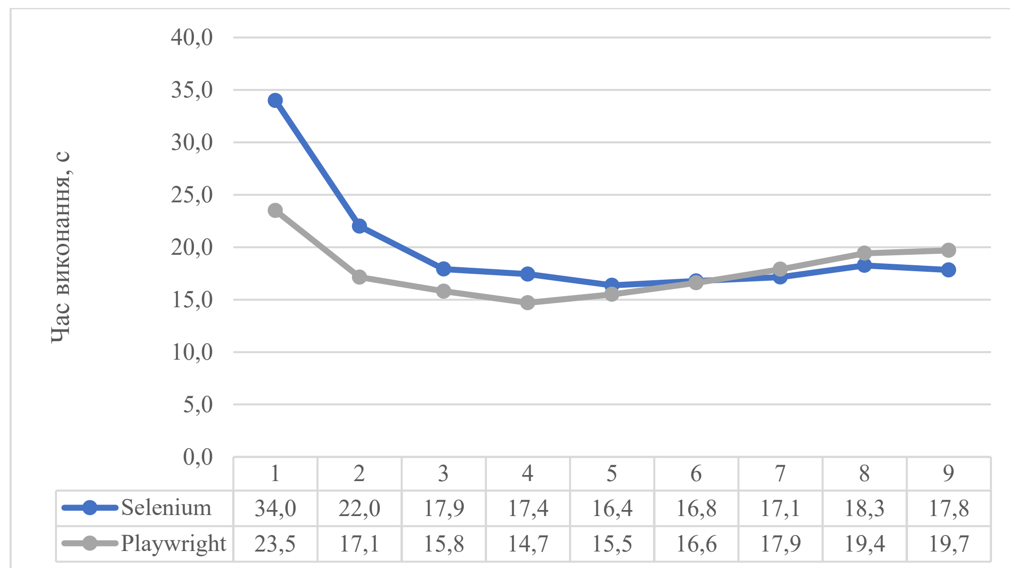


Рисунок 3.15 – Графік порівняння Selenium та Playwright при виконанні тестів без збою у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Windows

3.1.16 Виконання тестів без збою у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Mac

Як і у попередніх випадках, на Mac результати Playwright значно кращі (Таб. 3.31). Проте, наявна деградація часу з кількістю часу.

Таблиця 3.31 – результати виконання тестів з Playwright без збою у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Mac

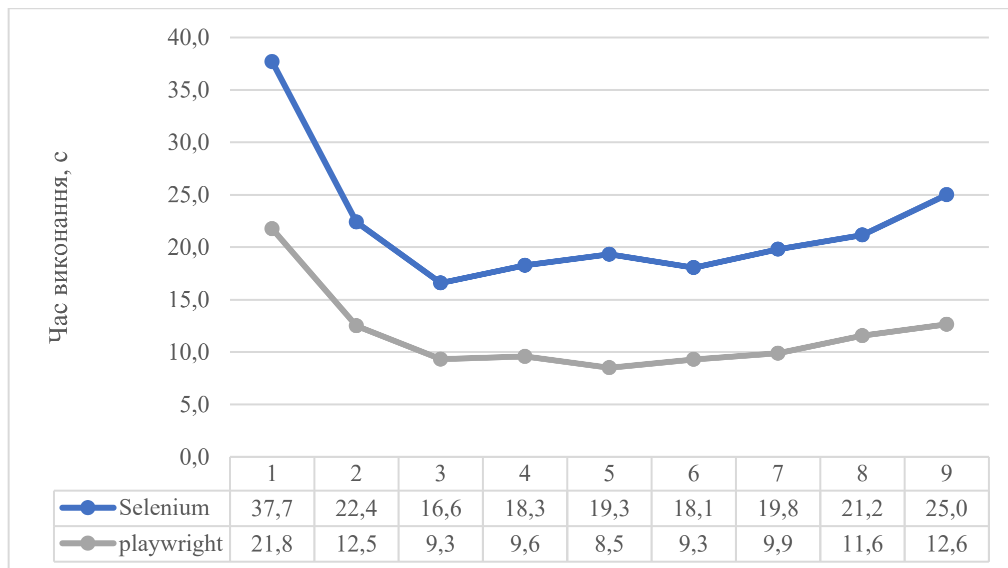
К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	21,8	21,8	22,0	21,5	21,7	21,8
2	12,4	12,2	12,8	12,5	12,6	12,5
3	9,7	9,9	9,2	8,9	8,9	9,3
4	9,6	9,4	9,9	9,3	9,7	9,6
5	8,7	8,5	8,5	8,2	8,6	8,5
6	9,4	9,4	9,3	9,1	9,3	9,3
7	9,7	10,5	10,7	8,8	9,7	9,9
8	10,0	11,1	9,7	10,7	16,3	11,6
9	11,8	11,7	14,6	14,7	10,4	12,6

Ймовірно, обидва інструменти, Playwright і Selenium, стикаються з певними обмеженнями або особливостями операційної системи Mac OS при відображенні браузера, що призводить до деградації часу виконання тестів (Таб. 3.31 та Таб.3.32). Можливо, в подальших дослідженнях можна спробувати знайти оптимальні налаштування або знайти шляхи для подолання цих обмежень.

Таблиця 3.32 – результати виконання тестів з Selenium без збою у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Mac

К-ість потоків	Номер спроби					Ср. час
	1	2	3	4	5	
1	38,4	37,7	37,4	37,5	37,4	37,7
2	23,3	22,4	22,8	21,8	21,7	22,4
3	16,5	16,6	16,5	16,6	16,7	16,6
4	18,7	17,7	17,9	18,5	18,6	18,3
5	18,8	20,7	19,7	17,7	19,6	19,3
6	19,0	18,6	19,3	17,3	16,1	18,1
7	17,9	19,1	20,3	18,3	23,5	19,8
8	21,6	20,4	21,4	20,6	21,7	21,2
9	24,7	20,9	27,4	27,8	24,1	25,0

Різниця у часі виконання тестів між Playwright та Selenium на Mac OS є помітною, і її діапазон досягає 7 секунд. Мінімальні значення становлять 8.5 секунд для 5 потоків з Playwright та 16.6 секунд для 3 потоків з Selenium. Ці відмінності можуть бути визначені різними підходами та оптимізаціями в реалізації обох інструментів для Mac OS.

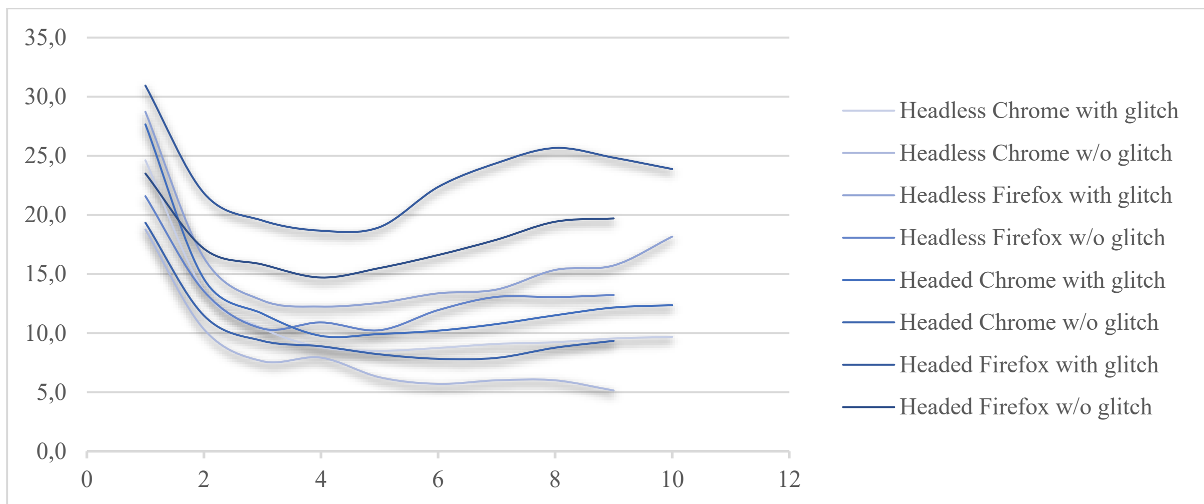


	1	2	3	4	5	6	7	8	9
Selenium	37,7	22,4	16,6	18,3	19,3	18,1	19,8	21,2	25,0
playwright	21,8	12,5	9,3	9,6	8,5	9,3	9,9	11,6	12,6

Рисунок 3.16 – Графік порівняння Selenium та Playwright при виконанні тестів без збою у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера на Mac

3.2 Аналіз результатів виконання наскрізного тестування у багатопоточному середовищі

Отже з отриманих результатів можна зробити певні висновки по динаміці графіків. Згідно з Рис. 3.17 Playwright на Windows має гірші показники для браузеру Firefox та найбільшої оптимальності виконання досягає саме під час використання Chromium. Також бажано заздалегідь відокремити тести, які мають проблеми у продуктивності. Такі тести називаються нестабільними і Playwright може самостійно їх помічати за часом виконання тесту.

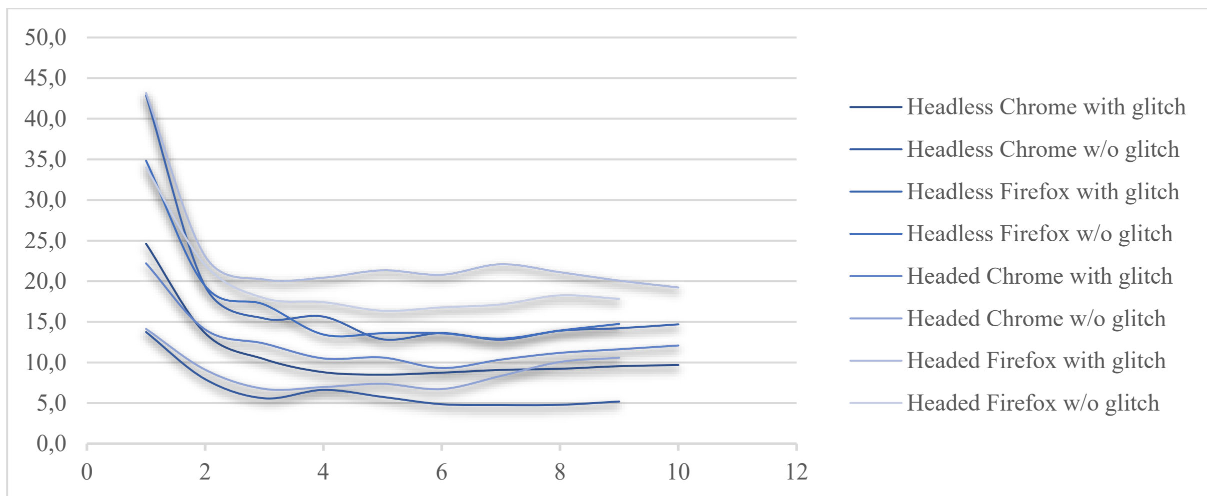


(***headless*** – без відображення інтерфейсу браузера; ***headed*** – з відображенням інтерфейсу браузера; ***w/o glitch*** - без збою у продуктивності застосунку; ***with glitch*** – зі збоєм у продуктивності застосунку)

Рисунок 3.17 – Графік відображення залежності часу від різних конфігурацій для Playwright на Windows

Як можна побачити на Рис 3.18 було отримано гірші результати для Firefox та кращі для Chrome. Також можна помітити значну стабільність результатів та їх стабільно гіперболічний характер. При оптимальному використанні потоків діапазон різниці часу дорівнює приблизно 15 секунд при різних конфігураціях.

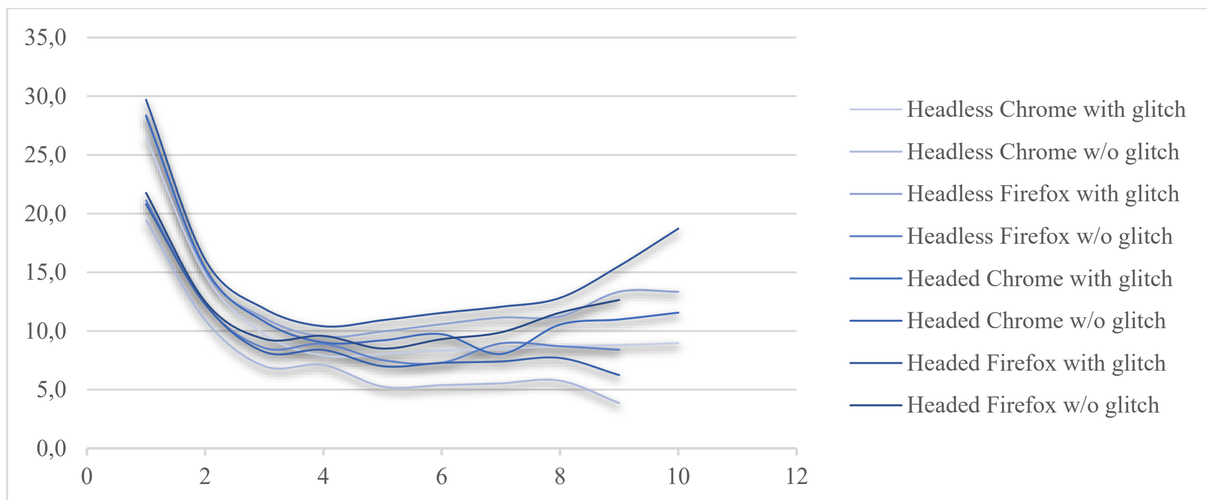
Слід зазначити з Рис. 3.17 та Рис 3.18, що Firefox має кращі показники саме на Windows для Playwright, у той час як для Mac найшвидший час отримує Chromium. Тож якщо перед тестувальниками буде поставати питання підтримки якості на різних браузерах, то краще запускати тести для Chrome на Mac, а для Firefox на Windows. Таким чином буде досягнуто максимального скорочення часу виконання процесу наскрізного тестування.



(***headless*** – без відображення інтерфейсу браузера; ***headed*** – з відображенням інтерфейсу браузера; ***w/o glitch*** - без збою у продуктивності застосунку; ***with glitch*** – зі збоєм у продуктивності застосунку)

Рисунок 3.18 – Графік відображення залежності часу від різних конфігурацій для Playwright на Mac

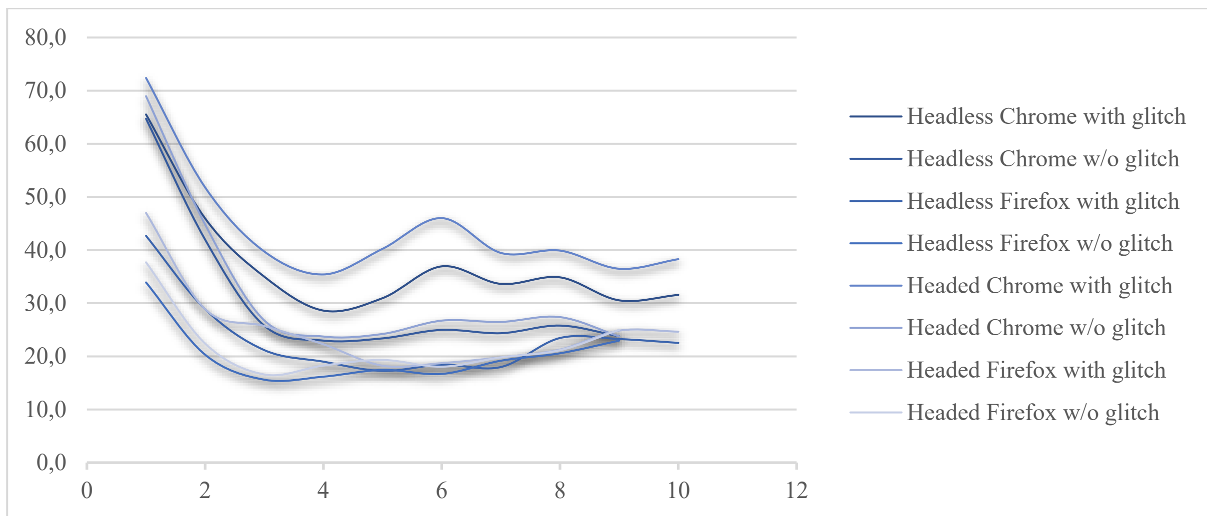
Натомість Selenium отримав протилежні результати для обох ОС. Selenium на Windows (Рис. 3.19) має менший діапазон розходжень часу при використанні більше 2 потоків та менше 8. Він складає до 10 секунд, що є показником стабільності результатів не залежно від конфігурації. Гіперболічність ліній каже про підпорядкованість оптимальному розподіленню тестів муж потоками.



(***headless*** – без відображення інтерфейсу браузера; ***headed*** – з відображенням інтерфейсу браузера; ***w/o glitch*** - без збою у продуктивності застосунку; ***with glitch*** – зі збоєм у продуктивності застосунку)

Рисунок 3.19 – Графік відображення залежності часу від різних конфігурацій для Selenium на Windows

Ситуація на Mac OC набагато гірша ніж на Windows (Рис 3.20). Мінімальний час якого було досягнуто на цій ОС дорівнює 15.6 секунд, іншими словами на кожен тест витрачається майже 2 секунди. Стан результатів Chrome значно гірший, адже як було раніше зауважено існує проблема у використані драйверу відповідної версії Chrome, що унеможливлює ефективне використання цього браузера як вручну, так і для автоматизованого тестування. Стабільність тестів на цій ОС для Selenium найгірша з усіх чотирьох результатів (Рис. 3.17 – Рис. 3.19). Діапазон значень коливається від 15 до 20 секунд, що перевищує попередні результати.



(***headless*** – без відображення інтерфейсу браузера; ***headed*** – з відображенням інтерфейсу браузера; ***w/o glitch*** – без збою у продуктивності застосунку; ***with glitch*** – зі збоєм у продуктивності застосунку)

Рисунок 3.20 – Графік відображення залежності часу від різних конфігурацій для Selenium на Mac

Підсумовуючи, найменший час було досягнуто для Playwright на Mac OC без збою у продуктивності застосунку у браузері Chrome, який дорівнює 3.86 секунд (Таб. 3.33). Проте, відстаючи на 890 мілісекунд, друге місце за найменшим часом посів Selenium на Windows. У Таблиці 3.33 наведено усі мінімальні значення часу для кожної конфігурації та ОС.

Таблиця 3.33 – мінімальні результати виконання тестів з різними конфігураціями та їх підсумки для Selenium та Playwright на різних ОС

№	Конфігурація	Play-wright на Windows	Selenium на Windows	Play-wright на Mac	Selenium на Mac	Мін. знач.
1	**Зі збоєм у продуктивності застосунку у Chrome без відображення інтерфейсу браузера**	8,50 с	8,19 с	7,90 с	28,61 с	7,90 с
2	**Без збою у продуктивності застосунку у**	5,14 с	4,75 с	3,86	22,96 с	3,86 с

№	Конфігурація	Play-wright на Windows	Selenium на Windows	Play-wright на Mac	Selenium на Mac	Мін. знач.
	Chrome без відображення інтерфейсу браузера					
3	**Зі збоєм у продуктивності застосунку у Firefox без відображення інтерфейсу браузера**	12,24 с	12,80 с	9,52 с	17,26 с	9,52 с
4	**Без збою у продуктивності застосунку у Firefox без відображення інтерфейсу браузера**	10,24 с	12,94 с	7,30 с	15,60 с	7,30 с
5	**Зі збоєм у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера**	9,76 с	9,31 с	8,04 с	35,41 с	8,04 с
6	**Без збою у продуктивності застосунку у Chrome з відображенням інтерфейсу браузера**	7,82 с	6,72 с	6,24 с	23,72 с	6,24 с
7	**Зі збоєм у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера**	18,66 с	19,23 с	10,40 с	18,23 с	10,40 с
8	**Без збою у продуктивності застосунку у Firefox з відображенням інтерфейсу браузера**	14,70 с	16,36 с	8,50 с	16,58 с	8,50 с
	Мін. знач.	5,14 с	4,75 с	3,86 с	15,60 с	3,86 с

Нижче на Рис. 3.21 наведено графічне представлення Таб. 3.33. Цю діаграму можна умовно поділити на дві частини – без відображення інтерфейсу застосунку та з відображенням. Аналізуючи ці результати можна прийти до висновку, що без відображення інтерфейсу застосунку можна досягти меншого

часу виконання тестів. Це обумовлено різними чинниками у тому числі витрачання можливостей процесору на малювання інтерфейсу.

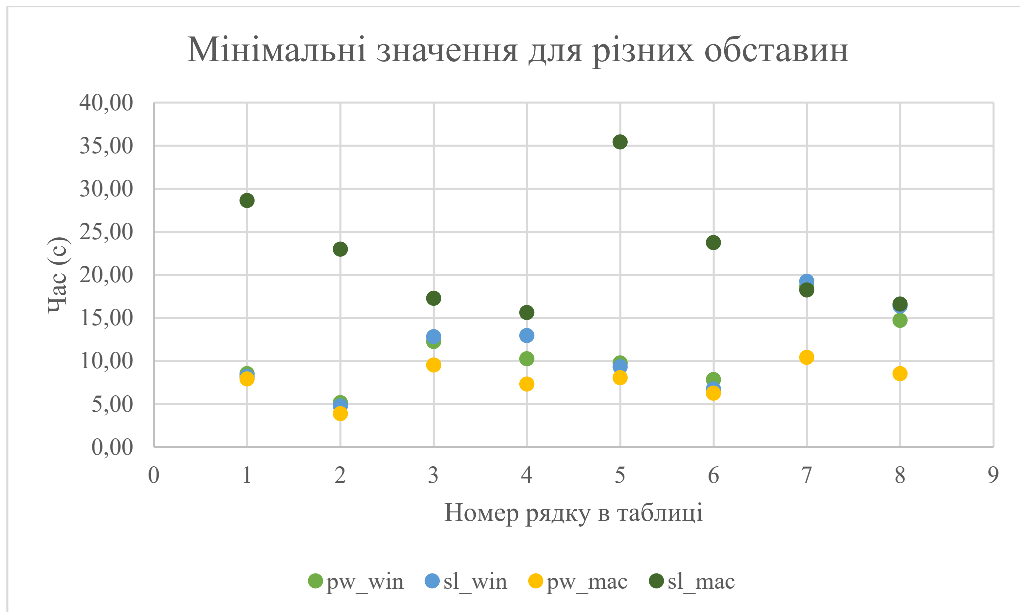


Рисунок 3.21 – Графік відображення часу при різних конфігураціях (***1-8*** – номер конфігурації у Таблиці 3.33; ***pw_win*** – Playwright на Windows; ***sl_win*** – Selenium на Windows; ***pw_mac*** – Playwright на Mac; ***sl_mac*** – Selenium на Mac)

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було розроблено два проєкти для автоматизації процесу наскрізного тестування у багатопоточному середовищі з використанням Selenium та Playwright, на мовах JavaScript та TypeScript Для досягнення поставленої мети, дослідження ефективності процесу наскрізного тестування сучасними засобами для автоматизації, було проведено запуск розроблених тестів при різних конфігураціях та ОС. Мінімальний час, який дорівнює 3.86 секунд, було досягнуто використовуючи Playwright на Mac ОС без збою у продуктивності застосунку у браузері Chrome.

Багатопоточний запуск програм – це дуже складний процес, який дуже часто складно оптимізувати. Це стосується і автоматизованого процесу наскрізного тестування. Багато чинників впливають на його швидкість виконання, у тому числі кількість ресурсів, оптимальність тестів, конфлікти та багато інших.

Узагальнюючи усі результати різних конфігурацій, можна зазначити, який із засобів автоматизації, серед проаналізованих, краще обрати для виконання тестів у багатопоточному середовищі:

— Якщо необхідно тестувати функційність лише на Windows ОС у Chrome, то Selenium впорається з цим краще ніж Playwright. Однак, потрібно буде витратити багато часу на налаштування паралельного запуску.
— Якщо проєкт потребує тестування на різних платформах та браузерах більш доречно буде обрати Playwright як засіб автоматизації.

Наукова новизна цього дослідження допоможе обрати інструменти для автоматизації тестів для різних реальних проєктів, які мають сумніви щодо ефективності їх у багатопоточному середовищі та легкості адаптації до різних потреб.

ПЕРЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Erich M. Oct. 26, 1992: Software Glitch Cripples Ambulance Service [електронний ресурс] / Musick Erich. – 2009. – режим доступу до ресурсу: https://www.wired.com/2009/10/1026london-ambulance-computer-meltdown/.
2. ОГЛЯД ВИДІВ ТЕСТУВАННЯ [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://training.qatestlab.com/blog/technical-articles/review-the-types-of-testing/.
3. Poliarush M. Future of test automation in 2023 [Електронний ресурс] / Mykhailo Poliarush. – 2022. – Режим доступу до ресурсу: https://testomat.io/blog/future-of-test-automation/.
4. Gazala S. 35 Best Test Automation Frameworks for 2024 [Електронний ресурс] / Saniya Gazala. – 2023. – Режим доступу до ресурсу: https://www.lambdatest.com/blog/best-test-automation-frameworks/.
5. GH. Best Test Automation Frameworks (Updated 2023) [Електронний ресурс] / GH, A. Badkar. – 2023. – Режим доступу до ресурсу: https://www.browserstack.com/guide/best-test-automation-frameworks.
6. The Selenium Browser Automation Project [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://www.selenium.dev/documentation/.
7. Playwright [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://playwright.dev/.
8. Das S. Parallel Testing: The Essential Guide [Електронний ресурс] / Sourojit Das. – 2022. – Режим доступу до ресурсу: https://www.browserstack.com/guide/what-is-parallel-testing.
9. tajawal. Page Object Model (POM) | Design Pattern [Електронний ресурс] / tajawal. – 2018. – Режим доступу до ресурсу: https://medium.com/tech-tajawal/page-object-model-pom-design-pattern-f9588630800b.
10. Smirnov A. How to use the Data Provider pattern in the project. [Електронний ресурс] / Anton Smirnov. – 2020. – Режим доступу до ресурсу:

https://antony-s-smirnov.medium.com/how-to-use-the-data-provider-pattern-in-the-project-ea12430d9275.

11. Playwright Ambassadors. Fixtures [Електронний ресурс] / Playwright Ambassadors. – 2022. – Режим доступу до ресурсу: https://playwright.dev/docs/test-fixtures.

ДОДАТОК А

Код програми

playwright\fixtures\fixture.ts

```
import { test as base } from '@playwright/test';
import LoginPage from '../pages/LoginPage';
import ProductsPage from '@pages/ProductsPage';

type TestOptions = {
  loginPage: LoginPage;
  productsPage: ProductsPage;
};

export const test = base.extend<TestOptions>({
  page: async ({ baseURL, page }, use) => {
    await page.goto(baseURL, { waitUntil: 'networkidle' });
    await use(page);
  },

  loginPage: async ({ page }, use) => {
    await use(new LoginPage(page));
  },

  productsPage: async ({ page }, use) => {
    await use(new ProductsPage(page));
  },
});
```

---

playwright\pages\LoginPage.ts

```
import { Locator, Page } from '@playwright/test';

export default class LoginPage {
  readonly input: {
    username: Locator;
```

```
  password: Locator;
};

readonly label: {
  siteName: Locator;
};

readonly button: {
  login: Locator;
};

readonly errorMessage: Locator;

constructor(page: Page) {
  this.input = {
    username: page.locator('#user-name'),
    password: page.locator('#password'),
  };

  this.label = {
    siteName: page.locator('div.login_logo'),
  };

  this.button = {
    login: page.locator('#login-button'),
  };

  this.errorMessage = page.locator('div.error > h3');
}

public async login(username: string, password: string) {
  await this.input.username.fill(username);
  await this.input.password.fill(password);
  await this.button.login.click();
```

```
  }

  public async clearAllInputs() {
    await this.input.username.clear();
    await this.input.password.clear();
  }

  public async getErrorMessage() {
    return await this.errorMessage.textContent();
  }

  public async getSiteLabel() {
    return await this.label.siteName.textContent();
  }
}
```

---

playwright\pages\ProductsPage.ts

```
import { Locator, Page } from '@playwright/test';

export default class ProductsPage {
  readonly link: {
    cart: Locator;
  };

  constructor(page: Page) {
    this.link = {
      cart: page.locator('a.shopping_cart_link'),
    };
  }
}
```

---

playwright\test_data\login-data.json

```json
{
  "positiveCases": [
    {
      "username": "standard_user"
    },
    {
      "username": "problem_user"
    },
    {
      "username": "performance_glitch_user"
    }
  ],
  "negativeCases": [
    {
      "username": "someone",
      "password": "",
      "error": "Epic sadface: Password is required"
    },
    {
      "username": "",
      "password": "someone",
      "error": "Epic sadface: Username is required"
    },
    {
      "username": "someone",
      "password": "someone",
      "error": "Epic sadface: Username and password do not match any user
in this service"
    },
    {
      "username": "standard_user",
      "password": "someone",
```

```
      "error": "Epic sadface: Username and password do not match any user
in this service"
    },
    {
      "username": "locked_out_user",
      "password": "secret_sauce",
      "error": "Epic sadface: Sorry, this user has been locked out."
    },
    {
      "username": "someone",
      "password": "secret_sauce",
      "error": "Epic sadface: Username and password do not match any user
in this service"
    }
  ]
}
```

playwright\tests\login.test.ts

```
import { expect } from '@playwright/test';
import loginData from '@data/login-data.json';
import { test } from '@fixture/fixture';

test.describe('Login test', async () => {
  test(`Smoke test on elements`, async ({ loginPage }) => {
    const siteLabel = 'Swag Labs';
    await expect(loginPage.label.siteName, `Site name should be
visible`).toBeVisible();
    await expect(loginPage.input.username, `Input field for username
should be visible`).toBeVisible();
    await expect(loginPage.input.password, `Input field for password
should be visible`).toBeVisible();
```

```
    await expect(loginPage.button.login, `Login button should be
visible`).toBeVisible();
    expect(await loginPage.getSiteLabel(), `Site label should be
${siteLabel}`).toBe(siteLabel);
    await expect(loginPage.input.password, `Password should be
masked`).toHaveAttribute('type', 'password');
  });

  for (const cases of loginData.positiveCases) {
    test(`Positive login test: ${cases.username}`, async ({ loginPage,
productsPage }) => {
      await loginPage.login(cases.username, process.env.password);
      await expect(productsPage.link.cart, `Cart link should be
visible`).toBeVisible();
    });
  }

  for (const cases of loginData.negativeCases) {
    test(`Negative login test: ${cases.username}, ${cases.password}`,
async ({ loginPage }) => {
      await loginPage.clearAllInputs();
      await loginPage.login(cases.username, cases.password);
      await expect(loginPage.errorMessage, `Error message should be
visible`).toBeVisible();
      expect(await loginPage.getErrorMessage(), `Error message should be
${cases.error}`).toBe(cases.error);
    });
  }
});
```

playwright\playwright.config.ts

```
import { defineConfig, devices } from '@playwright/test';
```

```
export default defineConfig({
  testDir: './tests',
  fullyParallel: true,
  forbidOnly: !!process.env.CI,
  retries: process.env.CI ? 2 : 0,
  workers: process.env.CI ? 1 : 1,
  reporter: 'html',
  use: {
    trace: 'on-first-retry',
    baseURL: 'https://www.saucedemo.com/',
    headless: !true,
  },
  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
    },
    {
      name: 'firefox',
      use: { ...devices['Desktop Firefox'] },
    },
  ],
});
```

---

selenium\pages\LoginPage.js

```
import { By } from 'selenium-webdriver';

export default class LoginPage {
  constructor(driver) {
    this.driver = driver;
    this.input = {
```

```
      username: driver.findElement(By.css('#user-name')),
      password: driver.findElement(By.css('#password')),
    };

    this.label = {
      siteName: driver.findElement(By.css('div.login_logo')),
    };

    this.button = {
      login: driver.findElement(By.css('#login-button')),
    };

    this.errorMessage = driver.findElement(By.css('div.error > h3'));
  }

  async login(username, password) {
    await this.input.username.sendKeys(username);
    await this.input.password.sendKeys(password);
    await this.button.login.click();
  }

  async clearAllInputs() {
    await this.input.username.clear();
    await this.input.password.clear();
  }

  async getErrorMessage() {
    return await (await this.getErrorElement()).getText();
  }

  async getErrorElement() {
    return await this.driver.findElement(By.css('div.error > h3'));
  }
```

```
  async getSiteLabel() {
    return await this.label.siteName.getText();
  }
}
```

selenium\pages\ProductsPage.js

```
import { By } from 'selenium-webdriver';

export default class ProductsPage {
  constructor(driver) {
    this.link = {
      cart: driver.findElement(By.css('a.shopping_cart_link')),
    };
  }
}
```

selenium\tests\parallel_1\test.spec.js

```
import { options } from '../../selenium.config.js';
import assert from 'assert';
import { setupDriver, setupParams } from '../../global/setup.js';
import LoginPage from '../../pages/LoginPage.js';
import loginData from '../../test_data/login-data.json' assert { type:
'json' };
import ProductsPage from '../../pages/ProductsPage.js';

describe('First script', function () {
  this.timeout(50000);
  let driver;
  let loginPage;
```

```
  this.beforeAll(async function () {
    await setupParams(this);
  });

  beforeEach(async function () {
    this.timeout(50000);
    driver = await setupDriver();
    await driver.get(options.baseURL);
    loginPage = new LoginPage(driver);
  });

  it(`Smoke test on elements`, async () => {
    const siteLabel = 'Swag Labs';
    assert.ok((await loginPage.label.siteName.isDisplayed()) == true,
`Site name should be visible`);
    assert.ok((await loginPage.input.username.isDisplayed()) == true,
`Input field for username should be visible`);
    assert.ok((await loginPage.input.password.isDisplayed()) == true,
`Input field for password should be visible`);
    assert.ok((await loginPage.button.login.isDisplayed()) == true,
`Login button should be visible`);
    assert.equal(await loginPage.getSiteLabel(), siteLabel, `Site label
should be ${siteLabel}`);
    assert.equal(await loginPage.input.password.getAttribute('type'),
'password', `Password should be masked`);
  });

  for (const cases of loginData.positiveCases) {
    it(`Positive login test: ${cases.username}`, async () => {
      await loginPage.login(cases.username, process.env.password);
      const productsPage = new ProductsPage(driver);
      assert.ok((await productsPage.link.cart.isDisplayed()) == true,
`Cart link should be visible`);
```

```
    }).timeout(10000);
  }

  for (const cases of loginData.negativeCases) {
    it(`Negative login test: ${cases.username}, ${cases.password}`, async
() => {
      await loginPage.clearAllInputs();
      await loginPage.login(cases.username, cases.password);
      assert.ok((await (await loginPage.getErrorElement()).isDisplayed())
== true, `Error message should be visible`);
      assert.equal(await loginPage.getErrorMessage(), cases.error, `Error
message should be visible`);
    });
  }

  afterEach(async () => {
    this.timeout(50000);
    await driver.quit();
  });
});
```

selenium\selenium.config.js

```
export const options = {
  baseURL: 'https://www.saucedemo.com/',
  timeout: 10 * 10000,
  disableExtensions: false,
  windowSize: {
    maximized: true,
  },
  disableGpu: true, // required for Windows,
  headless: !true,
  browserDrivers: {
```

```
    name: 'firefox',
    path: '../drivers/geckodriver.exe',
    //   name: 'chrome',
    //   path: '../drivers/chrome.exe',
  },
};
```

ДОДАТОК Б

Перелік файлів на диску

Ім’я файла	**Опис**
Пояснювальні документи	
Диплом_Ларикова.doc	Пояснювальна записка роботи. Документ Word.
Диплом_Ларикова.pdf	Пояснювальна записка роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація_Ларикова.ppt	Презентація роботи