МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Організація тестування Веб-застосунків засобами фреймворку Playwright**

Виконав: студент 4 курсу, групи 6.1212-пзс
спеціальності 121 Інженерія програмного Забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
Д.Ю. Самонов
(ініціали та прізвище)
Керівник доцент, к.ф.-м.н. Г.П. Коломоєць
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент Директор ТОВ «Дісітел»
П. О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 Інженерія програмного забезпечення
(код та назва)

Освітня програма Програмне забезпечення систем
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри Тетяна КРИТСЬКА

“ 01 ” березня 2026 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Самонову Дмитру Юрійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Організація тестування Веб-застосунків засобами фреймворку Playwright

керівник роботи Коломоєць Геннадій Павлович, к.ф.-м.н., доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджена наказом ЗНУ від 23.12.2025 р. № 1830-с

2. Строк подання студентом кваліфікаційної роботи 02.06.2026
3. Вихідні дані бакалаврської роботи
   - комплект нормативних документів ;
   - технічне завдання до роботи.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   - огляд та збір літератури стосовно теми кваліфікаційної роботи;
   - гляд та аналіз існуючих рішень та аналогів;
   - дослідження підходів до системного тестування веб-застосунків;
   - розробка автоматизованого тестового проєкту та його опис;
   - визначення вимог і тестових сценаріїв для веб-застосунку;
   - розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) слайдів презентації

6. Консультанти розділів бакалаврської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата
		Завдання прийняв

7. Дата видачі завдання 01.03.2026

КАЛЕНДАРНИЙ ПЛАН

з/п	Назва етапів бакалаврської роботи	Строк виконання етапів бакалаврської роботи	Примітка
1.	Аналіз предметної області.	01.03-24.03	виконано
2.	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником.	25.03	виконано
3.	Аналіз програм-аналогів.	26.03-29.04	виконано
4.	Вибір технологій для реалізації.	01.04-02.04	виконано
5.	Аналіз Java-інструментів автоматизації веб-тестування.	02.04-25.04	виконано
6.	Проектування внутрішнього формату-посередника.	02.04-05.04	виконано
7.	Побудова архітектури застосунку.	06.04-08.04	виконано
8.	Розробка модулів імпорту для оглянутих форматів.	08.04-16.05	виконано
9.	Реалізація автоматизованого тестового проєкту з Playwright for Java.	10.04-11.04	виконано
10.	Розробка користувацького інтерфейсу.	12.04-13.04	виконано
11.	Розробка модулів експорту для оглянутих форматів.	08.04-16.05	виконано
12.	Розширення функціоналу застосунку переглядачем мета-інформації.	14.05-16.05	виконано
13.	Рефакторинг коду.	16.05-22.05	виконано
14.	Системне тестування веб-застосунку Sauce Demo	23.05-28.05	виконано
15.	Оформлення звіту.	29.05-06.05	виконано
16.	Оформлення презентації і отримання рецензії.	07.05-01.06	в процессі

Студент _____ _____Самонов Д.Ю.

Керівник роботи _____ _____ Коломоєць Г.П.

Нормоконтроль пройдено

Нормоконтролер _____ _____ Скрипник І.А

АНОТАЦІЯ

Сторінок — 90.

Рисунків — 14.

Джерел — 26.

Самонов Д. Ю. Організація тестування Веб-застосунків засобами фреймворку Playwright: кваліфікаційна робота бакалавра спеціальності 121 Інженерія програмного забезпечення / наук. керівник Г. П. Коломоєць. Запоріжжя : ЗНУ, 2026, 90 с.

У роботі розглянуто процес системного тестування веб-застосунків та можливості його автоматизації засобами Playwright for Java. Проаналізовано підходи до організації end-to-end перевірок, синхронізації дій, роботи з локаторами, ізоляції браузерних контекстів і діагностичних артефактів.

Метою роботи є дослідження технологічних можливостей Playwright for Java та розроблення автоматизованого набору системних тестів для веб-застосунку Sauce Demo.

Практична частина включає реалізацію тестового проєкту на Java 25 з використанням Maven, JUnit Jupiter 6.1.0 і Playwright for Java 1.60.0, перевірку ключових користувацьких сценаріїв та аналіз результатів виконання тестів.

Ключові слова: *системне тестування, автоматизація тестування, Playwright for Java 1.60.0, Java 25, JUnit Jupiter 6.1.0, Maven, Sauce Demo, end-to-end тестування.*

ABSTRACT

Pages — 90.

Figures — 14.

Sources — 26.

Organization of Web Application Testing using the Playwright Framework: qualification work of a bachelor in the specialty 121 “Software Engineering” / scientific supervisor H. P. Kolomoicts. Zaporizhzhia : ZNU, 2026, 90 p.

The thesis examines the process of system testing for web applications and the possibilities of its automation using Playwright for Java. It analyzes end-to-end testing, synchronization, locator strategies, browser-context isolation and diagnostic artifacts.

The aim of the work is to study the technological capabilities of Playwright for Java and to develop an automated set of system tests for the Sauce Demo web application.

The practical part includes implementing a Java 25 test project with Maven, JUnit Jupiter 6.1.0 and Playwright for Java 1.60.0, verifying key user scenarios, and analyzing the results of test execution.

Keywords*: system testing, test automation, Playwright for Java 1.60.0, Java 25, JUnit Jupiter 6.1.0, Maven, Sauce Demo, end-to-end testing.*

ЗМІСТ

ВСТУП 8
РОЗДІЛ 1. ОРГАНІЗАЦІЯ СИСТЕМНОГО ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКІВ ТА ВИБІР ІНСТРУМЕНТАРІЮ АВТОМАТИЗАЦІЇ 11
1.1 Поняття системного тестування веб-застосунків та місце автоматизації 11
1.2 Огляд Java-фреймворків для автоматизованого тестування веб-застосунків 16
1.3 Порівняння фреймворків за критеріями та обґрунтування вибору інструмента 21
РОЗДІЛ 2. ТЕХНОЛОГІЧНІ МОЖЛИВОСТІ PLAYWRIGHT FOR JAVA ТА ПІДГОТОВКА ДО СИСТЕМНОГО ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКУ 28
2.1 Загальна характеристика Playwright for Java та архітектура інструмента 28
2.2 Особливості Java-реалізації Playwright та організація тестового проєкту 31
2.3 Робота з елементами сторінки у Playwright for Java 34
2.4 Очікування завантаження сторінок, очікування запитів, таймінги та retry-механізми 40
2.5 Підтримувані браузери, режими запуску, паралельне виконання та засоби діагностики 45
2.6 Робота з API та network interception 49
2.7 Playwright Inspector та автоматична генерація тестових сценаріїв 51
2.8 Обрання веб-застосунку та визначення функціональних вимог 52
РОЗДІЛ 3. ПРОЄКТУВАННЯ ТА РОЗРОБКА АВТОМАТИЗОВАНИХ ТЕСТ-ВИМОГ І СИСТЕМНИХ END-TO-END СЦЕНАРІЇВ ДЛЯ ВЕБ-ЗАСТОСУНКУ SAUCE DEMO 57
3.1 Проєктування тест-вимог за визначеними функціональними вимогами 57
3.2 Структура автоматизованого тестового проєкту 61
3.3 Реалізація Page Object Model для сторінок Sauce Demo 68
3.4 Розробка позитивних і негативних тестових сценаріїв 72

3.5 Реалізація тестових наборів, параметризації та запуску через Maven 79
3.6 Аналіз результатів виконання системних end-to-end тестів 82
ВИСНОВКИ 88
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 89

ВСТУП

Системне тестування веб-застосунків у цій роботі розглядається як перевірка повних користувацьких маршрутів через браузер. Для Sauce Demo такими маршрутами є вхід у систему, робота з каталогом, кошиком і checkout-формою. Ручне повторення цих дій займає час і може пропускати регресійні помилки, тому в роботі використано Playwright for Java. Він дає змогу запускати браузерні сценарії, ізолювати стан кожного тесту через BrowserContext і зберігати діагностичні артефакти для аналізу результатів.

Актуальність теми пов’язана з тим, що для веб-застосунків важливо швидко перевіряти не окремий елемент, а завершений сценарій користувача. У дипломному проєкті автоматизація потрібна для повторюваної перевірки авторизації, каталогу товарів, кошика, checkout-процесу, навігації та негативних сценаріїв. Playwright for Java обрано через підтримку Chromium, Firefox і WebKit, автоматичні очікування, ізоляцію контекстів та trace-файли.

Мета роботи – дослідити технологічні можливості Playwright for Java та розробити автоматизований набір системних тестів для веб-застосунку Sauce Demo.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

1) розглянути сутність системного тестування веб-застосунків та місце автоматизації в його організації;

2) проаналізувати Java-інструменти для автоматизованого тестування веб-застосунків і обґрунтувати вибір Playwright for Java;

3) дослідити архітектуру, функціональні можливості та особливості Java-реалізації Playwright;

4) охарактеризувати засоби роботи з локаторами, елементами сторінки, очікуваннями, браузерними режимами та діагностичними артефактами;

5) дослідити можливості використання APIRequestContext і network interception для підготовки даних та стабілізації системних сценаріїв;

6) обґрунтувати вибір веб-застосунку Sauce Demo як об’єкта практичної перевірки;

7) визначити функціональні вимоги та ключові користувацькі сценарії, що підлягають автоматизованому тестуванню;

8) реалізувати автоматизований тестовий проєкт на Java із застосуванням Playwright;

9) виконати системні перевірки обраного веб-застосунку та проаналізувати отримані результати.

Об’єкт дослідження – процес системного тестування веб-застосунків.

Предмет дослідження – методи, засоби та практичні підходи до автоматизації системного тестування веб-застосунків із використанням Playwright for Java.

У роботі використано такі методи дослідження: аналіз наукової, навчальної та технічної літератури з тестування програмного забезпечення; порівняльний аналіз інструментів автоматизації браузерних перевірок; узагальнення підходів до організації тестового проєкту в середовищі Java; моделювання користувацьких сценаріїв веб-застосунку; проєктування структури автоматизованого набору тестів; експериментальна перевірка функціональних вимог шляхом виконання браузерних системних сценаріїв; аналіз результатів прогону та діагностичних артефактів. [26]

Структура роботи складається зі вступу, трьох розділів, висновків і списку використаних джерел. У першому розділі розглянуто теоретичні засади системного тестування веб-застосунків, подано огляд інструментів автоматизації та обґрунтовано вибір Playwright for Java. У другому розділі досліджено технологічні можливості обраного інструмента, особливості його використання в Java-проєкті, засоби взаємодії з інтерфейсом, механізми синхронізації,

кросбраузерного запуску та підхід до вибору цільового веб-застосунку. У третьому розділі подано практичну реалізацію автоматизованого тестового проєкту, опис сценаріїв перевірки, результати виконання тестів і їх аналіз.

РОЗДІЛ 1 ОРГАНІЗАЦІЯ СИСТЕМНОГО ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКІВ ТА ВИБІР ІНСТРУМЕНТАРІЮ АВТОМАТИЗАЦІЇ

1.1 Поняття системного тестування веб-застосунків та місце автоматизації

Системне тестування розглядає програмний продукт як цілісну систему і перевіряє виконання вимог у сценаріях, що відтворюють експлуатаційні умови. Для веб-застосунку перевіряється узгоджена робота інтерфейсу в браузері, серверної логіки, бази даних, механізмів керування сесією, файлових операцій, поштових або платіжних інтеграцій, а також зовнішніх API, якщо вони є частиною бізнес-потоку. На цьому рівні виявляються дефекти, які не видно в модульних перевірках, наприклад помилки маршрутизації, некоректні конфігурації середовища, розбіжності між версіями залежностей, проблеми з політиками cookie, розсинхронізація стану клієнта і сервера [1, 3].

Веб-застосунок має властивості, що впливають на організацію системних перевірок. Інтерфейс формується DOM-деревом, стилями та скриптами, а значна частина стану підтримується на боці клієнта. Дані часто надходять асинхронно через HTTP-запити, тому елементи з’являються, змінюють текст або стають доступними для взаємодії із затримкою. У застосунках із подієвою навігацією сторінка може не перезавантажуватися повністю, але змінювати вміст секцій, URL і метадані. Додаткові складнощі створюють політики безпеки браузера, зокрема правила CORS і CSP, параметри cookie, робота зі сховищами, обмеження на доступ до буфера обміну, камери, мікрофону тощо. Через ці фактори системний тест має не лише натиснути кнопку, а й дочекатися фактичної готовності інтерфейсу та перевірити результат у вигляді стану сторінки, повідомлення або зміни даних.

У межах системного рівня поширені два підходи до перевірок. Перший підхід орієнтований на функціональні потоки через інтерфейс користувача, тобто end-to-end тести (повні сценарії від дій користувача в браузері до відповіді серверної частини) в браузері. Другий підхід поєднує UI-перевірку з API-кроками, коли підготовка даних і частина перевірок виконуються через HTTP, а інтерфейс використовується для підтвердження поведінки з точки зору користувача. Така комбінація корисна, коли повна підготовка через UI є тривалою, наприклад створення каталогу товарів або завантаження великого набору даних. При цьому важливо не перетворити системний тест на набір розрізнених API-перевірок, оскільки метою залишається перевірка потоку через браузер [3, 10, 15].

Планування системного тестування починається з формування тестових вимог. Тестова вимога має бути перевірюваною і однозначною, наприклад «користувач із валідними обліковими даними отримує доступ до особистого кабінету» або «форма не приймає порожнє поле і показує повідомлення про помилку». Для кожної вимоги визначають позитивні та негативні сценарії, вхідні дані, очікуваний результат і критерії завершення тестування. Якщо застосунок має багатомовний інтерфейс або різні профілі доступу, вимоги фіксують, для яких умов виконується перевірка. Така конкретизація зменшує непорозуміння під час аналізу збою, оскільки видно, що саме перевірялося [1, 3].

Автоматизація системного тестування забезпечує повторюваність і швидке відтворення регресії. Браузерні тести перевіряють повний шлях користувача, тому вони дорогі за часом виконання і потребують обмеження за кількістю. Набір доцільно будувати на основі ризиків. Важливі потоки включають авторизацію, роботу з формами, основні операції з даними, навігацію між базовими сторінками, перевірку помилкових введень. Менш ризиковані елементи інтерфейсу можуть перевірятися нижчими рівнями або ручними перевірками.

Такий відбір зменшує тривалість прогонів і зосереджує ресурси на сценаріях, де дефект має найбільший вплив [2, 3].

Керування станом і тестовими даними є обов’язковим для відтворюваності. Якщо кожний тест створює дані через інтерфейс, прогін стає повільним і крихким. Практичне рішення полягає у підготовці даних через API або службові сценарії перед UI-частиною, а підтвердження результату здійснюється через інтерфейс. Для повторюваності бажано робити тести ідемпотентними. Це досягається використанням унікальних значень у введенні, ізоляцією середовища, або очищенням створених об’єктів після виконання. Під час паралельного запуску важливо уникати спільних ресурсів, що призводять до конфліктів, зокрема одного й того самого облікового запису, спільних черг обробки або фіксованих записів у базі.

Стабільність системного набору значною мірою визначається синхронізацією тестових дій зі станом сторінки. Типові помилки синхронізації проявляються як «елемент не знайдено», «елемент не готовий до кліку», «очікуваний текст не з’явився». Небажаною практикою є довільні паузи, оскільки вони уповільнюють прогін і не гарантують успіху. Натомість застосовують очікування умов. Для сторінок з асинхронним завантаженням доцільно чекати появи індикатора готовності, стабілізації DOM або завершення мережевих запитів. Для динамічних списків корисно чекати конкретної зміни, наприклад появи рядка з очікуваним значенням. Для модальних вікон та діалогів потрібно чекати не лише видимості, а й готовності до взаємодії, оскільки елемент може бути видимим, але перекритим іншим шаром.

Під час виконання браузерних тестів можливі нестабільні падіння, які не пов’язані зі зміною застосунку. Їх можуть спричиняти навантаження CI-агента, коливання мережі, затримки DNS, обмеження ресурсів або конфлікти тестових даних. У такому разі тест не варто просто запускати повторно без аналізу: потрібно визначити причину, перевірити сценарій і за потреби змінити оточення

або дані. Для контролю якості прогону корисно відстежувати кількість падінь, тривалість виконання та повторюваність збоїв.

Не менш важливою є доказова база при невдачах. Для швидкого аналізу збою потрібні скриншоти, відеозапис, журнали кроків, консольні повідомлення, дані про мережеві виклики, а також можливість відтворити стан сторінки. Якщо артефакти прив’язані до конкретного тесту і зберігаються разом зі звітом прогону, час пошуку причини скорочується. Це важливо для помилок, що проявляються лише в CI або лише в певному браузері, коли ручне відтворення займає багато часу [10, 17].

Системний набір тестів має узгоджені правила запуску. На практиці використовують розподіл на швидкі перевірки, що виконуються на кожен коміт, і розширену регресію, яка запускається за розкладом або перед релізом. Для цього тести маркують тегами, групують у набори і керують конфігурацією через профілі. У CI важливо мати параметризацію базового URL, облікових даних і режимів браузера, щоб один і той самий код запускався на різних середовищах. Додатково потрібні механізми повторного підняття середовища після збоїв, наприклад перезапуск контейнерів, очищення кешу, відновлення тестової бази. Якщо середовище нестабільне, зростає частка помилкових падінь і зменшується довіра до результатів прогону.

Документування системного тестування включає опис середовища, набір припущень, правила підготовки даних, а також відповідність тестів вимогам. Зв’язок тестів з вимогами корисний для перевірки покриття і для аргументації, чому обрано саме ці сценарії. У звіті тестування фіксують, які вимоги перевірялися, які результати отримано, які дефекти знайдено. Для дефектів бажано додавати артефакти - скриншот, відео, лог браузерної консолі, дані мережевих викликів або трасування - щоб скоротити час відтворення для розробника. Під час повторних прогонів важливо відстежувати, чи зменшилася

кількість падінь, чи змінився час виконання і чи не з’явилися нові нестабільні тести після змін інтерфейсу [2, 17].

Окрему увагу приділяють керуванню конфігурацією запуску. Параметри середовища, зокрема URL, облікові дані, прапорці функціональності та ключі доступу до зовнішніх сервісів, не варто фіксувати безпосередньо у коді тестів. Практичний підхід - використовувати змінні середовища, секрети CI або конфігураційні файли з розділенням за профілями. Це дозволяє запускати однаковий набір локально і на агенті збірки, не змінюючи тестовий код, а також знижує ризик випадкового витоку даних. Для відтворюваності корисно фіксувати версії браузерів та залежностей і вести журнал змін конфігурації прогонів.

З позиції організації тестового коду корисно визначити правила найменування тестів, структуру пакетів і підхід до повторного використання компонентів. Практично це означає поділ на модулі за функціональністю, виділення спільних фікстур, утиліт для роботи з даними та сторінкових об’єктів. У проєктах, де є авторизація, варто стандартизувати створення сесії, наприклад через збережений стан або швидкий вхід через API, щоб інші тести не дублювали один і той самий шлях. Також доцільно обмежувати кількість перевірок у межах одного end-to-end сценарію, щоб при падінні було очевидно, на якому кроці виникла проблема, а виправлення не зачіпало непов’язані тести.

Для довіри до автоматизованих прогонів важливо відокремлювати стабільні тести від нестабільних. Якщо сценарій падає випадково, його причину потрібно зафіксувати: проблема локатора, очікування, даних або середовища. У CI для цього використовують звіти прогонів, час виконання та історію падінь. Паралельний запуск додатково потребує контролю спільного стану, щоб тести не впливали один на одного через сесію або дані [18, 20].

Окремим правилом є підхід до локаторів. Локатор має бути стабільним при зміні верстки і водночас читабельним для підтримки. Найкраще підходять спеціальні атрибути для тестів, наприклад data-testid або семантичні атрибути,

пов’язані з бізнес-значенням елемента. Селектори, що залежать від глибокої структури DOM або від випадкових класів, швидко ламаються при редизайні. Якщо немає тестових атрибутів, бажано використовувати видимі підписи, заголовки або унікальні значення, що існують для користувача. Єдиний стиль локаторів у проєкті зменшує розбіжності між тестами і спрощує рев’ю.

У Java-проєктах автоматизовані системні перевірки зазвичай будують навколо тестового фреймворку, системи складання та окремого шару Page Object. JUnit Jupiter відповідає за структуру тестів, теги й параметризацію, Maven, Gradle - за залежності та запуск, а Page Object виносить локатори й дії зі сторінками в окремі класи. [7, 18, 19].

1.2 Огляд Java-фреймворків для автоматизованого тестування веб-застосунків

Для автоматизації браузерних системних перевірок у Java найпоширеніші три фреймворки, що відрізняються архітектурою та рівнем абстракції. Selenium WebDriver керує браузером через стандарт WebDriver і забезпечує базові примітиви взаємодії [4]. Selenide працює поверх WebDriver і надає компактніший стиль тестів з автоматичними очікуваннями [8]. Playwright for Java керує браузером через власний драйверний механізм і додає розширені засоби синхронізації, ізоляції та збирання артефактів [10].

Selenium WebDriver у типовій конфігурації працює так. Тест створює екземпляр WebDriver, встановлює налаштування браузера через опції та capabilities (набір параметрів сесії браузера та його можливостей), відкриває сторінки, знаходить елементи й виконує дії. У реалізації системних тестів важливе правильне керування життєвим циклом драйвера - ініціалізація перед тестом або перед набором, закриття після виконання, очищення стану. Для кросбраузерних прогонів застосовують параметризацію запуску, де один і той

самий набір виконується з різними опціями браузера. Для масштабування використовують Selenium Grid або хмарні сервіси, що дозволяють запускати тести віддалено на різних агентах [4].

Пошук елементів у Selenium здійснюється через стратегії локаторів. Якщо в елемента є стабільний і унікальний атрибут id, його доцільно використовувати першим. За відсутності такого id застосовують CSS-селектори, а для складних випадків - XPath. Локатори name, className, linkText і partialLinkText корисні для окремих типів елементів, але залежать від структури форми, тексту посилання або класів верстки, тому не завжди є універсальними [5]. Для взаємодії з інтерфейсом Selenium підтримує роботу з фреймами та вкладками, алертами, файлами завантаження, прокруткою, виконанням JavaScript. Для складніших дій використовується Actions API, який моделює дії миші та клавіатури, зокрема наведення, drag and drop (перетягування мишею), комбінації клавіш [4].

Найскладнішою частиною Selenium у системних тестах є синхронізація. Неявне очікування впливає лише на пошук елемента, але не гарантує готовності до взаємодії. Явні очікування реалізуються через WebDriverWait або FluentWait і дозволяють чекати умов, наприклад видимості, клікабельності, зміни тексту, зникнення елемента. Для стабільності важливо підбирати умови, що відображають реальний стан, який потрібний тесту. Очікування присутності не гарантує, що елемент не перекритий модальним вікном, а очікування видимості не гарантує завершення анімації. Через це в Selenium-проєктах часто створюють власні утиліти, які стандартизують очікування і повторні спроби. Такий підхід покращує стабільність, але збільшує обсяг інфраструктурного коду [6].

Ще один практичний момент - керування сумісністю браузера і драйвера. При оновленнях браузера необхідно оновлювати драйвер управління браузером, що використовується тестами, а в CI це проявляється як раптові падіння без зміни тестового коду. Частково проблему знижують механізми автоматичного керування драйверами [9], але проєкт все одно повинен мати керування версіями

і відтворюваність середовища. Для стабільного headless режиму (запуск браузера без відображення вікна) інколи потрібно налаштовувати параметри графічної підсистеми, шрифти, системні бібліотеки. Для завантаження файлів через браузер застосовують налаштування профілю та директорій, але поведінка може відрізнятися між браузерами. У тестах із завантаженням файлів часто необхідно перевіряти наявність файла у файловій системі і чистити директорії перед запуском, щоб уникати конфліктів [4].

Selenium має розвинені можливості віддаленого виконання. Це зручно, коли потрібно тестувати у різних ОС або версіях браузера. Разом із тим віддалені прогони ускладнюють діагностику, тому проєкт зазвичай додає логування команд, знімки DOM, скриншоти при невдачі та інтеграцію зі звітністю. Для аналізу мережевих проблем можуть застосовувати проксі або логування в браузері, але це додаткові компоненти [4]. У підсумку Selenium є гнучким інструментом, але стабільність і зручність сильно залежать від того, як проєкт організує власну інфраструктуру тестів.

Selenide спрощує написання UI-сценаріїв поверх Selenium. Він пропонує об’єкти елементів із методами дій і перевірок, а також автоматичні очікування. Тест не пише WebDriverWait для кожного кроку, а задає умову - наприклад елемент має бути видимим або містити текст - і фреймворк виконує повторні перевірки до таймауту. Це зменшує кількість помилок синхронізації і робить сценарії коротшими. Для колекцій елементів доступні перевірки розміру, наявності значення, видимості, що зручно для таблиць, списків і результатів пошуку. У конфігурації Selenide можна задавати таймаути, режим запуску, параметри вікна, каталоги для артефактів і поведінку при збереженні скриншотів.

Для взаємодії з елементами Selenide автоматично виконує прокрутку до елемента, намагається працювати зі станами видимості та активності, а у повідомленнях про помилки показує, яку умову не виконано. Це допомагає швидко зрозуміти причину збою без відкриття браузера. Також корисною є

підтримка різних підходів до сторінок - використання шаблону проєктування тестів Page Object або статичних методів, коли сторінка описується набором елементів і дій. Конфігураційні параметри легко підключаються через системні властивості, тому один і той самий проєкт можна запускати локально та в CI з різними налаштуваннями без зміни коду.

Selenide спрощує роботу зі звітами. При невдачі тесту за типовими налаштуваннями можна отримати скриншот і HTML сторінки, що корисно для аналізу у CI. Для детальнішого відображення кроків часто застосовують інтеграцію з Allure. При цьому фундаментальні властивості WebDriver зберігаються. Команди все одно проходять через драйвер, а розширені сценарії на кшталт перехоплення мережі або підміни відповідей потребують зовнішніх рішень. Тому Selenide є зручним інструментом для швидкої розробки сценаріїв у межах WebDriver, але не змінює саму модель керування браузером [8].

Playwright for Java пропонує альтернативну архітектуру. Інструмент керує браузером через власний драйвер, тому отримує доступ до ширшого набору подій сторінки і механізмів мережі. Підтримуються три рушії - Chromium, Firefox і WebKit - з однаковим API. Це спрощує кросбраузерні прогони, оскільки відмінності між рушіями враховуються на рівні реалізації, а тестовий код залишається одним. Фреймворк працює в headless та headed режимах, що зручно для CI та налагодження [10].

У тестах використовується поняття контексту браузера. Контекст ізольований - має власні cookies, кеш і локальне сховище - тому тести можна запускати паралельно без спільного профілю. Це знижує ризик конфліктів стану, коли один тест впливає на інший через залишені дані. Контекст також легко налаштовується під сценарій, наприклад через viewport, user agent, геолокацію, часовий пояс, мову інтерфейсу, дозволи на роботу з окремими можливостями браузера. Такі налаштування корисні, коли потрібно перевірити поведінку в

різних умовах користувача або відтворити проблему, яка проявляється лише у певних параметрах.

Модель локаторів орієнтована на підтримуваність. Локатор описує спосіб знаходження елемента і повторно виконує пошук перед дією, тому перерендеринг компонентів рідше спричиняє помилку. Доступні локатори за CSS, XPath, текстом, атрибутами, data-testid, а також за ознаками доступності через ARIA-атрибути. Для взаємодії доступні кліки, заповнення полів, вибір у списках, робота з діалогами, завантаження файлів і відстеження завантаження файлів із браузера. Дії за замовчуванням виконуються з автоочікуваннями, тобто інструмент сам чекає, доки елемент стане готовим, а навігація завершиться [11, 12].

Робота з мережею дозволяє перехоплювати запити, змінювати маршрутизацію, підміняти відповіді, блокувати ресурси, перевіряти параметри викликів. Це корисно для системних сценаріїв, де інтерфейс залежить від сторонніх сервісів або де потрібно перевірити обробку помилок сервера. Окремо доступний HTTP-клієнт, який можна використати для підготовки даних, отримання токенів або очищення стану перед тестом. Таким чином одна інструментальна база покриває як UI, так і допоміжні API-кроки.

Для діагностики Playwright може зберігати скриншоти, відео та trace-файли. Trace Viewer дає змогу подивитися кроки сценарію, стан сторінки, мережеві події та знімки DOM. У Java-проєкті ці можливості підключаються через методи життєвого циклу й розширення JUnit Jupiter, щоб артефакти створювалися автоматично під час прогону або при падінні тесту [17, 18].

Паралельність у Java-проєкті можна організувати на рівні JUnit Jupiter. Для цього кожен тест повинен отримувати власний BrowserContext, інакше стан авторизації, cookies або кошика може перейти в інший сценарій. У реалізованому проєкті саме такий підхід застосовано: браузер спільний для класу, але контекст і сторінка створюються окремо перед кожним тестом.

Практична цінність інструмента для системних перевірок проявляється у сценаріях, де тест має працювати з непередбачуваними затримками або помилками мережі. Якщо інтерфейс залежить від стороннього сервісу, корисно мати можливість стабілізувати прогін шляхом блокування частини ресурсів або підміни відповіді на запит. Це дозволяє відтворювати ситуації з помилковими кодами відповіді, затримками або порожніми даними без ручного втручання. Такі перевірки допомагають оцінити, як застосунок обробляє проблемні умови, чи показує повідомлення користувачу і чи не переходить у неконсистентний стан. Додатково корисною є можливість зберігати стан після авторизації і використовувати його в інших тестах, щоб зменшити час прогону та прибрати зайве навантаження на сервіс автентифікації. У поєднанні з відео, скриншотами та трасуванням це формує повний набір доказів, який полегшує аналіз невдач у CI і робить регресію більш керованою.

Отже, Selenium WebDriver дає базовий стандарт керування браузером, Selenide спрощує сценарії в межах WebDriver і автоматизує частину очікувань, а Playwright for Java пропонує розширені можливості ізоляції, синхронізації, мережевих сценаріїв і діагностики, що є важливими для системних end-to-end перевірок.

1.3 Порівняння фреймворків за критеріями та обґрунтування вибору інструмента

Порівняння інструментів автоматизації доцільно виконувати за критеріями, що впливають на стабільність системного набору, час виконання і трудомісткість підтримки. У межах роботи використано п’ять критеріїв - стабільність, швидкодія, зручність написання тестів, підтримка браузерів, складність налаштування. Оцінювання проводиться з погляду типового Java-

проєкту з інтеграцією тестів у CI, де важливі відтворюваність і швидкий аналіз невдач. Узагальнення наведено в таблиці 1.1.

Стабільність у браузерних тестах означає відтворюваність результату за незмінної поведінки застосунку. Для Selenium стабільність найбільше залежить від дисципліни очікувань і якості локаторів. Якщо очікування задані неповно, тест може взаємодіяти з елементом до завершення рендерингу або анімації. Це дає випадкові падіння, які важко відтворити локально. Selenide зменшує частоту таких проблем за рахунок очікувань на рівні дій і умов, але він залежить від WebDriver і типових обмежень драйверного підходу. У Playwright стабільність підтримується поєднанням автоочікувань, локаторів з повторним пошуком і ізольованих контекстів. Додатково доступні інструменти мережі, які дозволяють блокувати нестабільні сторонні ресурси, що інколи є причиною випадкових затримок.

Швидкодія визначається не лише часом одного сценарію, а й ефективністю паралельного запуску. Selenium і Selenide у багатьох конфігураціях використовують окремий браузер на потік або віддалену інфраструктуру, що підвищує накладні витрати. При великій кількості тестів прогін вимагає більше агентів CI або складнішого Grid. У Playwright контексти легкі і створюються швидко, тому кілька тестів можуть використовувати один процес браузера, залишаючись ізольованими за станом. Це скорочує витрати ресурсів і полегшує масштабування прогону. Додатковий інструмент оптимізації - керування ресурсами сторінки, наприклад блокування завантаження медіа або сторонніх скриптів у регресійному наборі.

Зручність написання тестів охоплює читабельність сценаріїв, кількість службового коду, підтримуваність локаторів, якість повідомлень про помилки та наявність артефактів. Selenium дає низькорівневі примітиви, тому в проєкті зазвичай з’являються утиліти для очікувань, повторів, логування і збору скриншотів. Ці елементи потрібно підтримувати разом із тестами. Selenide дає

компактні сценарії, оскільки очікування і перевірки інтегровані в API, а типові артефакти доступні без значних доробок. У Playwright поєднано компактний API дій з автоочікуваннями і вбудованими матеріалами для розслідування - відео, скриншоти, трасування. Для системного тестування це суттєво, оскільки помилки часто пов’язані із середовищем або кросбраузерністю, і без артефактів їх складно пояснити.

Підтримка браузерів впливає на можливість запускати один набір у різних рушіях рендерингу. Selenium підтримує багато браузерів, але практична якість залежить від драйверів і сумісності версій. У CI це може призводити до проблем після оновлень браузера або драйвера. Selenide має ті самі можливості, оскільки використовує WebDriver. У Playwright підтримуються Chromium, Firefox і WebKit, а перемикання рушія відбувається через конфігурацію запуску при незмінному коді тестів. Для системних перевірок це зручно, оскільки дозволяє швидко перевірити, чи пов’язаний дефект з конкретним рушієм.

Окремо для кросбраузерної перевірки важливо враховувати можливість одночасного запуску одного набору тестів у різних браузерах. У Selenium WebDriver це зазвичай реалізують через Selenium Grid або сумісні хмарні платформи: тести створюють віддалені сесії RemoteWebDriver з різними capabilities, а Grid розподіляє їх між вузлами з потрібними браузерами та ОС. Selenide використовує той самий підхід через інфраструктуру WebDriver, тому підключення до Grid і паралельний запуск налаштовують на рівні конфігурації remote драйвера, CI та тестового раннера. У Playwright for Java одночасний запуск у Chromium, Firefox і WebKit зазвичай організовують без окремого Grid - через паралельні задачі CI або параметризовані прогони, а ізольовані контексти браузера зменшують конфлікти стану між тестами.

Складність налаштування охоплює інсталяцію залежностей, керування драйверами або браузерами, стабільність запуску в контейнерах, відтворюваність оточення між локальним запуском і CI. У Selenium потрібен керування версіями

драйверів і налаштування браузерних опцій, а також додаткові параметри для headless запуску в Linux-оточенні. Selenide зменшує обсяг налаштувань за рахунок типових конфігурацій, але залежність від драйверів зберігається. У Playwright браузери встановлюються керовано і використовуються з передбачуваними версіями, що спрощує відтворюваність. Це зменшує кількість ситуацій, коли набір працює локально, але падає на агенті збірки через інше оточення.

Критерії взаємопов’язані. Надмірні паузи підвищують тривалість прогону і маскують помилки синхронізації, а кросбраузерні запуски збільшують кількість прогонів та навантаження інфраструктури. Якщо інструмент не формує артефактів виконання, розробка тестів може бути швидшою, але розслідування падінь у CI займає більше часу. Для роботи потрібен підхід, який поєднує автоочікування, ізоляцію тестів, запуски в кількох браузерах та збір скриншотів, відео й трасування без значного обсягу власних надбудов.

Таблиця 1.1 – *Порівняння Selenium WebDriver, Selenide та Playwright for Java*

Критерій	Selenium WebDriver	Selenide	Playwright for Java 1.60.0
Стабільність	Потребує чітких явних очікувань і дисципліни локаторів, випадкові невдачі можливі при асинхронному UI.	Автоматичні очікування знижують частоту помилок синхронізації, але поведінка залежить від WebDriver.	Автоочікування, локатори з повторним пошуком, ізольовані контексти та інструменти мережі підвищують відтворюваність.
Швидкодія	Накладні витрати протоколу і драйверів, паралельність часто потребує Grid або окремих інстансів браузера.	Подібна швидкість до Selenium, виграш здебільшого у швидшій розробці сценаріїв.	Легкі контексти й ефективний паралельний запуск, можливість оптимізувати завантаження сторінок.
Зручність тестів	Потрібні власні утиліти для очікувань, логування та артефактів, сценарії часто об’ємніші.	Компактний DSL, умови елементів і типові артефакти при падінні, менше службового коду.	Компактний API, автоочікування, вбудовані відео і трасування, швидкий аналіз невдач у CI.
Браузери	Широкий перелік через WebDriver, сумісність залежить від драйверів та версій браузера.	Ті самі браузери, що і в Selenium WebDriver, оскільки основа спільна.	Chromium, Firefox, WebKit з однаковим API та простим перемиканням рушія у конфігурації.
Налаштування	Контроль драйверів і сумісності, інколи додаткові налаштування в CI або контейнерах.	Простіший запуск за рахунок конфігурацій, але залежності WebDriver і драйвери залишаються.	Кероване встановлення браузерів і передбачуваний запуск, простіше забезпечити однакове оточення.

Для практичного застосування наведеного порівняння важливо окремо уточнити, які саме механізми пошуку елементів і синхронізації підтримує кожен інструмент. Саме від набору локаторів та стратегій очікування безпосередньо залежать стабільність тестів, зручність їх підтримки та поведінка сценаріїв під час паралельних і кросбраузерних запусків. Тому в таблиці 1.2 наведено узагальнення підтримуваних локаторів і підходів до очікування для Selenium WebDriver, Selenide та Playwright for Java.

Таблиця 1.2 - *Підтримувані локатори та стратегії очікування*

Фреймворк	**Підтримувані локатори**	**Стратегії очікування**
Selenium WebDriver	By.id, By.name, By.className, By.tagName, By.linkText, By.partialLinkText, By.cssSelector, By.xpath.	Неявні очікування (implicit wait); явні очікування через WebDriverWait та ExpectedConditions; FluentWait (таймаут, інтервал опитування, ігноровані винятки); користувацькі умови очікування.
Selenide	CSS і XPath через $(), $$(), $x(), $$x(); локатори Selenium By (id, name, className, cssSelector, xpath тощо); текстові селектори через Selectors.byText(), withText(), withTagAndText().	Вбудовані очікування в should/shouldBe/shouldHave для елементів і колекцій; умови Condition та CollectionCondition; за потреби waitUntil/waitWhile або використання очікувань Selenium через WebDriver.
Playwright for Java 1.60.0	page.locator() (CSS, XPath та інші селектори); getByRole(), getByText(), getByLabel(), getByPlaceholder(), getByAltText(), getByTitle(), getByTestId(); фільтрація та композиція локаторів.	Автоочікування (actionability checks) для дій і навігації; locator.waitFor(); page.waitForSelector(), page.waitForURL(), page.waitForLoadState(); очікування мережевих подій через waitForRequest() і waitForResponse().

Порівняння показує, що Selenium WebDriver є універсальною основою для керування браузером і має розвинену екосистему, проте вимагає значних зусиль для побудови стабільних очікувань і підтримки інфраструктури виконання. Selenide робить сценарії коротшими і бере на себе значну частину синхронізації, але не прибирає обмеження драйверного підходу та потребу у зовнішніх рішеннях для мережевих сценаріїв і розширеної діагностики.

Для задачі організації системного тестування в межах роботи доцільно обрати інструмент, який забезпечує керування синхронізацією, ізоляцією і діагностикою без великої кількості власних надбудов. Playwright for Java відповідає цим вимогам завдяки автоочікуванням, ізольованим контекстам, підтримці трьох рушіїв, роботі з мережею та можливості зберігати відео, скриншоти і трасування виконання. Це дозволяє побудувати набір системних

end-to-end сценаріїв, показати методику стабілізації тестів і забезпечити прозорий аналіз причин невдач у CI.

РОЗДІЛ 2 ТЕХНОЛОГІЧНІ МОЖЛИВОСТІ PLAYWRIGHT FOR JAVA ТА ПІДГОТОВКА ДО СИСТЕМНОГО ТЕСТУВАННЯ ВЕБ-ЗАСТОСУНКУ

2.1 Загальна характеристика Playwright for Java та архітектура інструмента

Playwright for Java є фреймворком автоматизації браузерних перевірок, призначеним для виконання функціональних, end-to-end і системних сценаріїв у веб-застосунках. Він забезпечує керування сторінкою на рівні дій користувача та дає змогу запускати однакові тести в кількох браузерних рушіях із єдиним стилем API.

Java-реалізація цієї технології входить до переліку офіційно підтримуваних мов разом із JavaScript/TypeScript, Python і .NET. Такий підхід спрощує використання одного інструментального підґрунтя в різних командах і дає можливість будувати тести в межах звичного для Java-проєкту середовища. Автоматизація не зводиться лише до відтворення окремих кліків або введення тексту. Бібліотека дозволяє відкривати сторінки, виконувати навігацію, працювати з елементами DOM, контролювати мережеву активність, обробляти кілька сторінок, фіксувати стан браузера та зберігати діагностичні артефакти. За рахунок цього сценарій може відтворювати завершений користувацький потік від входу в систему до перевірки результату в інтерфейсі [10].

Підтримка Chromium, Firefox і WebKit формує основу кросбраузерного виконання та забезпечує перевірку роботи веб-застосунку в браузерах із різними рушіями та особливостями реалізації веб-стандартів. Один і той самий набір перевірок запускається в різних рушіях без зміни загальної логіки тесту, що спрощує організацію тестового набору та дає змогу зберігати єдиний підхід до

побудови сценаріїв. Це дає можливість виявляти розбіжності рендерингу, поведінки елементів, обробки подій і навігації в браузерах із різною реалізацією веб-стандартів.

Технологічні характеристики інструмента, що визначають його придатність для системного тестування, узагальнено в таблиці 2.2.

Таблиця 2.2 – *Основні характеристики Playwright for Java*

Характеристика	**Зміст**	**Застосування у системному тестуванні**
Підтримувані мови	Java, JavaScript/TypeScript, Python, .NET	Використання інструмента в різних технологічних стеках
Підтримувані браузери	Chromium, Firefox, WebKit	Кросбраузерний запуск одного набору сценаріїв
Модель взаємодії	Пряме керування браузером без окремого WebDriver-сервера	Менші накладні витрати під час виконання дій
Ізоляція сесій	Незалежні BrowserContext із власними cookies, cache і storage	Розділення тестових даних і паралельні прогони
Робота зі сторінками	Використання об’єкта Page для вкладки або сторінки	Опис дій користувача на рівні браузерного інтерфейсу
Синхронізація	Вбудовані механізми очікування та перевірки готовності елемента	Зменшення кількості нестабільних перевірок
Діагностичні артефакти	Скриншоти, відео, trace	Аналіз причин збоїв локально та в CI
Робота з мережею	Контроль HTTP-запитів і відповідей, перехоплення ресурсів	Підготовка даних і стабілізація окремих сценаріїв

Архітектура Playwright складається з кількох рівнів: Java-код тесту викликає API Playwright, а той керує браузерним застосунком Chromium, Firefox або WebKit. У браузері створюються контексти та сторінки, через які тест взаємодіє з веб-застосунком. На відміну від класичного WebDriver-підходу, у цій моделі не потрібно окремо налаштовувати WebDriver-сервер, що спрощує

локальний запуск і CI-конфігурацію. Узагальнену структуру взаємодії між цими компонентами подано на рисунку 2.1.

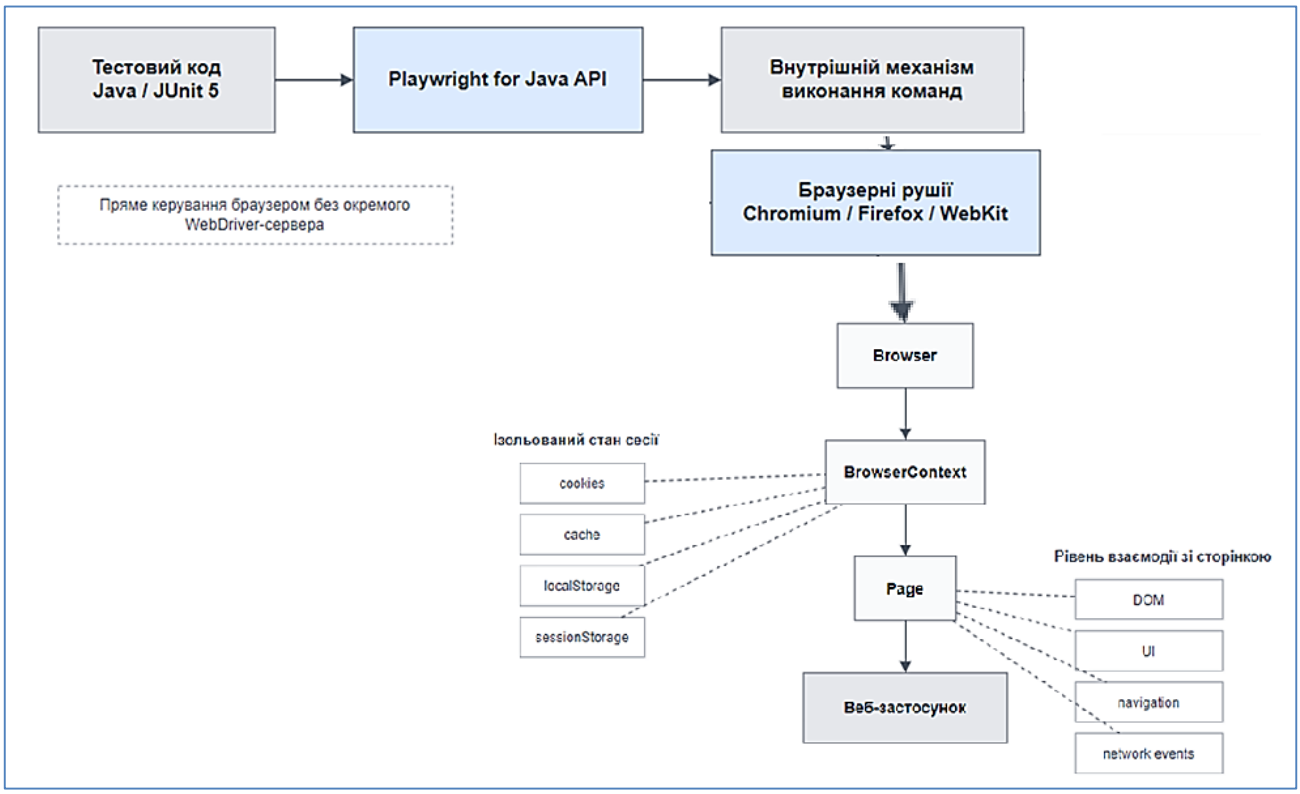


Рисунок *2.1 – Узагальнена архітектура Playwright for Java*

У практичній роботі найважливішими об’єктами є Browser, BrowserContext і Page. Browser представляє запущений браузер, BrowserContext ізолює cookies, localStorage та інші дані сесії, а Page відповідає за конкретну вкладку. Завдяки окремому контексту кожен тест у наборі починається з чистого стану. Це особливо важливо для Sauce Demo, де авторизація й стан кошика можуть впливати на наступні сценарії.

Об’єкт Page відповідає окремій вкладці або сторінці в межах контексту. Через нього виконуються переходи за URL, побудова локаторів, взаємодія з DOM, перевірка тексту, стану елементів і результатів навігації. Такий розподіл відповідальності створює чітку ієрархію: браузер керує процесом, контекст

ізолює сесію, сторінка працює з інтерфейсом. Це спрощує структуру тестового коду та дає змогу однаково організовувати ініціалізацію, виконання і завершення сценаріїв.

Сукупність наведених властивостей формує середовище, придатне для відтворюваного та кросбраузерного запуску автоматизованих перевірок із контрольованим станом сесії, прямою взаємодією з браузером і можливістю детального аналізу виконання. Така архітектура добре узгоджується з вимогами системного тестування веб-застосунків.

2.2 Особливості Java-реалізації Playwright та організація тестового проєкту

Java-реалізація цього фреймворку інтегрується у звичне для тестового проєкту середовище через систему керування залежностями та модуль тестування. Найпоширенішим підходом є підключення бібліотеки через Maven або Gradle з подальшим запуском сценаріїв за допомогою JUnit Jupiter. Така комбінація дозволяє зберігати конфігурацію, залежності й правила виконання в єдиній структурі проєкту.

На рівні складання проєкту Maven або Gradle виконує кілька функцій одночасно. Через файл pom.xml або build.gradle фіксуються версії бібліотек, підключаються тестові залежності, налаштовуються профілі запуску та формується відтворюване оточення. Для Java-проєкту це важливо через те, що конфігурація тестового набору не розподіляється по окремих скриптах, а зберігається в одному зрозумілому місці. Під час перенесення набору на іншу машину або на CI-агент достатньо відтворити залежності й параметри запуску, після чого структура роботи залишається сталою. У такому підході зменшується кількість прихованих налаштувань, які не видно з репозиторію, а отже, легше підтримувати однакову поведінку локального і серверного прогону. Саме тому

зв’язка Java, менеджера збірки та фреймворку тестування розглядається як базова технічна основа для побудови керованого автоматизованого набору.

JUnit Jupiter у цій моделі виконує роль тестового раннера: задає життєвий цикл сценаріїв, групування, параметризацію й паралельний запуск. У браузерних перевірках це дозволяє винести створення й закриття Playwright-об’єктів у @BeforeAll, @BeforeEach, @AfterEach і @AfterAll. У результаті тестові методи залишаються зосередженими на сценаріях, а інфраструктурний код знаходиться в одному базовому класі.

Організація тестового проєкту в Java будується за принципом розділення відповідальності між пакетами. Окремо розміщують тестові класи, що описують сценарії перевірок; окремо — класи Page Object, де інкапсулюються локатори та дії над конкретними сторінками або фрагментами інтерфейсу; окремо — допоміжні класи для конфігурації, читання параметрів середовища, підготовки тестових даних, роботи з файлами, логування та збереження артефактів. Якщо всі ці елементи змішані в одному пакеті або в одному класі, підтримка набору швидко ускладнюється: локатори дублюються, сценарії втрачають читабельність, а зміна одного елемента інтерфейсу потребує редагування багатьох файлів. За наявності структурованих пакетів тестовий код стає ближчим до моделі застосунку: сторінка описується окремим класом, користувацька дія — окремим методом, конфігурація запуску — окремим шаром. Така організація не усуває потребу в дисципліні написання тестів, але зменшує зв’язність між частинами проєкту та полегшує повторне використання компонентів. Типову структуру тестового проєкту Playwright for Java наведено на рисунку 2.2.

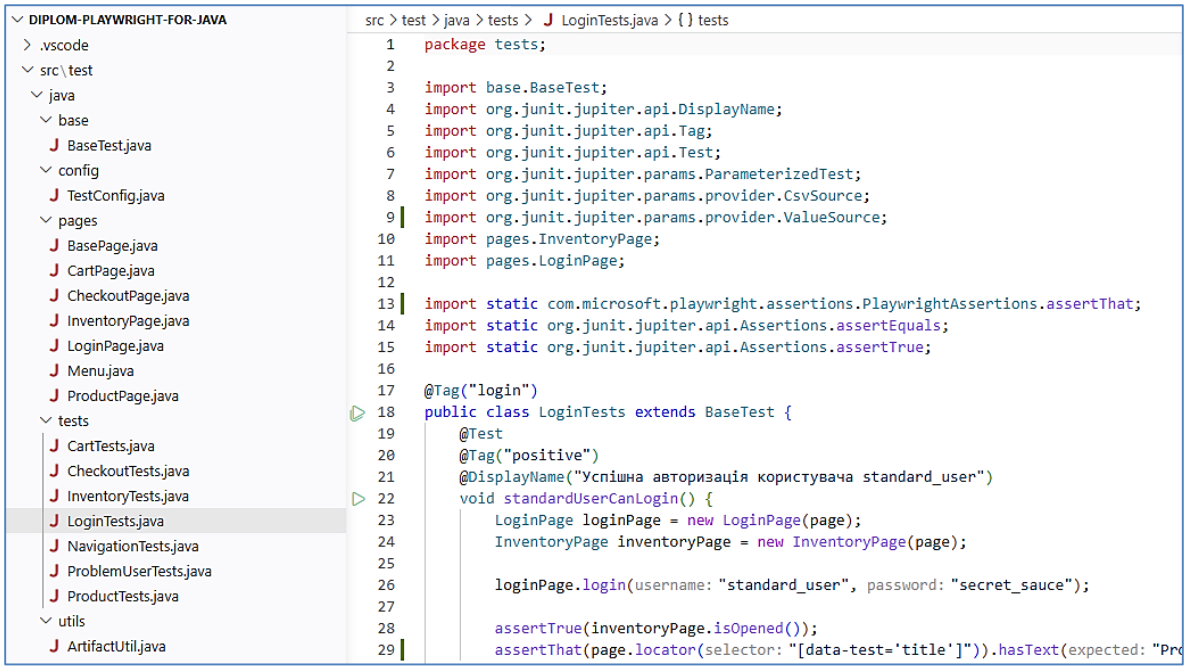


Рисунок 2.2 – *Типова структура тестового проєкту Playwright for Java*

Page Object використовується для того, щоб тестові класи не працювали напряму з усіма локаторами сторінки [7]. Наприклад, LoginPage інкапсулює поля username/password і кнопку login, InventoryPage - каталог товарів, а CheckoutPage - форму оформлення замовлення. Якщо зміниться локатор або логіка взаємодії зі сторінкою, правка вноситься у відповідний Page Object, а не в усі тести.

Допоміжні класи потрібні для спільної конфігурації й артефактів. У цьому проєкті TestConfig зберігає базовий URL, браузер, headless-режим і шляхи до target/screenshots та target/traces. ArtifactUtil створює каталоги, формує безпечні назви файлів і зберігає скриншоти. Такий поділ робить код простішим для перевірки й повторного запуску.

Життєвий цикл Playwright-об’єктів у проєкті побудовано послідовно. У BaseTest один раз для тестового класу створюються Playwright і Browser, перед кожним тестом відкриваються нові BrowserContext і Page, а після тесту

зберігається trace-файл і контекст закривається. Це дає ізоляцію сценаріїв без запуску нового браузера для кожного тесту.

Окрему увагу варто приділити тому, як структура коду впливає на підтримуваність тестового набору. Якщо сценарій безпосередньо створює браузер, оголошує локатори, містить усю логіку підготовки даних і ще й перевіряє результат, такий клас швидко стає перевантаженим. Його важко читати, складно перевикористовувати, а будь-яке розширення збільшує ризик помилки. Коли ж інфраструктурний код, сторінкові класи, тестові дані та самі перевірки розведені по окремих елементах проєкту, кожен фрагмент виконує власну функцію. Це полегшує внесення змін, спрощує рев'ю, зменшує дублювання й робить набір більш передбачуваним у розвитку. Для довготривалого супроводу це має безпосереднє значення, оскільки автоматизований набір майже завжди змінюється разом із веб-застосунком. Чим чіткіше відокремлено структуру сторінок, конфігурацію запуску та логіку сценаріїв, тим легше масштабувати перевірки без втрати керованості.

Сукупність описаних рішень формує зрозумілу модель Java-проєкту, у якій залежності, запуск, життєвий цикл браузерних об'єктів і структура пакетів узгоджені між собою. За такої організації тестовий набір зберігає читабельність, простіше розширюється і краще переноситься між середовищами виконання.

2.3 Робота з елементами сторінки у Playwright for Java

Взаємодія з елементами сторінки становить основу браузерної автоматизації, оскільки саме на цьому рівні тест відтворює дії користувача в інтерфейсі. Для цього фреймворк надає набір локаторів і методів, через які сценарій знаходить елементи, виконує над ними дії та перевіряє зміну стану сторінки.

Пошук елементів може будуватися як через універсальний виклик page.locator(), так і через вбудовані семантичні методи getByRole(), getByText(), getByLabel(), getByPlaceholder(), getByTitle() та getByTestId(). Універсальний локатор зручний тоді, коли елемент описується CSS, XPath або складнішою комбінацією селекторів. Водночас семантичні методи орієнтуються на текст, атрибут title, тестовий атрибут або доступну назву елемента. Така модель пошуку краще узгоджується з реальним інтерфейсом, де користувач сприймає кнопку через напис, поле - через підпис або placeholder, а не через глибину вкладеності DOM-вузлів. У матеріалах з автоматизації цим інструментом саме такі локатори подаються як рекомендований набір для роботи з веб-сторінками, тоді як CSS і XPath залишаються універсальним резервом для складніших випадків [11].

Семантичні локатори краще підходять для підтримки тестового набору, ніж громіздкі CSS- або XPath-конструкції. Якщо локатор побудований на ланцюгу з кількох вкладених контейнерів, він легко ламається після зміни верстки, навіть коли поведінка сторінки для користувача не змінилася. Коли ж пошук виконується за текстом кнопки, підписом поля або data-testid, тест прив’язується до стабільнішої ознаки. Це не означає повної відмови від CSS і XPath, однак робить їх допоміжним інструментом, а не основним способом ідентифікації елементів. Саме тому при організації системних сценаріїв доцільно спочатку використовувати семантичні способи пошуку, а складні селектори залишати для нетипових або технічно закритих фрагментів інтерфейсу [11].

Метод Locator getByRole(AriaRole role, GetByRoleOptions options) у Playwright використовується для пошуку елементів вебсторінки за їхньою семантичною роллю, тобто за тим, як елемент сприймається користувачем і допоміжними технологіями доступності. Такий підхід є більш стійким для користувацьких сценаріїв, ніж довгі CSS-селектори або XPath-ланцюги, оскільки орієнтується не лише на технічну структуру DOM, а й на призначення елемента в інтерфейсі: кнопка, посилання, заголовок, чекбокс, текстове поле тощо [11].

Параметр AriaRole role задає ARIA-роль елемента, який потрібно знайти. Наприклад, AriaRole.BUTTON використовується для кнопок, AriaRole.CHECKBOX для прапорців, AriaRole.HEADING для заголовків, AriaRole.LINK для посилань, AriaRole.TEXTBOX для текстових полів. Роль може бути задана явно через атрибут role, але багато HTML-елементів мають неявну семантичну роль, наприклад <button> автоматично сприймається як кнопка.

Параметр GetByRoleOptions options є необов'язковим і дозволяє уточнити пошук. Найчастіше використовується setName(), який задає доступне ім'я елемента, тобто текст або ARIA-мітку, за якою користувач розпізнає елемент. Наприклад, кнопка <button>Submit</button> може бути знайдена через роль BUTTON і назву "Submit".

Основні групи дій над елементами сторінки узагальнено в таблиці 2.2.

Таблиця 2.2 – *Основні групи операцій з елементами сторінки у Playwright for Java*

Група операцій	**Приклади засобів**	**Призначення**
Пошук елементів	page.locator(), getByRole(), getByText(), getByLabel(), getByPlaceholder(), getByTitle(), getByTestId()	Ідентифікація елементів інтерфейсу
Базові дії	click(), dblclick(), hover(), fill(), clear(), press()	Імітація дій користувача
Робота з формами	введення значень, очищення полів, вибір у списках, встановлення прапорців	Заповнення та перевірка форм
Діалоги	обробка alert, confirm, prompt	Контроль браузерних повідомлень
Вікна та вкладки	відкриття popup, робота з новою сторінкою, перемикання між сторінками	Перевірка багатовіконних сценаріїв
Додаткові випадки	модальні вікна, таблиці, динамічні списки, Shadow DOM	Робота з ускладненими компонентами

На рисунку 2.3 зображений фрагмент сторінки та код з відповідними локаторами. Зазвичай, локатори й дії розносять за групами: пошук, введення даних, вибір, підтвердження та робота з додатковими вікнами.

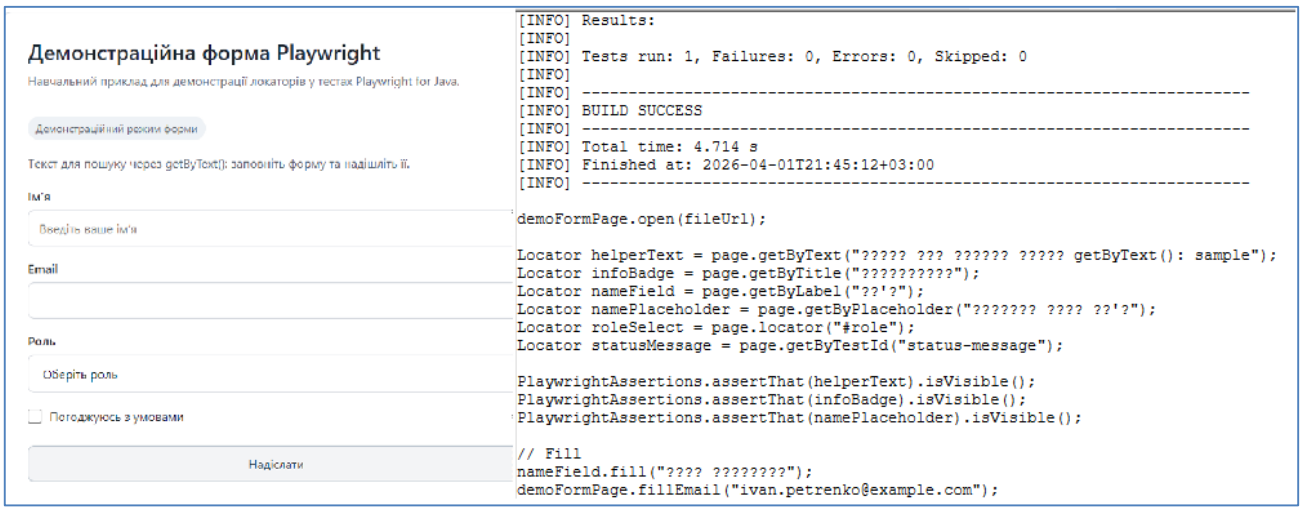


Рисунок 2.3 – *Приклад локаторів для елементів Веб-сторінки*

Найуживаніші дії над елементами пов’язані з формами. Для текстових полів застосовують введення значення через fill(), а за потреби імітації послідовного набору можуть використовуватися натискання клавіш. Очищення поля виконується або через окремий виклик очищення, або повторним заповненням порожнім значенням, залежно від структури сценарію та версії API. Для кнопок, посилань і клікабельних елементів використовують click(), а для наведення — hover(). Усі ці дії мають сенс лише тоді, коли елемент знайдено стабільним локатором і його стан відповідає очікуванню сценарію. Якщо тест виконує авторизацію, зміну профілю, створення запису чи пошук даних, саме зв’язка «локатор + дія» формує мінімальну технічну одиницю сценарію. Через це локатори не можна розглядати окремо від дій: помилка в пошуку елемента одразу переходить у помилку взаємодії, а надто складний спосіб пошуку підвищує вартість супроводу всього тестового набору.

Операції з елементами вибору мають власну специфіку. Для стандартних випадаючих списків використовують методи вибору значення, видимого тексту або індексу, а для мультивибору — передачу набору значень. Прапорці зазвичай обробляються через check() і uncheck(), що дозволяє явно фіксувати потрібний стан елемента, а не покладатися лише на клік. Такий підхід зменшує неоднозначність у сценаріях, де важливо перевірити, чи прапорець саме встановлений, а не просто натиснутий. Подібна логіка корисна і для перемикачів, радіокнопок та елементів керування фільтрами, оскільки в інтерфейсі вони часто змінюють не лише зовнішній вигляд, а й набір доступних даних на сторінці. У системних перевірках це дозволяє переходити від простого натискання до контролю фактичного стану інтерфейсу після дії.

Окрему групу становлять браузерні діалоги типу alert, confirm і prompt. Їх обробка в автоматизованому сценарії не може зводитися до звичайного пошуку DOM-елемента, оскільки ці повідомлення належать до браузерного середовища. Для цього використовуються події та обробники, які дозволяють прийняти діалог, відхилити його або передати текст у випадку prompt. Крім звичайних діалогів, у веб-застосунках часто використовуються модальні вікна, реалізовані вже засобами самого інтерфейсу. На відміну від alert або confirm, вони є частиною DOM і тому взаємодія з ними здійснюється через звичайні локатори, перевірку видимості, кнопки закриття, поля введення та підтвердження дії. Розмежування цих двох випадків є необхідним: браузерний діалог потребує роботи через подію, а модальне вікно — через пошук елементів усередині інтерфейсу. Якщо цього не враховувати, сценарій може бути технічно коректним, але фактично не взаємодіяти з потрібним об’єктом. У виданнях, присвячених цій платформі автоматизації, JavaScript alerts, confirm, prompt, tabs і popup-вікна розглядаються як окрема категорія ускладнених елементів, що потребують окремих прийомів роботи.

Робота з вкладками та новими вікнами також не зводиться до звичайного кліку по посиланню. Коли дія користувача відкриває popup або нову вкладку, сценарій має не лише ініціювати подію, а й отримати посилання на новий об’єкт сторінки, дочекатися його завантаження та виконувати подальші перевірки вже в новому контексті сторінки. Зазвичай, це реалізується через очікування події popup і подальшу роботу з новою сторінкою після завершення завантаження. Такий механізм потрібний для перевірок авторизації через зовнішні сервіси, відкриття довідкових сторінок, переходу до документів, друкованих форм або додаткових інформаційних вікон. Коли сценарій працює лише з початковою сторінкою й ігнорує появу нового вікна, він втрачає частину бізнес-потоку та не перевіряє завершений ланцюг взаємодії.

У сучасному інтерфейсі трапляються і складніші випадки: компоненти всередині Shadow DOM, динамічні таблиці, списки з фільтрацією, елементи, що змінюють стан після запиту до сервера. У таких ситуаціях універсальність page.locator() і композиція локаторів дозволяють поєднувати загальні та семантичні підходи. Наприклад, поле з чітким підписом доцільно шукати через getByLabel(), а вкладений елемент усередині нестандартного контейнера - через locator() з додатковою фільтрацією. Саме комбінування різних способів пошуку дає змогу працювати не лише зі стандартними формами, а й з інтерфейсами, де окремі елементи рендеряться динамічно або приховуються всередині компонентної структури. Такий підхід узгоджується з природою веб-застосунків, у яких DOM часто перебудовується без повного перезавантаження сторінки, а дії користувача викликають не ізольовану зміну одного поля, а послідовну зміну кількох областей інтерфейсу [11].

Сукупність локаторів і дій формує технічну основу взаємодії автоматизованого сценарію з веб-інтерфейсом. За умови коректного вибору способу пошуку, явного керування станами елементів і розмежування між DOM-

компонентами, діалогами та новими вікнами тестовий набір стає зрозумілішим у супроводі та стійкішим до змін верстки.

У Playwright для Java перевірки виконуються через клас PlaywrightAssertions. Він містить перевантажені методи assertThat(), які створюють об’єкти для подальших перевірок стану сторінки, локатора або HTTP-відповіді. На відміну від звичайних асертів JUnit Jupiter, ці асерти підтримують автоматичне очікування: Playwright повторно перевіряє умову до її виконання або до завершення тайм-ауту. Тому для перевірок вебінтерфейсу рекомендується використовувати саме асерти Playwright, оскільки вони зменшують нестабільність тестів, пов’язану з асинхронним завантаженням сторінки.

2.4 Очікування завантаження сторінок, очікування запитів, таймінги та retry-механізми

Стабільність браузерного сценарію залежить не лише від локаторів і структури коду, а й від того, наскільки коректно тест синхронізований зі станом сторінки. Якщо дія виконується раніше, ніж інтерфейс готовий до взаємодії, з’являються випадкові падіння, що не пов’язані зі зміною логіки застосунку.

При використанні Playwright значна частина синхронізації виконується автоматично. Перед кліком, введенням тексту або іншою взаємодією перевіряється, чи елемент присутній, видимий, доступний для дії та не перекритий іншим шаром. Саме ці перевірки зазвичай позначаються як actionability checks. Вони зменшують кількість ситуацій, коли сценарій формально знаходить елемент у DOM, але фактично не може натиснути кнопку або ввести значення в поле. За такого підходу тест орієнтується не на формальну наявність вузла в дереві документа, а на реальний стан готовності елемента до взаємодії. Це добре узгоджується з динамічними інтерфейсами, де після

натискання кнопки можуть запускатися анімації, часткове оновлення контенту або мережеві запити.

Автоматичне очікування не скасовує потреби в явній синхронізації там, де тест очікує конкретну подію. У Playwright для цього використовують waitForLoadState(), waitForURL(), waitForSelector(), waitForRequest(), waitForResponse(). Такі методи допомагають прив’язати перевірку не до довільної паузи, а до фактичного стану сторінки або мережевої події. Основні механізми синхронізації, очікування та повторних спроб узагальнено в таблиці 2.4.

Таблиця 2.4 – *Основні механізми синхронізації, очікування та повторних спроб у Playwright for Java*

Механізм	**Призначення**	**Приклад застосування**
Actionability checks	Перевірка готовності елемента до взаємодії	Клік по кнопці лише після того, як вона видима і не перекрита
page.waitForLoadState()	Очікування стану завантаження сторінки	Завершення переходу після відкриття сторінки або перезавантаження
page.waitForURL()	Очікування зміни адреси сторінки	Перевірка переходу після авторизації або натискання меню
page.waitForSelector()	Очікування появи, зникнення або зміни стану елемента	Відображення повідомлення, таблиці, індикатора завантаження
waitForRequest()	Очікування вихідного HTTP-запиту	Контроль відправлення форми або запуску пошуку
waitForResponse()	Очікування відповіді сервера	Перевірка завершення API-виклику перед валідацією інтерфейсу
Таймаути дій	Обмеження часу для кліків, введення, пошуку елементів	Захист від нескінченного очікування при збої інтерфейсу
Таймаути навігації	Обмеження часу на завантаження сторінки або перехід	Контроль зависання переходів між сторінками

Auto-retrying assertions	Повторна перевірка умови до досягнення таймауту	Очікування появи тексту, видимості або URL
Retry запуску тесту	Повторний запуск сценарію після збою	Відсікання випадкових інфраструктурних збоїв

Метод page.waitForLoadState() використовується тоді, коли потрібно синхронізувати сценарій із завантаженням сторінки після переходу або оновлення. Метод page.waitForURL() придатний у випадках, де зміна маршруту сама по собі є частиною очікуваного результату, наприклад після авторизації, натискання пункту меню або завершення оформлення дії. Метод page.waitForSelector() дозволяє очікувати появу або зміну стану конкретного елемента на сторінці, тому він зручний для роботи з динамічними списками, повідомленнями про успіх, індикаторами помилки або модальними вікнами. Для сценаріїв, де стан інтерфейсу безпосередньо залежить від мережевої взаємодії, застосовують waitForRequest() і waitForResponse(). У такій моделі тест не просто натискає кнопку й одразу перевіряє DOM, а узгоджує перевірку з моментом відправлення запиту або отримання відповіді від сервера. Це зменшує кількість ситуацій, коли асинхронний інтерфейс ще не встиг оновитися, хоча дія вже формально виконана.

Використання Thread.sleep() у браузерних перевірках є значно слабшим підходом, ніж очікування подій або станів. Довільна пауза не враховує, скільки часу сторінка фактично потребує для завершення дії. Якщо пауза надто коротка, тест усе одно впаде; якщо надто довга, виконання штучно сповільнюється. Крім цього, фіксована затримка не дає відповіді на питання, чого саме очікує сценарій: зміни URL, появи елемента, завершення анімації чи мережевого виклику. Саме тому в описі системного тестування небажані довільні паузи прямо протиставляються очікуванню умов, які відображають реальний стан сторінки. Для асинхронних інтерфейсів це має прямий технічний наслідок: сценарій стає

прив’язаним не до штучного часу очікування, а до фактичної події. Через це прогін краще адаптується до відмінностей між локальним запуском, CI-середовищем і різними браузерами. Навіть коли інтерфейс спрацьовує швидко, умовне очікування завершується одразу після досягнення потрібного стану, тоді як Thread.sleep() завжди відпрацьовує весь заданий інтервал. У підсумку зменшується і тривалість набору, і кількість випадкових помилок синхронізації.

Таймаути визначають верхню межу очікування для дій, навігації та перевірок. У Playwright Java це реалізовано через класи опцій конкретних методів. Наприклад, в інтерфейсі Page є Page.CheckOptions, Page.ClickOptions, Page.DragAndDropOptions, Page.NavigateOptions, Page.WaitForNavigationOptions тощо; багато з них містять метод setTimeout(double timeout), який задає максимальний час очікування для відповідної операції. Для перевірок таймаут також задається через класи опцій assertion API, наприклад LocatorAssertions.IsVisibleOptions або LocatorAssertions.IsCheckedOptions. Без них сценарій може зависати надто довго в разі реального збою або недоступності сервера. Якщо таймаут надто малий, тест починає падати раніше, ніж інтерфейс встигає відреагувати в повільнішому середовищі. Якщо він завеликий, збої виявляються пізно, а прогін втрачає швидкість. Через це таймаути мають підбиратися не довільно, а відповідно до характеру дії: коротші для елементів інтерфейсу, довші для навігації, мережевих викликів і важких сторінок. З технічної точки зору таймаут є не лише засобом захисту від зависання, а й способом формалізувати припустиму тривалість операції в конкретному середовищі виконання [10, 13].

Retry-механізми існують у двох різних площинах, і ці випадки не варто змішувати. Перший випадок — це повторна перевірка умови всередині самої дії або assert-перевірки. Auto-retrying assertions, наприклад, toBeVisible(), toHaveText(), toHaveURL() та інші повторюються до досягнення таймауту.

(// Перевірка, що конкретна кнопка є видимою.

```
assertThat(
   page.getByRole(
      AriaRole.BUTTON,
      new Page.GetByRoleOptions().setName("Submit")
   )
).isVisible();
// Очікування до 10 секунд, що індикатор завантаження стане видимим.
assertThat(page.locator(".loading-spinner"))
   .isVisible(new LocatorAssertions.IsVisibleOptions().setTimeout(10_000));)
```

Це означає, що тест не падає миттєво в момент першої невідповідності, а дає сторінці час перейти в очікуваний стан. Другий випадок — це повторний запуск усього сценарію після його невдачі. Такий механізм іноді застосовується в CI для відсікання випадкових інфраструктурних проблем, але він не повинен замінювати коректну синхронізацію. Якщо тест регулярно падає без повторного запуску, причина найчастіше криється в логіці очікування, нестабільних даних, конфліктах стану або в зовнішньому середовищі. Повторний запуск може приховати проблему, але не усуває її. Саме тому retry варто розглядати як допоміжний запобіжник від разових збоїв, а не як основний спосіб стабілізації тестового набору.

У сценаріях, де сторінка активно взаємодіє з бекендом, правильна синхронізація часто потребує поєднання кількох механізмів. Наприклад, після натискання кнопки збереження може знадобитися очікування відправлення запиту, отримання відповіді, зміни URL і появи повідомлення в інтерфейсі. Якщо обмежитися тільки одним типом очікування, перевірка може виявитися передчасною. Якщо ж сценарій фіксує весь причинно-наслідковий ланцюг, його поведінка стає передбачуванішою. Такий підхід добре працює і для сторінок з частковим оновленням DOM, і для SPA-інтерфейсів, де браузер не виконує повне перезавантаження після дії користувача. У таких умовах саме комбінація очікування сторінки, URL, елемента й мережевої події дає найбільш надійну модель синхронізації.

Коректно налаштовані очікування, обґрунтовані таймаути й обмежене використання retry-механізмів зменшують частку flaky-тестів і роблять результат прогону більш відтворюваним. За такої організації набір перевірок спирається на фактичний стан сторінки й мережевих подій, а не на довільні затримки.

2.5 Підтримувані браузери, режими запуску, паралельне виконання та засоби діагностики

Java-реалізація Playwright підтримує три браузерні рушії: Chromium, Firefox і WebKit (браузер Safari використовує рушій WebKit. Його розробкою керує компанія Apple, а сам рушій є відкритим вихідним кодом. Крім того, через вимоги Apple усі інші браузери (наприклад, Chrome або Firefox) для iOS та iPadOS також вимушені працювати на базі WebKit. Натомість на macOS інші браузери можуть використовувати власні рушії). Такий набір дає змогу запускати один тестовий сценарій у кількох середовищах відображення сторінки без зміни загальної логіки коду [10].

Кросбраузерний запуск потрібний для перевірки того, як веб-застосунок поводиться в рушіях із різною реалізацією HTML, CSS, JavaScript та механізмів рендерингу. Навіть коли сторінка коректно працює в Chromium, це не гарантує однакової поведінки у Firefox або WebKit. Відмінності можуть проявлятися у верстці, порядку фокусування елементів, відпрацюванні анімацій, роботі випадаючих списків, діалогів, файлових операцій або сценаріїв навігації. Якщо системний тест запускається лише в одному браузері, частина дефектів залишається поза перевіркою. Підтримка трьох рушіїв через єдиний API зменшує витрати на розширення набору перевірок, оскільки одна і та сама логіка сценарію може бути виконана в різних браузерах із мінімальними змінами конфігурації. У навчальній та практичній літературі з Playwright кросбраузерний запуск прямо розглядається як одна з його базових можливостей [10, 23].

Окремо слід враховувати, що Chromium, Firefox та WebKit представляють не просто різні програми, а різні браузерні сімейства з відмінними підходами до рендерингу сторінок, обробки DOM, виконання JavaScript і підтримки окремих веб-стандартів. Через це перевірка в трьох рушіях дає ширше покриття, ніж багаторазовий запуск в одному середовищі з різними налаштуваннями, оскільки дозволяє виявляти відмінності, зумовлені саме особливостями браузерної реалізації, а не лише конфігурацією запуску. Такий підхід є важливим для системного тестування, коли необхідно переконатися, що веб-застосунок зберігає коректну поведінку, однакову логіку взаємодії та передбачуване відображення інтерфейсу в різних браузерних середовищах.

Порівняння підтримуваних браузерів, режимів запуску та діагностичних артефактів наведено в таблиці 2.5.

Таблиця 2.5 – *Підтримувані браузери, режими запуску та артефакти діагностики у Playwright for Java*

Група	**Варіант**	**Характеристика**	**Використання**
Браузерні рушії	Chromium	Підтримка сценаріїв у середовищі на базі Chromium	Перевірка більшості типових веб-сценаріїв
Браузерні рушії	Firefox	Окремий рушій із власною поведінкою рендерингу та подій	Виявлення розбіжностей у DOM і UI-поведінці
Браузерні рушії	WebKit	Рушій, близький до середовища Safari	Перевірка сумісності з іншим класом браузерів
Режими запуску	Headless	Виконання без відображення вікна браузера	Швидкі локальні та CI-прогони
Режими запуску	Headed	Запуск із візуальним вікном браузера	Налагодження, аналіз поведінки сторінки
Паралельність	Кілька потоків / кілька контекстів	Одночасне виконання незалежних сценаріїв	Скорочення часу регресії
Ізоляція	BrowserContext	Окремі cookies, cache, storage для кожного сценарію	Запобігання конфліктам стану
Артефакти	Скриншоти	Фіксація стану сторінки в певний момент	Аналіз падінь і помилок інтерфейсу

Артефакти	Відео	Запис проходження сценарію	Відтворення послідовності дій
Артефакти	Trace	Повна трасування виконання з кроками, мережею та станами	Дослідження причин збоїв

На рисунку 2.4 зображена схема паралельного запуску тестів на різних браузерах.

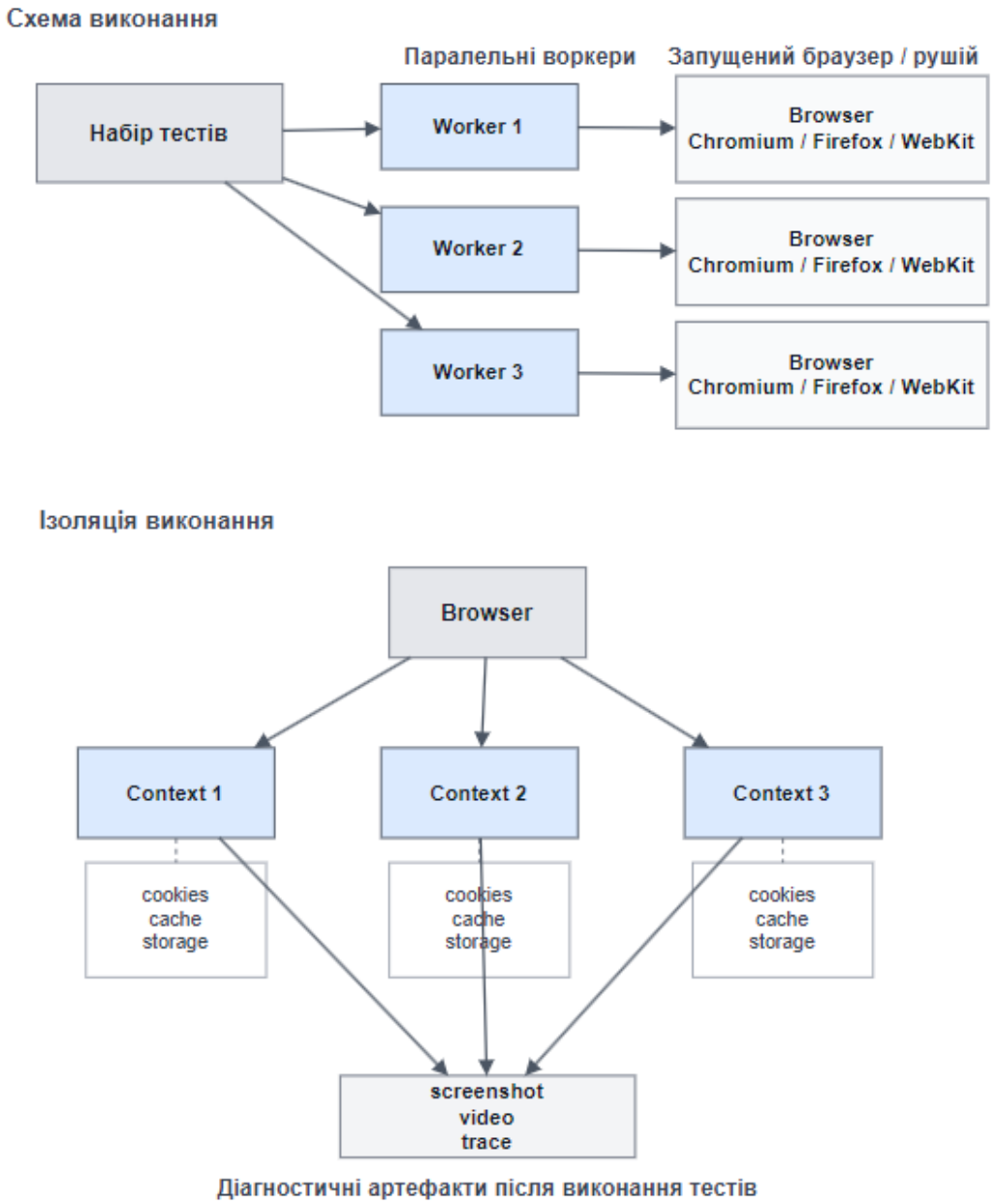


Рисунок 2.4 – *Схема паралельного запуску тестів і формування діагностичних артефактів*

На ній показано, що один набір тестів може бути розподілений між кількома браузерами або потоками виконання, а результати кожного запуску супроводжуються скриншотами, відео та trace-файлами. Запуск у режимах headless і headed відрізняється передусім способом виконання браузерного процесу. У headless-режимі браузер працює без відображення графічного вікна, тому такий варіант зазвичай застосовується для автоматизованих прогонів, де потрібні швидкість, економія ресурсів і можливість виконання на сервері без графічного інтерфейсу. У headed-режимі браузер запускається як звичайна візуальна програма, і це полегшує аналіз поведінки сторінки під час налагодження. Коли необхідно побачити порядок відкриття вкладок, появу модального вікна, зміну URL, анімацію або момент блокування елемента, візуальний режим дає більше інформації, ніж сухий лог виконання. Водночас для тривалих регресійних прогонів headed-режим менш економний, оскільки споживає більше системних ресурсів і не дає переваг там, де сценарій уже стабілізовано. Отже, headless-режим краще підходить для масового запуску, а headed-режим — для аналізу та виправлення проблемних сценаріїв.

Паралельне виконання зменшує сумарний час проходження регресійного набору. Якщо запускати всі сценарії послідовно, тривалість прогону зростає пропорційно кількості тестів, сторінок і перевірок. Коли ж сценарії розподіляються між кількома потоками, тестовий набір виконується швидше без зміни логіки окремих тестів. Для цього потрібна ізоляція стану, і саме тут використовується BrowserContext. Кожен контекст зберігає власні cookies, кеш і сховища, тому сценарії не впливають один на одного через спільну сесію. Така модель особливо корисна для наборів, де одночасно виконуються авторизація, створення записів, перевірка різних профілів доступу або переходи між кількома сторінками. Якщо паралельний запуск не супроводжується ізоляцією, тести починають конфліктувати через спільні дані, і швидкість досягається ціною

нестабільності. У матеріалах, присвячених інтеграції з CI/CD, паралельність і шардінг прямо розглядаються як засоби прискорення виконання без відмови від кросбраузерної перевірки [14, 18].

Засоби діагностики становлять окрему групу можливостей. Скриншот фіксує стан інтерфейсу в конкретний момент і дозволяє побачити повідомлення про помилку, зміщення верстки, відсутність елемента або інший результат дії. Відеозапис показує послідовність проходження сценарію від початку до завершення, тому з нього можна відновити, на якому саме етапі з’явилася проблема. Trace-файл містить детальнішу інформацію: кроки сценарію, мережеві події, зміну станів сторінки, вкладені дії та часову послідовність виконання. Playwright має засіб перегляду trace-файлу та його покрокового відтворення. За допомогою розширень JUnit Jupiter можна налаштувати запис trace-файлу або скриншота залежно від результату виконання тесту.. Поєднання цих артефактів дає змогу досліджувати не лише факт помилки, а й її розвиток у часі, що значно полегшує пошук причин збою.

Сумісність із трьома рушіями, наявність двох режимів запуску, підтримка паралельного виконання та формування діагностичних артефактів створюють технічне середовище, придатне для стабільного й керованого автоматизованого прогону веб-сценаріїв. Така сукупність властивостей добре відповідає вимогам системного тестування.

2.6 Робота з API та network interception

Автоматизоване системне тестування веб-застосунку не обмежується взаємодією з елементами інтерфейсу. Значна частина підготовки сценарію, контролю мережевої поведінки та стабілізації перевірок може виконуватися на рівні HTTP-викликів. У Java-реалізації Playwright для цього передбачено окремі

засоби роботи з API та мережею, які поєднуються з браузерною частиною сценарію в межах одного тестового середовища.

Для надсилання HTTP-запитів використовується APIRequestContext. Через нього можна виконувати GET, POST, PUT і DELETE-виклики, передавати заголовки, параметри, тіло запиту та обробляти відповіді сервера без відкриття сторінки в браузері. Результатом виклику HTTP-методів з APIRequestContext є об’єкт APIResponse, який містить метадані відповіді, HTTP-статус, заголовки та тіло відповіді [15, 16].

Такий підхід зручний тоді, коли тесту необхідно підготувати дані до початку UI-сценарію, створити сутність у системі, отримати токен доступу, очистити стан після виконання перевірки або перевірити відповідь сервера без проміжних дій через інтерфейс. Якщо, наприклад, сценарій має перевірити відображення списку замовлень, тест може попередньо створити потрібний запис через API, а вже потім відкрити сторінку та перевірити його наявність у браузері. За такого підходу час виконання зменшується, а залежність від зайвих кроків у UI послаблюється. У виданні, присвяченому автоматизації за допомогою цього інструмента, API testing, authentication і session storage розглядаються як окремий функціональний блок, що безпосередньо доповнює браузерні перевірки [23].

HTTP-запити у Playwright можна використовувати для підготовки тестових даних або скорочення зайвих UI-кроків. У межах цієї дипломної реалізації основний акцент зроблено на браузерних end-to-end сценаріях Sauce Demo, тому API-підготовка даних не є частиною коду проєкту. Ця можливість розглядається як напрям подальшого розширення набору.

Перехоплення мережевих запитів у Playwright дає змогу перевіряти або змінювати HTTP-взаємодію сторінки [10, 15]. В практичному проєкті основний результат досягнуто через UI-сценарії, Page Object Model, trace-файли й запуск тестів через Maven. Взаємозв’язок UI-сценарію, API-підготовки даних та network interception показано на рисунку 2.5.

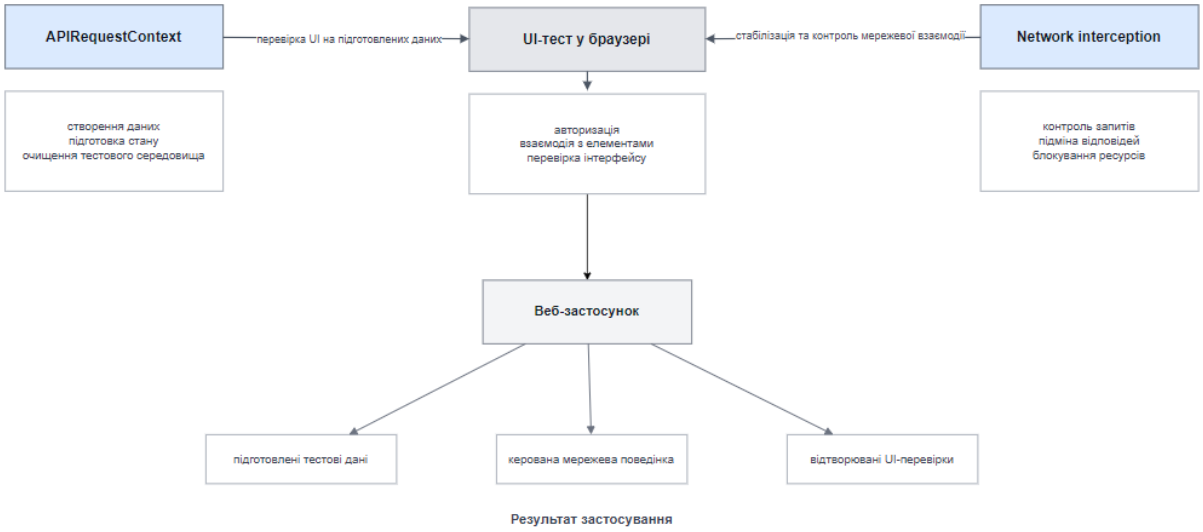


Рисунок 2.5 – *Поєднання UI-сценарію, API-підготовки даних та network interception у Playwright for Java*

Блокування або підміна ресурсів корисні для моделювання помилок сервера, затримок або нестандартних відповідей. У цьому дипломному проєкті такі сценарії не реалізовувалися, оскільки об’єктом тестування був демонстраційний застосунок Sauce Demo зі стабільними навчальними потоками. Для реального комерційного проєкту цей механізм можна додати як окремий крок перевірок.

2.7 Playwright Inspector та автоматична генерація тестових сценаріїв

Playwright Inspector і режим codegen використовуються для швидкого створення початкового варіанта тестового сценарію. Інструмент відкриває браузер, записує дії користувача та генерує код для Java API Playwright: переходи між сторінками, кліки, введення тексту, вибір елементів і базові перевірки.

Згенерований код не слід безпосередньо переносити у фінальний тестовий набір без аналізу, але він корисний як чернетка для визначення локаторів і послідовності дій [24].

У межах дипломного проєкту такий підхід доцільно використовувати на етапі дослідження інтерфейсу Sauce Demo. Наприклад, можна вручну пройти сценарій авторизації, додавання товару до кошика та checkout, отримати початковий код, а потім перенести стабільні локатори й послідовність кроків у Page Object-класи. Це дозволяє поєднати швидкість автоматичної генерації з підтримуваною архітектурою тестового проєкту.

Окремою перевагою Inspector є можливість покрокового налагодження сценарію. Тестувальник може призупинити виконання, переглянути доступні локатори, перевірити стан сторінки й уточнити, який елемент фактично обирається. Тому Playwright Inspector розглядається не як заміна проєктування тестів, а як допоміжний засіб для первинного аналізу сторінки, перевірки локаторів і прискорення розробки Page Object-методів.

2.8 Обрання веб-застосунку та визначення функціональних вимог

Як застосунок для тестування засобами Playwright обрано Sauce Demo - https://www.saucedemo.com. Це демонстраційний веб-магазин Sauce Labs, у якому є повний користувацький шлях: авторизація, перегляд каталогу, відкриття картки товару, додавання позицій до кошика, checkout і завершення замовлення. Така структура підходить для дипломного проєкту, бо дозволяє перевірити реальні end-to-end сценарії без залежності від випадкових зовнішніх даних [21, 25].

Sauce Demo має кілька наперед підготовлених користувачів з однаковим паролем secret_sauce. У роботі використано standard_user для позитивних сценаріїв, locked_out_user для перевірки блокування, problem_user для перевірки некоректних зображень і неможливості видалення товару з каталогу,

performance_glitch_user для сценарію повільного входу, а також error_user і visual_user як профілі для розширення негативних перевірок. Завдяки цьому об’єкт тестування містить як нормальну, так і проблемну поведінку без потреби створювати власні тестові дані [21].

Функціональна структура цього застосунку дозволяє виділити шість груп перевірок. Перша група пов’язана з авторизацією та обмеженням доступу до внутрішніх сторінок без входу. Друга група охоплює каталог товарів: відображення списку, кількість товарів, сортування, перехід до картки товару, додавання та видалення товару з каталогу. Третя група пов’язана зі сторінкою товару, де користувач переглядає деталі, додає товар до кошика або видаляє його. Четверта група охоплює кошик: склад замовлення, правильність кількості товарів, видалення позицій і повернення до каталогу. П’ята група охоплює checkout і навігаційне меню: оформлення замовлення, валідацію полів, перехід до каталогу, About, logout і скидання стану застосунку. Шоста група стосується проблемних користувачів Sauce Demo, зокрема problem_user, performance_glitch_user, error_user і visual_user. Якщо розглядати цей застосунок як об’єкт системного тестування, він дає змогу перевіряти як базову взаємодію з UI, так і послідовні бізнес-потоки, що складаються з кількох залежних кроків. Карту ключових користувацьких сценаріїв Sauce Demo наведено на рисунку 2.6. Для формалізації майбутніх автоматизованих перевірок необхідно виділити перелік функціональних вимог, які будуть покладені в основу сценаріїв. Такі вимоги мають бути конкретними, перевірюваними й пов’язаними з видимим результатом у браузері або зі зміною доступного стану сторінки. Узагальнення таких вимог наведено в таблиці 2.6.

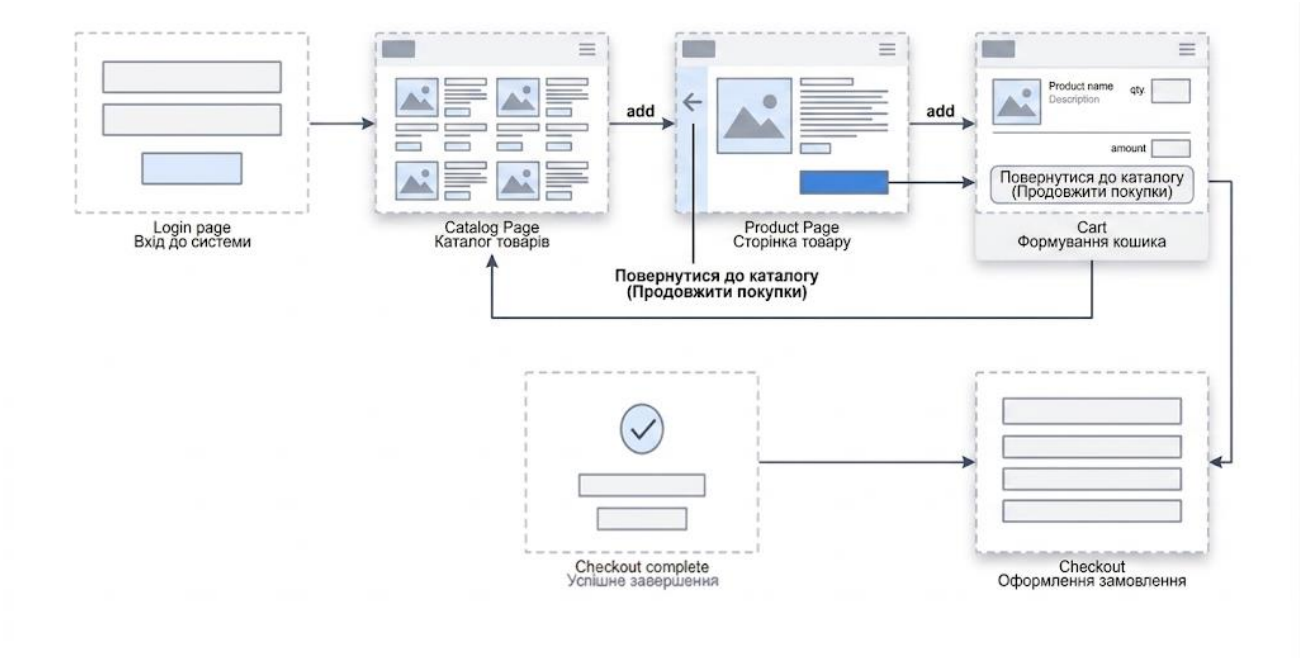


Рисунок 2.6 – *Карта ключових користувацьких сценаріїв веб-застосунку SauceDemo*

Таблиця *2.6 – Функціональні вимоги до веб-застосунку Sauce Demo*

Група перевірок	**Функціональна вимога**	**Очікуваний результат**
Авторизація	Користувач із валідними обліковими даними повинен успішно увійти до системи.	Відкривається сторінка каталогу Products.
Авторизація	Користувач із невалідними даними не повинен отримувати доступ до каталогу.	Відображається повідомлення про помилку авторизації.
Авторизація	Заблокований користувач не повинен входити до системи.	Відображається повідомлення про блокування облікового запису.
Доступ	Внутрішні сторінки inventory, cart і checkout не повинні відкриватися без авторизації.	Користувач повертається на сторінку входу й бачить повідомлення про обмеження доступу.
Каталог	Після входу користувач повинен бачити список товарів.	На сторінці відображається каталог із шістьма доступними позиціями.
Каталог	Користувач повинен мати можливість сортувати товари.	Порядок товарів змінюється відповідно до вибраної опції сортування.

Товар	Користувач повинен мати можливість відкрити картку товару з каталогу.	Відкривається сторінка товару з назвою, описом і ціною.
Товар	Користувач повинен мати можливість додати та видалити товар зі сторінки товару.	Лічильник кошика змінюється після додавання й повертається до нуля після видалення.
Кошик	Користувач повинен мати можливість додати товар до кошика з каталогу.	Лічильник кошика змінюється відповідно до кількості доданих позицій.
Кошик	Користувач повинен мати можливість видалити товар із каталогу або кошика.	Товар зникає з кошика, а лічильник зменшується.
Кошик	Користувач повинен бачити правильну кількість товарів у кошику.	Кількість позицій і badge кошика відповідають фактичному складу кошика.
Кошик	Користувач повинен мати можливість повернутися з кошика до каталогу.	Відкривається сторінка Products.
Checkout	Користувач повинен мати можливість розпочати checkout-процес.	Відкривається форма введення персональних даних.
Checkout	Checkout-форма не повинна приймати порожні обов’язкові поля.	Відображається повідомлення про відповідну помилку валідації.
Checkout	Після коректного заповнення форми користувач повинен завершити замовлення.	Відображається повідомлення Thank you for your order!.
Навігація	Користувач повинен мати можливість відкрити бокове меню.	У меню доступні All Items, About, Logout і Reset App State.
Навігація	Користувач повинен мати можливість перейти з меню до каталогу та сторінки About.	Відкривається каталог або формується коректне посилання на Sauce Labs.
Навігація	Користувач повинен мати можливість скинути стан застосунку та виконати logout.	Кошик очищується, а після logout відкривається сторінка входу.
Проблемні користувачі	Для problem_user потрібно перевіряти некоректні зображення та неможливість видалення товару з каталогу.	Тест фіксує відмінну від стандартного користувача поведінку.
Проблемні користувачі	Для performance_glitch_user, error_user і visual_user потрібно перевіряти використання як об’єктів негативних сценаріїв.	Користувачі авторизуються, а їхні особливості можуть бути покладені в основу окремих негативних перевірок.

Для побудови практичного набору достатньо однієї узагальненої схеми користувацького маршруту. Після успішного входу користувач потрапляє на сторінку каталогу Products, з якої може відкрити сторінку окремого товару або одразу додати товар до кошика. Додати й видалити товар можна як із каталогу, так і зі сторінки товару. Зі сторінки кошика користувач може повернутися до каталогу через Continue Shopping або перейти до checkout. Після заповнення checkout-форми відкривається сторінка огляду замовлення, а після завершення - фінальне повідомлення про успіх. Така схема не дублює окремі екрани, а показує правильну логіку переходів між ними.

РОЗДІЛ 3. ПРОЄКТУВАННЯ ТА РОЗРОБКА АВТОМАТИЗОВАНИХ ТЕСТ-ВИМОГ І СИСТЕМНИХ END-TO-END СЦЕНАРІЇВ ДЛЯ ВЕБ-ЗАСТОСУНКУ SAUCE DEMO

3.1 Проєктування тест-вимог за визначеними функціональними вимогами

Проєктування тест-вимог виконувалося з урахуванням того, що системний тест має перевіряти поведінку веб-застосунку як цілісної системи. Тому в основу набору покладено не ізольовані перевірки окремих методів, а завершені користувацькі потоки. Наприклад, перевірка кошика не обмежується фактом існування кнопки додавання товару. Сценарій спочатку виконує авторизацію, потім додає товар до кошика, відкриває сторінку кошика і лише після цього перевіряє, що обраний товар відображається в потрібному стані інтерфейсу. Такий підхід відповідає сутності системного тестування, де результат оцінюється через поведінку застосунку в умовах, наближених до реального використання.

У межах реалізації використано поділ тестів на позитивні (P) та негативні (N). Позитивні сценарії підтверджують коректне виконання основних бізнес-процесів: вхід у систему, перегляд каталогу, відкриття картки товару, додавання товару до кошика, перехід до оформлення замовлення, заповнення форми, завершення замовлення, роботу меню та навігації. Негативні сценарії перевіряють реакцію системи на неправильні або неповні дані, заблокований обліковий запис, доступ до внутрішніх сторінок без авторизації, помилки checkout-форми та особливу поведінку проблемних користувачів Sauce Demo. Поділ на дві групи дає змогу підтвердити працездатність основних потоків і оцінити коректність обмежень, валідації та повідомлень про помилки.

Оскільки об’єктом перевірки є веб-застосунок з готовими тестовими обліковими записами, підготовка даних виконувалася через фіксовані тестові

значення, надані Sauce Demo [21]. Для позитивної авторизації використано користувача standard_user з паролем secret_sauce. Для перевірки забороненого доступу використано locked_out_user. Для негативних сценаріїв застосовано problem_user, у якого проявляються некоректні зображення товарів і проблеми з видаленням товару з каталогу, а також performance_glitch_user, error_user і visual_user як окремі профілі для дослідження проблемної поведінки застосунку.

На етапі проєктування забезпечено простежуваність між функціональними вимогами, тестовими сценаріями та класами автоматизованих тестів. Для цього кожна вимога отримала умовний ідентифікатор FR, а кожний тестовий сценарій - умовний ідентифікатор TC. Такий зв’язок спрощує перевірку покриття: для кожної вимоги визначено тестовий клас, вхідні дані та очікуваний результат.

Критерієм успішності для більшості сценаріїв є не лише відсутність технічної помилки під час виконання тесту, а й перевірка конкретного стану інтерфейсу. Для авторизації таким станом є відображення сторінки Products. Для кошика - наявність доданого товару та зміна лічильника. Для checkout-форми - поява повідомлення про помилку або підтвердження успішного замовлення. Отже, автоматизований набір перевіряє результат з точки зору користувача, а не лише факт виконання кліку або переходу за URL.

Наведені параметри задають відтворюване середовище для запуску тестів і фіксації результатів. Систематизацію тестових вимог подано в таблиці 3.1.

Таблиця 3.1 – *Ідентифікатори тестових вимог*

ID	**Група перевірок**	**Функціональна вимога**	**Клас**	**Тип**
FR01	Авторизація	Користувач із валідними обліковими даними може авторизуватися.	LoginTests	P

FR02	Авторизація	Користувач із невалідним паролем не може авторизуватися.	LoginTests	N
FR03	Авторизація	Форма входу показує помилки для порожнього username або password.	LoginTests	N
FR04	Авторизація	Заблокований користувач не може увійти до системи.	LoginTests	N
FR05	Доступ	Внутрішні сторінки недоступні без входу.	LoginTests	N
FR06	Каталог	Після авторизації відкривається сторінка Products.	InventoryTests	P
FR07	Каталог	На сторінці Products відображається список із шести товарів.	InventoryTests	P
FR08	Каталог	Користувач може сортувати товари.	InventoryTests	P
FR09	Товар	Користувач може відкрити сторінку товару з каталогу.	ProductTests	P
FR10	Товар	Користувач може додати й видалити товар зі сторінки товару.	ProductTests	P
FR11	Кошик	Користувач може додати товар до кошика з каталогу.	InventoryTests, CartTests	P
FR12	Кошик	Користувач може видалити товар із каталогу або кошика.	InventoryTests, CartTests	P

FR13	Кошик	Лічильник і кількість позицій у кошику змінюються коректно.	InventoryTests, CartTests	P
FR14	Кошик	Користувач може повернутися з кошика до каталогу.	CartTests, NavigationTests	P
FR15	Checkout	Користувач може перейти з кошика до checkout-форми.	CartTests, CheckoutTests	P
FR16	Checkout	Checkout-форма приймає валідні дані та завершує замовлення.	CheckoutTests	P
FR17	Checkout	Checkout-форма показує помилки для порожніх обов’язкових полів.	CheckoutTests	N
FR18	Навігація	Користувач може відкрити меню та перейти до каталогу.	InventoryTests, NavigationTests	P
FR19	Навігація	Користувач може перевірити About, Reset App State і Logout.	NavigationTests	P
FR20	Проблемні користувачі	problem_user відображає некоректні зображення товарів.	ProblemUserTests	N
FR21	Проблемні користувачі	problem_user не може видалити товар з каталогу стандартним способом.	ProblemUserTests	N
FR22	Проблемні користувачі	performance_glitch_user, error_user для негативних перевірок.	ProblemUserTests	N

Матриця відповідності показує, що автоматизований набір охоплює основні функціональні блоки Sauce Demo: авторизацію, доступ до захищених сторінок, каталог, сторінку товару, кошик, checkout, меню та проблемних

користувачів. Найбільша кількість сценаріїв належить до авторизації, каталогу, кошика, навігації та негативних профілів, оскільки ці блоки формують основний бізнес-потік і водночас містять найбільше ризиків для системного тестування.

Під час формування тест-вимог враховано ризики веб-інтерфейсу. Для форми входу значущими є перевірки доступу та повідомлень про помилки. Для каталогу критичними є наявність товарів, додавання в кошик, видалення та сортування. Для сторінки товару важливо перевірити перехід із каталогу й операції Add/Remove. Для кошика потрібна перевірка правильності кількості позицій, видалення та повернення до каталогу. Для checkout-процесу центральними є перевірка обов'язкових полів і підтвердження завершення замовлення. Для проблемних користувачів важливо зафіксувати поведінку, що відрізняється від standard_user.

3.2 Структура автоматизованого тестового проєкту

Автоматизований тестовий проєкт реалізовано як Maven-проєкт з розміщенням тестового коду в каталозі src/test/java. Така структура відповідає типовій організації Java-проєктів, де production-код і тестовий код розділяються на рівні стандартних директорій. Оскільки завданням роботи є саме автоматизація браузерних перевірок, основна логіка зосереджена в тестовому модулі, а не в src/main/java.

У файлі pom.xml описано залежності й параметри запуску тестового проєкту. Playwright for Java 1.60.0 забезпечує керування браузером, JUnit Jupiter 6.1.0 використовується для структури тестів, параметризації та тегів. Maven Surefire Plugin запускає тести й підтримує відбір сценаріїв через властивість groups, а Maven Compiler Plugin налаштовано на компіляцію з release 25 через встановлений JDK 25 [18, 19, 20].

Лістинг *3.1 – Фрагмент pom.xml з основними залежностями*

```
    <properties>
        <maven.compiler.release>25</maven.compiler.release>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
        <junit.jupiter.version>6.1.0</junit.jupiter.version>
        <playwright.version>1.60.0</playwright.version>
        <browser>chromium</browser>
        <headless>true</headless>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.microsoft.playwright</groupId>
            <artifactId>playwright</artifactId>
            <version>${playwright.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>${junit.jupiter.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>${junit.jupiter.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-params</artifactId>
            <version>${junit.jupiter.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

У проєкті використано п’ять основних пакетів: base, config, pages, tests та utils. Структуру автоматизованого тестового проєкту Playwright for Java наведено на рисунку 3.1.

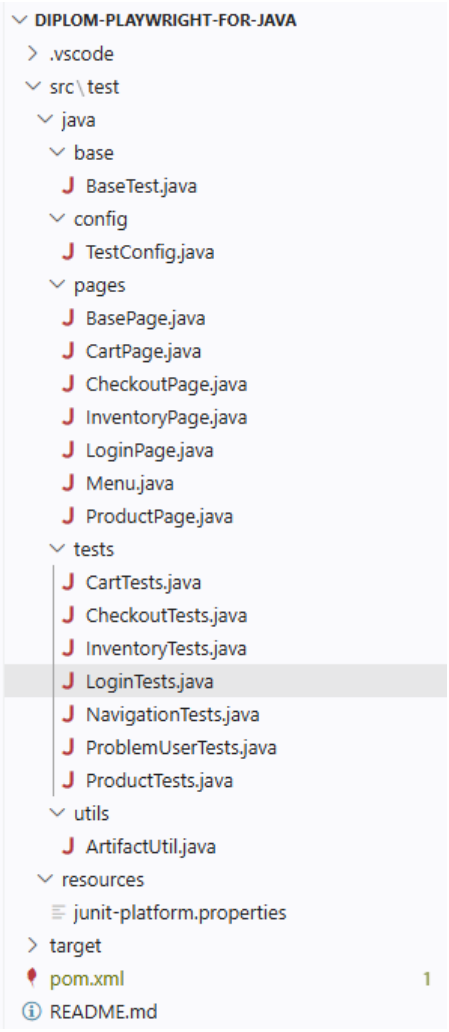


Рисунок 3.1 – *Структура автоматизованого тестового проєкту Playwright for Java*

Пакет config містить TestConfig з параметрами запуску. Пакет base містить BaseTest, який керує життєвим циклом Playwright, Browser, BrowserContext і Page. Пакет pages містить BasePage, Menu і Page Object-класи для сторінок Sauce Demo: LoginPage, InventoryPage, ProductPage, CartPage і CheckoutPage. Пакет tests містить тестові класи LoginTests, InventoryTests, ProductTests, CartTests, CheckoutTests, NavigationTests і ProblemUserTests. Пакет utils містить ArtifactUtil для створення каталогів і шляхів до артефактів.

Обрана структура зменшує дублювання коду та дозволяє підтримувати тести як керований набір. Наприклад, логіка авторизації описана в LoginPage і

викликається з різних тестових класів, але локатори полів username, password і кнопки login не дублюються в кожному тесті. Аналогічно, дії з кошиком інкапсульовані в CartPage, а дії з формою оформлення - у CheckoutPage. У випадку зміни верстки достатньо буде оновити відповідний Page Object, а не всі тестові сценарії.

У класі BaseTest реалізовано життєвий цикл браузерного сценарію з урахуванням ефективності. Екземпляр Playwright і браузер створюються один раз для тестового класу в @BeforeAll, а закриваються в @AfterAll. Перед кожним тестом створюється окремий BrowserContext і Page, тому cookies, localStorage та кошик не змішуються між сценаріями. Каталоги для артефактів створюються перед запуском браузера, а trace-файл зберігається після кожного тесту в target/traces. Такий підхід поєднує ізоляцію тестів із меншими витратами на запуск браузера. Тестове середовище та інструментальні засоби реалізації узагальнено в таблиці 3.2.

Таблиця 3.2 – *Тестове середовище та інструментальні засоби реалізації*

Компонент	Значення	Призначення
Мова програмування	Java 25	Використовується для реалізації тестових класів, Page Object та службових утиліт.
Система складання	Maven	Керує залежностями, запуском тестів і формуванням результатів у target-каталозі.
Фреймворк браузерної автоматизації	Playwright for Java 1.60.0	Керує браузером, контекстами, сторінками, локаторами, скриншотами та trace-файлами. Забезпечує assert-методи.
Модуль тестування	JUnit Jupiter 6.1.0	Забезпечує анотації тестів, параметризацію, теги, життєвий цикл і паралельний запуск.
Об’єкт тестування	Sauce Demo	Веб-застосунок із типовими сценаріями електронної комерції.
Основний браузер запуску	Chromium	Використовується за замовчуванням; додатково підтримуються Firefox і WebKit.
Артефакти виконання	screenshots, traces	Зберігаються для аналізу збоїв та документування результатів прогону.

Особливістю реалізації є підтримка кількох браузерів через системну властивість browser. За замовчуванням запускається Chromium, а за значеннями параметра browser той самий набір виконується у Firefox або WebKit. Це узгоджується з обраним підходом до кросбраузерного запуску, де кросбраузерність визначено однією з переваг Playwright for Java. Перемикання браузера через параметр не потребує зміни коду тестів і підвищує переносимість набору. Структуру тестового проєкту Playwright for Java подано в таблиці 3.3.

Таблиця 3.3 – *Структура тестового проєкту Playwright for Java*

Пакет	**Файл**	**Призначення**
config	TestConfig.java	Централізоване зберігання URL, браузера, headless-режиму, шляхів до артефактів.
base	BaseTest.java	Спільний життєвий цикл Playwright і Browser у @BeforeAll/@AfterAll; окремий BrowserContext і Page для кожного тесту.
pages	BasePage.java	Батьківський клас сторінок із заголовком, посиланням на кошик, badge кошика та об’єктом Menu.
pages	Menu.java	Опис бокового меню: перехід до каталогу, About, Logout і Reset App State.
pages	LoginPage.java	Опис сторінки авторизації, полів username/password, кнопки login і повідомлень про помилки.
pages	InventoryPage.java	Опис каталогу Products, списку товарів, сортування, додавання й видалення товарів.
pages	ProductPage.java	Опис сторінки окремого товару, ціни, опису, кнопок Add/Remove і повернення до каталогу.
pages	CartPage.java	Опис кошика, перевірки товарів, кількості позицій, видалення, checkout і повернення до каталогу.
pages	CheckoutPage.java	Опис checkout-форми, обов’язкових полів, повідомлень валідації та завершення замовлення.

tests	LoginTests.java	Позитивні й негативні сценарії авторизації, включно з обмеженням доступу до внутрішніх сторінок.
tests	InventoryTests.java	Перевірки каталогу, списку товарів, кількості, сортування, додавання, видалення та меню.
tests	ProductTests.java	Перевірки сторінки товару та додавання/видалення товару безпосередньо з неї.
tests	CartTests.java	Перевірки складу кошика, кількості позицій, видалення, checkout і повернення до каталогу.
tests	CheckoutTests.java	Позитивний checkout і параметризовані негативні сценарії обов’язкових полів.
tests	NavigationTests.java	Перевірки переходів, бокового меню, About, Reset App State і Logout.
tests	ProblemUserTests.java	Негативні сценарії для problem_user, performance_glitch_user, error_user і visual_user.
utils	ArtifactUtil.java	Створення каталогів, безпечних назв файлів, збереження скриншотів і trace-шляхів.

Подана структура розділяє конфігурацію, сторінкові об’єкти, сценарії та засоби збереження артефактів.

Лістинг 3.2 – Фрагмент ініціалізації тестового середовища в BaseTest

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
    public abstract class BaseTest {
        protected Playwright playwright;
        protected Browser browser;
        protected BrowserContext context;
        protected Page page;

        @RegisterExtension
        FailureArtifactExtension failureArtifactExtension =
new FailureArtifactExtension();
```

```
        @BeforeAll
        void startBrowser() {
            ArtifactUtil.createArtifactDirectories();
            playwright = Playwright.create();
            browser = launchBrowser();
        }

        @BeforeEach
        void setUp(TestInfo testInfo) {
            context = browser.newContext(new
Browser.NewContextOptions()
                    .setViewportSize(1440, 900));
            context.tracing().start(new
Tracing.StartOptions()
                    .setScreenshots(true)
                    .setSnapshots(true)
                    .setSources(true));
            page = context.newPage();
            page.navigate(TestConfig.BASE_URL);
        }
```

У наведеному фрагменті видно, що Playwright і Browser не створюються перед кожним тестом. Перед кожним сценарієм формується лише новий BrowserContext і Page, запускається трасування та виконується перехід на базову адресу Sauce Demo. Глобальні таймаути в BaseTest не зменшуються: сценарії покладаються на стандартні механізми auto-waiting Playwright, а додаткові таймаути задаються тільки там, де це виправдано конкретною дією або перевіркою.

Клас ArtifactUtil забезпечує доказову базу виконання. У разі падіння тесту створюється скриншот, а після кожного тесту зберігається trace-файл. Для іменування файлів використовується назва тесту та часовий маркер. Це дає змогу зіставити конкретний артефакт із конкретним сценарієм і швидко знайти дані для аналізу невдачі.

Лістинг 3.3 – Централізована конфігурація тестового запуску

```
public final class TestConfig {
        public static final String BASE_URL =
"https://www.saucedemo.com/";
```

```
        public static final String BROWSER =
System.getProperty("browser", "chromium").toLowerCase();
        public static final boolean HEADLESS =
Boolean.parseBoolean(System.getProperty("headless", "true"));
        public static final Path SCREENSHOTS_DIR =
Path.of("target", "screenshots");
        public static final Path TRACES_DIR =
Path.of("target", "traces");

        private TestConfig() {
        }
    }
```

Винесення конфігурації в окремий клас робить тестовий набір гнучкішим. Значення browser і headless зчитуються із системних властивостей, тому один і той самий код запускається в різних режимах. Шляхи screenshots і traces фіксують, де зберігаються артефакти. Окремої константи загального таймауту в TestConfig немає: сценарії покладаються на стандартні auto-waiting механізми Playwright, а додаткові очікування задаються локально там, де вони потрібні.

3.3 Реалізація Page Object Model для сторінок Sauce Demo

Для організації взаємодії з інтерфейсом використано Page Object Model [7, 22]. Цей підхід дозволяє відокремити технічні деталі пошуку елементів від логіки тестових сценаріїв. Тестовий клас описує, що перевіряється, а Page Object описує, як виконуються дії на конкретній сторінці. Такий розподіл підвищує читабельність і придатність автоматизованого набору до супроводу.

У межах реалізації створено Page Object-класи: BasePage, Menu, LoginPage, InventoryPage, ProductPage, CartPage і CheckoutPage. BasePage містить спільні елементи сторінок після авторизації: заголовок застосунку, заголовок сторінки, посилання на кошик, badge кошика та об’єкт Menu. Menu винесено окремим класом, оскільки бокове меню повторюється на різних сторінках і містить спільні дії: перехід до каталогу, About, logout і reset стану застосунку. ProductPage описує

сторінку окремого товару, що дає змогу перевірити додавання та видалення товару не лише з каталогу, а й зі сторінки товару.

Перевагою застосунку Sauce Demo є те що більшість його Веб-елементів мають data-test атрибути. Такі атрибути стабільніші за складні CSS-ланцюги або XPath, що залежать від вкладеності DOM. Наприклад, поле логіна визначається через [data-test='username'], поле пароля - через [data-test='password'], кнопка входу - через [data-test='login-button']. Якщо зміниться розташування елементів у верстці, але атрибути залишаться незмінними, тести продовжать працювати без виправлення селекторів.

Методи Page Object-класів мають бізнес-орієнтовані назви. Замість того щоб у тесті щоразу писати заповнення двох полів і клік по кнопці, використовується метод login(String username, String password). Для каталогу передбачено метод addProductToCart(String productName), який шукає товар за назвою та натискає відповідну кнопку. Для checkout-форми є методи fillFirstName, fillLastName, fillPostalCode, submitForm і finishOrder. Завдяки цьому тестові сценарії читаються як послідовність дій користувача.

Окремо реалізовано методи перевірки стану сторінок. Наприклад, isOpened у LoginPage перевіряє видимість полів входу та кнопки, isOpened в InventoryPage перевіряє заголовок Products, isOpened у CartPage - заголовок Your Cart. Для завершеного замовлення метод isOrderComplete перевіряє наявність повідомлення Thank you for your order!. У такий спосіб перевірки не розпорошуються між тестами, а залишаються поряд із відповідною сторінкою.

Лістинг 3.4 – Фрагмент BasePage зі спільними елементами сторінок

```
public abstract class BasePage {
        protected final Page page;
        private final Locator appLogo;
        private final Locator title;
        private final Locator cartLink;
        private final Locator cartBadge;
        private final Menu menu;
```

```
        protected BasePage(Page page) {
            this.page = page;
            this.appLogo = page.locator(".app_logo");
            this.title = page.locator("[data-test='title']");
            this.cartLink = page.locator("[data-
test='shopping-cart-link']");
            this.cartBadge = page.locator("[data-
test='shopping-cart-badge']");
            this.menu = new Menu(page);
        }

        public Menu menu() {
            return menu;
        }
```

Фрагмент BasePage демонструє винесення повторюваних елементів сторінок у батьківський клас. Завдяки цьому InventoryPage, ProductPage, CartPage і CheckoutPage не дублюють локатори заголовка, кошика та меню. Якщо структура header або бокового меню зміниться, правка вноситься в одному місці.

Лістинг 3.5 – Фрагмент роботи з товарами на сторінці Products

```
    public void addProductToCart(String productName) {
            productItem(productName).locator("button").click(
);
        }

        public void removeProductFromCatalog(String
productName) {
            productItem(productName).locator("button").click(
);
        }

        public ProductPage openProduct(String productName) {
            productItem(productName).locator("[data-
test='inventory-item-name']").click();
            return new ProductPage(page);
        }

        public void sortProducts(String value) {
            sortSelect.selectOption(value);
        }
```

У InventoryPage для пошуку товару використано фільтрацію елементів списку за текстом назви. Це дає змогу не прив’язувати сценарій до порядку товарів у каталозі. Якщо користувач сортує список, метод addProductToCart продовжить знаходити потрібний товар за назвою, а не за позицією. Такий підхід знижує крихкість тестів і робить сценарії ближчими до реальної поведінки користувача. Приклад сторінки Products після успішної авторизації користувача наведено на рисунку 3.2.

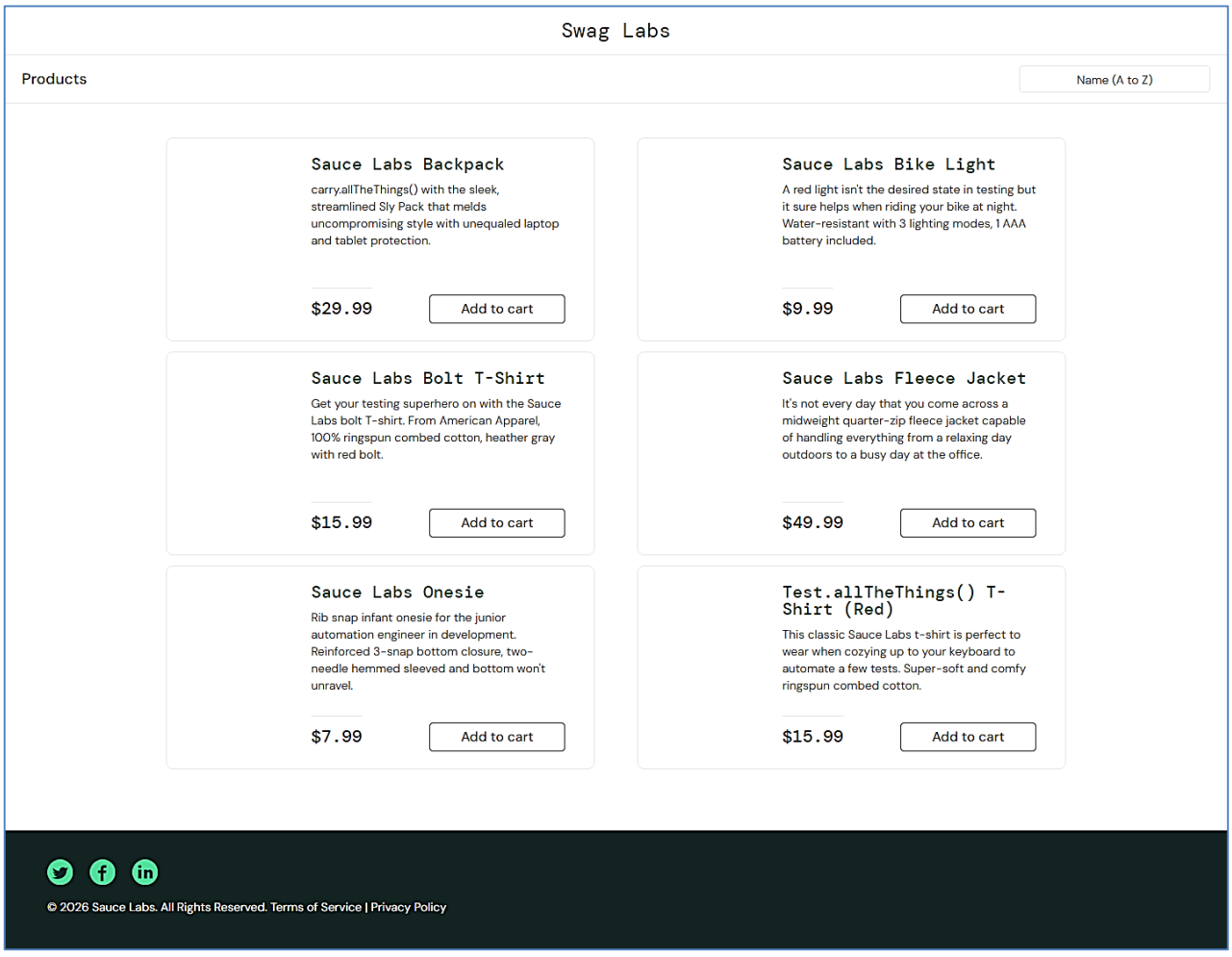


Рисунок 3.2 – *Сторінка Products після успішної авторизації користувача*

На рисунку 3.2 показано стан застосунку після успішного входу стандартного користувача. На сторінці відображається заголовок Products, список товарів, ціни, кнопки Add to cart і елемент сортування. Саме цей стан є очікуваним результатом позитивного сценарію авторизації та початковою точкою для більшості тестів каталогу, кошика й checkout-процесу.

3.4 Розробка позитивних і негативних тестових сценаріїв

Тестові сценарії реалізовано у семи класах: LoginTests, InventoryTests, ProductTests, CartTests, CheckoutTests, NavigationTests і ProblemUserTests. Такий поділ відповідає функціональним модулям застосунку та спрощує вибірковий запуск. За допомогою тегів окремо запускаються тести авторизації, каталогу, сторінки товару, кошика, checkout, навігації, доступу без входу або проблемних користувачів.

Позитивні сценарії перевіряють основні користувацькі потоки. LoginTests підтверджує успішну авторизацію standard_user. InventoryTests перевіряє сторінку Products, кількість товарів, додавання, видалення, сортування та меню. ProductTests перевіряє відкриття сторінки товару та операції Add/Remove зі сторінки товару. CartTests тестує додавання, відображення, видалення, кількість позицій, перехід до checkout і повернення до каталогу. CheckoutTests проходить повний шлях оформлення замовлення. NavigationTests перевіряє кошик, меню, About, Reset App State і logout.

Негативні сценарії реалізовано для функцій із валідацією, доступом і проблемними профілями користувачів. LoginTests перевіряє неправильний пароль, порожній username, порожній password, locked_out_user та спроби відкрити внутрішні сторінки без авторизації. CheckoutTests перевіряє обов’язкові поля форми оформлення. ProblemUserTests фіксує відмінності problem_user,

performance_glitch_user, error_user і visual_user. Такий набір показує не лише стандартну роботу застосунку, а й очікувані проблемні сценарії Sauce Demo.

Використання @CsvSource є доцільним для компактного опису наборів даних. Для кожного рядка задається комбінація вхідних значень і очікуваний результат. Під час виконання JUnit Jupiter 6.1.0 створює окремий прогін тестового методу для кожного рядка. З точки зору звітності це означає, що кілька варіантів негативної перевірки обробляються як окремі сценарії, але код залишається коротким і підтримуваним.

Додатково в Page Object-класах перевіряється видимість елементів і текст заголовків. Такий підхід відповідає рекомендаціям щодо побудови стабільних UI-тестів: тест має чекати не довільну кількість секунд, а конкретний стан елемента або сторінки [12, 13].

Лістинг 3.6 – Параметризований негативний тест авторизації

```
    @ParameterizedTest(name = "{index}: {2}")
    @CsvSource(value = {
          "standard_user|wrong_password|Epic sadface: Username
and password do not match any user in this service",
           "''|secret_sauce|Epic sadface: Username is
required",
           "standard_user|''|Epic sadface: Password is
required",
           "locked_out_user|secret_sauce|Epic sadface: Sorry,
this user has been locked out."
    }, delimiter = '|')
    @Tag("negative")
    @DisplayName("Перевірка помилок авторизації")
    void loginValidationErrorsAreDisplayed(String username,
String password, String expectedError) {
        LoginPage loginPage = new LoginPage(page);

        loginPage.login(username, password);

        assertEquals(expectedError,
loginPage.getErrorMessage());
    }
```

Параметризований тест авторизації перевіряє кілька негативних умов одним методом. Він зберігає єдину логіку дій: відкрити сторінку, ввести логін і пароль, натиснути кнопку входу та порівняти фактичне повідомлення з очікуваним. Змінюються лише дані, що надходять із CsvSource. Такий підхід корисний для форм, де багато правил валідації мають базову структуру перевірки.

Лістинг 3.7 – Параметризований тест перевірки обов’язкових полів checkout-форми

```
@ParameterizedTest(name = "{index}: {3}")
    @CsvSource({
            "'', Petrenko, 01001, Error: First Name is
required",
            "Ivan, '', 01001, Error: Last Name is required",
            "Ivan, Petrenko, '', Error: Postal Code is
required"
    })
    @Tag("negative")
    @DisplayName("Перевірка помилок checkout-форми")
      void  checkoutRequiredFieldsValidation(String  firstName,
String lastName, String postalCode, String expectedError) {
        checkoutPage.fillFirstName(firstName);
        checkoutPage.fillLastName(lastName);
        checkoutPage.fillPostalCode(postalCode);
        checkoutPage.submitForm();

        assertEquals(expectedError,
checkoutPage.getErrorMessage());
    }
```

У checkout-сценаріях негативна перевірка виконується після попередньої підготовки стану: користувач авторизується, додає товар до кошика, переходить у кошик і відкриває форму оформлення. Форма не доступна ізольовано від бізнес-процесу, тому навіть негативна перевірка обов’язкових полів залишається end-to-end сценарієм, який проходить через кілька сторінок застосунку. Узагальнений перелік розроблених тестових сценаріїв наведено в таблиці 3.4.

Таблиця 3.4 – *Розроблені тестові сценарії автоматизованого набору*

ID	Сценарій	Дані та очікуваний результат
TC01	Успішна авторизація standard_user (Login; P)	standard_user / secret_sauce; відкрито Products.
TC02	Авторизація з неправильним паролем (Login; N)	Показано повідомлення про невідповідність логіна й пароля.
TC03	Авторизація без username (Login; N)	Показано Username is required.
TC04	Авторизація без password (Login; N)	Показано Password is required.
TC05	Авторизація locked_out_user (Login; N)	Показано повідомлення про блокування користувача.
TC06	Відкриття inventory.html без входу (Access; N)	Сторінка не відкривається без авторизації.
TC07	Відкриття cart.html без входу (Access; N)	Сторінка не відкривається без авторизації.
TC08	Відкриття checkout-step-one.html без входу (Access; N)	Сторінка не відкривається без авторизації.
TC09	Відкриття Products після входу (Inventory; P)	Заголовок Products видимий.
TC10	Відображення списку товарів (Inventory; P)	Список товарів видимий і не порожній.
TC11	Перевірка кількості товарів (Inventory; P)	У каталозі відображається 6 товарів.
TC12	Додавання товару до кошика (Inventory; P)	Badge кошика дорівнює 1.
TC13	Видалення товару з каталогу (Inventory; P)	Badge кошика повертається до 0.
TC14	Зміна badge після додавання двох товарів (Inventory; P)	Badge кошика дорівнює 2.
TC15	Сортування товарів (Inventory; P)	Перший товар змінюється відповідно до сортування za.
TC16	Відкриття навігаційного меню (Inventory; P)	У меню доступні основні дії.
TC17	Logout через меню (Inventory; P)	Відкрито сторінку авторизації.
TC18	Відкриття сторінки товару з каталогу (Product; P)	Відкрито картку Sauce Labs Backpack з ціною.
TC19	Додавання й видалення товару зі сторінки товару (Product; P)	Badge змінюється 1 -> 0, повернення до Products.
TC20	Додавання товару у кошик (Cart; P)	Badge кошика дорівнює 1.

TC21	Відображення товару в кошику (Cart; P)	Товар є на сторінці Your Cart.
TC22	Видалення товару з кошика (Cart; P)	Кошик порожній.
TC23	Порожній кошик після видалення (Cart; P)	Кількість позицій дорівнює 0.
TC24	Зміна кількості після додавання й видалення двох товарів (Cart; P)	Кількість змінюється 2 -> 1 -> 0.
TC25	Перехід із кошика до checkout (Cart; P)	Відкрито Checkout: Your Information.
TC26	Повернення з кошика до каталогу (Cart; P)	Відкрито Products.
TC27	Успішне оформлення замовлення (Checkout; P)	Показано Thank you for your order!.
TC28	Checkout без first name (Checkout; N)	Показано Error: First Name is required.
TC29	Checkout без last name (Checkout; N)	Показано Error: Last Name is required.
TC30	Checkout без postal code (Checkout; N)	Показано Error: Postal Code is required.
TC31	Перехід на сторінку кошика (Navigation; P)	Відкрито Your Cart.
TC32	Повернення з кошика до каталогу (Navigation; P)	Відкрито Products.
TC33	Відкриття меню (Navigation; P)	Меню доступне.
TC34	Перехід до каталогу з меню (Navigation; P)	Відкрито Products.
TC35	Перевірка About з меню (Navigation; P)	Посилання веде на https://saucelabs.com/.
TC36	Reset App State через меню (Navigation; P)	Після скидання badge кошика дорівнює 0.
TC37	Logout через меню (Navigation; P)	Відкрито сторінку авторизації.
TC38	Перевірка сторінки після logout (Navigation; P)	Видимі поля username і password.
TC39	problem_user має некоректні зображення (ProblemUser; N)	Зафіксовано проблемні placeholder-зображення.
TC40	problem_user не може видалити товар з каталогу (ProblemUser; N)	Badge залишається 1 після спроби видалення.

TC41	performance_glitch_user відкриває каталог із затримкою (ProblemUser; N)	Products відкривається після повільнішого входу.
TC42	error_user доступний для негативних перевірок (ProblemUser; N)	Користувач авторизується й відкриває Products.
TC43	visual_user доступний для візуальних негативних перевірок (ProblemUser; N)	Користувач авторизується й відкриває Products.

Набір сценаріїв охоплює позитивні й негативні перевірки авторизації, каталогу, кошика, checkout-форми та навігації. Приклад повідомлення про помилку авторизації при некоректних даних наведено на рисунку 3.3.

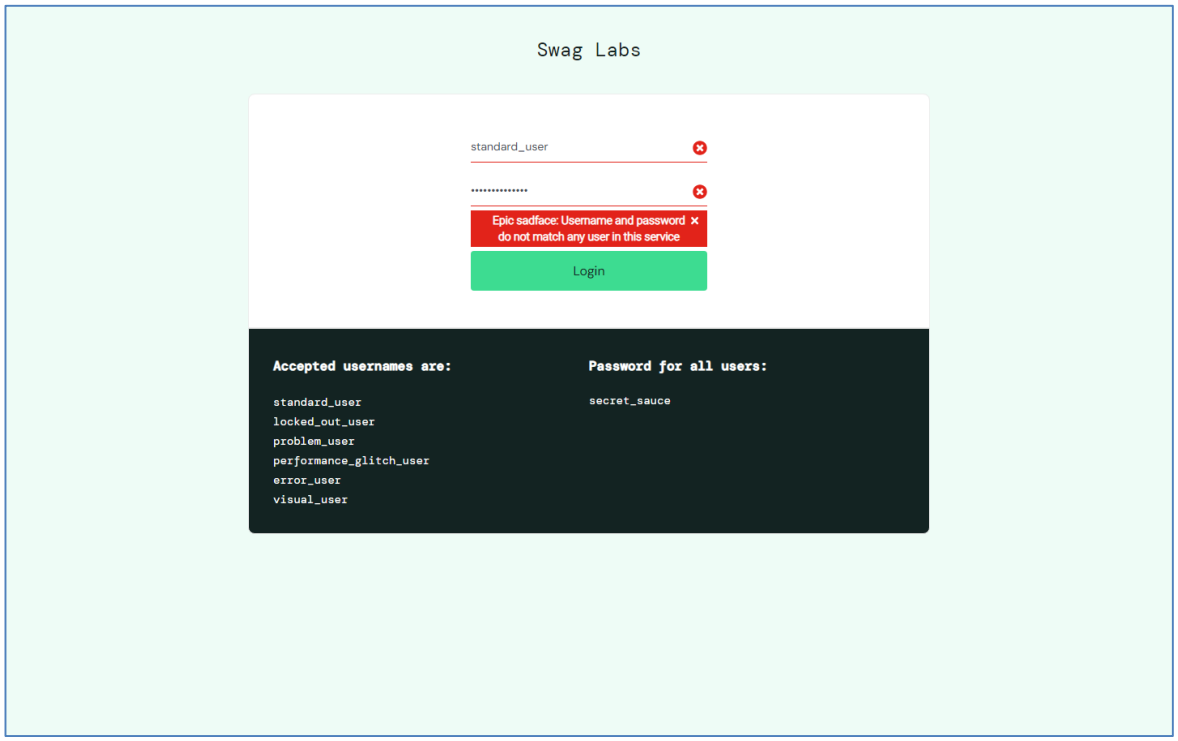


Рисунок 3.3 – *Повідомлення про помилку авторизації при некоректних даних*

На рисунку 3.3 наведено приклад негативного результату авторизації. Система залишається на сторінці входу і показує повідомлення Epic sadface. У тесті такий стан перевіряється через текст повідомлення про помилку, що дає змогу впевнено підтвердити не лише сам факт відмови у вході, а й правильність реакції інтерфейсу. Сторінку кошика з доданим товаром показано на рисунку 3.4.

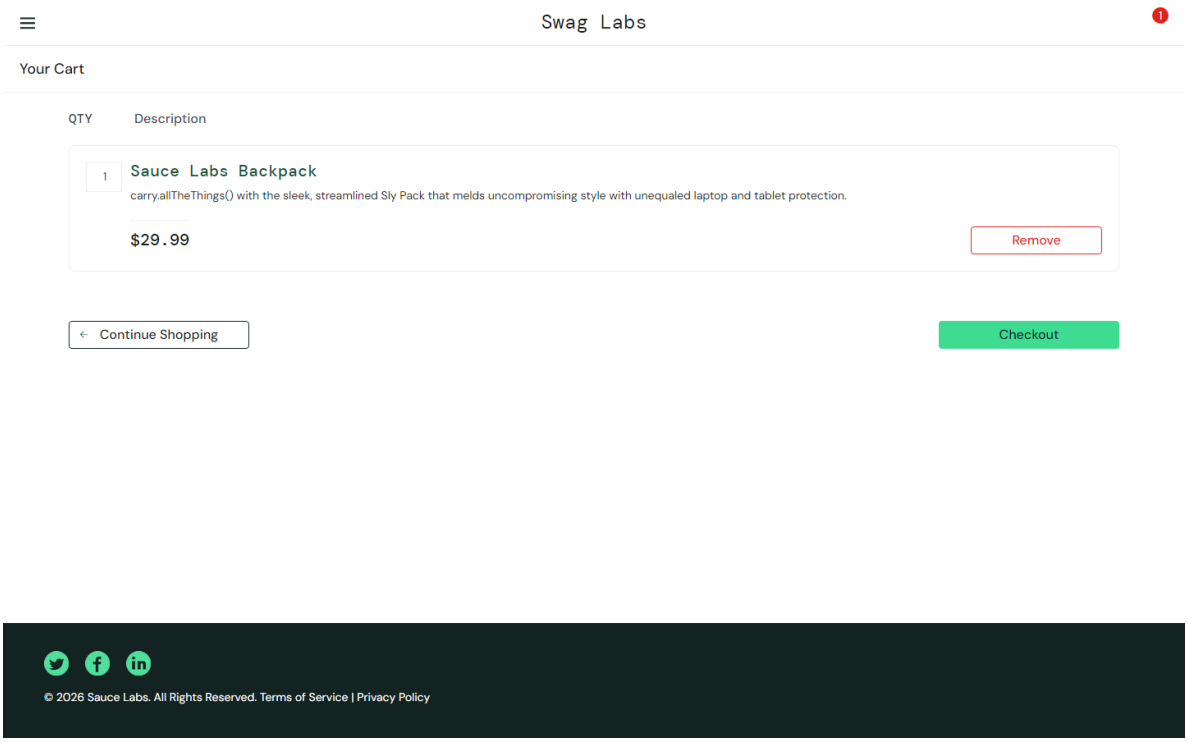


Рисунок 3.4 – *Сторінка кошика з доданим товаром*

Рисунку 3.4 фіксує проміжний стан бізнес-процесу після додавання товару Sauce Labs Backpack і переходу до кошика. Для системного тестування цей стан підтверджує передачу даних між сторінкою Products і сторінкою Your Cart. Якщо товар був доданий, але не відобразився в кошику, це свідчило б про дефект у взаємодії між модулями застосунку.

3.5 Реалізація тестових наборів, параметризації та запуску через Maven

Для групування тестів використано анотацію @Tag. Кожен тестовий клас має функціональний тег: login, inventory, product, cart, checkout, navigation або problem-users. Додатково сценарії позначаються тегами positive, negative, access, problem-user або performance. Це дозволяє запускати повний набір або вибіркові групи. Після зміни форми авторизації виконуються login-тести, після зміни сторінки товару - product-тести, а для перевірки проблемних профілів - problem-users [18].

Запуск тестів виконується через Maven. Базова команда mvn test запускає весь набір. Відбір сценаріїв виконується через властивість groups, яку Maven Surefire передає JUnit Platform: наприклад, mvn test -Dgroups=negative або mvn test -Dgroups=checkout. Такий спосіб запуску зручний для CI, оскільки одна і та сама кодова база може виконувати різні набори залежно від етапу pipeline [19, 20].

У файлі junit-platform.properties увімкнено паралельне виконання на рівні тестових класів. Для цього задано параметри junit.jupiter.execution.parallel.enabled=true, junit.jupiter.execution.parallel.mode.default=same_thread, junit.jupiter.execution.parallel.mode.classes.default=concurrent і junit.jupiter.execution.parallel.config.fixed.parallelism=3. У поєднанні з окремим BrowserContext для кожного тесту це дозволяє скоротити тривалість прогону, не змішуючи cookies, localStorage та інші дані сесії між сценаріями [14, 18].

Паралельний запуск вимагає дисципліни в роботі зі спільним станом. У реалізованому проєкті тестові класи не використовують спільний статичний Page або один загальний BrowserContext. Кожен сценарій починає виконання з нової сторінки та нового контексту. Без такої ізоляції залишки стану після

попереднього тесту, наприклад товар у кошику або активна сесія, могли б спотворити результат наступної перевірки.

Підтримка trace-файлів підсилює діагностику реалізації. Trace містить інформацію про кроки, мережеві події, знімки сторінок і стан DOM, тому застосовується для покрокового аналізу падіння. Це підтверджує, що автоматизоване системне тестування охоплює запуск перевірок і збір даних для подальшого аналізу [17]. Команди запуску автоматизованого тестового набору наведено в таблиці 3.5.

Таблиця 3.5 – *Команди запуску автоматизованого тестового набору*

Команда	**Призначення**
mvn test	Запуск повного набору з 43 системних тестів.
mvn test -Dgroups=positive	Запуск лише позитивних сценаріїв.
mvn test -Dgroups=negative	Запуск лише негативних сценаріїв.
mvn test -Dgroups=login	Запуск сценаріїв авторизації.
mvn test -Dgroups=access	Запуск перевірок доступу до внутрішніх сторінок без входу.
mvn test -Dgroups=inventory	Запуск сценаріїв каталогу Products.
mvn test -Dgroups=product	Запуск сценаріїв сторінки окремого товару.
mvn test -Dgroups=cart	Запуск сценаріїв кошика.
mvn test -Dgroups=checkout	Запуск сценаріїв checkout-процесу.
mvn test -Dgroups=navigation	Запуск сценаріїв навігації.
mvn test -Dgroups=problem-users	Запуск негативних сценаріїв для проблемних користувачів.
mvn test -Dbrowser=firefox	Запуск набору у Firefox.
mvn test -Dbrowser=webkit	Запуск набору у WebKit.
mvn test -Dheadless=false	Запуск у headed-режимі для візуального налагодження.

У процесі роботи з автоматизованим набором використовуються різні режими запуску. Повний запуск потрібний для регресійної перевірки. Вибіркові

запуски за тегами корисні під час локального налагодження або після змін в окремому функціональному модулі. Запуск у різних браузерах дозволяє виявити проблеми кросбраузерної сумісності. Headed-режим використовується тоді, коли потрібно візуально спостерігати за діями тесту або швидко зрозуміти, на якому кроці виникає проблема. Checkout-форму для введення даних користувача показано на рисунку 3.5.

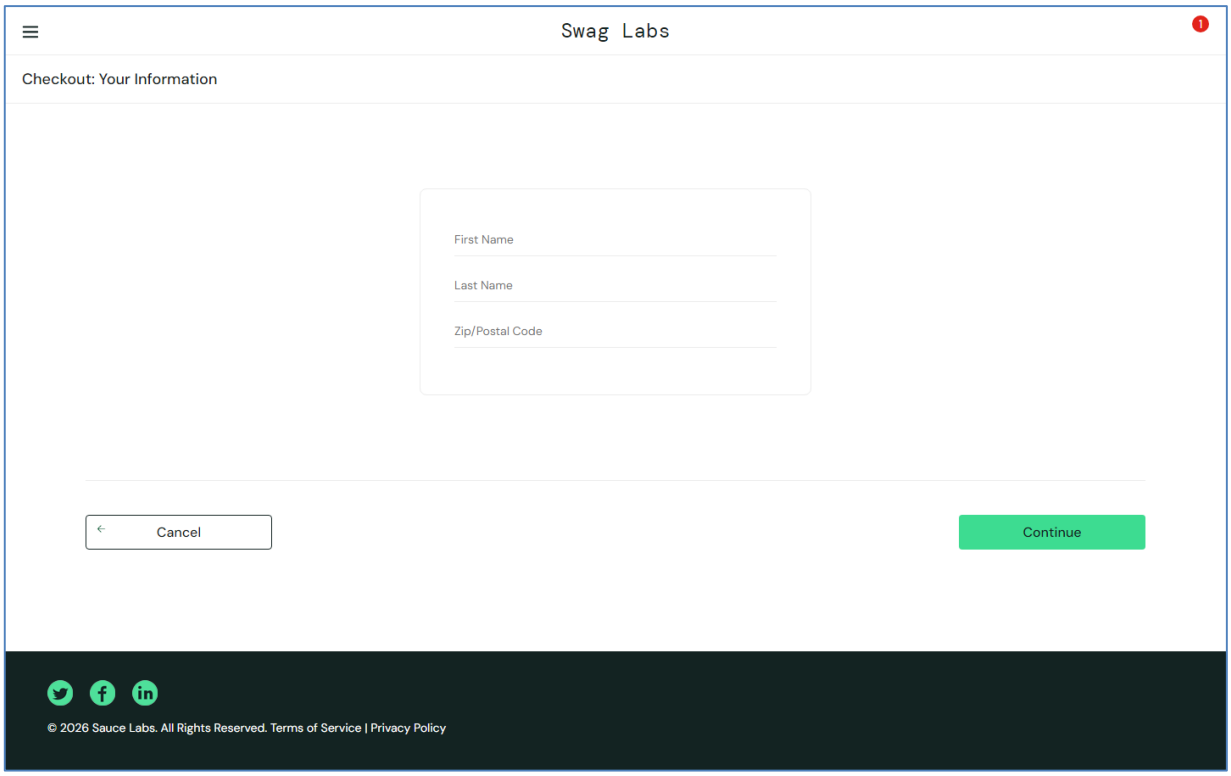


Рисунок 3.5 – *Checkout-форма для введення даних користувача*

На рисунку 3.5 показано форму введення даних користувача під час оформлення замовлення. Ця форма є центральною для перевірки правил валідації. Автоматизований тест заповнює поля first name, last name і postal code, після чого натискає кнопку Continue. У позитивному сценарії користувач переходить до наступного кроку оформлення, а в негативних сценаріях система повинна залишитися на формі та показати відповідне повідомлення. Приклад

повідомлення про помилку при порожньому полі First Name наведено на рисунку 3.6.

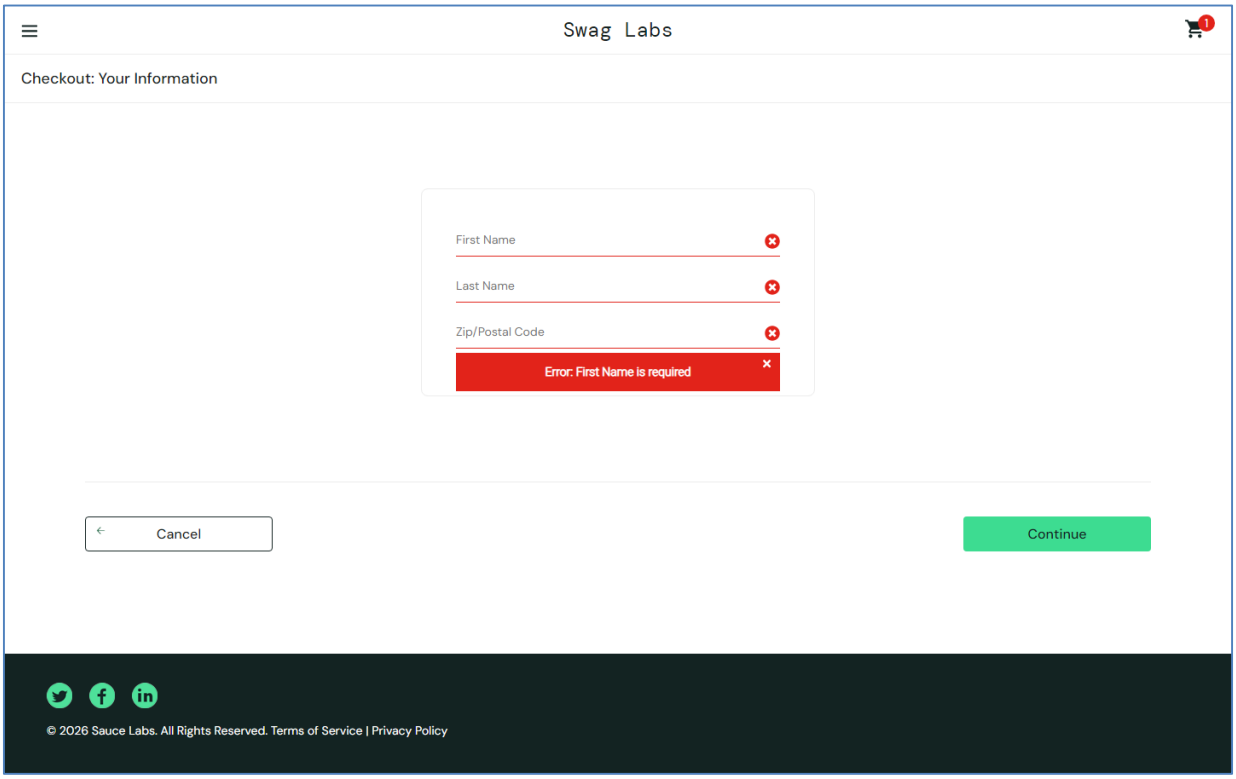


Рисунок 3.6 – *Повідомлення про помилку при порожньому полі First Name*

Рисунку 3.6 ілюструє один із негативних checkout-сценаріїв. Поле First Name залишено порожнім, тому після спроби продовжити оформлення з'являється повідомлення Error: First Name is required. Такий сценарій перевіряє не тільки наявність валідації, а й те, що повідомлення відображається в інтерфейсі в зрозумілому для користувача місці.

3.6 Аналіз результатів виконання системних end-to-end тестів

Після реалізації тестового проєкту виконано системний прогін основних і негативних сценаріїв. Загальна кількість автоматизованих перевірок становить 43

сценарії, з яких 28 належать до позитивних, а 15 - до негативних. Такий розподіл відповідає структурі реалізованого набору: позитивні перевірки підтверджують головні користувацькі потоки, а негативні охоплюють помилки авторизації, обмеження доступу, валідацію checkout-форми та проблемних користувачів Sauce Demo.

Результати прогону оцінювалися за двома групами ознак. Перша група - це технічний результат JUnit: тест завершився успішно або з помилкою. Друга група - функціональний результат: фактичний стан інтерфейсу відповідає очікуваному результату, зафіксованому в сценарії. Наприклад, успішне завершення тесту checkout означає не лише відсутність винятку в коді, а й появу повідомлення Thank you for your order! на фінальній сторінці.

Під час аналізу результатів потрібно враховувати, що end-to-end тести проходять через кілька шарів застосунку. Якщо падає тест оформлення замовлення, причина може бути не лише в самій checkout-формі. Помилка може виникнути на етапі авторизації, додавання товару, переходу в кошик або натискання кнопки Checkout. Тому в проєкті передбачено trace-файли і скриншоти: вони дозволяють швидко визначити точний крок, на якому фактична поведінка відрізнилася від очікуваної.

Результати реалізації показують, що обраний стек працює для системного тестування Sauce Demo. Playwright for Java виконує браузерні дії й зберігає trace-файли, JUnit Jupiter 6.1.0 організує тести, теги й параметризовані сценарії, Maven запускає весь набір або окремі групи через -Dgroups. Page Object Model з BasePage і Menu зменшує дублювання локаторів.

Отримані скриншоти й trace-файли демонструють основні стани веб-застосунку: помилку авторизації, відкритий каталог товарів, сторінку товару, кошик із доданим товаром, checkout-форму, помилку обов’язкового поля та успішне завершення замовлення. Ці стани підтверджують виконання позитивних

і негативних сценаріїв та фіксують фактичну поведінку інтерфейсу. Узагальнені результати автоматизованого тестового набору подано в таблиці 3.6.

Таблиця 3.6 – *Узагальнені результати автоматизованого тестового набору*

Тестовий клас	**Кількість**	**Розподіл**	**Область і стан**
LoginTests	8	P: 1; N: 7	Авторизація, помилки входу та обмеження доступу. Стан: виконано.
InventoryTests	9	P: 9; N: 0	Каталог, кількість товарів, сортування, додавання/видалення, меню. Стан: виконано.
ProductTests	2	P: 2; N: 0	Сторінка товару, додавання й видалення з картки товару. Стан: виконано.
CartTests	7	P: 7; N: 0	Кошик, кількість позицій, видалення, checkout і повернення до каталогу. Стан: виконано.
CheckoutTests	4	P: 1; N: 3	Оформлення замовлення й валідація обов’язкових полів. Стан: виконано.
NavigationTests	8	P: 8; N: 0	Переходи, меню, About, reset стану та logout. Стан: виконано.
ProblemUserTests	5	P: 0; N: 5	Проблемні користувачі Sauce Demo. Стан: виконано.
Усього	43	P: 28; N: 15	Основні й негативні end-to-end потоки Sauce Demo. Стан: виконано.

Зведення результатів підтверджує проходження запланованих перевірок без падінь у межах виконаного прогону. Діагностичні артефакти виконання тестів наведено в таблиці 3.7.

Таблиця 3.7 – *Діагностичні артефакти виконання тестів*

Артефакт	Розташування	Зміст	Призначення
Скриншоти	target/screenshots	Створюються при падінні тесту через FailureArtifactExtension.	Швидкий візуальний аналіз фактичного стану сторінки.
Trace-файли	target/traces	Зберігаються після виконання сценарію з даними про кроки, DOM, мережу і знімки.	Покрокове відтворення сценарію в Playwright Trace Viewer.
JUnit-звіт	target/surefire-reports	Формується Maven Surefire Plugin після запуску тестів.	Фіксація статусу тестових методів і часу виконання.
Консольний вивід Maven	terminal	Показує кількість запущених, успішних і помилкових тестів.	Оперативна перевірка результату прогону.

Артефакти забезпечують відтворення стану сторінки під час аналізу прогону та помилок.

Приклад відкритого trace-файлу наведено на рисунку 3.7: у Trace Viewer відображаються кроки сценарію, стан сторінки, часову шкалу виконання та деталі вибраної дії. Фінальну сторінку після успішного оформлення замовлення показано на рисунку 3.8.

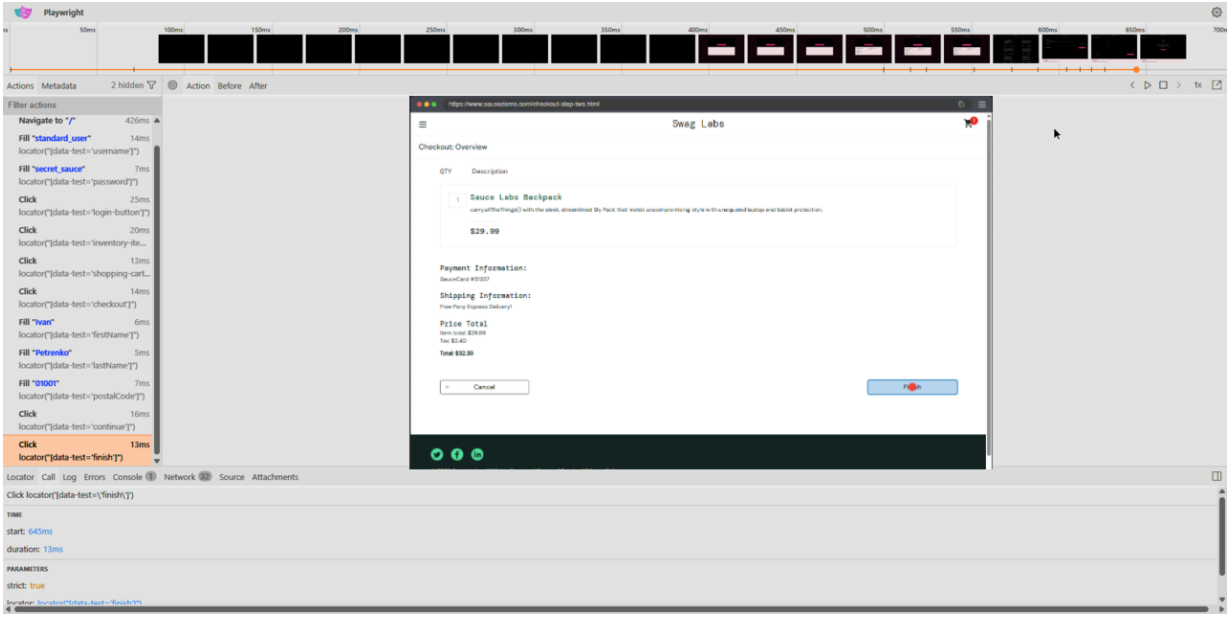


Рисунок 3.7 – *Приклад перегляду trace-файлу у Playwright Trace Viewer*

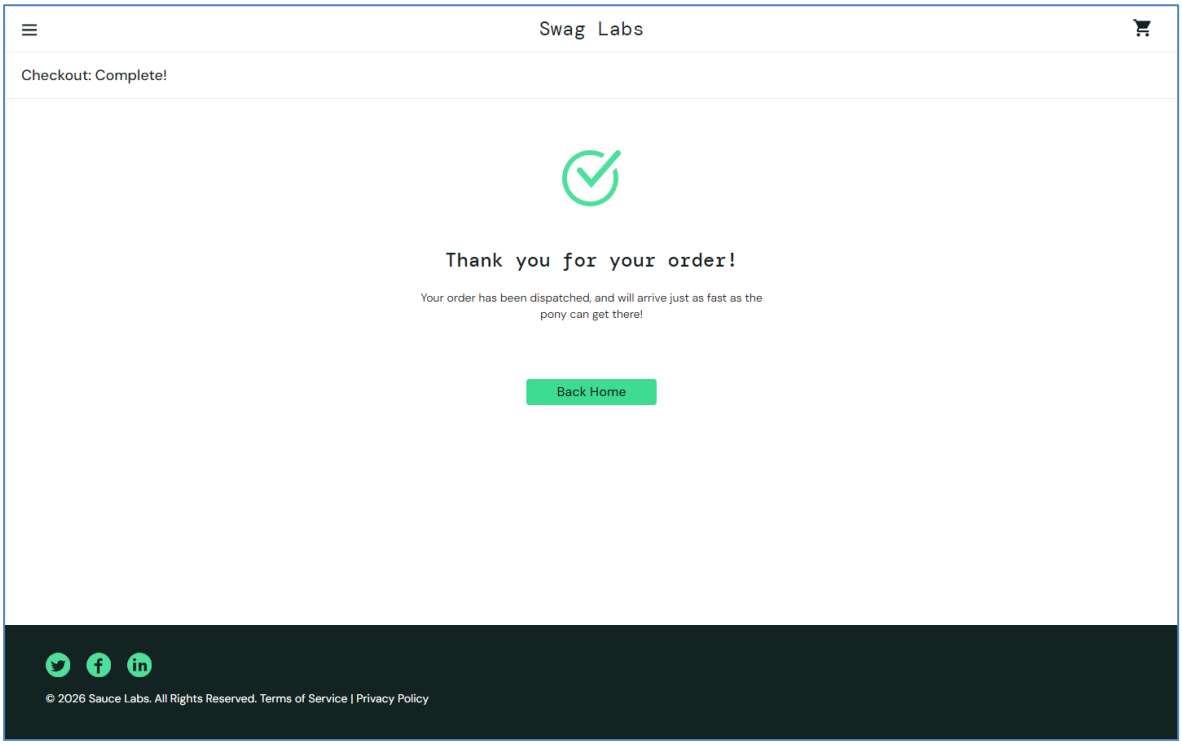


Рисунок 3.8 – *Фінальна сторінка після успішного оформлення замовлення*

На рисунку 3.8 зображено фінальний стан успішного checkout-сценарію. Повідомлення Thank you for your order! є очікуваним результатом повного бізнес-потоку: авторизація, додавання товару до кошика, перехід до checkout, заповнення форми та завершення оформлення. Саме такий сценарій є типовим прикладом end-to-end тесту, оскільки він охоплює кілька сторінок і перевіряє завершений шлях користувача.

Реалізований набір можна використовувати як регресійну перевірку Sauce Demo. Він покриває основні UI-потоки, але не замінює модульні, API або візуальні тести. Надалі проєкт можна розширити Allure-звітами, CI/CD-запуском і окремими перевірками для інших браузерів.

Відтворюваність забезпечується тим, що залежності описані в pom.xml, тестовий код зберігається у стандартній Maven-структурі, а параметри browser і

headless передаються через системні властивості. Після встановлення JDK 25 і браузерів Playwright проєкт запускається стандартною командою mvn test.

У практичній частині сформовано тест-вимоги, реалізовано 43 сценарії у 7 тестових класах, створено Page Object-класи BasePage, Menu, LoginPage, InventoryPage, ProductPage, CartPage і CheckoutPage, підключено параметризацію JUnit Jupiter 6.1.0, скриншоти при падіннях і trace-файли після кожного тесту. Це підтверджує придатність Playwright for Java для підтримуваного end-to-end набору системних тестів.

ВИСНОВКИ

У результаті роботи встановлено, що Playwright for Java підходить для автоматизації повних браузерних сценаріїв. У проєкті це проявилось у перевірках авторизації, каталогу, кошика, checkout-процесу, меню та проблемних користувачів. Auto-waiting і web-first assertions зменшують потребу в довільних паузах, а локатори Playwright дозволяють перевіряти стан інтерфейсу через видимі елементи.

Ізоляція стану реалізована через BrowserContext. У проєкті Playwright і Browser створюються один раз для тестового класу, а для кожного тесту відкриваються нові BrowserContext і Page. Завдяки цьому cookies, localStorage, авторизація та стан кошика не переходять між сценаріями.

Кросбраузерна підтримка Chromium, Firefox і WebKit дає змогу запускати той самий набір у різних браузерних рушіях без переписування тестів. У реалізації це підтримано через системну властивість browser, яка дозволяє перемикати Chromium, Firefox або WebKit під час запуску Maven.

Діагностика результатів забезпечується скриншотами при падіннях і trace-файлами після кожного тесту. Trace Viewer показує послідовність дій, DOM, мережеві події та знімки сторінки, тому помилку можна аналізувати не лише за консольним повідомленням, а й за фактичним станом інтерфейсу.

Практична реалізація підтвердила, що Playwright for Java, Maven, JUnit Jupiter 6.1.0 і Page Object Model достатні для побудови відтворюваного end-to-end набору. У проєкті реалізовано 43 сценарії: 28 позитивних і 15 негативних. Вони охоплюють авторизацію, обмеження доступу, каталог, сторінку товару, кошик, checkout, навігаційне меню та проблемних користувачів Sauce Demo.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Авраменко А. С., Авраменко В. С., Косенюк Г. В. Тестування програмного забезпечення: навчальний посібник. Черкаси: ЧНУ ім. Б. Хмельницького, 2017. 284 с.

2. Золотухіна О. А., Негоденко О. В., Резник С. Ю., Разіна С. Я. Якість та тестування інформаційних систем: навчальний посібник. Київ: ННІІТ ДУТ, 2020. 128 с.

3. ISTQB Glossary. System testing; Test automation . URL: https://istqb-glossary.page/ (дата звернення: 21.04.2026).

4. SeleniumHQ. Selenium WebDriver Documentation . URL: https://www.selenium.dev/documentation/webdriver/ (дата звернення: 23.04.2026).

5. SeleniumHQ. Tips on working with locators . URL: https://www.selenium.dev/documentation/test_practices/encouraged/locators/ (дата звернення: 23.04.2026).

6. SeleniumHQ. Waiting Strategies . URL: https://www.selenium.dev/documentation/webdriver/waits/ (дата звернення: 23.04.2026).

7. SeleniumHQ. Page Object Models . URL: https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/ (дата звернення: 23.04.2026).

8. Selenide Team. Selenide Documentation . URL: https://selenide.org/documentation.html (дата звернення: 23.04.2026).

9. Garcia B. WebDriverManager . URL: https://bonigarcia.dev/webdrivermanager/ (дата звернення: 29.04.2026).

10. Microsoft. Playwright for Java Documentation . URL: https://playwright.dev/java/ (дата звернення: 14.05.2026).

11. Microsoft. Playwright for Java: Locators . URL: https://playwright.dev/java/docs/locators (дата звернення: 14.05.2026).

12. Microsoft. Playwright for Java: Auto-waiting . URL: https://playwright.dev/java/docs/actionability (дата звернення: 14.05.2026).

13. Microsoft. Playwright for Java: Assertions . URL: https://playwright.dev/java/docs/test-assertions (дата звернення: 14.05.2026).

14. Microsoft. Playwright for Java API: BrowserContext . URL: https://playwright.dev/java/docs/api/class-browsercontext (дата звернення: 14.05.2026).

15. Microsoft. Playwright for Java API: APIRequestContext . URL: https://playwright.dev/java/docs/api/class-apirequestcontext (дата звернення: 14.05.2026).

16. Microsoft. Playwright for Java API: APIResponse . URL: https://playwright.dev/java/docs/api/class-apiresponse (дата звернення: 14.05.2026).

17. Microsoft. Playwright for Java: Trace Viewer . URL: https://playwright.dev/java/docs/trace-viewer (дата звернення: 14.05.2026).

18. JUnit Team. JUnit User Guide . URL: https://docs.junit.org/current/user-guide/ (дата звернення: 16.05.2026).

19. Apache Software Foundation. Apache Maven Project Documentation . URL: https://maven.apache.org/guides/index.html (дата звернення: 21.05.2026).

20. Apache Software Foundation. Maven Surefire Plugin . URL: https://maven.apache.org/surefire/maven-surefire-plugin/ (дата звернення: 21.05.2026).

21. Sauce Labs. Sauce Demo Web Application . URL: https://www.saucedemo.com/ (дата звернення: 22.05.2026).

22. Kovalenko D. Selenium Design Patterns and Best Practices. Birmingham: Packt Publishing, 2014. ISBN 978-1-78398-271-4. URL:

https://openlibrary.org/books/OL39586454M/Selenium_Design_Patterns_and_Best_Practices (дата звернення: 22.05.2026).

23. Pathak K. Web Automation Testing Using Playwright: End-to-end, API, accessibility, and visual testing using Playwright. New Delhi: BPB Publications, 2025. ISBN 978-93-6589-800-2. URL: https://bpbonline.com/products/web-automation-testing-using-playwright (дата звернення: 22.05.2026).

24. Microsoft. Playwright for Java: Codegen. URL: https://playwright.dev/java/docs/codegen-intro (дата звернення: 22.05.2026).

25. Sauce Labs. Playwright Quickstart. URL: https://docs.saucelabs.com/web-apps/automated-testing/playwright/quickstart/ (дата звернення: 22.05.2026)

26. Самонов Д. Автоматизація системного тестування веб-застосунків із використанням Playwright for Java Збірник наукових праць студентів, здобувачів PhD, докторантів і молодих вчених, університетська конференція «МОЛОДА НАУКА-2026» Том V «Секції Інженерного навчально-наукового інституту ім. Ю.М. Потебні Запорізького національного університету» : м. Запоріжжя, 20–24 квітня 2026 р. Запоріжжя, 2026. С..

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Самонов Дмитро Юрійович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ, спеціальність 121 Інженерія програмного забезпечення, освітня програма Програмне забезпечення систем, підтверджую, що написана мною кваліфікаційна робота на тему «Організація тестування Веб-застосунків засобами фреймворку Playwright» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

- згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 24.05.2026 _____ Самонов Дмитро Юрійович
(студент)

Дата 24.05.2026 _____ Коломоєць Геннадій Павлович
(керівник)