POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Natalia Chotza
Nr albumu s19322

Automatyczne testowanie aplikacji webowych w podejściu End-to-End: Przegląd technologii i studium przypadku.

Praca magisterska napisana pod kierunkiem:

Dr inż. Mariusz Trzaska

Warszawa, lipiec 2025

Streszczenie

Praca dotyczy istotnego problemu, jakim jest wybór odpowiedniego narzędzia do pisania testów automatycznych. Duża ilość rozwiązań dostępnych na rynku powoduje, że wybór optymalnego programu, który będzie w stanie zaspokoić potrzeby z jakimi zmagają się testerzy automatyczni stał się wyzwaniem. Dobrze dobrany system ma bezpośredni wpływ na efektywność pracy zespołu testerskiego oraz jakość dostarczanego oprogramowania. Wybór ten może również wpływać na koszty utrzymania testów oraz możliwość ich rozbudowy w przyszłości.

Praca ma na celu porównanie dwóch najpopularniejszych bibliotek, pod kątem łatwości implementacji, wydajości testów oraz czasu ich wykonania.

Aby zaprezentować różnice i przeanalizować co oferują oba frameworki wykonano wiele krótkich testów jak również kilka testów *End-to-End*.

Słowa kluczowe: testy automatyczne, testy *End-to-End*, aplikacja webowa, Selenium, Playwright

Spis treści

1. Wstęp

W dzisiejszych czasach na rynku istnieje niezliczona liczba aplikacji internetowych charakteryzująca się ogromną dynamiką i zróżnicowaniem dostępnych rozwiązań. Rosnąca konkurencja w tym obszarze powoduje, że coraz trudniej wyróżnić się wyłącznie innowacyjnymi funkcjonalnościami oferowanymi na stronach internetowych. Z tego też powodu rośnie znaczenie aspektów jakościowych, takich jak estetyka interfejsu, jego responsywność, niezawodność działania jak i intuicyjność obsługi.

Aby spełnić ciągle rosnące wymagania użytkowników końcowych oraz zapewnić zgodność aplikacji z postawionymi założeniami biznesowymi, niezastąpionym etapem procesu wytwarzania oprogramowania jest jego dokładne przetestowanie. Kluczową rolę odgrywają tutaj testy funkcjonalne [1], które umożliwiają sprawdzenie poprawności działania aplikacji oraz identyfikacje potencjalnych błędów jeszcze na środowisku testowym przed komercjalizacją.

W ostatnich latach coraz większą popularnością cieszą się testy automatyczne, pozwalające na przyspieszenie i usprawnienie procesu walidacji jakości oprogramowania. Szczególne miejsce wśród nich zajmują testy automatyczne typu *End-to-End* [2], które umożliwiają kompleksową symulację rzeczywistych interakcji użytkownika z systemem. Objemują one pełny przebieg danych - zaczynając od interfejsu użytkownika, poprzez warstwę logiki aplikacji, aż po komunikację z zewnętrznymi usługami i bazami danych.

Zastosowanie testów pozwala nie tylko na sprawdzenie poprawności działania poszczególnych funkcjonalności ale również pomaga w ocenie ogólnego doświadczenia użytkownika końcowego. W aplikacjach komercyjnych ma to istotne znaczenie dla utrzymania satysfakcji klienta oraz lojalności, co skutkuje wysoką pozycją na rynku. Nawet niewielkie błędy lub opóźnienia w działaniu systemu mogą spowodować niezadowolenie oraz frustracje konsumentów, a w konsekwencji, ich rezygnacji z korzystania z danej usługi.

Testy automatyczne, dzięki możliwości wielokronego odtwarzania zdefiniowanych scenariuszy oraz szybkiej identyfikacji regresji, stanowią niezbędny etap zapewnienia jakości w nowoczesnych projektach informatycznych. Ich odpowiednie zaplanowanie, wdrożenie oraz ciągła aktualizacja znacząco wpływają na stabilność i niezawodność aplikacji internetowych w warunkach rzeczywistego użytkowania.

1.1.Cel Pracy

Celem pracy jest przeprowadzenie analizy porównawczej dwóch popularnych narzędzi służących do automatyzacji testów typu *End-to-End* [2] w kontekście aplikacji webowych. Opracowanie koncentruje się na ocenie wpływu tych narzędzi na jakość oprogramowania oraz na porównaniu sposobów implementacji scenariuszy testowych z ich wykorzystaniem. W ramach części praktycznej wykonane zostaną testy na wybranych stronach internetowych, różniących się zakresem oraz charakterem dostępnych funkcjonalności. Takie podejście umożliwi ocenę elastyczności, efektywności i skuteczności obu narzędzi w odmiennych warunkach i środowiskach testowych.

Dzięki temu możliwe będzie przedstawienie realnych korzyści oraz identyfikacja potencjalnych ograniczeń wynikających z wyboru konkretnego rozwiązania. Analiza ta dostarczy wartościowych wniosków, kóre mogą stanowić wsparcie przy podejmowaniu decyzji dotyczących doboru narzędzi w projektach programistycznych ukierunkowanych na zapenienie najwyższej jakości oprogamowania.

1.2. Rozwiązania przyjęte w pracy

Bardzo ważnym aspektem pracy jest dobór aplikacji poddanym testom oraz precyzyjne opracowanie scenariuszy przypadków. Z uwagi na różnorodność funkcjonalności zdecydowano się na wykorzystanie zarówno skomercjalizowanych stron internetowych jak i aplikacji internetowych stworzonych wyłącznie w celach edukacyjncyh, służących nauce tworzenia testów automatycznych.

Językiem programowania wybranym do realizacji części praktycznej pracy jest *Java*, co wynika z wysokiego poziomu znajomości tej technologii przez autorkę pracy oraz jej szerokiego zastosowania w projektach komercyjnych i systemach testowych.

Spośród dostępnych narzędzi na rynku do automatyzacji testów *End-to-End* [2], wybrano dwa aktualnie najpopularniejsze i konkurujące ze sobą rozwiązania wspierające Javę:

- Selenium,
- Playwright.

Oba frameworki zyskały szerokie uznanie wśród specjalistów z zakresu testowania serwisów internetowych. Bezpośrednie porównanie sposobu implementacji, czasu wykonania oraz zużycia pamięci podczas wykonywanych testów, umożliwi identyfikację zarówno zalet, jak i ograniczeń wynikających z zastosowania każdego rozwiązania.

1.3. Rezultat pracy

Rezultatem pracy jest szczegółowa analiza wybranych narzędzi do wykonywania testów automatycznych oraz ich ocena z podziałem na funkcjonalności.

Opis przypadków oraz ich wyników zostały poprzedzone rozpisaniem odpowiednich scenariuszy oraz ich implementacją.

1.4. Organizacja pracy

Pracę otwiera rozdział poświęcony charakterystyce testów automatycznych, ich znaczeniu w procesie wytwarzania oprogramowania oraz podstawowym rodzajom weryfikacji stosowanym w praktyce. W dalszej części zostały szczegółowo przybliżone narzędzia poddane analizie oraz ich krótka historia na rynku międzynarodowym. W rozdziale 4 zawarto opis aplikacji poddanych testom, w celu przybliżenia czytelnikowi kontekstu.

Rozdziały od 5 do 13 zawierają szczegółowe studia przypadków przeprowadzonych skryptów. Dla każdej funkcjonalności przedstawiono:

- scenariusze testowe,
- implementację testów w obu narzędziach,
- ocenę czasu wykonania oraz stabilności automatycznych weryfikacji,
- krótkie porównanienie różnic między badanymi narzędziami.

W ostatnim rozdziale zamieszczono podsumowanie całej pracy, wskazując mocne i słabe strony analizowanych rozwiązań, a także ogólną ocenę ich przydatności w automatyzacji testów funkcjonalnych [1] i *End-to-End* [2]. Na tej podstawie sformułowano wnioski oraz zawarto wyniki w formie tabeli z punktacją dla poszczególnych frameworków. Wskazano również możliwe kierunki dalszych badań.

2. Testy automatyczne

Głównym celem testów automatycznych jest szybkie i efektywne sprawdzenie działania oprogramowania w różnych scenariuszach oraz uproszczenie trywialnych przypadków testowych pozostawiając obsługę trudniejszych funkcjonalności do testów manualnych. Dzieje się to poprzez użycie specjalnych narzędzi oraz skryptów. Zwykle automatyczna weryfikacja może powodować większe koszty podczas pierwszych faz projektowych redukując nakłady finansowe w utrzymaniu. Pomagają za to zapobiegać potencjalnym błędom, które tester mógłby pominąć.

Automatyzacja testów przydatna jest w sytuacjach :

- powtarzalnych przypadków testowych,
- testach wydajnościowych,
- testach różniących się konfiguralnością środowisk programistycznych [3].

W odróżnieniu od manualnych weryfikacji, które wymagają udziału człowieka, programowo sterowane sprawdzanie poprawności, umożliwia wielokrotne wykonanie tych samych scenariuszy w krótkim czasie. Automatyzacja według [4] pozwala:

- przyspieszyć proces testowania,
- zredukować kosztów,
- zwiększyć dokładności i powtarzalności testów.

Autorka pracy z własnych obserwacji wywnioskowała, że w dużych firmach z rozbudowanym systemem jest wręcz niemożliwe zastosowanie wyłącznie testów automatycznych w codziennej pracy, a jedynie użycie ich jako dodatek i dopełnienie testowania pewnych funkcjonalności.

Według raportu Capgemeni [5], organizacje stosujące praktyki skryptów automatycznych utrzymują się na wysokiej pozycji w zakresie niezawodności systemu oraz skróconego czasu wytwarzania oprogramowania.

3. Wykorzystane narzędzia i technologie

W tym rozdziale przedstawione zostały narzędzia do pisania skryptów testowych będące przedmiotem oceny.

3.1. Opis narzedzi użytych do testów automatycznych

Na rynku dostępnych jest wiele różnorodnych programów służących do pisania testów automatycznych. Na potrzeby niniejszej pracy przeanalizowane zostały dwie biblioteki, które należą do najpopularniejszych i najczęśniej wykorzystywanych rozwiązań w tej dziedzinie. Jednocześnie zostały wybrane z uwagi na wysoką pozycję na rynku udokumentowną w rankingu [6] oraz możliwość pisania skryptów testowych w języku *Java*.

Rozdziały 3.2 - 3.3 stanowią wprowadzenie do tematyki każdego z omawianych narzędzi, zawierając również krótką historią ich rozwoju. W tej części celowo pominięto szczegóły zasad implementacji, ponieważ zostały one dogłębnie omówione w rozdziałach 5 – 13, które porównują realizację poszczególnych testów.

3.2. Selenium

Selenium zostało stworzone w 2004 roku w Chicago przez Jasona Hugginsa, który napisał to narzędzie na własne potrzeby do testowania aplikacji, nad którą wówczas pracował. Według [7] po udostępnieniu rozwiązania współpracownikom, projekt spotkał się z pozytywnym odzewem, co poskutkowało przekształceniem go we wspólny projekt *open source* rozwijany przez międzynarodową społeczność.

Obecnie Selenium to jedno z najpopularniejszych i najczęściej wykorzystywanych bibliotek *open source* do automatyzacji testów aplikacji webowych. Istnieje i jest aktywnie rozwijane od ponad 20 lat, dzięki czemu stanowi dojrzałe, sprawdzone i szeroko stosowane rozwiązanie w branży. Jego głównym celem jest umożliwienie tworzenia testów symulujących rzeczywiste interakcje użytkownika z przeglądarką internetową.

Selenium obsługuje wiele popularnych języków programowania, takich jak:

- *Java*,
- *Python*,
- *C#*,
- *Ruby*,
- *JavaScript*

Dzięki wsparciu dla tak wielu technologii Selenium jest narzędziem elastycznym, łatwo integrującym się z różnymi środowiskami programistycznymi oraz platformami *CI/CD,* takimi jak:

- Jenkins,
- GitLab CI,
- Bamboo,
- CircleCI

Jednym z najczęściej stosowanych sposobów implementacji Selenium w języku *Java* jest integracja z frameworkiem Spring, który umożliwia zarządzanie zależnościami poprzez definiowanie komponentów jako beanów [8]. Dzięki wykorzystaniu adnotacji takich jak *@Autowired* oraz *@Inject*,

możliwe jest automatyczne wstrzykiwanie zależności, co zwiększa modularność aplikacji oraz upraszcza konfigurację środowiska testowego.

Architektura Selenium składa się z trzech głównych komponentów opisanych w rozdziałach 3.2.1 – 3.2.3.

3.2.1. Selenium WebDriver

WebDriver [9] to kluczowy komponent Selenium umożliwiający bezpośrednią komunikację z przeglądarką internetową za pomocą natywnych sterowników dostarczanych przez producentów przeglądarek. Testy tworzone z użyciem *WebDrivera* naśladują rzeczywiste interakcje użytkownika z aplikacją webową — klikają przyciski, wprowadzają dane do formularzy, nawigują po stronach oraz weryfikują elementy DOM.

WebDriver wspiera popularne przeglądarki, takie jak:

- Google Chrome,
- Mozilla Firefox,
- Microsoft Edge,
- Safari,
- oraz przeglądarki mobilne w połączeniu z narzędziami typu Appium.

Dzięki temu *WebDriver* umożliwia realizację testów niezależnych od używanego sterownika, co ma kluczowe znaczenie przy testowaniu kompatybilności aplikacji webowych.

3.2.2. Selenium IDE

Selenium Integrated Development Environment (IDE) [10] to narzędzie typu „*record and playback”*, przeznaczone głównie dla początkujących testerów oraz do szybkiego prototypowania testów. *IDE* działa jako rozszerzenie przeglądarki (dla Chrome i Firefox), umożliwiając nagrywanie interakcji użytkownika z aplikacją webową i automatyczne generowanie skryptów testowych.

Chociaż *Selenium IDE* nie oferuje takiej elastyczności i możliwości rozbudowy jak *WebDriver*, może być użyteczne w prostych przypadkach testowych, szybkich regresjach lub jako narzędzie wspierające dokumentację procesów.

3.2.3. Selenium Grid

Selenium Grid [11] to komponent umożliwiający równoległe wykonywanie testów na wielu kombinacjach przeglądarek, systemów operacyjnych i urządzeń. *Grid* umożliwia rozproszenie środowiska testowego poprzez uruchamianie testów w różnych węzłach (ang. *nodes*), które mogą być fizycznymi lub wirtualnymi maszynami w sieci lokalnej lub chmurze.

Typowy przypadek użycia *Selenium Grid* obejmuje:

- testowanie wieloprzeglądarkowe (z ang. *cross-browser testing*),
- testy równoległe (z ang. *parallel testing*),
- skrócenie czasu trwania całego procesu przy dużej liczbie przypadków testowych.

Selenium Grid może być skonfigurowane lokalnie lub wykorzystane w ramach platform chmurowych, takich jak Sauce Labs [12], BrowserStack [13] czy LambdaTest [14], co pozwala na dostęp do szerokiego wachlarza środowisk bez konieczności ich fizycznego utrzymywania.

3.3. Playwright

Playwright [15] to nowoczesne, open-source'owe narzędzie do automatyzacji testów E*nd-to-End*, rozwijane przez firmę Microsoft od 2020 roku. Jego architektura oraz bogaty zestaw funkcjonalności zostały zaprojektowane z myślą o nowoczesnych aplikacjach webowych i mobilnych, w których istotną rolę odgrywa:

- dynamiczne generowanie treści,
- interakcje asynchroniczne,
- oraz komponenty aplikacji jednostronicowej (z ang. *Single Page Application)* [16].

Dzięki wysokiej wydajności, prostocie użycia i szerokim możliwościom konfiguracji, Playwright szybko zyskał popularność wśród społeczności testerskiej i programistycznej.

Początkowo Playwright obsługiwał wyłącznie język *JavaScript*, jednak w kolejnych wersjach wsparcie rozszerzono o:

- *TypeScript,*
- *Python,*
- *C#,*
- *Java*

Dzięki temu możliwa jest integracja z różnorodnymi środowiskami oraz platformami testowymi. Najszersze wsparcie oraz dostęp do najnowszych funkcjonalności oferowane są jednak w środowisku *Node.js,* które stanowi główny ekosystem rozwoju tego narzędzia.

Playwright umożliwia pełną kontrolę nad następującymi przeglądarkami internetowymi:

- Chromium,
- Firefox,
- WebKit (co oznacza również możliwość testowania na silniku wykorzystywanym przez Safari).

3.3.1. Automatyczne oczekiwanie na stabilność elementów

Jedną z kluczowych zalet narzędzia Playwright [15] jest wbudowany mechanizm automatycznego oczekiwania na widoczność, aktywność oraz stabilność elementów *DOM* przed wykonaniem interakcji jak kliknięcie lub wprowadzenie tekstu opisana dogłębnie w [17]. Funkcjonalność ta znacząco ogranicza ryzyko błędów wynikających z problemów synchronizacji i eliminuje konieczność ręcznego stosowania opóźnień (*sleep, wait*), co przekłada się na większą stabilność oraz szybkość testów.

3.3.2. Obsługa wielu kontekstów przeglądarki

Playwright umożliwia tworzenie wielu niezależnych kontekstów przeglądarki [18] w ramach jednej sesji testowej. Każda instancja działa jako odizolowana część – z własnymi ciasteczkami, lokalną pamięcią i sesją, co pozwala na równoległe testowanie scenariuszy z udziałem wielu użytkowników lub

ról. Jest to funkcjonalność szczególnie przydatna w testowaniu aplikacji z autoryzacją, systemami uprawnień lub interakcją wielu kont użytkowników.

3.3.3. Obsługa testowania mobilnego i emulacji urządzeń

Playwright [15] oferuje wbudowane wsparcie dla emulacji urządzeń mobilnych [19], w tym rozdzielczości ekranów, dotyku, orientacji ekranu oraz nagłówków *User-Agent*. Dzięki temu możliwe jest testowanie aplikacji w warunkach zbliżonych do rzeczywistych, bez konieczności używania fizycznych urządzeń mobilnych.

3.3.4. Wbudowany framework testowy

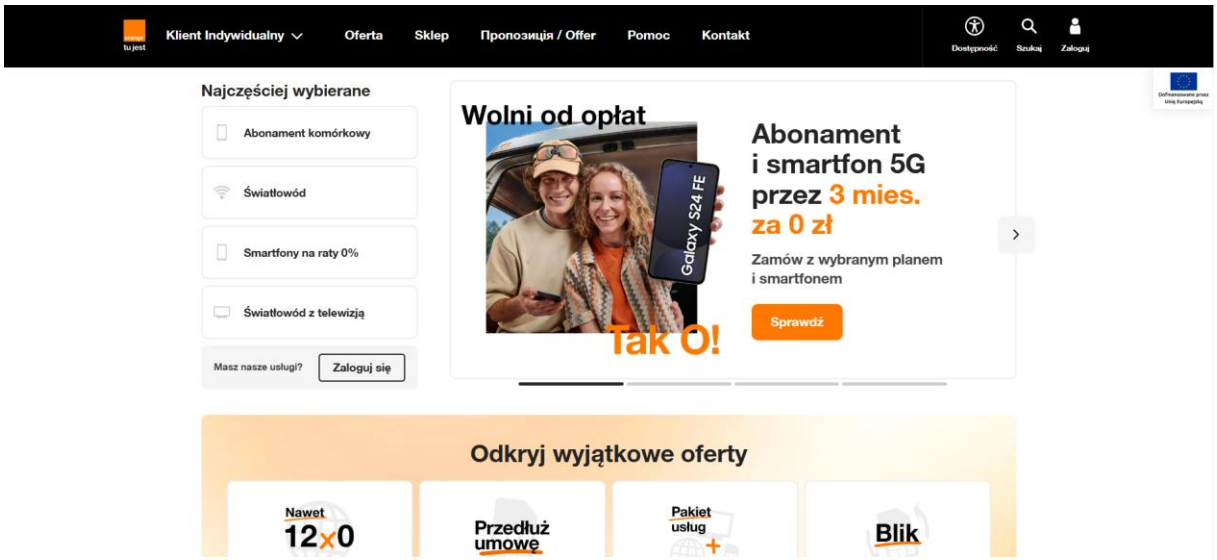
Microsoft dostarcza wraz z Playwright lekki framework testowy — *Playwright Test* [20], który umożliwia:

- asercje,
- konfigurację retry, timeoutów i równoległości,
- automatyczne generowanie raportów z testów.

4. Opis przetestowanych aplikacji webowych

W niniejszym rozdziale przedstawiono krótki opis aplikacji internetowych służących jako baza do wykonywania testów. Scharakteryzowano ich przezaczenie, zakres działania oraz pozycję na rynku. Wybór zróżnicowanych serwisów miał na celu sprawdzenie możliwości narzędzi testujących w różnych środowiskach – od komercyjnych stron, przez platformy edukacyjne, po aplikacje demonstracyjne wykorzystywane w testach automatycznych.

4.1.Orange.pl

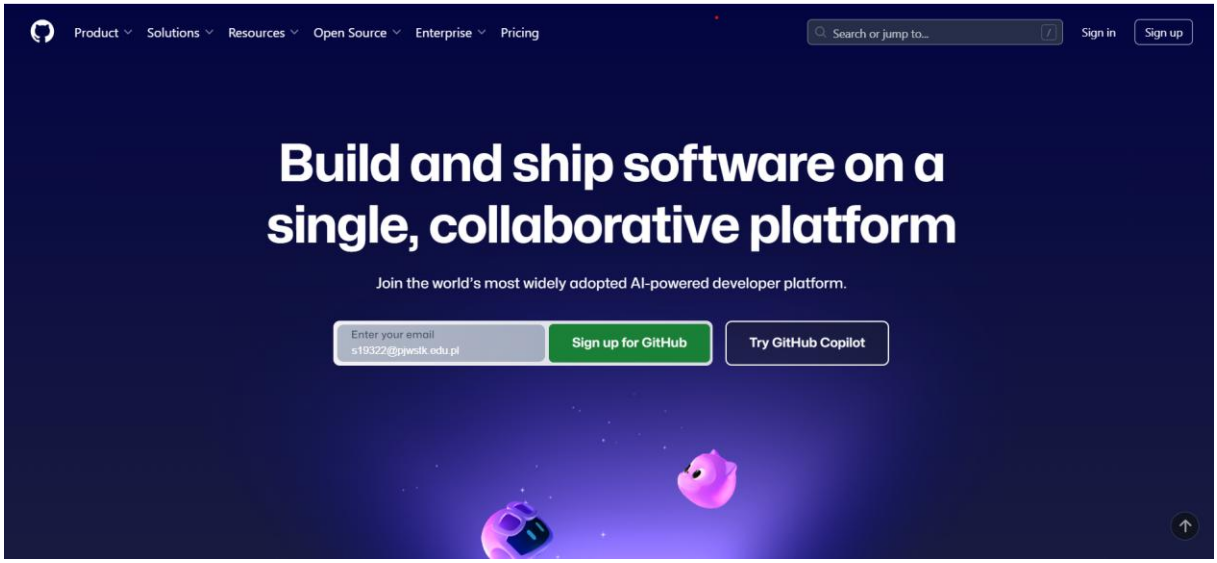


Rysunek 1 Witryna strony Orange.pl. Źródło: [21]

Rysunek 1 przedstawia witrynę oficjalnej strony internetowej jednego z największych operatorów sieci komórkowych i dostawców usług telekomunikacyjnych w Polsce. Serwis umożliwia klientom dostęp do informacji o ofercie oraz zamawianie usług online.

Decyzja o uwzględnieniu tej aplikacji została podjęta ze względu na miejsce pracy autorki oraz jej znajomość architektury systemu i procesów komunikacji między serwerami. W przeszłości elementy te niejednokrotnie stanowiły wyzwania zarówno dla zespołów deweloperskich, jak i testerów, co czyni tę aplikację nietrywialnym przypadkiem do analizy w kontekście testów automatycznych.

4.2.Github.com



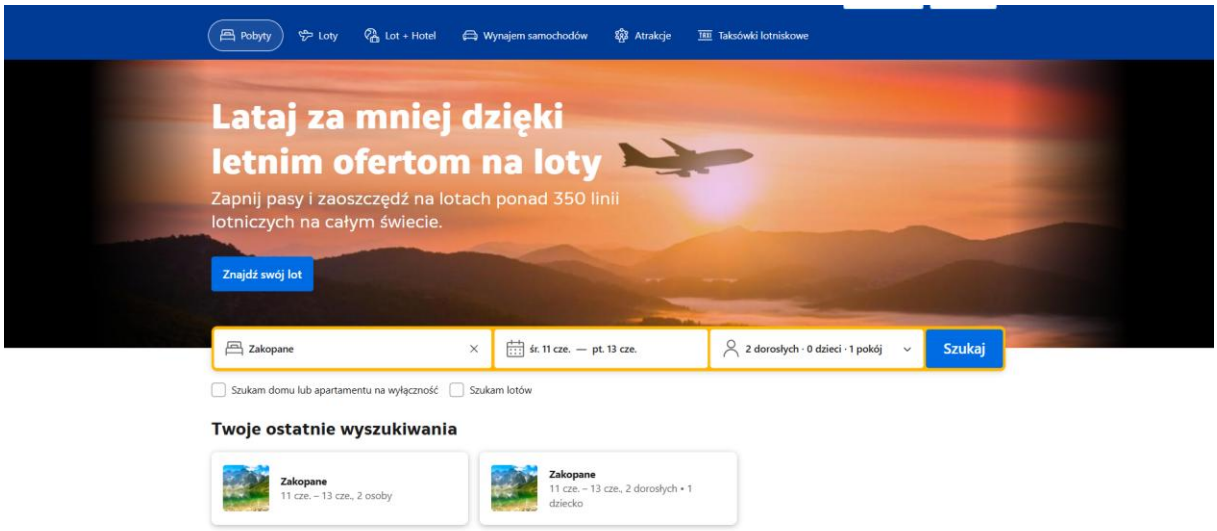
Rysunek 2 Witryna strony Github.com. Źródło: [22].

GitHub.com jest platformą internetową służącą do hostowania i zarządzania projektami programistycznymi wykorzystującymi system kontroli wersji *Git*. Umożliwia programistom wspólną pracę nad kodem źródłowym oraz współdzielenie projektów w sposób zorganizowany i transparentny.

Stronę wybrano ze względu na różne możliwości logowania. Rysunek 2 przedstawia witrynę strony github.com, na której można zalogować się poprzez m.in:

- podanie loginu i hasła,
- logowanie przez konto Google lub Apple

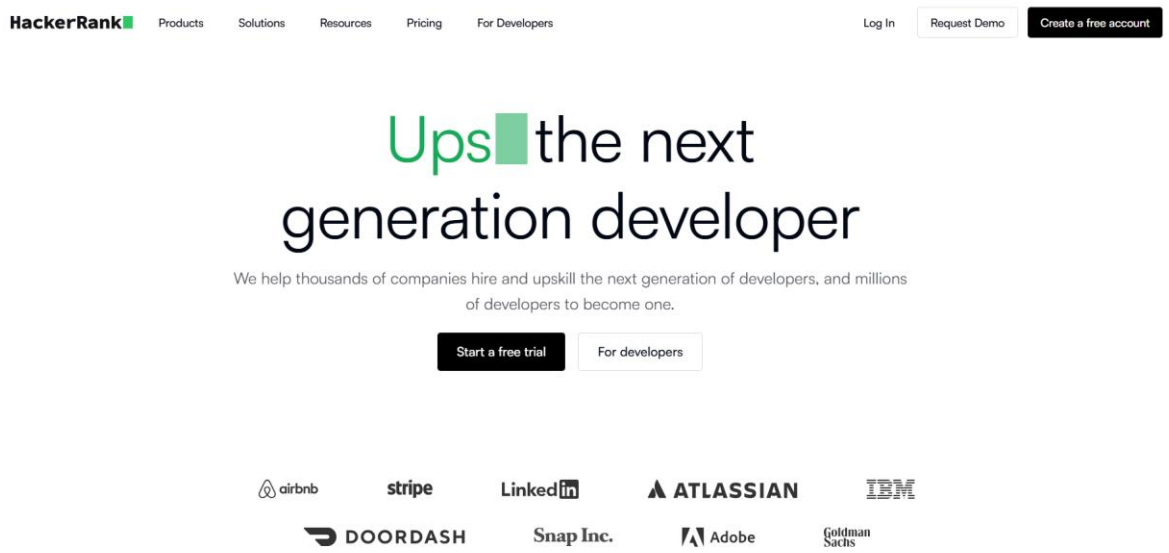
4.3.Booking.com



Rysunek 3 Witryna strony Booking.com. Źródło: [23].

Rysunek 3 prezentuje platformę międzynarodową służącą do rezerwacji noclegów, lotów oraz usług turystycznych. Serwis charakteryzuje się statyczną strukturą stron, gdzie treści są ładowane głównie podczas przechodzenia pomiędzy podstronami. Dzięki temu Booking.com stanowi dobre środowisko do testowania podstawowych funkcjonalności aplikacji webowych, takich jak wyszukiwanie, filtrowanie oraz weryfikacja formularzy rezerwacyjnych. Wybór strony pozwala na sprawdzenie działania narzędzi testujących w warunkach typowych dla dużych, komercyjnych serwisów internetowych.

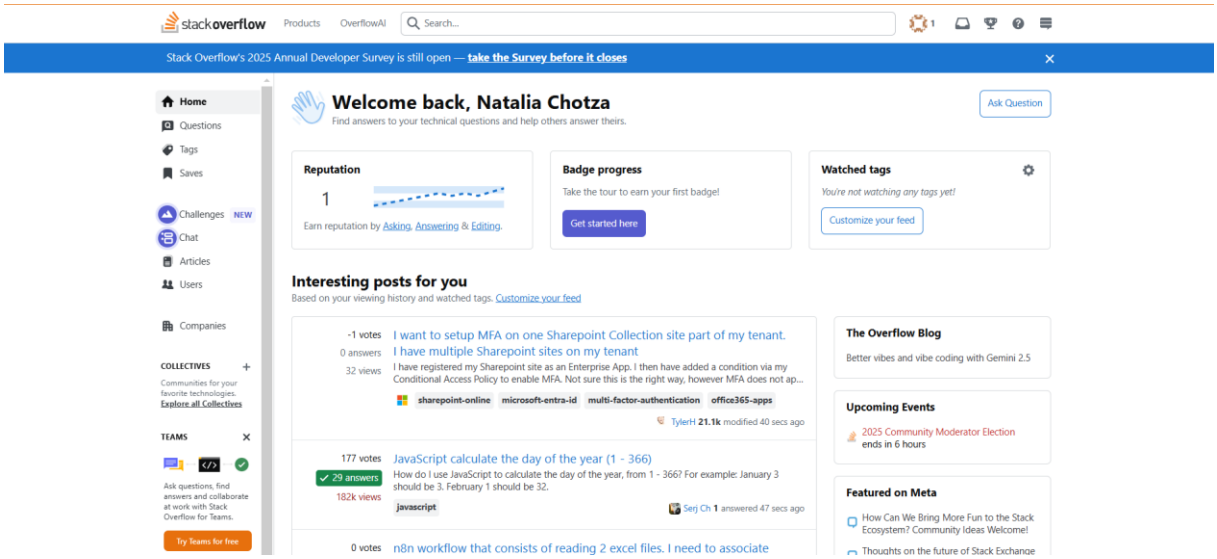
4.4.HackerRank.com



Rysunek 4 Witryna strony HackerRank.com. Źródło: [24].

Popularna platforma internetowa umożliwiająca programistom rozwiązywanie zadań algorytmicznych i uczestniczenie w konkursach programistycznych. Rysunek 4 przedstawia serwis wykorzystywany również przez firmy rekrutacyjne do weryfikacji umiejętności kandydatów.

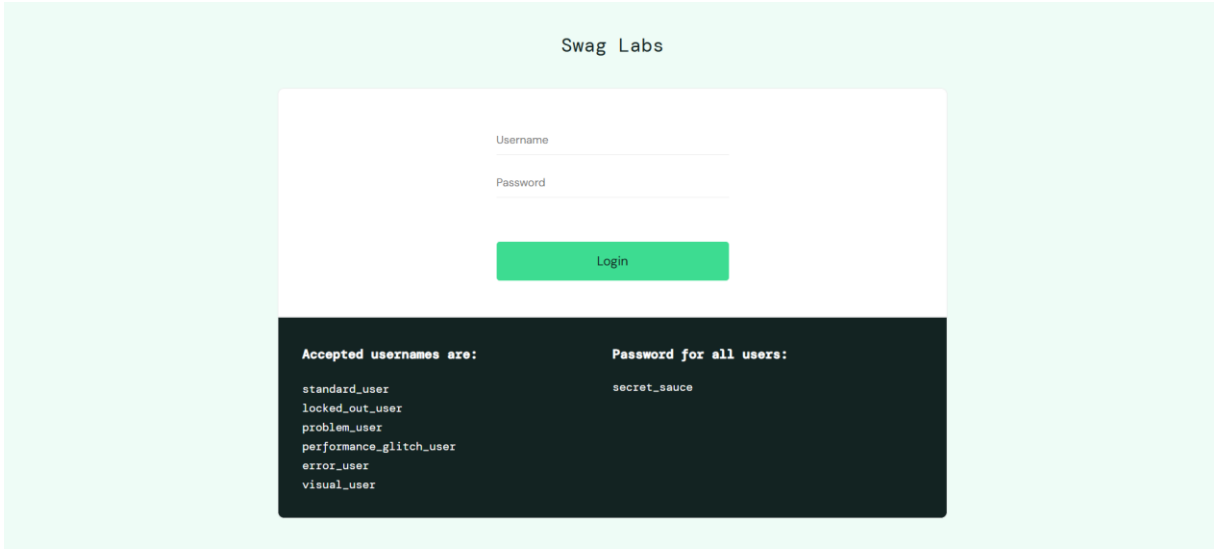
4.5.StackOverFlow.com



Rysunek 5 Witryna strony StackOverFlow.com. Źródło: [25].

Rysunek 5 prezentuje stronę StackOverFlow, jedną z największych społeczności programistycznych online, pozwalającą na zadawanie i odpowiadanie na pytania z zakresu kodowania i technologii informacyjnych. Posiada funkcjonalność logowania do konta użytkownika.

4.6.SauceDemo.com



Rysunek 6 Witryna strony SauceDemo.com. Źródło: [26].

Przykładowa aplikacja demonstracyjna przygotowana z myślą o testowaniu automatycznym. Udostępnia proste funkcjonalności sklepu internetowego, co pozwala na szybkie i kontrolowane testy funkcji logowania, dodawania produktów do koszyka oraz finalizacji zamówienia. Rysunek 6 prezentuje stronę główną platformy SauceDemo.

4.7. The-internet-herokuapp.com

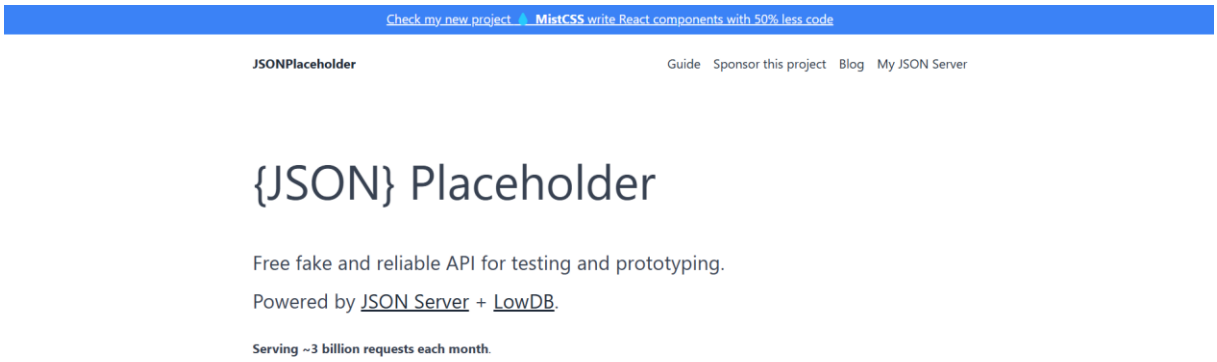
Rysunek 7 Witryna strony The-internet-herokuapp.com. Źródło: [27].

Rysunek 7 przedstawia witrynę aplikacji demonstracyjnej TheInternet, zawierającą różnego rodzaju komponenty webowe wykorzystywane w testach automatycznych, takie jak:

- alerty,
- przyciski,
- formularze,
- dynamiczne elementy,
- czy tabele.

Stanowi wygodne środowisko do sprawdzania działania narzędzi testujących w różnych scenariuszach.

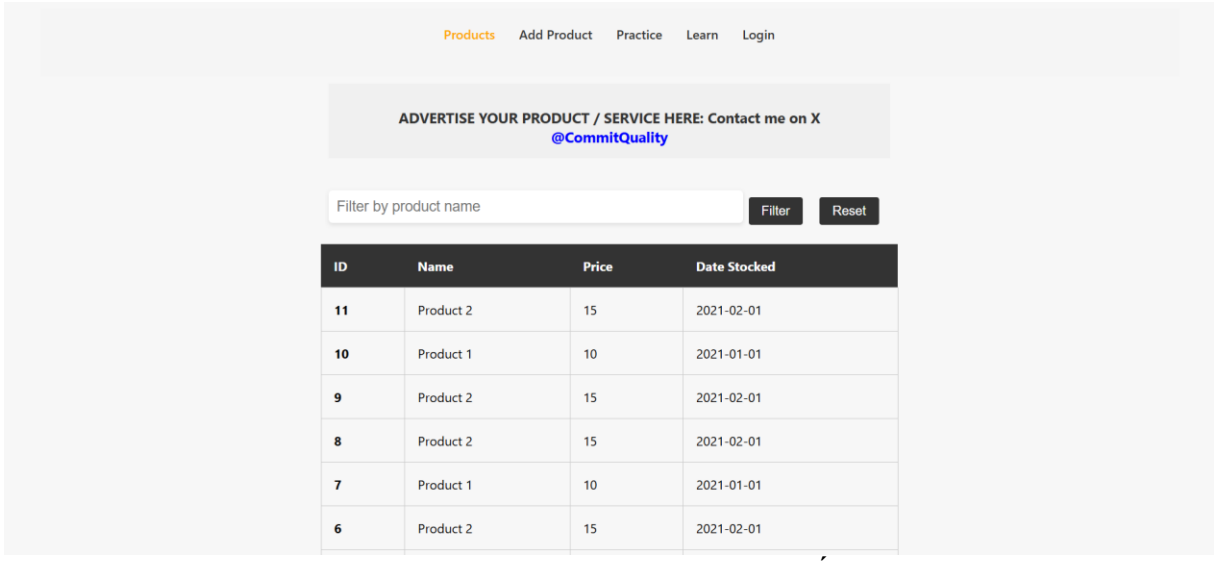
4.8.JsonPlaceholder.typecode.com



Rysunek 8 Witryna strony JsonPlaceholder.typecode.com. Źródło: [28].

Aplikacja udostępniająca darmowe odpowiedzi dla testowania zapytań typu *REST API,* które zostało opisane bardziej szczegółowo w rozdziale 6. Rysunek 8 prezentuje witrynę strony JsonPlaceHolder.

4.9.CommitQuality.com



Rysunek 9 Witryna strony CommitQuality.com. Źródło: [29].

Rysunek 9 przedstawia ogólnodostępną aplikację internetową stworzoną przez testera dla społeczności testerskiej do możliwości nauki pisania testów automatycznych. Serwis webowy posiada wiele funkcjonalności jak np.

- Pola wyboru,
- Formularze,
- Itp.

5. Testy z wbudowanym frameworkiem testowym w Playwright

W tym rozdziale przedstawione zostały dwa sposoby sprawdzania poprawności wykonywania testów w Playwright [15] przy użyciu funkcjonalności opisanej w rozdziale 3.3.4 oraz stosując bibliotekę Junit5 [30]. Z uwagi na brak wbudowanego frameworka do testowania w bibliotece Selenium [8] narzędzie zostało pominięte w porównaniu.

Testy przeprowadzono na stronie Sauce Demo opisanej w rozdziale 4.6. Na końcu przeanalizowano czasy wykonywania badanych przypadków i krótko podsumowano wyniki.

W dalszych etapach pracy autorka zdecydowała się na wykorzystanie biblioteki Junit5 jako podstawowego narzędzia testowego, niezależnie od opisanych wyników opisanych, aby zapewnić jak największą porównywalność i spójność testów.

5.1.Scenariusze Testów

Opis scenariusza testowego dla składania zamówienia na stronie internetowej Sauce Demo.

Tabela 1 Scenariusz dla testu z wbudowanym frameworkiem testowym w Playwright. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna	Wprowadź adres https://www.saucedemo.com/ do przeglądarki i przejdź do strony głównej.
2.	Logowanie	W odpowiednie pola wpisz testową nazwę użytkownika: *standard_user* oraz testowe hasło *secret_sauce.* Zaloguj się na stonę wciskając przycik: *Login.*
3.	Strona z produktami	Wybierz przycisk dodawania do koszyka dedykowanego: • dla lampki rowerowej, • dla czerwonej bluzki. W prawym górnym rogu kliknij w ikonę koszyka. Zweryfikuj czy na stronie produktów dodanych do koszyka istenieją poprawne nazwy.
4.	Strona składania zamówienia	Przejdź do strony składania zamówienia. W odpowiednie miejsca wpisz dane potrzebne do złożenia zamówienia: *firstName – testName* *lastName - testLastName* *postalCode -03-333* Naciśnij przycisk *Continue.*

5.	Strona podsumowania	Zweryfikuj: • czy istnieje sekcja podsumowania, • czy sekcja płatności posiada tytuł *„Payment Information”,* informacje o karcie oraz tytuł „*Price Total”,* • czy dostawa posiada tytuł *„Shipping Information”* oraz informacje o sposobie dostawy, Naciśnij przycisk *Finish*
6.	Strona końcowa	Zweryfikuj czy na stronie końcowej pojawia się tytuł : „Thank you for your order!”

5.2. Asercje przy wykorzystaniu Junit5

Junit5 [30] jest popularnym frameworkiem pozwalającym programiście pisać testy jednostkowe, które automatycznie sprawdzają poprawność działania kodu w aplikacjach bazujących na Wirtualnej Maszynie Javy.

Aby umożliwić korzystanie z mechanizmów testowania przy użyciu asercji oferowanych przez bibliotekę Junit5, konieczne było uprzednie dodanie odpowiednich zależności (z ang. *dependencies*) do pliku *pom.xml*. Jest to podstawowy plik konfiguracyjny wykorzystywany przez system budowania Maven [31], odpowiedzialny za zarządzanie bibliotekami zewnętrznymi oraz konfigurację projektu.

Listing 1 Zależność JUnit 5

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.11.3</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.11.3</version>
  <scope>test</scope>
</dependency>
```

Podstawowym mechanizmem wykorzystywanym podczas tworzenia testów jednostkowych z użyciem frameworka Junit5 są asercje, które służą do sprawdzania, czy określony warunek został spełniony. W przypadku, gdy dany warunek nie zostanie potwierdzony, test automatycznie kończy się niepowodzeniem. Listing 1 zawiera zależność Junit5.

W zaprezentowanym teście zastosowano dwie różne konstrukcje asercji. Pierwszym przykładem jest metoda `Assertions.assertEquals()`, której zadaniem jest porównanie dwóch argumentów, wartości oczekiwanej oraz wartości rzeczywistej, wyliczonej podczas wykonywania kodu. Listing 2, przedstawia porównanie liczby całkowitej przekazanej jako obiekt typu `String`, z tekstem wyświetlanym nad ikoną koszyka. Celem tego sprawdzenia było potwierdzenie, czy system poprawnie oznacza liczbę dodanych elementów do podsumowania.

Listing 2 *AssertEquals* z biblioteki Junit.

```
Assertions.assertEquals("2",sauceDemo.getShoppingCartBadge().textContent();
```

Listing 3, korzysta z metody `isVisible()` z klasy `Locator` [32] z biblioteki Playwright w celu weryfikacji czy lista produktów jest widoczna na stronie. Funkcja zwraca wartość logiczną typu `Boolean`, która przyjmuje jeden z dwóch możliwych parametrów: `true` lub `false`. Do sprawdzania tego typu warunków w testach jednostkowych wykorzystuje się metodę `Assertions.assertTrue()`. Jej zadaniem jest potwierdzenie, że przekazany warunek logiczny jest spełniony. W przypadku, gdy `isVisible()` zwróci wartość `false`, test automatycznie zakończy się niepowodzeniem, sygnalizując tym samym niezgodność stanu aplikacji z założeniami testowymi.

Listing 3 *AssertTrue* z bibliotek JUnit z użyciem metody *isVisible()*.

```
Assertions.assertTrue(cartPage.getCartList().isVisible());
```

Listing 4 również wykorzystuje metodę `Assertions.assertTrue` co Listing 3, jednak w tym przypadku sprawdzono, czy element sekcji podsumowania opłat zawiera odpowiedni tytuł. Weryfikacja została zrealizowana przy użyciu metody `contains()` z klasy `String`, która zwraca wartość logiczną typu `Boolean`, przyjmującą parametr `true` w przypadku odnalezienia wskazanego ciągu znaków, bądź `false`, gdy podany tekst nie występuje w analizowanym napisie.

Następnie, aby ocenić rezultat tego sprawdzenia zastosowano `Assertions.assertTrue()`. W sytuacji, gdy metoda `contains()` zwróci `false`, test zakończy się niepowodzeniem, sygnalizując brak oczekiwanego tytułu w elemencie podsumowania.

Listing 4 *AssertTrue* z biblioteki JUnit z użyciem metody *contains()*.

```
Assertions.assertTrue(summaryPage.getPriceTotalValue()
.contains("Price Total"));
```

5.3.Asercje przy wykorzystaniu wbudowanego frameworka testowego

Konstrukcja każdej asercji w tym rozwiązaniu opiera się na metodzie `assertThat()`, do której jako argument przekazywany jest kod odpowiedzialny za wykonanie sprawdzenia. Funkcja ta zwraca obiekt typu `LocatorAssertions` [33] z pakietu `com.microsoft.playwright.assertions`, na którym można wywołać jedną z wielu dostępnych strategii asercyjnych. Każda z tych metod posiada w swojej implementacji domyślny mechanizm oczekiwania na spełnienie warunku, co pozwala uwzględnić ewentualne opóźnienia w widoczności elementów na stronie. Po zakończeniu weryfikacji zwracana jest wartość typu `Boolean`, wskazująca, czy dane założenie zostało spełnione.

Listing 5, przedstawia zastosowanie metody `hasText()`, której zadaniem było porównanie liczby całkowitej przekazanej jako obiekt typu `String` z wartością tekstową wyświetlaną nad ikoną koszyka. W przypadku, gdy numer nad badanym elementem nie będzie zgodny z wartością przekazaną w argumencie, test jednostkowy zostanie zakończony niepowodzeniem.

Listing 5 Użycie metody *hasText()*, na obiekcie *LocatorAssertions*.

```
assertThat(sauceDemo.getShoppingCartBadge()).hasText("2");
```

Listing 6, przedstawia użycie `isVisible()` na obiekcie typu `LocatorAssertions`, której zadaniem jest sprawdzenie, czy lista wybranych produktów jest widoczna na stronie.

Listing 6 Użycie metody *isVisible()* na obiekcie *LocatorAssertions*.

```
assertThat(cartPage.getCartList()).isVisible();
```

Jeśli element reprezentujący spis produktów nie będzie dostępny w momencie wykonywania testu, asercja zakończy się niepowodzeniem, informując o niespełnieniu oczekiwanego warunku.

Listing 7, zawiera implementację metody `containsText()`. Umożliwia ona sprawdzenie, czy tekst zawarty w obiekcie wstrzykniętym do `assertThat()` posiada oczekiwany ciąg znaków. Porównuje zawartość tekstowego elementu z podanym w argumencie parametrem i zwraca wynik w postaci zmiennej typu logicznego – *true/false.*

Listing 7 Użycie metody *containsText()* na obiekcie *LocatorAssertions*.

```
assertThat(summaryPage.getPriceTotalValueLocator())
.containsText("Price Total");
```

W sytuacji, gdy tekst kontrolowanego elementu nie będzie składał się z podanego ciągu znaków, test zakończy się niepowodzeniem, sygnalizując niespełnienie założonego warunku.

5.4.Analiza Porównawcza

Porównując implementację obu rozwiązań dla scenariusza testowego składania zamówienia (Tabela 1), autorka pracy dostrzegła większą łatwość w korzystaniu z konstrukcji oferowanych przez wbudowany w Playwright framework testowy [21]. Sposób definiowania asercji okazał się bardziej intuicyjny i przejrzysty w porównaniu do rozwiązań dostępnych w bibliotece Junit5 [30]. Kod napisany z wykorzystaniem natywnego frameworka jest czytelniejszy, a jego analiza pozwala już przy pobieżnym spojrzeniu zrozumieć intencję autora, bez konieczności szczegółowego zapoznawania się z implementacją.

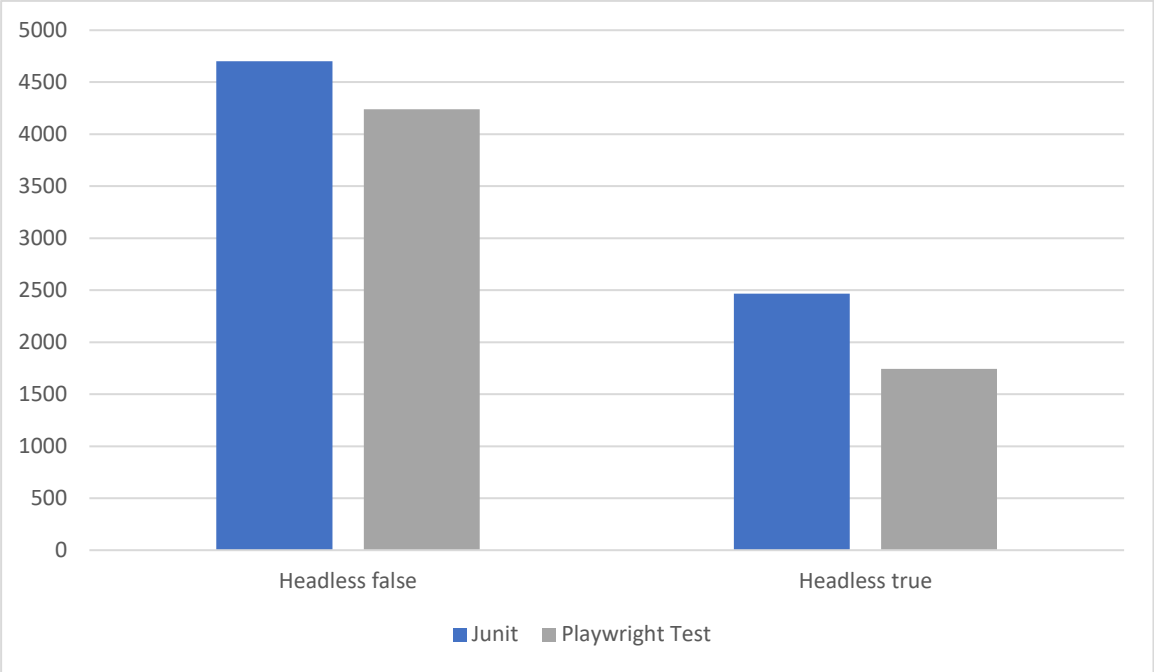
Warto jednak zaznaczyć, że klasa `LocatorAssertions` [33] nie oferuje tak szerokiego zakresu funkcji jak Junit5 i nie umożliwia na przykład bezpośredniego porównywania dwóch obiektów. Tego typu operacje są możliwe do zrealizowania za pomocą metody `Assertions.assertEquals()` dostępnej w bibliotece Junit5, która zapewnia większą elastyczność w definiowaniu warunków testowych i obsłudze bardziej złożonych przypadków.

Testy wykonano w dwóch trybach z włączonym - w przypadku ustawienia parametru *headless* na *false* oraz wyłączonym interfejsem graficznym - parametr *headless* ma wartość *true*

Analiza wyników dotyczących czasów wykonania poszczególnych testów (Wykres 1) oraz analiza przyrostu zużycia pamięci *RAM*, który utrzymywał się na poziomie około 1 MB, wykazała bardzo niewielkie różnice pomiędzy użytymi rozwiązaniami. Jednakże krótszy okres wykonywania testów świadczy o wyższej wydajności wbudowanego frameworka w bibliotece Playwright. Ponadto,

- szeroki zakres dostępnych konstrukcji,
- brak konieczności importowania zewnętrznych narzędzi,
- oraz brak dodatkowych zależności w pliku konfiguracyjnym

czyni go rozwiązaniem przewyższającym nad JUnit.



Wykres 1 Porównanie czasu wykonania w milisekundach z wbudowanym frameworkiem Playwright, a Junit (mniej=lepiej). Źródło: opracowanie własne.

6. Testy obsługi podstawowych zapytań REST API

W tym rozdziale omówiono sposoby implementacji zapytań *REST API* [34] z wykorzystaniem Selenium oraz Playwright. Rozdział rozpoczyna się od przedstawienia jednego scenariusza testowego (Tabela 2), który w tym przypadku wygląda tak samo dla każdej metody *HTTP*, a kończy analizą porównawczą efektywności każdego z obu narzędzi wraz z porównaniem czasowym badań.

REST (*Representational State Transfer*) to styl architektury, który wykorzystuje protokół *HTTP* do komunikacji pomiędzy klientem a serwerem. *REST API* udostępnia zestaw punktów końcowych (tzw. endpointów), umożliwiających wykonywanie operacji na zasobach — takich jak pobieranie, tworzenie, aktualizowanie czy usuwanie danych.

Do implementacji wykorzystano stronę testową JsonPlaceHolder.com (rozdział 4.8).

Pod koniec rozdziału zawarto implementację graficzną średnich czasów wykonania z podziałem na metody i narzędzia testowane.

6.1. Scenariusz testowy

Opis scenariusza testowego dla zapytań typu *REST API*.

Tabela 2 Scenariusz testowy *REST API* metoda *GET/PUT/POST/DELETE*. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Metoda GET/PUT/POST/DELETE	Zweryfikuj, że odpowiedź ma status 200

6.2. Implementacja w Selenium z użyciem RestAssured

W środowisku testów automatycznych Selenium nie ma wbudowanych mechanizmów obsługi zapytań *REST API* [34]. W związku z powyższym, w celu realizacji testów konieczne było wykorzystanie dodatkowej biblioteki RestAssured [35]. Ułatwia ona wysyłanie zapytań typu *REST* w języku *Java* poprzez zestaw wbudowanych metod.

W pierwszym etapie niezbędne było rozszerzenie pliku konfiguracyjnego *pom.xml*. Listing 7 zawiera stosowną zależność, umożliwiającą integrację `RestAssured` z projektem opartym na systemie budowania Maven [31].

Listing 8 Dodanie biblioteki *RestAssured* do *pom.xml* w Selenium

```
<dependency>
            <groupId>io.rest-assured</groupId>
            <artifactId>rest-assured</artifactId>
            <version>5.5.1</version>
            <scope>test</scope>
</dependency>
```

W celu ułatwienia wykonywania zapytań *REST API*, zastosowano mechanizm inicjalizacji w metodzie oznaczonej `@BeforeAll`. Funkcja z tą annotacją wykonywana jest jednokrotnie przed

rozpoczęciem wszystkich testów w danej klasie.W jej obrębie nadpisano domyślny adres *URL*, który wykorzystywany będzie we wszystkich zapytaniach realizowanych przez bibliotekę `RestAssured`.

Listing 9 Nadpisanie domyślnego adresu *URL* w Selenium

```
RestAssured.baseURI = "https://jsonplaceholder.typicode.com";
```

Aby Listing 9 był możliwy do wykonania należało określić cykl życia instancji klasy testowej, co umożliwia lepsze zarządzanie jej stanem w trakcie wykonywania testów. W tym celu wykorzystano adnotację `@TestInstance` [36] z odpowiednim parametrem, umożliwiającym utworzenie jednego obiektu współdzielonego przez wszystkie metody testowe.

Listing 10 Annotacja do testu *RestAPI* w Selenium

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
```

Po skończonej konfiguracji rozpoczęto implementację kodu odpowiedzialnego za obsługę zapytań *REST API.* Listing 11 zawiera przykład zrealizowania metody typu *GET,* której celem było pobranie danych z serwera.

Listing 11 Implementacja metody *GET* w Selenium

```
Response response = given()
        .when()
        .get("/posts/1")
        .then()
        .statusCode(200)
        .extract()
        .response();
```

Proces realizacji zapytania został podzielony na kilka logicznych etapów:

- `given()` – metoda wbudowana w `RestAssured` umożliwiająca opcjonalne skonfigurowanie parametrów wykorzystywanych do wywołania zapytania, nagłówków lub danych przesyłanych w ciele żądania. W prezentowanym przykładzie funkcja została wywołana bez parametrów konfiguracyjnych, co oznacza wysłanie żądania z domyślnymi ustawieniami,

- `get()` – procedura odpowiedzialna za wysyłania zapytania *HTTP* typu *GET.* Przyjmuje jako argument końcowy fragment ścieżki zasobu, który zostaje dołączony do wcześniej zdefiniowanego adresu bazowego. W efekcie tworzone jest kompletne wywołanie pod wskazany *endpoint*,

- `then()` – wywołanie służące definiowaniu asercji oraz warunków weryfikujących poprawność otrzymanej odpowiedzi. W połączeniu z metodą `statusCode(200)` umożliwia sprawdzenie, czy odpowiedź serwera zawiera kod statusu *HTTP* równy 200, co oznacza poprawne przetworzenie zapytania,

- `extract().response()` – zestaw metod umożliwiających zapisanie pełnej odpowiedzi serwera w postaci obiektu klasy `Response`.

Po zapisaniu danych zwrotnych w postaci klasy `Response` możliwa jest szczegółowa analiza otrzymanych informacji, w tym sprawdzenie wartości statusu, odczytanie nagłówków czy wyodrębnienie danych z odpowiedzi w formacie *Json*.

Dzięki konstrukcji `statusCode(200)`, jeśli serwer zwróciłby inny kod statusu niż oczekiwany, test automatycznie zakończyłby się niepowodzeniem już na etapie realizacji żądania, bez konieczności stosowania dodatkowej zewnętrznej asercji w kodzie testowym.

W celu realizacji zapytania *HTTP* typu *POST*, służącego do wysyłania danych do wskazanego zasobu aby je zmodyfikować lub dodać jako nowy obiekt, zastosowano analogiczny schemat budowania żądania jak w przypadku metody *GET*, z wykorzystaniem metody `given()`. Listing 12 zawiera dalszy zestaw funkcji oraz konfigurację ulegającą rozszerzeniu ze względu na konieczność przesłania danych w ciele zapytania oraz ustawienia odpowiednich nagłówków.

Listing 12 Implementacja metody *POST* w Selenium

```
Response response = given()
        .header("Content-Type", "application/json")
        .body("{ \"title\": \"Test Post\", \"body\": \"This is a
         test\", \"userId\": 1 }")
        .post("https://jsonplaceholder.typicode.com/posts");
```

Proces budowania i wysyłania operacji typu *POST,* został podzielony na kilka etapów:

- `header()` – metoda służąca do budowania nagłówków w zapytaniach *HTTP*. Przesłana tu została informacja o formacie danych jakie będą wysyłane do serwera,
- `body()` – funkcja umożliwiająca dodanie danych do ciała żądania. W przykładzie przekazano obiekt tekstowy reprezentujący strukturę *Json,*
- `post()` – realizuje *HTTP POST* na wskazany adres *URL*.

Metody *POST* używa się również do przesyłania danych w formacie, który jest wykorzystywany tradycyjnie do wysyłania formularzy w aplikacjach webowych oraz w prostych zapytaniach wymagających przekazania par klucz–wartość w ciele wywołania.

Listing 13 Implementacja z danych w formacie formularza w Selenium

```
Response response = given()
  .contentType(ContentType.URLENC.withCharset("UTF-8"))
  .formParam("firstName", "Natalia")
  .formParam("lastName", "Natalia")
  .post("https://postman-echo.com/post");
```

- `contentType(ContentType.URLENC.withCharset("UTF-8"))` — metoda określająca nagłówek *Content-Type* jako informację na temat formatu danych przesyłanych w żądaniu. W przykładzie wybrano *format application/x-www-form-urlencoded* przyjmujący postać par klucz–wartość, kodowanych w standardzie *URL*. Jednakże można zamienić go na `ContentType.JSON` co nadaje elastyczność rozwiązania,
- `formParam()` — dodaje parametr formularza w formie pary klucz–wartość do ciała żądania.

Kolejnym ważnym typem zapytania *HTTP,* wykorzystywanym w procesie testowania usług *REST API,* jest metoda *PUT.* Służy ona do aktualizacji zasobu znajdującego się pod wskazanym adresem *URL*. Pod względem implementacji jest zbliżona do metody *POST*, jednak znaczącą różnicą stanowi wywołanie metody `put()` zamiast `post()`, a także obowiązek przekazania pełnej, zaktualizowanej reprezentacji zasobu w ciele żądania.(Listing 14)

Listing 14 Implementacja metody *PUT* w Selenium

```
Response response = given()
        .header("Content-Type", "application/json")
        .body("{ \"id\": 1, \"title\": \"Updated Post\", \"body\":
      \"Updated content\", \"userId\": 1 }")
        .put("https://jsonplaceholder.typicode.com/posts/1");
```

W celu usunięcia odpowiedniego zasobu na serwerze z wykorzytsaniem protokołu *HTTP* wykorzystuje się metodę *DELETE*. Do realizacji implementacji metody *DELETE* użyto metody z klasy `Response`, która potrzebuje podanie dokładnego adresu obiektu (Listing 15)

Listing 15 Implementacja metody *DELETE* w Selenium

```
Response response = delete("https://jsonplaceholder.typicode.com/posts/1");
```

Na końcowym etapie testów, po wykonaniu żądań *HTTP*, kluczowym elementem była weryfikacja, czy odpowiedź zwrócona przez serwer posiadała oczekiwany kod statusu *HTTP*. Listing 16 wykorzystuje do tego strukturę asercji udostępnianą przez framework JUnit [30].

Listing 16 Implementacja weryfkacji poprawnego zakończenia testów Selenium

```
assertEquals(200, response.statusCode());
```

6.3.Implementacja w Playwright

Playwright posiada wbudowany interfejs `APIRequest`, który umożliwia testowanie wysyłania zapytań typu *REST API.* Aby z niego korzystać należy najpierw na jego obiekcie utworzyć nowy kontekst aplikacji `APIRequestContext` [37]. Posłuży on dalej do wywoływania zapytań. Każda metoda *REST API* ma swoją dedykowaną metodę w interfejsie `APIRequestContext`.

Aby zaimplementować metodę *GET* używaną do pobierania informacji z serwera autorka pracy użyła gotowej metody `get(String url)`, dostając przy tym odpowiedź w formacie *Json*. (Listing 17)

Listing 17 Implementacja metody *GET* w Playwright

```
APIResponse response =
      request.get("https://jsonplaceholder.typicode.com/posts/1");
```

Listing 18, implementuje wbudowaną w Playwright metodę `post(String url, RequestOptions options)`, która służy do wysyłania żądań *HTTP POST* na wskazany adres *URL*.

Listing 18 Implementacja metody *POST* w Playwright

```
APIResponse response =
request.post("https://jsonplaceholder.typicode.com/posts",
      RequestOptions.create()
      .setHeader("Content-Type", "application/json")
      .setData("{ \"title\": \"Test Post\", \"body\": \"This is a test\",
\"userId\": 1 }"));
```

- `RequestOptions.create()` - metoda służąca do tworzenia instancji nowego obiektu implementujacego interfejs `RequestOptions`,

- `setHeader()` – ustawia nagłówek *HTTP*, w tym przypadku informujący o tym, że wysyłane dane będą w formacie *Json*,

- `setData()` - pozwala na przesłanie informacji jako obiekt *Json*, który zawiera treść żądania (tytuł, treść posta i identyfikator użytkownika),

Różnica pomiędzy operacjami *HTTP PUT* a *HTTP POST* sprowadza się głównie do zastosowania odpowiedniej metody, przy czym struktura zapytania oraz konfiguracja parametrów pozostają niemal identyczne (Listing 19).

Listing 19 Implementacja metody *PUT* w Playwright

```
APIResponse response =
request.put("https://jsonplaceholder.typicode.com/posts/1",
RequestOptions.create()
            .setHeader("Content-Type","application/json"
            .setData("{ \"id\": 1, \"title\": \"Updated Post\", \"body\":
      \"Updated content\", \"userId\": 1 }"));
```

W niektórych starszych aplikacjach dane nadal przesyłane są w formacie formularza, gdzie informacje trafiają w ciele żądania jako pary klucz–wartość, zamiast w popularnym obecnie formacie *Json*. W przypadku testowania takich rozwiązań, Playwright umożliwia wysyłanie *HTTP POST* z danymi formularza przy pomocy metody `post()` z interfejsu `APIRequestContext` [37] *(*Listing 20*)*

Listing 20 Implementacja metody *POST* z danymi w formacie formularza w Playwright

```
APIResponse response = request.post("https://postman-echo.com/post",
        RequestOptions.create().setForm(
                FormData.create()
                        .set("firstName", "Natalia")
                        .set("lastName", "Natalia")
        ));
```

- `RequestOptions.create().setForm()` – tworzy nowe opcje żadania dla wysyłania danych w formacie formularza,
- `FormData().create()` – tworzy obiekt implementujący interfejs `FormData`.

W celu weryfikacji innego formatu należy użyć metody `setData()` z klasy `RequestOptions`, do której można podać np. format *Json*. Listing 21 implementuje mapę zawierającą dane w postaci klucz wartość. Kolekcja następnie jest przekazana do funkcji wysyłającej zapytanie.

Listing 21 Implementacja przesyłania formatu *Json* w Playwright przy użyciu metody *POST*

```
Map<String, Object> jsonData = new HashMap<>();
jsonData.put("firstName", "Natalia");
jsonData.put("lastName", "Natalia");

APIResponse response = request.post("https://postman-echo.com/post",
       RequestOptions.create()
                     .setData(jsonData));
```

Z kolei do zaimplementowania *HTTP DELETE* przy użyciu interfejsu `ApiRequestContext`, użyto dedykowanej do tego metody, z przekazanym *URL* jako parametr (Listing 22).

Listing 22 Implementacja metody *DELETE* w Playwright

```
APIResponse response=
      request.delete("https://jsonplaceholder.typicode.com/posts/1");
```

W celu poprawnego zakończenia każdego z testów zastosowano standardowe asercje z biblioteki *JUnit5*, które weryfikują, czy odpowiedź serwera jest zgodna z oczekiwaniami (Listing 23).

Listing 23 Implementacja weryfikacji statusów dla metody *GET/PUT/POST/DELETE* w Playwright

```
assertEquals(200, response.status());

assertTrue(response.ok());
```

- `response.status()` – metoda zwracająca kod statusu HTTP z obiektu `APIResponse`.
- `response.ok()` - funkcja zwraca wartość typu `boolean`. Jeśli kod statusu odpowiedzi mieści się w przedziale 200–299, daje wynik `true`.

6.4.Analiza porównawcza

W ramach przeprowadzonej analizy, porównano dwie różne implementacje: Selenium z użyciem zewnętrznej biblioteki `RestAssured` [38] oraz Playwright z natywnym wsparciem `APIRequestContext` [37]. Oba podejścia realizują podstawowe operacje *HTTP (GET, POST, PUT, DELETE)* oraz obsługę przesyłania danych w formacie *Json* i formularza (*application/x-www-form-urlencoded*), jednak różnią się pod względem:

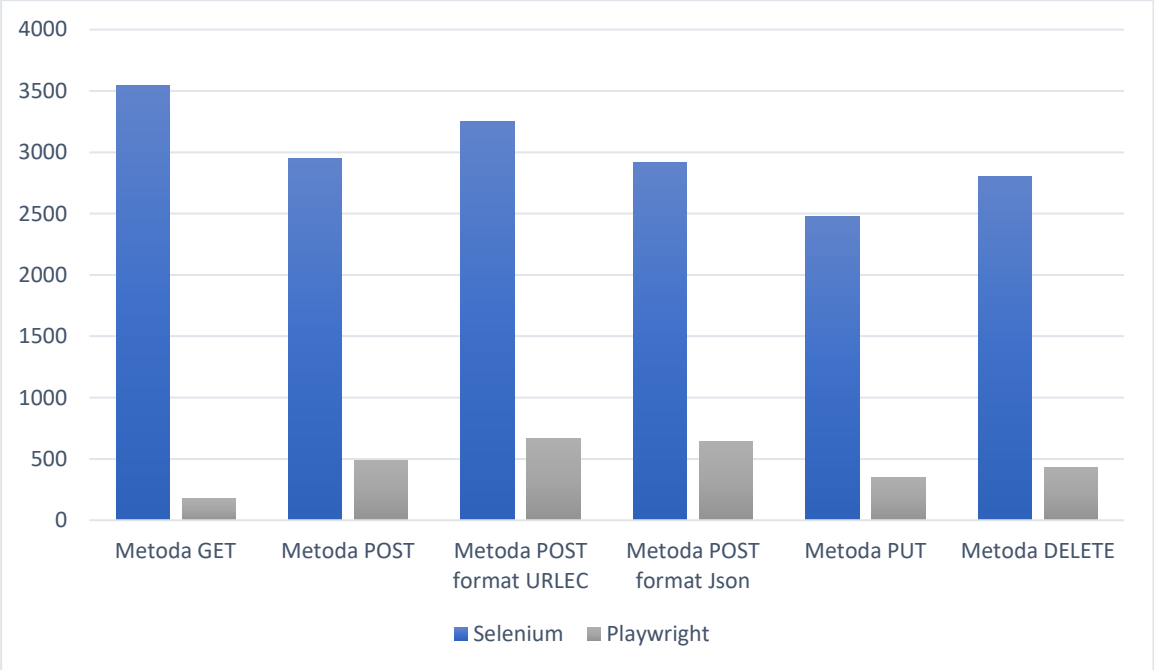
- architektury,
- sposobu integracji
- oraz wygody użytkowania.

W przypadku Selenium brak natywnej obsługi zapytań *REST API* wymaga rozszerzenia projektu o bibliotekę `RestAssured`, co wiąże się z koniecznością dodania zależności w pliku konfiguracyjnym projektu oraz ręcznym zarządzaniem ustawieniami i cyklem życia testów. Istotnym elementem jest możliwość natychmiastowej weryfikacji poprawności kodu odpowiedzi *HTTP* za pomocą metody `then().statusCode()`, co powoduje automatyczne zakończenie testu w przypadku niezgodności statusu.

Z kolei Playwright posiada wbudowany interfejs `APIRequestContext`, który natywnie obsługuje zapytania *HTTP*, eliminując potrzebę dodawania zewnętrznych bibliotek. Wykorzystanie dedykowanych metod `get()`, `post()`, `put()`, `delete()` wraz z konfigurowalnym obiektem `RequestOptions` pozwala na prostą i przejrzystą implementację testów *API*.

Playwright upraszcza proces konfiguracji testów interfejsów programistycznych oraz warstw prezentacji użytkownika w jednym środowisku, co może znacznie zwiększyć efektywność testów *End-to-End*. Metody asercji wykorzystują standardowe mechanizmy JUnit5, z możliwością weryfikacji statusu zwracanego przez serwer zarówno za pomocą kodu odpowiedzi, jak i metody `ok()`, potwierdzającej powodzenie zapytania.

Każdą implementację poddano trzykrotnemu wywołaniu w odstępstwach po 5s. Wykres 2 zawiera wyniki przeprowadzonych testów. Analiza grafu jednoznacznie pokazuje, że wbudowany interfejs *API* dostępny w bibliotece Playwright zapewnia znacznie szybszy mechanizm wysyłania zapytań *HTTP* w porównaniu do rozwiązania opartego na Selenium. Średnie czasy odpowiedzi w milisekundach dla Playwright są czterokrotnie, a w niektórych przypadkach pięciokrotnie niższe od wyników uzyskanych przy wykorzystaniu Selenium, co jednoznacznie wskazuje na wyraźną przewagę Playwright pod względem wydajności.



Wykres 2 Porównanie czasów wykonania w milisekundach dla zapytań *REST API* (mniej=lepiej). Źródło: opracowanie własne.

Biorąc pod uwagę czytelność kodu oraz zużycie pamięci, oba narzędzia zapewniają porównywalnie wysoki poziom w tego rodzaju testach.

Jedynym istotnym parametrem różnicującym, był czas wykonania testów, który zdecydowanie przemawiał na korzyść biblioteki Playwright. Sugeruje to, że to własnie ona stanowi bardziej efektywne rozwiązanie w testach, w których kluczowe znaczenie ma szybkość odpowiedzi serwera. Dodatkowo, krótszy okres wykonania testu może przyczyniać się do większej wydajności całego procesu testowania.

7. Testy logowania

W niniejszym rozdziale przedstawiono implementację przykładowych metod logowania z wykorzystaniem Selenium oraz Playwright. Do przeprowadzenia testów wybrano trzy różne serwisy: TheInternet (rozdział 4.7), GitHub.com (rozdział 4.2) oraz StackOverFlow.com (rozdział 4.5).

W celu dokonania rzetelnego porównania wybrano trzy metody weryfikacji o różnym poziomie trudności. Dla każdej z nich opracowano szczegółowy scenariusz testowy (Tabela 3) oraz zamieszczono implementację w obu narzędziach.

Na zakończenie rozdziału zebrano różnice oraz podobieństwa, tworząc podsumowanie, które zawiera również porównanie czasów wykonania testów oraz zużycia pamięci *CPU* przedstawione w formie tabelarycznej.

7.1.Podstawowe uwierzytelnianie HTTP *(Basic Authentication)*

Podstawowe uwierzytelnianie *HTTP* [39] jest jednym z najprostszych mechanizmów kontroli dostępu do zasobów sieciowych. Polega ono na przesłaniu w nagłówku *HTTP* zakodowanych w formacie *Base64* poświadczeń użytkownika, czyli loginu oraz hasła.

Ze względu na brak szyfrowania danych*, Basic Authentication* jest podatne na łatwe przechwycenie i odszyfrowanie przesyłanych informacji, co czyni go odpowiednim jedynie do zastosowań o niskim poziomie wymagań bezpieczeństwa lub w środowiskach testowych. Dlatego też zaleca się, aby stosować ten mechanizm wyłącznie w połączeniu z protokołem *HTTPS*, który zapewnia kodowanie transmisji danych i chroni przed podsłuchiwaniem.

Pomimo swoich ograniczeń, *Basic Authentication* jest często wykorzystywane ze względu na prostotę implementacji i łatwość automatyzacji, co czyni go popularnym wyborem podczas testów funkcjonalnych aplikacji webowych.

Do przeprowadzenia analizy wykorzystano stronę testową TheInternet (rozdział 4.7), która udostępnia mechanizm podstawowego uwierzytelniania *HTTP*, umożliwiając testowanie bez konieczności używania prawdziwych danych logowania, co eliminuje ryzyko przechwycenia informacji personalnych.

7.1.1. Scenariusz testowy

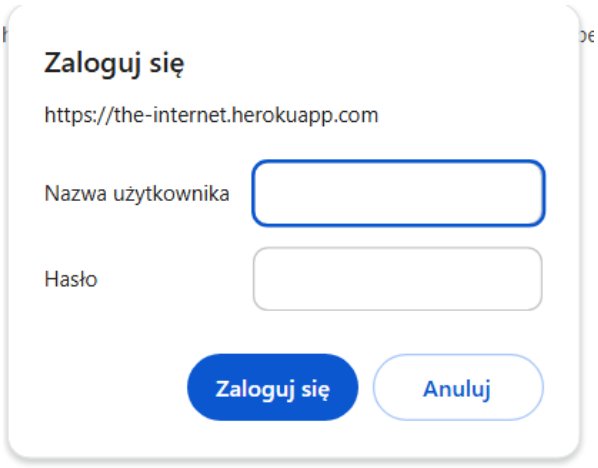
Opis scenariusza testowego dla podstawowego uwierzytelniania *HTTP*.

Tabela 3 Scenariusz testowy uwierzytelnianie *HTTP*. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna	Wprowadź adres *"https://theinternet.herokuapp.com/basic_auth"* do przeglądarki i przejdź do strony głównej. Wpisz dane uwierzytelniające użytkownika : Nazwa użytkownika : „admin’’ Hasło: „admin”

		Zweryfikuj, że autoryzacja przeszła pomyślne ukazując na stronie głównej tytuł: „Congratulations! You must have the proper credentials.”

7.1.2. Implementacja w Selenium



Rysunek 10 Okno dialogowe do logowania poprzez *Basic Authentication* na stronie TheInternet. Źródło: [28].

Implementacja testu automatycznego podstawowego logowania *HTTP* nie należała do skomplikowanych. Rysunek 10 przedstawia okno dialogowe po przejściu do aplikacji webowej, wymagające podania danych uwierzytelniających. Po otwarciu głównego adresu TheInternet, autorka pracy pozyskała login oraz hasło, które w obu przypadkach przyjmowało wartość *admin*. W celu weryfikacji użytkownika zastosowano konstrukcję adresu URL zawierającego dane logowania (Listing 24).

Listing 24 Implementacja *URL Basic Authentication* w Selenium

```
String username = "admin";
String password = "admin";
String url = "https://" + username + ":" + password + "@the-
            internet.herokuapp.com/basic_auth";

driver.get(url)
```

- `get()` – metoda z klasy *WebDriver* [9] odpowiedzielna za przekierowywanie użytkownika na dedykowaną stronę internetową.

Etap weryfikacji autorka pracy zdecydowała się zaimplementować poprzez użycie funkcji `contains()` z klasy `String` aby sprawdzić czy na stronie głównej pojawił się tytuł oznaczający pozytywną autoryzację użytkownika (Listing 25).

Listing 25 Weryfikacja pozytywnej autoryzacji z użyciem *Basic Authentication* w Selenium

```
String pageSource = driver.getPageSource();
Assertions.assertTrue(pageSource.contains("Congratulations! You must have
   the proper credentials."));
```

- `getPageSource()` - metoda zwracająca całą zawartość strony jako obiekt klasy *String.*

7.1.3. Implementacja w Playwright

Implementując logowanie *HTTP* w Playwright autorka pracy skorzystała z ustawień kontekstu przeglądarki wbudowanych w bibliotekę. Listing 26 przedstawia użycie klasy *BrowserContext* [40]. Obiekt jest wbudowanym elementem instacji aplikacji internetowej, który przechowuje informacje o ciasteczkach, sesjach oraz danych lokalnych.

Listing 26 Implementacja *Basic Authentication* w Playwright

```
BrowserContext context
BrowserContextUtils.getBrowser().newContext(new
      Browser.NewContextOptions()
        .setHttpCredentials(new HttpCredentials("admin", "admin")));

Page page = context.newPage();

page.navigate("https://the-internet.herokuapp.com/basic_auth");
```

- `setHttpCredentials()` – metoda umożliwia przekazanie danych logowania w formie obiektu `HttpCredentials` [41].

Biblioteka Playwright udostępnia wbudowaną funkcjonalność odpowiednią do stosowania logowania *Basic Authentication*, dzięki temu poświadczenia automatycznie zostają dodane do nagłówka żądania.

Weryfikację pozytywnej autoryzacji podobnie jak w Selenium zaimplementowano korzystając z funkcji `contains()` z klasy `String` (Listing 27).

Listing 27 Weryfikacja pozytywnej autoryzacji *Basic Authentication* w Playwright

```
String pageText = page.textContent("body");
Assertions.assertTrue(pageText.contains(("Congratulations! You must have
the proper credentials.")));
```

- `page.textContent()` – metoda z klasy `Page` [42] do pobierania zawartości elementu podanego jako argument.

7.1.4. Analiza porównawcza

Podczas implementacji logowania *HTTP* autorka pracy zaobserwowała, że oba narzędzia — zarówno Selenium, jak i Playwright — umożliwiają w stosunkowo prosty sposób przygotowanie kodu testowego pozwalającego na weryfikację poprawności procesu autoryzacji. Zwróciła jednak uwagę, że

Playwright, dzięki wbudowanym funkcjom konfiguracyjnym, pozwala na realizację tego typu mechanizmu w sposób bardziej przejrzysty i logiczny.

Rozwiązanie oferowane przez bibliotekę Microsoftu wyróżnia się klarownością i bezpieczeństwem implementacji. W przeciwieństwie do Selenium, gdzie proces uwierzytelnienia realizowany jest poprzez bezpośrednią modyfikację adresu *URL*. Może to prowadzić do prostych, lecz istotnych błędów :

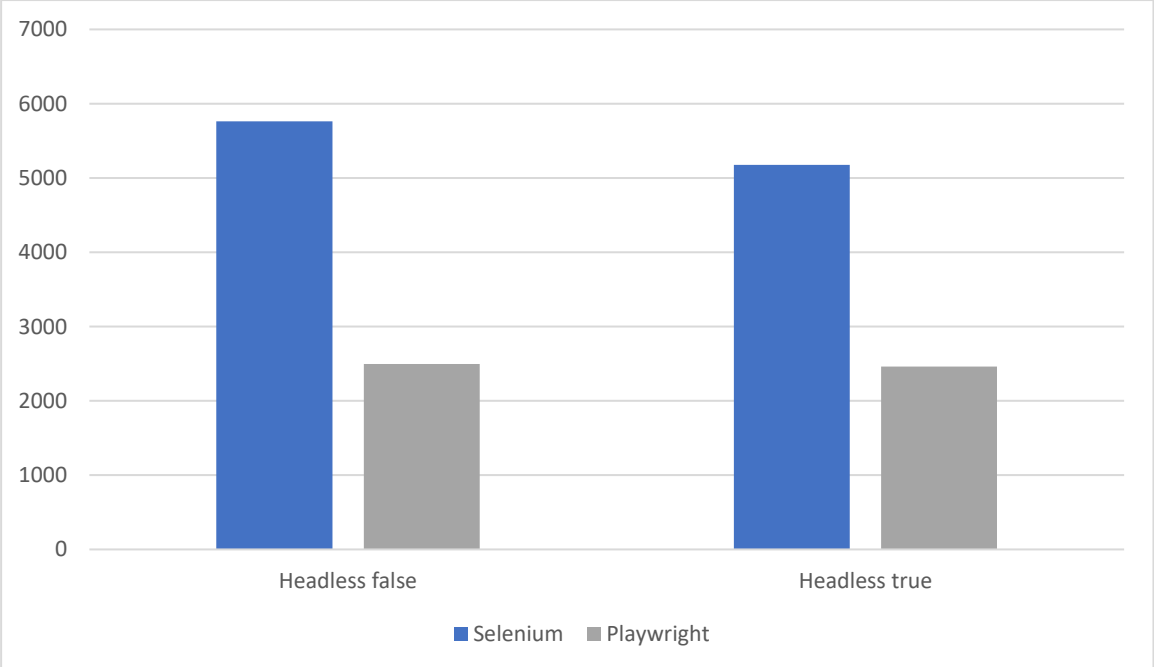
- związanych nieprawidłowym formatowaniem,
- lub niewłaściwym zakodowaniem znaków specjalnych.

Playwright umożliwia przekazanie danych uwierzytelniających w dedykowany sposób — poprzez konfigurację kontekstu przeglądarki. Takie podejście ogranicza ryzyko błędów i sprzyja utrzymaniu wysokiej czytelności oraz spójności kodu testowego.

Każdy z testów został uruchomiony trzykrotnie, a następnie wyliczono średnią czasu wykonania w milisekundach. Wykres 3 przedstawia wyniki przeprowadzonych testów, w których zauważono, niezależnie od trybu widoczności interfejsu graficznego, że Playwright wykonuje testy dużo szybciej niż Selenium.

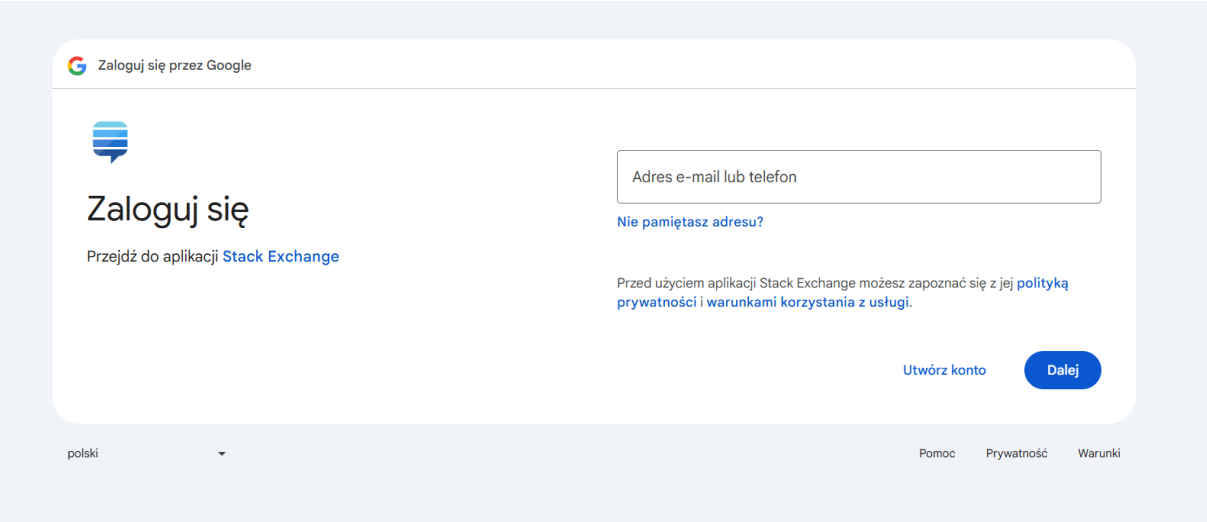
Testy podstawowego logowania wykonywane są lepiej przez narzędzie Playwright na każdym opisanym obszarze porównawczym:

- sposób implementacji,
- mniejsza podatność na błędy,
- oraz czas wykonania.



Wykres 3 Porównanie czasów wykonania w milisekundach przy logowaniu *HTTP* (mniej=lepiej). Źródło: opracowanie własne.

7.2.OAuth2



Rysunek 11 Przykład witryny logowania przy użyciu usługi Google.com. Źródło: [22].

Logowanie z wykorzystaniem *OAuth* (*Open Authorization*) zostało opracowane w celu umożliwienia bezpiecznej autoryzacji użytkowników bez konieczności udostępniania ich nazwy uwierzytelniającej i hasła każdej aplikacji, z którą wchodzą w interakcję. Takie podejście znacząco minimalizuje ryzyko naruszenia bezpieczeństwa poświadczenia. Dzięki zastosowaniu protokołu *OAuth* użytkownik może przyznać aplikacjom zewnętrznym dostęp do wybranych danych i funkcji swojego konta, bez konieczności przekazywania im wrażliwych informacji, takich jak hasło.

Dodatkowo według [44] mechanizm ten upraszcza pracę deweloperów, eliminując potrzebę tworzenia wielu niezależnych systemów logowania i umożliwiając wykorzystanie jednolitego standardu autoryzacji w różnych aplikacjach i usługach.

Testy dla tego typu logowania zostały wykonane na komercyjnej stronie StackOverFlow (rozdział 4.5) wykorzystujec logowania za pośrednictwem aplikacji Google.

7.2.1. Scenariusz testowy

Opis scenariusza testowego dla logowania przy użyciu logowania *Open Authentication.*

Tabela 4 Scenariusz testowy logowanie poprzez OAuth. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna	Wprowadź adres „*https://stackoverflow.com/users/login*” do przeglądarki i przejdź do strony głównej. Naciśnij przycisk autoryzacji poprzez *Google.*

2.	Logowanie	Wypełnij pole email: s19322@pjwstk.edu.pl Wciśnij „Dalej”.
3.	Logowanie na stronie PJWSTK	Wypełnij pola odpowiednimi danymi: Email : s19322@pjwstk.edu.pl Password: [password] Naciśnij przycisk „Dalej’’.
4.	Strona główna po autoryzacji	Zweryfikuj, że na stronie głównej widnieje tytuł : „Welcome back, Natalia Chotza”.

7.2.2. Implementacja w Selenium

Listing 28 przedstawia obsługę dynamicznego wyszukiwania elementów na stronie internetowej, w celu realizacji scenariusza logowania (Tabela 4) przy użyciu konta Google (Rysunek 11). Dla zapewnienia poprawności działania oraz odporności na ewentualne opóźnienia w ładowaniu interfejsu użytkownika, w kodzie wykorzystano mechanizm `WebDriverWait` [43].

Listing 28 Implementacja kliknięcia w przycisk reprezentujący aplikację Google.

```
WebElement googleLoginButton = wait.until(
ExpectedConditions.elementToBeClickable(By.cssSelector("button[data-
   provider='google']"))
);

googleLoginButton.click();
```

- `ExpectedConditions.elementToBeClickable()` – interakcja z elementem poprzedzona jest oczekiwaniem na jego pojawienie się lub możliwość kliknięcia,

Autorka pracy zdecydowała się na zalogowanie poprzez konto uczelni PJWSTK, w tym celu w momencie pojawienia się opcji wyboru innego konta logowania należało wybrać opcję „użyj innego konta” (Listing 29).

Listing 29 Implementacja logowania z użyciem konta uczelnianego w Selenium

```
List<WebElement> otherAccountOptions =driver
      .findElements(By.xpath("//*[text()='Użyj innego konta']"));

if (!otherAccountOptions.isEmpty()) {
    WebElement otherAccountOption = wait.until(
       ExpectedConditions
       .elementToBeClickable(By.xpath("//*[text()='Użyj innego konta']")));
   otherAccountOption.click();
}else {
    WebElement emailInput = wait.until(


ExpectedConditions
      .visibilityOfElementLocated(By.xpath("//*[@aria-label='Adres e-mail
      lub telefon']")));
```

```
emailInput.sendKeys("s19322@pjwstk.edu.pl");

WebElement nextButton = wait.until(ExpectedConditions
   .elementToBeClickable(By.xpath("//span[text()='Dalej']/..")));
nextButton.click();
}
```

- `ExpectedConditions.visibilityOfElementLocated()` – metoda odpowiedzialna za oczekiwanie aż element na stronie będzie widoczny,

- `emailInput.sendKeys()` – funkcja umożliwiająca przekazanie argumentu tekstowego jako wartość do elementu, na którym zostaje wywołana,

- `By.xpath()` – służy do wyszukiwania obiektow poprzez użycie dokładnej ścieżki w dokumencie *HTML* [45].

W zależności od wprowadzonego adresu e-mail, mechanizm logowania Google kieruje użytkownika bezpośrednio na stronę StackOverFlow lub do dedykowanej aplikacji obsługującej proces autoryzacji. W przypadku korzystania z konta uczelnianego, weryfikacja realizowana jest z wykorzystaniem zewnętrznego systemu uwierzytelniania instytucji. W analizowanym przypadku oznacza to dodatkowe przekierowanie na stronę logowania PJWSTK w celu przeprowadzenia sprawdzenia tożsamości oraz potwierdzenia autoryzacji. Dopiero po zezwoleniu na dostęp następuje przekierowanie do właściwej aplikacji (Listing 30).

Listing 30 Przekazanie danych użytkownika na stronie uczelni PJWSTK

```
WebElement emailInput =
   wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("userName
Input")));

emailInput.sendKeys("s19322@pjwstk.edu.pl");
stackOverFlowLogin.fillPJWSTKPassword();

driver.findElement(By.id("submitButton")).click();
```

- `stackOverFlowLogin.fillPJWSTKPassword()` – metoda utworzona przez autorkę pracy w celu ukrycia danych wrażliwych. Implementuje funkcję `sendKeys()` do wysłania hasła jako wartości tekstowej.

W przypadku korzystania z Selenium, przeglądarka domyślnie uruchamiana jest w standardowym trybie, w którym zapisywane są m.in. aktywne sesje użytkownika. W konsekwencji, podczas procesu logowania przy użyciu mechanizmu *OAuth*, na ekranie pojawia się dodatkowe zapytanie o treści „Czy to Twoje konto?”. Warto zaznaczyć, że to okno dialogowe nie pojawia się przy każdym uruchomieniu testu, co wynika z mechanizmu przechowywania sesji przez przeglądarkę. Nieregularne wyświetlanie tego komunikatu stanowiło istotne utrudnienie w realizacji testów automatycznych. W szczególności wpływało to na stabilność skryptów, które wywoływały metody oczekujące na obecność określonych elementów interfejsu. W sytuacji, gdy elementy te nie pojawiały się z powodu dodatkowego zapytania *OAuth*, testy kończyły się niepowodzeniem (Listing 31).

W celu ograniczenia tego problemu, autorka pracy zdecydowała się na uruchomienie przeglądarki w trybie prywatnym (z ang. *incognito*). Taki tryb uniemożliwia zapisywanie danych sesji oraz plików ciasteczkowych (z ang. *cookies*) pomiędzy kolejnymi testami, co pozwoliło wyeliminować losowe pojawianie się komunikatu „Czy to Twoje konto?”. Dzięki temu możliwe było uzyskanie bardziej powtarzalnych i przewidywalnych rezultatów testów, a tym samym ograniczenie liczby błędów

wynikających z nieoczekiwanych interakcji interfejsu z użytkownikiem. Listing 32 przedstawia implementację obejścia problemu przy użyciu klasy `ChromeOptions` [46], do zarządzania ustawieniami przeglądarki.

Listing 31 Implementacja kodu, który powodował niepowodzenie testów w Selenium

```
boolean isVisibleNextButton = driver
   .findElement(By.xpath("//span[text()='Dalej']/..")).isDisplayed();
if (isVisibleNextButton)
    driver.findElement(By.xpath("//span[text()='Dalej']/..")).click();
```

Listing 32 Implementacja dodania flagi trybu prywatnego w Selenium

```
chromeOptions.addArguments("--incognito");
```

- `chromeOptions.addArguments` – metoda umożliwiająca przekazywanie parametrów do przeglądarki Chrome poprzez użycie obiektu klasy `ChromeOptions`.

Listing 33 prezentuje weryfikację poprawności autoryzacji. W tym celu zaimplementowano sprawdzenie zgodności tytułu z tekstem oczekiwanym w scenariuszu testowym.

Listing 33 Weryfikacja pozytywnej autoryzacji na stronie StackOverFlow.com w Selenium

```
boolean isHeaderVisible;
try {
 wait.until(ExpectedConditions
      .visibilityOfElementLocated(By.xpath("//*[contains(text(),' Welcome
back, Natalia Chotza')]")));

    isHeaderVisible = true;
} catch (TimeoutException e) {
    isHeaderVisible = false;
}
Assertions.assertTrue(isHeaderVisible, "Login failed: Header not visible");
```

W trakcie realizacji testów w trybie bez interfejsu graficznego (z ang. *headless*) napotkano istotny problem polegający na niepowodzeniu wszystkich wywołań. Po szczegółowej analizie stwierdzono, iż przyczyną takiego stanu rzeczy było blokowanie zapytań pochodzących z przeglądarki działającej w trybie `headless` przez mechanizmy antybotowe zastosowane na stronie docelowej. Strona wykrywała specyficzne cechy przeglądarki, takie jak odmienny nagłówek *User-Agent* [47]. W celu obejścia tego problemu, autorka pracy zdecydowała się na modyfikację parametrów wywoływania przeglądarki poprzez dodanie odpowiednich argumentów i ustawień, mających na celu ukrycie faktu działania w trybie `headless`. Dzięki tej konstrukcji możliwe było pomyślne przeprowadzenie testów automatycznych bez ingerencji mechanizmów zabezpieczających strony internetowej (Listing 34).

Listing 34 Implementacja dodania sztucznego agenta aby wykonać testy w trybie bez interfejsu graficznego w Selenium.

```
chromeOptions.addArguments("user-agent=Mozilla/5.0 (Windows NT 10.0; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36");
```

7.2.3. Implementacja w Playwright

Podczas implementacji w narzędziu Playwright autorka pracy zastosowała wbudowane procedury do wyszukiwania elementu z rolą przycisku z nazwą *„google”* (Listing 35).

Listing 35 Implementacja kliknięcia w przycisk logowania do Google w Playwright.

```
StackOverFlowLogin stackOverFlowLogin = new StackOverFlowLogin(page);
page.getByRole(AriaRole.BUTTON, new Page.GetByRoleOptions()
      .setName("google"))
      .click();
```

- `page.getByRole()` – metoda służąca do wyszukania elementu na stronie o podanych argumentach,
- `AriaRole.BUTTON` – charakter elementu, do którego będzie odnosił się kod w tym przypadku wyszukiwany jest obiekt o roli przycisku (z ang. *button*),
- `GetByRoleOptions().setName()` – funkcja wyszukująca instancje o podanym tekście w argumencie.

Listing 35 implementuje logowanie na stronie uczelni PJWSTK niezbędne do ukończenia przykładu logowania *OAuth.* Użyto w tym celu wbudowanych metod wyszukiwania. W momencie znalezienia odpowiedniego przycisku zostaje on kliknięty, w innym przypadku należało wpisać dedykowany adres *emial* do pola tekstowego i przejść dalej (Listing 36).

Listing 36 Implementacja logowania z użyciem konta uczelnianego w Playwright.

```
Locator selectOtherUser = page
      .getByRole(AriaRole.BUTTON, new Page.GetByRoleOptions()
      .setName("Użyj innego konta"));

if (selectOtherUser.isVisible()) {
    selectOtherUser.click();

} else {

    page
      .getByRole(AriaRole.TEXTBOX, new Page.GetByRoleOptions()
      .setName("Adres e-mail lub telefon"))
      .fill("s19322@pjwstk.edu.pl");

      page.getByRole(AriaRole.BUTTON, new Page.GetByRoleOptions()
      .setName("Dalej"))
      .click();
}
```

- `AriaRole.TEXTBOX` – charakter elementu, do którego będzie odnosił się kod w tym przypadku wyszukiwany jest obiekt o roli pola tekstowego (z ang. *textbox*),

Na stronie logowania uczelni PJWSTK wymagane było wprowadzenie danych uwierzytelniających oraz zatwierdzenie ich za pomocą przycisku autoryzującego, co inicjowało dalszy proces logowania (Listing 37).

W przypadku zastosowania narzędzia Playwright, przeglądarka uruchamiana jest w obrębie nowego kontekstu dla każdego testu, co powoduje, że każdorazowo tworzona jest nowa sesja oraz oddzielny zestaw ciasteczek. Rozwiązanie to zapewnia większą elastyczność oraz powtarzalność wykonywanych testów, eliminując wpływ danych sesyjnych z wcześniejszych uruchomień na aktualny przebieg kodu (Listing 37).

Listing 37 Przekazanie danych użytkownika na stronie uczelni PJWSTK w Playwright.

```
page.locator("#userNameInput").fill("s19322@pjwstk.edu.pl");
stackOverFlowLogin.fillPJWSTKPassword();
page.locator("#submitButton").click();
```

- `page.locator()` – metoda służąca do wyszukiwania elementów na stronie poprzez nazwę klasy poprzedzoną znakiem #.

Ostatnim etapem realizowanego testu była weryfikacja poprawności zakończenia procesu autoryzacji. W tym celu w kodzie wykorzystano pomocniczą metodę wbudowaną w obiekt klasy *Page* [42], której zadaniem było odczekanie na załadowanie wszystkich niezbędnych zasobów strony dynamicznej oraz sprawdzenie widoczności oczekiwanego tytułu. Początkowo autorka pracy zaimplementowała rozwiązanie, które wyszukiwało określony tekst na stronie, weryfikując przy tym również obecność białych znaków. Taki sposób realizacji okazał się jednak nieefektywny i narażony na błędy implementacyjne, co skutkowało niepowodzeniami testów (Listing 38).

Listing 39 wprowadza z kolei poprawne rozwiązanie, gdzie pobierane są wszystkie elementy nagłówków na stronie, a następnie ich zawartość jest filtrowana w celu odnalezienia jednego, zawierającego oczekiwany tekst. Takie podejście pozwoliło na wyeliminowanie problemów z interpretacją białych znaków. [48]

Listing 38 Weryfikacja pozytywnej autoryzacji na stronie StackOverFlow.com w Playwright.

```
page.waitForLoadState(LoadState.NETWORKIDLE);
boolean isHeaderVisible = page.getByRole(AriaRole.HEADING, new
      Page.GetByRoleOptions()
      .setName(" Welcome back, Natalia Chotza"))
      .isVisible();
```

- `page.waitForLoadState(LoadState.NETWORKIDLE)` [49] – metoda, oczekująca na zakończenie wykonywania się wszystkich zapytań na stronie internetowej przed realizacją kolejnych akcji,

Listing 39 Właściwa implementacja weryfikacji pozytywnej autoryzacji na stronie StackOverFlow.com w Playwright

```
boolean isHeaderVisible page.getByRole(AriaRole.HEADING)
            .filter(new Locator
            .FilterOptions().setHasText("Welcome back, Natalia Chotza"))
            .isVisible();
```

- `AriaRole.HEADING` – charakter elementu, do którego będzie odnosił się kod, w tym przypadku wyszukiwany jest obiekt o roli nagłówka (z ang. *heading*),

Podczas wywoływania testów w trybie bez interfejsu graficznego napotkano problem niepowodzenia testów. Spowodowany był dwoma czynnikami :

- zabezpieczeniem stron komercyjnych przed działaniami robotów,
- niedoładowaniem wszystkich elementów.

Pierwszy z powodów trudno było odnaleźć, okazało się, że większość stron komercyjnych zapobiega działaniu bez interfejsu graficznego np. poprzez zmianę atrybutów, którą trudno wykryć pisząc testy automatyczne. Listing 40 przedstawia dodanie dodatkowej konfiguracji, która w przypadku testów na stronie StackOverFlow, umożliwiła autorce pracy dokończyć weryfikację z pozytywnym rezultatem.

W trybie bez interfejsu graficznego często brakuje informacji dotyczących przeglądarki, takie połączenia są blokowane. Rozwiązaniem zostało podanie sztucznych wartości aby oszukać silnik [50].

Listing 40 Implementacja konfiguracji dla testów z brakiem interfejsu graficznego w Playwright

```
setBrowserContext(browser.newContext(new Browser.NewContextOptions()
      .setUserAgent("Mozilla/5.0 (Windows NT 10.0; Win64; x64)
      AppleWebKit/537.36 (KHTML, like Gecko) Chrome/131.0.0.0
      Safari/537.36")));

BrowserContext context = browser.newContext(new Browser.NewContextOptions()
        .setViewportSize(1280, 720)
        .setJavaScriptEnabled(true)
        .setIgnoreHTTPSErrors(true));
```

- `setUserAgent()` - metoda wywołana na kontekście przeglądarki w celu dodania sztucznego agenta przeglądarki,
- `setViewportSize()` – funkcja ustawiająca rozmiar okna,
- `setJavaScriptEnabled(true)` - włączenie kontekstu *JavaScript* w przeglądarce,
- `setIgnoreHTTPSErrors(true)` – ustawienie ignorowania ewentualnych błędów.

Listing 38 przedstawia użycie metody `waitForLoadState(LoadState.NETWORKIDLE)`, która stała się rozwiązaniem do drugiego w kolejności problemu związanego z niedoładowaniem wszystkich elementów na czas. Funkcja wyczekuje na przesłanie wszystkich zasobów na stronę. Wykrycie brakującego obiektu zajęło autorce pracy nieco więcej czasu, z uwagi na to, że jeden na dziesięć testów kończył się powodzeniem. Autorka zdecydowała się na użycie konstrukcji wykonywania zrzutów ekranów [51] (z ang. *screen shot*), wbudowaną w Playwright w celu debugowania.

Listing 41 zawiera funkcjonalność uchwycenia obrazu, którą dodano w miejscu odczytywania nagłówka w ostatnim kroku scenariusza testowego (Tabela 4). Otrzymano zdjęcie pustego ekranu, co pokazało autorce pracy, zbyt szybkie wykonanie kodu odnoszącego się do elementu strony. Po dodaniu odpowiedniej metody każde wywołanie testu kończyło się pozytywnym rezultatem.

Listing 41 Funkcjonalność zrzutów ekranu w Playwright.

```
page.screenshot(new Page.ScreenshotOptions()
  .setPath(Paths.get("header.png")));
```

7.2.4. Analiza porównawcza

Porównując implementację w obu narzędziach autorka pracy zauważyła różnicę w wielkości kodu potrzebnego do osiągnięcia tego samego celu. W Selenium objętość kodu zajęła prawie dwa razy więcej niż w Playwright. Dodatkowo zauważono pozytywny aspekt występowania wbudowanych narzędzi oczekiwania na elementy w Playwright. W Selenium należało dodać dodatkowy obiekt `WebDriverWait` [44], który umożliwił zapobieganie odnoszenia się do nieistniejących elementów na stronie internetowej.

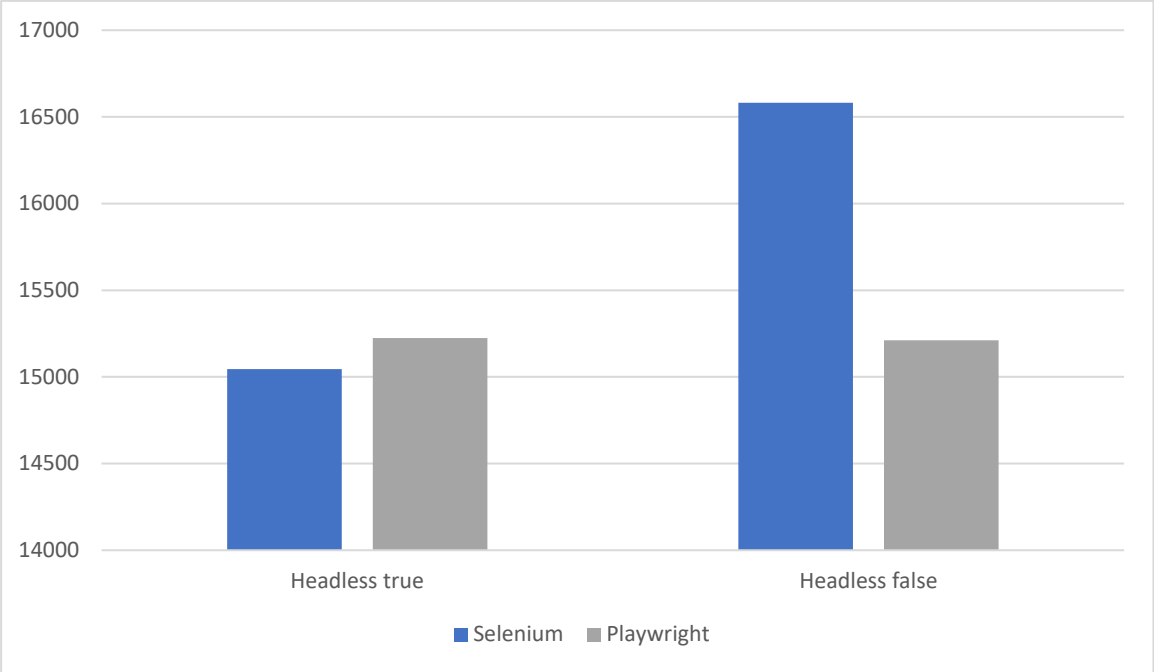
W trakcie wykonywania testów przy użyciu obu narzędzi napotkano utrudnienia związane z zabezpieczeniami antybotowymi zastosowanymi w aplikacji webowej StackOverFlow. W celu obejścia tego problemu konieczne było ręczne zalogowanie się na stronę, a następnie uruchomienie testów automatycznych. Z tego względu nie wszystkie testy zakończyły się powodzeniem.

Dodatkowo, w przypadku Selenium niezbędne było uruchomienie przeglądarki w trybie prywatnym (z ang. *incognito*), aby uniknąć zapisywania sesji oraz problemów związanych z nieregularnym wyświetlaniem komunikatów *OAuth* podczas logowania. W odróżnieniu od tego rozwiązania, w narzędziu Playwright każdorazowe wykonanie testu odbywało się w ramach nowego kontekstu przeglądarki, co oznaczało automatyczne tworzenie oddzielnych sesji i plików *cookies* dla każdego testu. Dzięki temu udało się wyeliminować problem w sposób natywny, co znacząco uprościło implementację oraz przyspieszyło proces przygotowania środowiska testowego.

W Playwright z kolei należało użyć jednego ze sposobów debugowania kodu, poprzez użycie zrzutów ekranu aby zaleźć miejsce, w którym implementacja kończyła się niepowodzeniem z powodu zbyt szybkiego wykonywania akcji odczytu elementów. W Selenium problem nie występował z uwagi na odpowiednie użycie metody z klasy `WebDriverWait`, przy odczytywaniu nagłówka strony.

Wykres 4 przedstawia średnie czasy wywoływania testów (w milisekundach), z podziałem na tryby widoczności interfejsu graficznego. Dane na grafie obejmują tylko testy przeprowadzone po wprowadzeniu usprawnień, bez testów w których konieczna była manualna interwencja podczas ich wykonywania.

Autorka zauważyła na wykresie słupkowym, że Playwright poradził sobie lepiej od Selenium przy parametrze z widocznym interfejsem graficznym. Z kolei przy zmienionym argumencie na `true`, Playwright wypadł nieco gorzej od Selenium. Powodem gorszego czasu mogły być warunki zewnętrzne jak i inaczej odczytywane sztuczne parametry przeglądarki. W przypadku testowania strony komercyjnej przy logowaniu z *Open Authentication* zauważono, że Playwright radzi sobie lepiej z włączonym interfejsem graficznym i z wbudowanymi argumentami przeglądarki.



Wykres 4 Porównanie czasów wykonania przy logowaniu *OAuth* (mniej=lepiej). Źródło: opracowanie własne.

7.3.Logowanie z wykorzystaniem sesji

Logowanie sesyjne (ang. *Session-Based Login*) [52] polega na jednorazowym uwierzytelnieniu użytkownika oraz, po pomyślnej weryfikacji, przydzieleniu mu unikalnego identyfikatora sesji wraz z ciasteczkami. Przeglądarka zarządza tymi danymi, automatycznie dołączając je do każdego wysyłanego żądania do serwera, co umożliwia utrzymanie stanu zalogowania bez konieczności ponownej manualnej autoryzacji użytkownika.

Testy implementujące rozwiązanie wykonano na stronie Github.com (rozdział 4.2).

7.3.1. Scenariusz testowy

Opis scenariusza testowego dla logowania z wykorzystaniem sesji.

Tabela 5 Scenariusz testowy logowanie z wykorzystaniem sesji. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna przed autoryzacją	Wprowadź adres url „*https://github.com/login”* do przeglądarki i przejdź do strony głównej. Wypełnij pola: email:”s19322@pjwstk.edu.pl” password: [password] Naciśnij przycisk *Login.*

2.	Nowa strona	Otwórz nową kartę w przeglądarce i wpisz adres *URL* „*https://github.com/login”*.
3.	Strona główna po autoryzacji	Zweryfikuj widoczność przycisku *New* na stronie głównej.

7.3.2. Implementacja w Selenium

W narzędziu Selenium pełne logowanie z wykorzystaniem sesji nie jest możliwe do zrealizowania z uwagi na brak natywnego mechanizmu obsługi cyklu działania przeglądarki w kontekście automatyzacji testów. W praktyce według [53] testerzy mogą jedynie zapisywać oraz odczytywać pliki *cookies*, jednak nie mają możliwości odwzorowania kompletnego logowania poprzez przekazanie wszystkich danych sesyjnych gdzie mogą być potrzebne inne dodatkowe atrybuty. Z tego względu autorka pracy była w stanie zaimplementować jedynie część wymagań określonych w scenariuszu testowym (Tabela 5).

Listing 42 zawiera proces implementacji rozpoczynający się od klasycznego logowania na stronie github.com poprzez przesłanie danych uwierzytelniających.

Listing 42 Implementacja logowania do strony Github.com w Selenium

```
driver.get("https://github.com/login");
driver.findElement(By.id("login_field")).sendKeys("s19322@pjwstk.edu.pl");
driver.findElement(By.id("password")).sendKeys(Utils.password);
driver.findElement(By.cssSelector("input[value='Sign in']")).click();
```

W kolejnym etapie testu następuje oczekiwanie na załadowanie żądanej witryny internetowej. Po potwierdzeniu poprawności adresu *URL* przy pomocy wbudowanego mechanizmu synchronizacji, zapisane zostają wszystkie ciasteczka sesyjne w obiekcie klasy `Set`. Zebrany zestaw danych zostaje następnie utrwalony w pliku z rozszerzeniem *.ser*, w którym są serializowane do postaci binarnej.

Implementacja pierwszej części kodu w Selenium kończy się na sprawdzeniu czy użytkownik został poprawnie zalogowany oraz czy plik z pobranymi ciasteczkami nie jest pusty. (Listing 43).

Listing 43 Zapis plików ciasteczkowych w Selenium.

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

wait.until(d -> d.getCurrentUrl().contains("github.com"));

Set<Cookie> cookies = driver.manage().getCookies();
try (ObjectOutputStream oos = new
ObjectOutputStream(Files.newOutputStream(COOKIE_PATH))) {
    oos.writeObject(new ArrayList<>(cookies));
}

WebElement h =wait.until(ExpectedConditions
      .visibilityOfElementLocated(driver
      .findElement(By
            .xpath("//span[@class='Button-label' and text()='New']"));
```

```
Assertions.assertTrue(h.isDisplayed(), "Nie widoczny nagłówek");
Assertions.assertTrue(Files.size(COOKIE_PATH) > 0);
```

- `getCurrentUrl()` – metoda do pobierania aktywnego adresu *URL*.

Mając wcześniej zapisane pliki ciasteczek sesyjnych w określonej lokalizacji na dysku lokalnym komputera, autorka pracy zaimplementowała mechanizm ich odczytu, wykorzystując standardowe praktyki programistyczne w języku *Java*. Po załadowaniu danych sesyjnych przeglądarka została ponownie uruchomiona, a następnie otworzono stronę internetową Github.com w celu weryfikacji skuteczności autoryzacji przy użyciu zapisanych atrybutów.

Niestety, mimo poprawnego wczytania danych, strona nadal pozostawała na ekranie logowania, oczekując ręcznego wprowadzenia danych uwierzytelniających przez użytkownika. Próba przeprowadzenia testu w ten sposób zakończyła się więc niepowodzeniem (Listing 44). Wskazuje to na ograniczenia Selenium w zakresie pełnego odwzorowania sesji uwierzytelniającej wyłącznie przy użyciu zapisanych ciasteczek.

Listing 44 Implementacja odczytania plików ciasteczkowych oraz próba zalogowania się z ich użyciem w Selenium

```
try (ObjectInputStream ois = new
      ObjectInputStream(Files.newInputStream(COOKIE_PATH))) {

      List<Cookie> cookies = (List<Cookie>) ois.readObject();

      for (Cookie cookie : cookies) {
              driver.manage().addCookie(cookie);
      }

} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}
driver.navigate().refresh();

WebElement h =wait.until(
     ExpectedConditions
      .visibilityOfElementLocated(driver
          .findElement(
            By.xpath("//span[@class='Button-label' and text()='New']"));

Assertions.assertTrue(h.isDisplayed(), "Nie widoczny nagłówek");
```

7.3.3. Implementacja w Playwright

Biblioteka Playwright według [54] oferuje natywne wsparcie dla logowania z wykorzystaniem sesji, umożliwiając zapis oraz odczyt danych sesyjnych w trakcie wykonywania testów automatycznych. W celu realizacji tego mechanizmu utworzono pusty plik w formacie *Json*, do którego — po pomyślnym zalogowaniu użytkownika — zapisano wszystkie aktualne atrybuty sesji pobrane ze strony internetowej (Listing 45).

W dalszym etapie testu przeprowadzono weryfikację poprawności autoryzacji użytkownika, a także sprawdzono integralność stworzonego pliku, kontrolując czy nie jest pusty.

Listing 45 Logowanie i zapis plików ciasteczkowych w formacie *Json*.

```
Path STORAGE_PATH = Paths.get("auth-storage.json");

Page page = BrowserContextUtils.getBrowserContext().newPage();
page.navigate("https://github.com/login");

page.getByLabel("Username").fill("s19322@pjwstk.edu.pl");
page.getByLabel("Password").fill(Utils.password);

Locator signInButton =  page.getByRole(AriaRole.BUTTON, new
                  Page.GetByRoleOptions()
                  .setName("Sign in").setExact(true));

signInButton.click();

BrowserContextUtils.getBrowserContext()
      .storageState(new BrowserContext
            .StorageStateOptions()
            .setPath(STORAGE_PATH));

page.waitForLoadState(LoadState.NETWORKIDLE);
Locator buttonLabel = page
            .getByRole(AriaRole.LINK, new Page.GetByRoleOptions()
                  .setName("New"));

Assertions.assertTrue(buttonLabel.isVisible());

Assertions.assertTrue(Files.size(STORAGE_PATH) > 0);
```

- `storageState()` – metoda służąca do zwracania plików ciasteczkowych oraz danych sesyjnych z przeglądarki [40].

Kolejnym etapem implementacji było utworzenie nowego kontekstu aplikacji webowej oraz przekazanie zmiennych sesji poprzez wskazanie ścieżki do pliku formatu *Json* utworzonego i uzupełnionego w poprzednim kroku. Po załadowaniu informacji kontekstowych oraz przejściu na stronę serwisu GitHub.com, użytkownik został automatycznie przekierowany na witrynę swojego profilu, co potwierdziło poprawne odtworzenie funkcjonalności logowania. W ten sposób zasymulowano mechanizm przekazywania danych, odzwierciedlający rzeczywiste działanie przeglądarki w warunkach interakcji konsumenta.

Weryfikacja skuteczności autoryzacji odbyła się poprzez odszukanie na stronie tego samego elementu interfejsu, który był wykorzystywany we wcześniejszym etapie testu do potwierdzenia zalogowania. Na zakończenie testu plik z danymi sesyjnymi został usunięty, celem umożliwienia jego ponownego utworzenia przy każdorazowym uruchomieniu kodu testowego, co zapewniło integralność i powtarzalność procesu testowania (Listing 46).

Listing 46 Implementacja wstrzykiwania pliku z danymi sesyjnymi w Playwright

```
BrowserContextUtils.createPlaywrightAndBrowserChromium();
BrowserContext context = BrowserContextUtils.getBrowser().newContext(
        new Browser.NewContextOptions().setStorageStatePath(STORAGE_PATH)
);
BrowserContextUtils.setBrowserContext(context);

page = context.newPage();
```

```
page.navigate("https://github.com/login");

buttonLabel = page.getByRole(AriaRole.LINK, new Page
            .GetByRoleOptions()
            .setName("New"));
Assertions.assertTrue(buttonLabel.isVisible());

Files.delete(STORAGE_PATH);
```

7.3.4. Analiza porównawcza

W narzędziu Selenium pełne logowanie z wykorzystaniem sesji nie jest możliwe do zrealizowania ze względu na brak natywnego mechanizmu obsługi danych sesyjnych w kontekście automatyzacji testów. W praktyce możliwe jest jedynie zapisanie i ponowne wczytanie ciasteczek do przeglądarki, jednak bez możliwości odwzorowania kompletnego procesu uwierzytelnienia, który często wymaga także dodatkowych atrybutów sesyjnych oraz danych kontekstowych. W efekcie implementacja logowania w Selenium ogranicza się do zapisania ciasteczek w pliku z rozszerzeniem *.ser* oraz próby ich ponownego użycia, co nie gwarantuje pełnej autoryzacji po odświeżeniu strony – testy zakończyły się niepowodzeniem.

Z kolei biblioteka Playwright oferuje natywne wsparcie dla logowania sesyjnego. Mechanizm ten umożliwia zapisanie do pliku w formacie *Json* pełnego stanu kontekstu przeglądarki, obejmującego zarówno :

- ciasteczka,
- dane sesyjne,
- jak i inne właściwości środowiska.

W kolejnym kroku dane te mogą zostać wstrzyknięte do nowego kontekstu przeglądarki, co pozwala na dokładne odwzorowanie sesji użytkownika i symulację rzeczywistej interakcji.

W testach przeprowadzonych przez autorkę pracy, logowanie przy użyciu sesji w Playwright zakończyło się powodzeniem — użytkownik został przekierowany na stronę swojego profilu, a autoryzację potwierdzono poprzez obecność oczekiwanego elementu na stronie (Tabela 5). Rozwiązanie to umożliwia także łatwe usuwanie i odtwarzanie plików sesyjnych w celu zapewnienia powtarzalności testów.

Z uwagi na brak możliwości przeprowadzenia pełnego porównania czasów wykonania w obu narzędziach, autorka pracy zrezygnowała z przedstawienia wyników w postaci graficznej. Okres wykonania testów automatycznych oscylował wokół wartości 1,5 minuty.

Jednoznacznie stwierdzono, że Playwright jest idealnym narzędziem do testowania logowania z wykorzystaniem sesji z uwagi na możliwość pełnego odzwierciedlenia funkcjonalności.

8. Testy operacji na plikach

W tym rozdziale omówiono różne operacje wykonywane na plikach w ramach testów *End-to-End*.

Przedstawiono następujące scenariusze:

- przesyłanie pliku na stronę,
- pobieranie dokumentu ze strony,
- przeciąganie i upuszczanie elementów za pomocą mechanizmu D*rag-and-Drop*.

Celem tych testów było sprawdzenie, jak wybrane narzędzia radzą sobie z obsługą operacji na plikach, które często występują w rzeczywistych aplikacjach webowych.

Scenariusze testowe są trywialne, natomiast w testach przedstawiono kilka możliwych sposobów na osiągnięcie opisanego celu.

Na zakończenie każdego z podrozdziałów przeprowadzono analizę napotkanych trudności oraz porównanie wydajności i funkcjonalności zastosowanych rozwiązań.

8.1.Przesyłanie pliku na stronę

W tym podrozdziale omówiono proces przesyłania dokumentu - zdjęcia do aplikacji webowej. Testy przeprowadzono na stronie internetowej TheInternet stworzonej typowo pod testowanie automatyczne (rozdział 4.7).

8.1.1. Scenariusz testowy

Opis scenariusza testowego dotyczący wysyłania pliku na stronę internetową.

Tabela 6 Scenariusz testowy przesyłania pliku na stronę. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna	Wprowadź adres https://the-internet.herokuapp.com/upload. Naciśnij przycisk wysyłania pliku. Zweryfikuj czy na stronie pojawił się nagłówek, który odpowiada nazwie przesłanego pliku.

8.1.2. Implementacja w Selenium

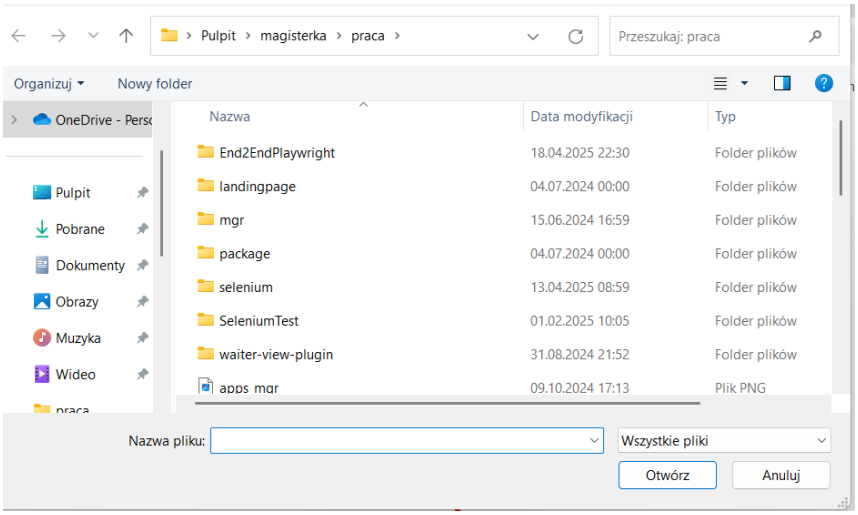
Listing 47, przedstawia sposób realizacji przesyłania dokumentu do aplikacji webowej przy wykorzystaniu biblioteki Selenium.

Listing 47 Implementacja przesyłania pliku do aplikacji webowej w Selenium

```
fileInput.sendKeys(fileToUpload.getAbsolutePath());
driver.findElement(By.id("file-submit")).click();
```

- `fileToUpload` – zmienna typu `File` wskazująca ścieżkę do pliku znajdującego się na lokalnym dysku,

- `sendKeys()` - metoda klasy `WebElement` [55] umożliwiająca symulację przesyłania wartości tekstowej do obiektu. W przedstawionym przypadku funkcja umożliwia przekazanie pełnej ścieżki bez konieczności korzystania z systemowego okna dialogowego, co pozwala na pełną automatyzację procesu przesyłania dokumentu w teście.



Rysunek 12 Okno graficzne do wpisania ścieżki z plikiem do przesłania na stronę internetową. Źródło: opracowanie własne.

Kolejnym etapem było zlokalizowanie przycisku potwierdzającego wysłanie obiektu i wywołanie na nim odpowiedniej metody wykonującej akcję. Weryfikacja poprawności procesu odbywała się poprzez sprawdzenie czy na stronie pojawiła się informacja zawierająca tytuł obiektu jaki został przesłany, co potwierdziło pomyślne zakończenie operacji (Listing 48).

Listing 48 Weryfikacja poprawności wysłania pliku w Selenium

```
driver.findElement(By.id("file-submit")).click();

Assertions.assertEquals("seleniumImage.png",
driver.findElement(By.id("uploaded-files")).getText());
```

8.1.3. Implementacja w Playwright

W bibliotece Playwright elementy strony internetowej są lokalizowane za pomocą dedykowanej klasy `Locator` [32], zaprojektowanej zgodnie z architekturą frameworka opracowanego przez firmę Microsoft. Zarówno w Playwright, jak i w Selenium, konieczne było zastosowanie obejścia opisanego w [56] umożliwiającego automatyczne wprowadzenie ścieżki pliku do systemowego okna dialogowego (Listing 49).

Listing 49 Implementacja przesyłania pliku w Playwright

```
Locator fileInput = page.locator("#file-upload");
fileInput.setInputFiles(fileToUpload.toPath());

page.locator("#file-submit").click();
Assertions.assertEquals("seleniumImage.png", page.locator("#uploaded-
files").textContent().trim());
```

- `fileInput.setInputFiles()-` metoda do ustawiania adresu pliku, przekazywanego do elementu na stronie. Przyjmuje parametr w podstaci obiektu klasy `Path`.

8.1.4. Analiza porównawcza

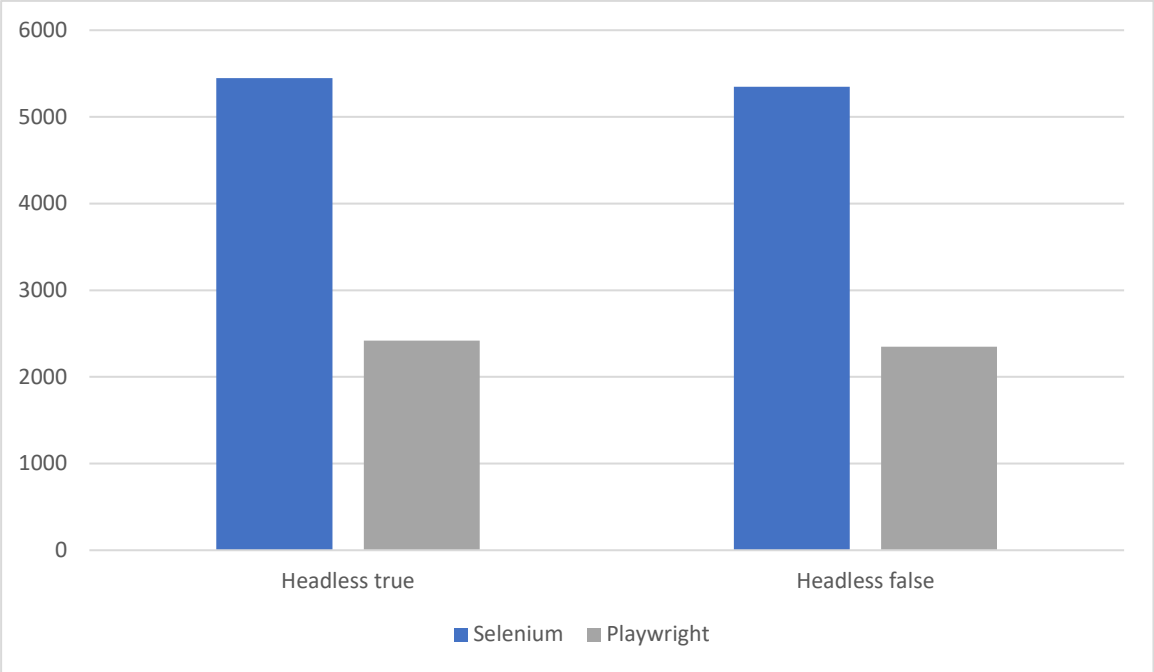
Podczas implementacji kodu w środowisku Playwright autorka pracy zauważyła znaczące podobieństwa z biblioteką Selenium w obsłudze formularzy przesyłania dokumentów. Istotną różnicą był sposób lokalizacji elementów na stronie oraz fakt, że funkcja służąca do symulacji wprowadzenia ścieżki dokumentu w Selenium przyjmuje wartość typu `String`, podczas gdy w Playwright wymaga obiektu typu `Path`.

Wykres 5 przedstawia czasy wykonania testów (w milisekundach). Podczas analizy zaobserwowano, że weryfikacje przeprowadzone z wykorzystaniem biblioteki Playwright wykonywały się niemal dwukrotnie szybciej niż analogiczne scenariusze zaimplementowane w Selenium. Różnica ta wystąpiła pomimo zbliżonego sposobu realizacji testów, które wykazywały odmienny charakter jedynie w zastosowaniu natywnych mechanizmów oraz konstrukcji właściwych dla każdej z bibliotek.

Wybór narzędzia do implementacji testów funkcjonalności przesyłania pliku na stronę nie jest w tym przypadku jednoznaczny. Autorka pracy zauważyła, że w sytuacji, gdy kluczowym kryterium jest szybkość wykonania testów, Playwright stanowi lepsze rozwiązanie. Natomiast oba narzędzia prezentują zbliżony poziom użyteczności i efektywności w przypadku, gdy priorytetem jest:

- łatwość implementacji,
- czasochłonność realizacji kodu,
- lub zużycie pamięci przez proces.

.



Wykres 5 Porównanie czasów w milisekundach wykonania przesyłania plików na stronę internetową (mniej=lepiej). Źródło: opracowanie własne.

8.2.Pobieranie pliku ze strony

W podrozdziale zawarto implementację oraz krótkie omówienie sposobów testowania automatycznego funkcjonalności pobierania plików z aplikacji webowych. Zawarto krótki scenariusz testowy (Tabela 7) oraz dokładny opis rozwiązań wykonanych przy pomocy obu narzędzi.

Testy przeprowadzono na stronie internetowej TheInternet (rozdział 4.7)

8.2.1. Scenariusz testowy

Opis scenariusza testowego dla testów pobierania pliku ze strony internetowej.

Tabela 7 Scenariusz testowy pobierania pliku ze strony internetowej. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna	Wprowadź adres ,, https://the-internet.herokuapp.com/download” do przeglądarki i przejdź do strony głównej. Wybierz jeden z dostępnych do pobrania plików. Zweryfikuj czy w podanym adresie znajduje się pobrany dokument.

		Sprawdź czy plik ma wielkość większą niż 0 Kb.

8.2.2. Implementacja w Selenium

Podczas testowania funkcjonalności pobierania plików z aplikacji internetowych, często pojawia się wyzwanie związane z obsługą systemowego okna dialogowego, które umożliwia użytkownikowi wybór lokalizacji zapisu dokumentu. W kontekście testów automatycznych jest to problematyczne, gdyż wymaga weryfikacji manualnej. Aby temu zapobiec, w przeglądarce Chromium można utworzyć konfigurację, która wyłączy wyskakujące okna i automatycznie wybierze katalog do zapisu [57]. Należy podkreślić, że takie ustawienia nie zostaną zapisane w prywatnym trybie przeglądarki (z ang. *incognito)*, ponieważ nie wykorzystuje on trwałych preferencji aplikacji webowej (Listing 50).

Listing 50 Implementacja konfiguracji w Chromium do blokowania wyskakującego okna wyboru katalogu plików w Selenium

```
public static void setChromeOptionsForDownloadFile(ChromeOptions
options,String downloadFilepath){

Map<String, Object> prefs = new HashMap<>();
prefs.put("download.default_directory", downloadFilepath);
prefs.put("download.prompt_for_download", false);

options.setExperimentalOption("prefs", prefs);
DriverUtils.setChromeOptions(options);
}
```

- `download.default_directory` – ustawienie stałej jako wartość dla podanego klucza oznacza przypisanie ścieżki do pobierania plików,

- `download.prompt_for_download` - wyłącza wyświetlanie okna dialogowego z pytaniem o potwierdzenie miejsca zapisu podczas pobierania.

Po skonfigurowaniu odpowiednich ustawień w obiekcie `ChromeOptions` [45], które następnie zostały przekazane do instancji `WebDriver` [9], aplikacja testowa zyskała możliwość blokowania systemowych dialogów. Takie rozwiązanie umożliwia pełną automatyzację procesu testowego bez konieczności manualnej interwencji użytkownika.

W kolejnym etapie implementacji zrealizowano lokalizację elementu odpowiedzialnego za inicjację pobierania pliku, a następnie wywołano na nim dedykowaną metodę, umożliwiającą rozpoczęcie operacji poboru danych w sposób kontrolowany i zautomatyzowany. W kodzie użyto wbudowanej klasy `WebDriverWait` [43] w celu przeczekania aż zakończy się pobieranie.

Weryfikację poprawności działania zaimplementowanego mechanizmu przeprowadzono poprzez dodatkowe sprawdzenie istnienia dokumentu w zadanej lokalizacji na dysku oraz potwierdzenie, że pobrany obiekt nie jest pusty, co stanowiło kryterium poprawnego zakończenia operacji (Tabela 7).

Dodatkowo, w celu zapewnienia powtarzalności wykonywania testów, zastosowano mechanizm usuwania pliku po zakończeniu testu. (Listing 51).

Listing 51 Implementacja weryfikacji pobranego pliku w Selenium

```
driver.findElement(By
            .xpath("//a[contains(@href, 'download/photo.png')]")).click();

String expectedFile = "photo.png";

File file = new File(downloadFilepath + "\\" + expectedFile);

WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(20));

wait.until(driver -> file.exists() && file.length() > 0);

Assertions.assertTrue(file.exists());
Assertions.assertTrue(file.length()>0);
file.delete();
```

- `downloadFilePath` - lokalna ścieżka do folderu z pobranymi plikami

8.2.3. Implementacja w Playwright

Biblioteka Playwright, posiada wbudowany mechanizm do nasłuchiwania i przechwytywania zdarzeń pobierania plików [58]. Rozwiązanie to znacząco ułatwia proces implementacji, eliminując konieczność dodatkowej konfiguracji oraz pozwalając skupić się bezpośrednio na pisaniu właściwego kodu testowego.

Pierwszym etapem było przejście do docelowej strony internetowej, a następnie, przy użyciu natywnego mechanizmu Playwright, określenie elementu, na którym wykonanie akcji spowoduje uruchomienie zdarzenia pobrania. (Listing 52).

Listing 52 Implementacja mechnizmu nasłuchiwania zdarzeń pobierania pliku w Playwright

```
Download download = page.waitForDownload(() -> {
    Locator fileElement = page
.locator("a[href*='download/selenium-snapshot.png']");
fileElement.click();
});
```

- `Download` – obiekt klasy reprezentujący pobrany plik wbudowany w biblitekę Playwright,
- `waitForDownload()` – metoda oczekująca na pobranie danych.

Dysponując obiektem klasy `Download` [59], autorka pracy zrealizowała operację zapisu dokumentu, wskazując lokalną ścieżkę docelową oraz pozyskując z obiektu jego nazwę. W efekcie powstał pełny adres zasobu.

Na zakończenie przeprowadzono podstawową weryfikację poprawności wykonanego procesu, sprawdzając istnienie pliku oraz potwierdzając, że jego rozmiar jest większy od zera. (Listing 53).

Listing 53 Implementacja zapisu pobranego pliku w Playwright.

```
Path path = Paths.get(downloadFilePath, download.suggestedFilename());
download.saveAs(path);

Assertions.assertTrue(Files.exists(path));
Assertions.assertTrue(Files.size(path)>0);
```

8.2.4. Analiza porównawcza

W podrozdziale przedstawiono dwie implementacje testów automatycznych funkcjonalności pobierania dokumentów na przykładzie strony internetowej TheInternet.

Pomimo realizacji tego samego scenariusza testowego (Tabela 7), podejścia zastosowane w Selenium i Playwright różnią się istotnymi aspektami technicznymi oraz stopniem automatyzacji.

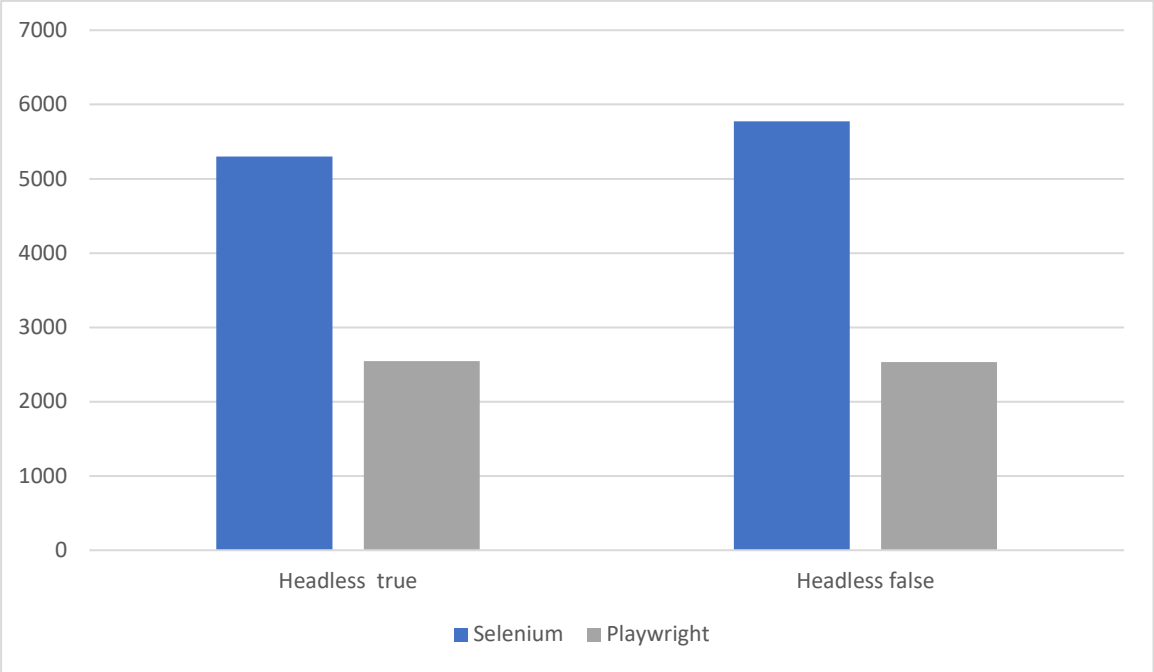
W przypadku Selenium głównym wyzwaniem była obsługa systemowego okna dialogowego wyboru lokalizacji zapisu, które w testach automatycznych stanowi przeszkodę uniemożliwiającą wykonanie procesu bez ingerencji manualnej. Aby obejść ten problem, stosuje się konfigurację przeglądarki poprzez odpowiednie ustawienia w klasie `ChromeOptions` [46], które wyłączają wyświetlanie dialogu i definiują stałą ścieżkę zapisu.

Z kolei Playwright oferuje natywny i wbudowany mechanizm nasłuchiwania oraz przechwytywania zdarzeń pobierania plików, co znacząco upraszcza proces implementacji. Metoda `waitForDownload()` pozwala na bezpośrednie oczekiwanie na zdarzenie pobrania bez konieczności konfigurowania przeglądarki czy obsługi dialogów systemowych. Po zakończeniu wykonywania procesu dostępny jest obiekt klasy `Download` [59], który umożliwia zapis pliku na dysk pod wskazany adres.

Aby dopełnić podsumowanie analizy, każdy z testów poddano trzykrotnemu wykonaniu, a średnią zmierzonych czasów w milisekundach zarejestrowano na wykresie ilustrującym różnice pomiędzy dwoma trybami pracy: z włączonym interfejsem graficznym oraz bez niego (Wykres 6).

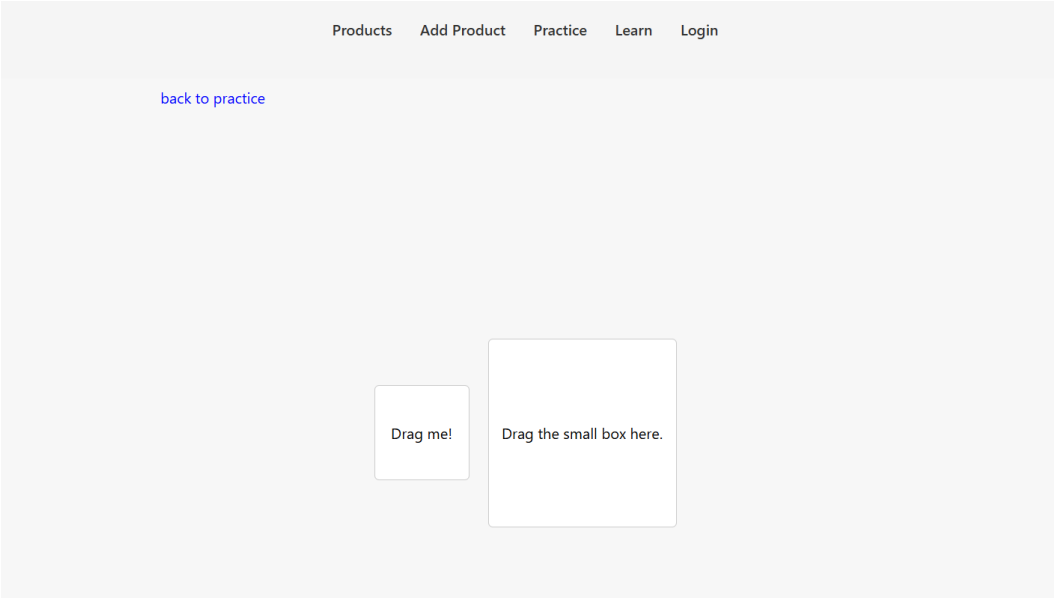
Analiza wyników badania jednoznacznie wskazuje, że Playwright jest lepszym rozwiązaniem od Selenium pod względem:

- krótszego wykonania testów,
- wygodniejszej implementacji,
- mniejszej podatności na błędy.



Wykres 6 Porównanie czasów wykonania w milisekunach przy pobieraniu pliku (mniej=lepiej). Źródło: opracowanie własne.

8.3.*Drag-and-Drop*



Rysunek 13 Witryna strony do testowania *Drag-and-Drop*. Źródło: [24]

Akcja przeciągnij i upuść (z ang *Drag-and-Drop*) stanowi coraz powszechniej wykorzystywaną funkcjonalność interfejsów użytkownika, oferującą intuicyjne i angażujące doświadczenia. Najczęściej znajduje zastosowanie w grach komputerowych, gdzie umożliwia przenoszenie elementów pomiędzy różnymi obiektami.

W ramach przeprowadzonych testów wykorzystano aplikację webową CommitQuality.com (rozdział 4.9), w której zdefiniowano dwa elementy na stronie umożliwiające testowanie funkcjonalności jaką jest *Drag-and-Drop.*

8.3.1. Scenariusz testowy

Opis scenariusza testowego dla testów operacji przeciągnij i upuść.

Tabela 8 Scenariusz testowy operacji *Drag-and-Drop*. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna	Wprowadź adres „https://commitquality.com/practice-drag-and-drop" do przeglądarki i przejdź do strony głównej. Znajdź element A i B na stronie. Przeciągnij obiekt A do elementu B i upuść. Zweryfikuj czy w zakresie obiektu B pojawił się tekst *„Success"*

8.3.2. Implementacja w Selenium

Narzędzie Selenium umożliwia realizację funkcjonalności *Drag-and-Drop* za pomocą wbudowanej klasy `Actions` [60], która pozwala na symulację interakcji użytkownika z interfejsem aplikacji webowej. Listing 54 przedstawia przykład implementacji z zastosowaniem tego obiektu. Wykorzystana metoda `dragAndDrop()` cechuje się wysoką czytelnością oraz prostotą implementacji, jednakże w wybranych przypadkach jej zastosowanie może być ograniczone ze względu na konieczność dodatkowej konfiguracji po stronie testowanej aplikacji webowej. Klasa `Actions` jest polecana aby zastępować nią symulacje klawiatury lub myszy.

Listing 54 Implementacja *Drag-and-Drop* z użyciem klasy *Actions* w Selenium

```
WebElement fileInput = driver.findElement(By.id("small-box"));
WebElement dropZone = driver.findElement(By.className("large-box"));

Actions actions = new Actions(driver);
actions.dragAndDrop(fileInput, dropZone).perform();

Assertions.assertEquals("Success!",dropZone.getText());
```

- `fileInput` – zdefiniowanie pierwszego elementu A,
- `dropZone` – zmienna odpowiadająca obiektowi B (Tabela 8).
- `dragAndDrop().perfom()` – metoda umożliwiająca wykonywanie akcji przeciągnij i upuść.

8.3.3. Implementacja w Playwright

Biblioteka Playwright udostępnia wbudowaną metodę `dragTo()` w klasie *Locator* [32], umożliwiającą symulację akcji przeciągania i upuszczania elementów na stronie internetowej. Dzięki temu możliwe jest efektywne odwzorowanie interakcji użytkownika bez konieczności implementowania złożonych procedur.

Przykładowy kod obejmuje przeciągnięcie elementu o identyfikatorze *„small-box”* na element *„large-box”*, co jest realizowane przez wywołanie odpowiedniej metody. Na zakończenie wykonuje się asercję, weryfikującą czy na docelowym obiekcie pojawiła się oczekiwana zmiana stanu, potwierdzająca poprawne wykonanie operacji (Listing 55).

Rysunek 13 przedstawia widok strony testowanej.

Listing 55 Implementacja *Drag-and-Drop* w Playwright

```
page.getByTestId("small-box").dragTo(page.getByTestId("large-box"));

Assertions.assertEquals("Success!",page.getByTestId("large-box")
.textContent());
```

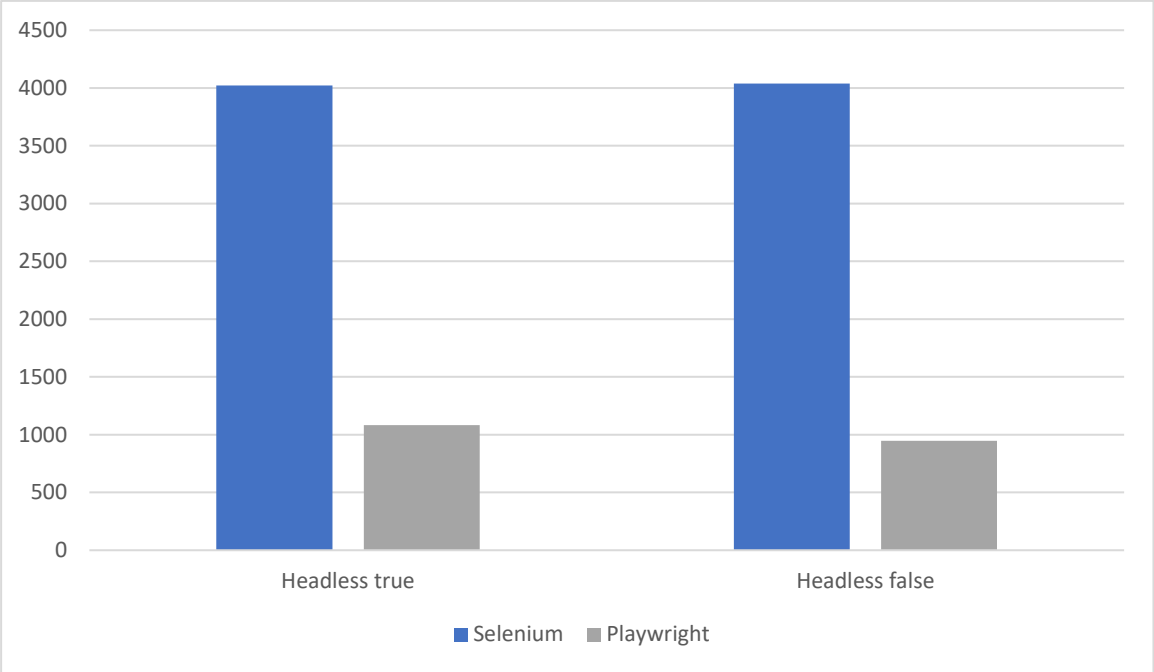
8.3.4. Analiza porównawcza

Obie biblioteki udostępniają wbudowane mechanizmy do osiągnięcia tego samego celu jakim jest symulacja akcji podnoszenia i upuszczania przez użytkownika.

Selenium umożliwia operację *Drag-and-Drop*, poprzez wbudowaną klasę `Actions` [60]. Użycie obiektu tej klasy oferuje proste rozwiązanie, które potrzebuje jedynie zdefiniowania obiektu źródła oraz celu. Niekiedy jednak może się okazać potrzeba wprowadzenia dodatkowej konfiguracji przez testera co może sprawić pewne trudności w użyciu tego prostego rozwiązania.

Playwright udostępnia metodę wbudowaną w obiekt klasy `Page` [42], którego nie trzeba dodatkowo formułować z uwagi na wcześniejsze jego zdefiniowanie chociażby do przekazania parametru *URL* strony, na której wykonywany jest test. Implementacja należy do jednej z prostszych, a użycie funkcji do wykonania akcji *Drag-and-Drop*, nie potrzebuje dodatkoweych ustawień w żadnym środowsku co powoduje, że jej użycie jest bardzo uniwersalne.

Biorąc pod uwagę wykres przedstawiający czas trwania testów przy użyciu obu narzędzi, autorka pracy zauważyła, że wykonanie w Playwright było dużo szybsze niż w testach wykonanych w Selenium (Wykres 7). Czyni to bibliotekę Playwright lepszym rozwiązaniem dla funkcjonalności *Drag-and-Drop.*



Wykres 7 Porównanie czasów wykonania w milisekundach dla testów typu *Drag-and-Drop* (mniej=lepiej). Źródło: opracowanie własne.

9. Testy obsługi Snapshot Aria

W niniejszym rozdziale przedstawiono sposób, w jaki wybrane frameworki do testowania automatycznego radzą sobie z obsługą mechanizmu *Snapshot Aria* [61]. W szczególności omówiono implementację krótkiego scenariusza testowego z wykorzystaniem bibliotek Selenium oraz Playwright na podstawie strony dokumentacji jednego z narzędzi, a także dokonano analizy uzyskanych wyników.

Na zakończenie rozdziału zamieszczono podsumowanie zrealizowanej implementacji oraz wnioski wynikające z przeprowadzonych testów, uzupełnione wykresem prezentującym średni czas ich wykonania.

Snapshot Aria, stanowi zapis struktury dostępności danej strony internetowej w określonym momencie. Reprezentuje on sposób, w jaki przeglądarka interpretuje semantykę elementów *HTML* wraz z atrybutami *ARIA* (z ang. *Accessible Rich Internet Applications*). Najczęściej zapis ten przyjmuje postać hierarchicznego układu w formacie *YAML*, odzwierciedlającego kompozycję elementów dostępnych dla technologii asystujących, takich jak czytniki ekranu.

Zastosowanie *Snapshot Aria* jest szczególnie istotne w procesach testowania dostępności aplikacji internetowych, gdyż umożliwia porównanie aktualnej struktury dostępności z jej oczekiwanym stanem. W efekcie możliwe jest szybkie wykrycie niezamierzonych zmian lub błędów w implementacji, które mogłyby wpłynąć na komfort i możliwości korzystania z serwisu przez osoby z niepełnosprawnościami.

9.1. Scenariusz testowy

Opis scenariusza testowego dla testów *Snapshot Aria*.

Tabela 9 Scenariusz testowy dla testów implementacji *Snapshot Aria*. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna	Wprowadź adres „https://playwright.dev/java/” do przeglądarki i przejdź do strony głównej. Zweryfikuj, że Snapshot Aria posiada wartości: "- banner:\n - heading \"Playwright enables reliable end-to-end testing for modern web apps." + "\" [level=1]\n - link \"Get started\"\n - link \"Star microsoft/playwright-java on GitHub\"\n - link /[\\d,.]+[bkmBKM]+\\+ stargazers on GitHub/"

9.2. Implementacja w Selenium

Framework Selenium nie oferuje natywnego wsparcia dla obsługi oraz weryfikacji *Snapshot Aria* [61]. W związku z tym jedyną dostępną metodą sprawdzenia zawartości strony internetowej w kontekście dostępności jest wykorzystanie podstawowych mechanizmów przeszukiwania struktury *DOM*. Podejście to polega na ręcznym wyszukiwaniu oraz weryfikowaniu wybranych elementów na podstawie ich właściwości i atrybutów.

W pierwszym etapie implementacji zaprojektowanego scenariusza testowego (Tabela 9) dokonano wyszukania elementu nagłówkowego strony na podstawie nazwy znacznika *HTML* (Listing 56).

Listing 56 Implementacja wyszukiania elementu po nazwie tagu w Selenium

```
WebElement headerElement = wait.until(ExpectedConditions
         .visibilityOfElementLocated(By.tagName("header")));
assertNotNull(headerElement, "Header element is missing.");
```

- `tagName()` – tag *HTML*, na podstawie którego wyszukano nagłówek.

W kolejnym etapie implementacji testu przeprowadzono zlokalizowanie drugiego elementu nagłówka pierwszego stopnia (z ang. *heading level one*) w obrębie wcześniej odnalezionej sekcji. Celem tej operacji była weryfikacja, czy zawartość tekstowa wskazanego obiektu odpowiada wartości oczekiwanej, zgodnej z projektem strony oraz specyfikacją dostępności (Listing 57).

Listing 57 Implementacja wyszukiwania elementu nagłówka w teście *Snapshot Aria* w Selenium

```
WebElement heading = headerElement
           .findElement(By.cssSelector("h1.hero__title"));
assertNotNull(heading, "Heading is missing.");
assertEquals("Playwright enables reliable end-to-end testing for modern web
apps.", heading.getText(), "Heading text does not match.");
```

- `By.cssSelector()`- selektor umożliwiający znalezienie elementu na podstawie atrybutu przekazanego w argumencie.

Kolejnym istotnym przypadkiem testowym było sprawdzenie obecności oraz poprawności linku przekierowującego do sekcji wprowadzającej w dokumentacji frameworka Playwright. W tym celu przeprowadzono operację wyszukania elementu odnośnika znajdującego się w obrębie głównego nagłówka strony. Następnie pobrana została wartość atrybutu `href` bezpośrednio z odnalezionego obiektu, co pozwoliło na sprawdzenie, czy adres *URL* odpowiada wartości oczekiwanej (Listing 58).

Listing 58 Implementacja pobierania atrybutu bezpośrednio z obiektu wyszukanego na stronie w Selenium

```
WebElement getStartedLink = headerElement
                 .findElement(By.className("getStarted_Sjon"));
assertNotNull(getStartedLink, "Get Started link is missing.");
assertEquals("https://playwright.dev/java/docs/intro",
            getStartedLink.getAttribute("href"),
             "'Get Started' link URL is incorrect.");
```

- `getStartedLink.getAttribute()` – metoda służąca do pobierania atrybutu z instancji klasy `WebElement` [55].

Pozostałe weryfikacje elementów znajdujących się w obrębie głównego obiektu strony, zgodnie z założeniami scenariusza testowego (Tabela 9), zostały zaimplementowane w analogiczny sposób.

Każda z operacji polegała na wyszukaniu odpowiedniego bytu przy wykorzystaniu selektorów *CSS*, a następnie przeprowadzeniu weryfikacji jego obecności bądź wartości wybranych atrybutów. Ze względu na konieczność osobnego adresowania poszczególnych elementów oraz powtarzalny charakter operacji, ostateczna objętość kodu testowego wzrosła do dwudziestu linii w edytorze.

9.3.Implementacja w Playwright

W przypadku implementacji testów przy użyciu frameworka Playwright, wykorzystano jego natywne mechanizmy wspierające weryfikację dostępności stron internetowych.

W Playwright *Aria Snapshot* [62] jako mechnizm reprezentuje drzewo strony w formacie *YAML*, który odzwierciedla sposób w jaki przeglądarka interpretuje strukturę witryny.

Do przeprowadzenia walidacji zawartości nagłówka wykorzystano wbudowaną metodę frameworka, która umożliwia sprawdzenie, czy analizowany układ strony zawiera atrybuty przekazane jako argumenty funkcji. Przykładowy test został zaimplementowany zgodnie z zalecanymi praktykami stosowanymi w bibliotece Playwright.

W celu zapewnienia poprawnego wywołania oraz weryfikacji działania metody `matchesAriaSnapshot()` wykorzystano natywny framework testowy dostarczany wraz z Playwright, co umożliwiło rzetelne i spójne przeprowadzenie ostatecznych asercji w testach dostępności. Ostateczna implementacja kodu zajęła cztery linijki w edytorze. (Listing 59).

Listing 59 Implementacja testu *Snapshot Aria* w Playwright

```
assertThat(page
        .getByRole(AriaRole.BANNER))
        .matchesAriaSnapshot(
"- banner:\n  - heading \"Playwright enables reliable end-to-end testing for
modern web apps." +"\" [level=1]\n  - link \"Get started\"\n  - link \"Star
microsoft/playwright-java on GitHub\"\n    - link /[\\d,.]+[bkmBKM]+\\+
stargazers on GitHub/");
```

- `AriaRole.BANNER` - rola elementu, do którego będzie odnosił się kod, w tym przypadku wyszukiwany jest obiekt o charakterze oznaczającym początek strony,

- `matchesAriaSnapshot()` – wbudowana metoda do sprawdzania poprawności *Aria Snapshot.*

9.4.Analiza porównawcza

Analiza implementacji testowania *Snapshot Aria* przy użyciu narzędzi Selenium oraz Playwright, wykazała istotne różnice w podejściu oraz dostępnych możliwościach obu rozwiązań.

Playwright, dzięki natywnemu wsparciu dla mechanizmu *Snapshot Aria*, umożliwia szybkie i efektywne porównywanie aktualnej kompozycji strony z wcześniej zdefiniowanym wzorcem. Rozwiązanie to wpływa korzystnie na czytelność implementowanych testów, ograniczając jednocześnie liczbę instrukcji niezbędnych do weryfikacji złożonej struktury *DOM*.

W przeciwieństwie do tego, Selenium nie oferuje wbudowanej obsługi *Snapshot Aria*, co wymusza ręczne wyszukiwanie poszczególnych elementów oraz indywidualne sprawdzanie ich właściwości i wzajemnych relacji. Mimo iż takie podejście zapewnia pełną kontrolę nad przebiegiem testu, wiąże się z większym nakładem pracy, ryzykiem popełnienia błędów oraz mniej przejrzystym kodem.

Pod względem wydajności pracy oraz intuicyjności tworzenia testów przewagę wykazuje Playwright. Należy jednak zauważyć, że rozwiązanie to cechuje się ograniczoną skalowalnością oraz podatnością na błędy w przypadku istotnych zmian w strukturze strony internetowej. Z kolei

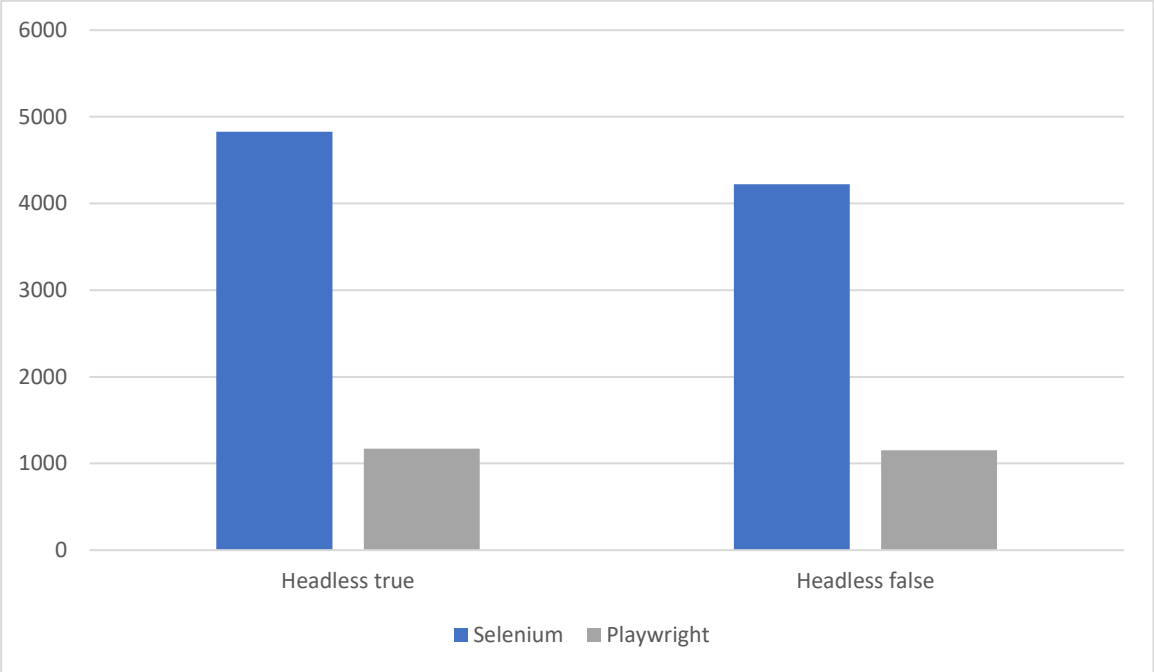
elastyczność oraz możliwość szczegółowego dostosowania zakresu ewentualnych modyfikacji na stronie przemawia na korzyść Selenium, co czyni to narzędzie odpowiednim w sytuacjach wymagających pełnej kontroli nad procesem.

W celu dopełnienia analizy porównawczej autorka pracy, przeprowadziła trzykrotne wywołanie obu testów, a następnie dokonała zapisu średnich czasów ich wykonania, wyrażonych w milisekundach. Uzyskane wyniki zostały przedstawione w formie graficznej, umożliwiającej przejrzyste porównanie efektywności obu rozwiązań (Wykres 8).

Na podstawie zestawienia czasowego oraz podsumowania wniosków, autorka pracy stwierdziła, że Playwright stanowi korzystniejsze rozwiązanie w zakresie:

- czasu realizacji testów,
- prostoty oraz szybkości implementacji,
- ograniczenia ryzyka popełnienia błędów przez testera.

Natomiast gorzej wypada w przypadku projektów, w których kluczowym kryterium jest wysoka skalowalność kodu.



Wykres 8 Porównanie czasów wykonania testów *Aria Snapshot* w milisekundach (mniej=lepiej). Źródło: opracowanie własne.

10. Testy przełączania między zakładkami

Przełączanie między otwartymi zakładkami stanowi jeden z podstawowych elementów interakcji użytkownika z aplikacjami webowymi. Ze względu na istotny wpływ tego mechanizmu na doświadczenie konsumenta, podjęto decyzję o przetestowaniu tej funkcjonalności oraz porównaniu sposobów jej implementacji w narzędziach Playwright i Selenium.

W ramach tego rozdziału wykonano testy na dynamicznej stronie komercyjnej Orange.pl (rozdział 4.1). Treść tej sekcji zawiera scenariusz testowy (Tabela 10) opisujący podjęte przez automatyczną weryfikację kroki, oraz implementację, która zawiera tylko najważniejsze fragmenty testów.

Na koniec autorka pracy zawarła krótką analizę obu narzędzi pod kątem przełączania się między oknami serwisu wraz z wykresem graficznym przedstawiającym czasy wykonania procesów.

10.1. Scenariusz testowy

Opis scenariusza testowego dla przełączania się pomędzy zakładkami.

Tabela 10 Scenariusz testowy dla przełączania się pomiędzy zakładkami. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna	Wprowadź adres https://www.orange.pl/ do przeglądarki i przejdź do strony głównej. Zaakceptuj ciasteczka. Kliknij przycisk „Światłowód” i otwórz sekcję w nowej karcie.
2.	Strona internetu domowego	Otwórz nową kartę wciskając : „Przedłuż umowę”.
3.	Strona przedłużenia internetu domowego	Zweryfikuj, że na ostatniej stronie widoczny jest przycisk do logowania. Przejdź do pierwszej zakładki.
4.	Strona główna	Zweryfikuj, że aktualna strona, na której się znajdujesz posiada *URL* : „https://www.orange.pl/”

10.2. Implementacja w Selenium

W zależności od implementacji danej strony internetowej, kliknięcie w odnośnik może skutkować otwarciem nowej karty lub pozostaniem w tym samym oknie przeglądarki podmieniając jej zawartość. W przypadku testowanej strony Orange.pl, łącza, których dotyczy test, nie powodują automatycznego otwarcia nowej zakładki. Z tego względu konieczne było ręczne zasymulowanie uruchomienia wskazanego adresu w nowym oknie.

Selenium posiada wbudowną klasę `Actions` [60], która została już użyta w implementacji do umożliwienia symulacji *Drag-and-Drop* (Listing 54). Klasa ta udostępnia szereg innych metod między innymi pozwala na bezpośrednie otwarcie strony w nowym oknie poprzez symulację wciśnięcia przycisku *CTRL* w połączeniu z lewym klawiszem myszy. Należy zaznaczyć, że to rozwiązanie jest nie możliwe dla przeglądarki Safari z uwagi na brak przycisku *CTRL* na klawiaturze w systemie *macOS*. W tym celu przedstawiono drugi sposób jakim jest użycie czystego fragmentu kodu *JavaScript* wywołanego poprzez interfejs `JavascriptExecutor` [64].

Listing 60, przedstawia wywołanie funkcji *JavaScript*, która została zastosowana do implementacji wciśnięcia przycisku „Światłowód", na stronie startowej. W pierwszym etapie pobrano adres *URL* z atrybutu `href` wskazanego odnośnika, a następnie przy użyciu metody `executeScript()` wywołano funkcję `window.open()`, przekazując do niej odczytany adres oraz parametr `'_blank'`, który odpowiada za otwarcie zasobu w nowej karcie.

Listing 60 Użycie *JavaScriptExecutor* do dodania atrybutu *'blank'* w Selenium

```
String url=Driver
            .findElement(By.xpath("//a[@href=\"/internet/internet-
            domowy\"]"))
            .getAttribute("href");

((JavascriptExecutor)driver)
      .executeScript("window.open(arguments[0], '_blank');", url);
```

- `getAttribute()` – metoda pobierająca atrybutów podany jako parametr z elementu strony,
- `executeScript()` – wywołuje funkcję *JavaScript*,
- `window.open(arguments[0], '_blank')` - określa otwarcie nowej karty, której *URL* podany jest jako pierwszy argument.

Po otwarciu nowego okna przeglądarki w trakcie wykonywania testów automatycznych, niezbędne jest poinformowanie obiektu `WebDriver` [9], który kontekst powinien być aktywny. W przeciwnym razie wszystkie kolejne akcje wykonywane byłyby w ramach pierwotnie otwartej strony, co mogłoby prowadzić do błędnie wykonanych testów lub interakcji z nieaktualnym widokiem. Implementacja została zrealizowana poprzez zastosowanie metody pomocniczej, w której zawarty jest mechanizm przełączania się między zakładkami przeglądarki (Listing 61).

Listing 61 Przełączanie sterowania *WebDriver* na nowo otwartą kartę

```
private String switchToNewWindow(ChromeDriver driver, int numberOfWindows){

WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(5));

wait.until(ExpectedConditions.numberOfWindowsToBe(numberOfWindows));

Set<String> windows = driver.getWindowHandles();

String lastWindow = windows
                  .toArray(new String[0])[windows.size() - 1];

driver.switchTo().window(lastWindow);
```

```
return lastWindow;
}
```

- *driver.getWindowHandles()* – metoda zwracająca identyfikatory wszystkich otwartych stron w ramach jednej sesji. [65]

- *driver.switchTo()* – umożliwia przełączenie się do podanej jako argument zakładki.

Listing 61, przedstawia implementację funkcji, która zwraca ostatnie aktywne okno.

Na etapie inicjalizacji sesji przeglądarki, za pomocą metody `getWindowHandle()`, pobrano identyfikator pierwszego aktywnego okna serwisu i przypisano go do zmiennej obiektu `String` (Listing 62), aby umożliwić wykonanie ostatniego kroku scenariusza testowego (Tabela 10), który sprawdza poprawność adresu *URL*.

Listing 62 Zapis identyfikatora pierwszej otwartej zakładki w Selenium.

```
String mainWindow = driver.getWindowHandle();
```

W ramach kolejnych etapów zaimplementowano mechanizm wciśnięcia zestawu poleceń *CTRL* z lewym klawiszem myszy (Listing 63).

Listing 63 Implementacja otwarcia karty w nowym oknie z użyciem obiektu klasy *Actions* w Selenium.

```
String prelongXPath = "//span[contains(@class, 'flex grow items-center
justify-center w-full py-2 text-size-12-bold') and text()='Przedłużam
umowę']";

WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(20));
wait.until(ExpectedConditions
            .elementToBeClickable(By.xpath(prelongXPath)));

Actions actions = new Actions(driver);

actions.keyDown(Keys.CONTROL)
.click(driver.findElement(By.xpath(prelongXPath)))
.keyUp(Keys.CONTROL).perform();
```

- `keyDown(Keys.CONTROL)` – metoda symulująca wciśnięcie i przytrzymanie klawisza klawiatury określonego w parametrze,

- `click()` – imituje zwykłe kliknięcie lewrgo klawisza myszy,

- `keyUp(Keys.CONTROL)` – funkcja symulująca zwolnienie wciśniętego przycisku na klawiaturze

- `perform()` – aktywuje wybrany zestaw przycisków.

Po przełączeniu okna na ostatnią zakładkę zatytułowaną „Przełużenie umowy Internetu domowego”, dokonano weryfikacji widoczności przycisku logowania na stronie, korzystając z metody opracowanej przez autorkę pracy (Listing 64).

Listing 64 Weryfikacja w Selenium czy tytuł strony „O nas" jest widoczny.

```
public boolean isHeadingVisible() {
    new WebDriverWait(driver, Duration.ofSeconds(10))
      .until(ExpectedConditions
      .visibilityOfElementLocated(By
            .id("Zyskaj_2×0_Zaloguj_się_GloriousSG_Button")));

return this.driver.findElement(By
      .id("Zyskaj_2×0_Zaloguj_się_GloriousSG_Button"))
      .isDisplayed();
}

Assertions.assertTrue(internet.isHeadingVisible());
```

Po zakończeniu weryfikacji widoczności przycisku logowania, konieczne było przywrócenie kontekstu przeglądarki do pierwotnej zakładki. W tym celu wykorzystano metodę do przełączania się pomiędzy oknami `switchTo().window(mainWindow)`, umożliwiającą przełączenie się do obiektu, którego identyfikator został zapisany w poprzednich krokach (Listing 62).

Następnie, w ramach walidacji poprawności przeprowadzonej operacji, sprawdzono, czy aktualny adres *URL* odpowiada wartości oczekiwanej. Pomyślne przejście tego etapu potwierdza prawidłowe przywrócenie kontekstu oraz zgodność adresu *URL* ze stanem przewidzianym w scenariuszu testowym (Listing 65).

Listing 65 Weryfikacja adresu URL przy przełączaniu zakładek w Selenium

```
Assertions.assertEquals("https://www.orange.pl/", driver.getCurrentUrl());
```

- `getCurrentUrl()` – metoda pobierająca aktualnie ustawione *URL* w obiekcie *WebDriver [9]*

10.3. Implementacja w Playwright

Podstawową konstrukcją reprezentującą stronę internetową w Playwright jest klasa `Page` [42]. Po otwarciu głównej zakładki tworzony jest obiekt tej klasy, umożliwiający wykonywanie operacji na danym zasobie internetowym. W przypadku otwarcia nowego okna i rozpoczęcia interakcji z kolejną stroną, należy utworzyć instancję klasy `Page`.

Nie każdy dostępny odnośnik przenosi użytkownika do nowo otwartej karty. Aby testy automatyczne były możliwe autorka pracy zastosowała dwa podejścia, które zostały umożliwione przez dostępność funkcji w Playwright.

Pierwszą z nich jest kliknięcie w link na stronie poprzez symulację kliknięcia środkowego przycisku myszy [66] (Listing 66).

Listing 66 Implementacja wciśnięcia środkowego klawisza myszki

```
page.locator("//a[@href='/internet/internet-domowy'
      and contains(normalize-space(.), 'Światłowód')]")
      .click(new Locator.ClickOptions()
      .setButton(MouseButton.MIDDLE));
```

- *Normalize-space(.)* – metoda umożliwiająca ignorowanie białych znaków podczas wyszukiwania tekstu,

- *click().setButton()* – wykonuje kliknięcie w element. W tym przykładze wraz dodatkową opcją imitacji kliknięcia w środkowy klawisz myszy co w większości przeglądarek oznacza otwarcie elementu w nowej karcie.

Drugim sposobem, jaki zastosowano w implementacji, jest kliknięcie w link przy użyciu symulacji wciśnięcia określonego zestawu klawiszy modyfikujących.W tym przypadku wykorzystano kombinację *CTRL*, która w przeglądarkach opartych na silniku *Chromium* umożliwia otwarcie odnośnika w nowej zakładce (Listing 67). Należy jednak zaznaczyć, że rozwiązanie to nie jest w pełni uniwersalne, gdyż w przeglądarce *Safari* (bazującej na silniku *WebKit*), wykorzystywanej w systemach *macOS*, brak klawisza *CTRL* powoduje, że taka symulacja nie działa poprawnie.

Listing 67 Implementacja symulacji zestawu klawiszy *CTRL* w Playwright

```
this.page.getByRole(AriaRole.LINK,new
         Page.GetByRoleOptions().setName("Przedłużam umowę"))
         .click(new Locator.ClickOptions()
           .setModifiers(List.of(KeyboardModifier.CONTROL)));
```

- `setModifiers(List.of(KeyboardModifier.CONTROL))`- imituje przytrzymanie klawisza *CTRL* i kliknięcie w link co powoduje otwarcie odnośnika w nowej karcie.

Playwright udostępnia wbudowaną metodę `waitForPage()`, służącą do obsługi zdarzeń z otwarciem nowej strony. Działa ona w kontekście aktualnie używanego obiektu `BrowserContext` [40], oczekując na zdarzenie wygenerowane przez określoną akcję. Metoda ta automatycznie tworzy nowy obiekt klasy *Page* dla każdej otwartej strony (Listing 68).

Listing 68 Implementacja utworzenia nowego obiektu Page

```
Page newPage1 = context.waitForPage(orangePage::clickInternetLink);
```

- `clickInternetLink`- Listing 66 przedstawia metodę wykonaną przez autorkę pracy. Odpowiada za kliknięcie w link na stronie Orange.pl

Po utworzeniu obiektu, zastosowano funkcję, która oczekuje na zakończenie pobierania wszystkich potrzebnych zasobów, aby upewnić się, że elementy zostaną zaczytane przed wykonaniem kolejnych akcji [49]. Metoda przydaje się najbardziej przy testowaniu stron dynamicznych, pozwala zminimalizować ryzyko błędów wynikających z niepełnego załadowania strony oraz poprawia stabilność wykonywanych testów automatycznych (Listing 69).

Listing 69 Implementacja oczekiwania na pełne załadowanie nowej strony internetowej w Playwright

```
newPage1.waitForLoadState(LoadState.NETWORKIDLE);
```

Należy pamiętać aby każdą kolejną akcję wykonywać na nowo powstałym obiekcie, co pozwala uniknąć wyjątków *NullPointerException,* które mogłyby wystąpić w przypadku odwoływania się do nieistniejących elementów strony internetowej.

Klasa `Page` oferuje szereg metod ułatwiających zarządzanie i interakcję w aplikacjach webowych. Jedną z nich jest wbudowana funkcja `bringToFront()`, pozwalająca na przeniesienie wybranej strony

na początek, imitując w ten sposób kliknięcie użytkownika w odpowiednią zakładkę. W implementacji wywołano tą funkcję na pierwszym obiekcie klasy `Page` aby aktywować stronę startową (Listing 70).

Listing 70 Implementacja przechodzenia na pierwszą otwartą stronę w Playwright

```
page.bringToFront();
page.waitForURL("https://www.orange.pl/");
```

Dodatkowo, w celu weryfikacji poprawności działania oraz zachowania integralności danych w sesji testowej, zastosowano asercję porównującą aktualny adres *URL* strony z oczekiwaną wartością (Listing 71).

Listing 71 Weryfikacja czy adres URL jest taki sam jak adres URL pierwszo otwartej strony

```
Assertions.assertEquals("https://www.orange.pl/", page.url());
```

- `url()` – metoda do pobierania *URL* z obiektu klasy *Page*

10.4. Analiza porównawcza

W Selenium podstawowy obiekt zarządza sesją przeglądarki jako pojedynczą instancją. W przypadku otwierania nowej karty narzędzie to nie tworzy osobnego obiektu reprezentującego aktualny widok. Zamiast tego konieczne jest ręczne pobranie identyfikatorów wszystkich otwartych zakładek oraz przełączanie się między nimi. Takie podejście wymaga dodatkowej logiki, kontrolującej liczbę aktywnych okien oraz ich kolejność w trakcie trwania sesji.

W odróżnieniu od tego, w Playwright każda strona otwarta w przeglądarce reprezentowana jest przez osobny obiekt klasy `Page`. Dzięki temu praca z wieloma kartami staje się bardziej przejrzysta i intuicyjna.

W kwestii weryfikacji aktualnego adresu *URL* oraz stanu załadowania strony również występują istotne różnice.

W Selenium należy upewnić się, że `WebDriver` znajduje się w odpowiednim kontekście okna, a następnie odczytać bieżący adres za pomocą metody `getCurrentUrl()`. W Playwright każdy obiekt `Page` posiada własną metodę *`url()`*, co pozwala bezpośrednio sprawdzić aktualną ścieżkę przypisaną do danej strony, bez konieczności zmiany kontekstu. Dodatkowo Playwright udostępnia metodę `waitForLoadState()`, która gwarantuje, że wszystkie niezbędne zasoby zostały w pełni załadowane przed wykonaniem dalszych operacji. Minimalizuje to ryzyko błędów wynikających z pracy na stronie, która nie jest jeszcze w pełni gotowa do interakcji.

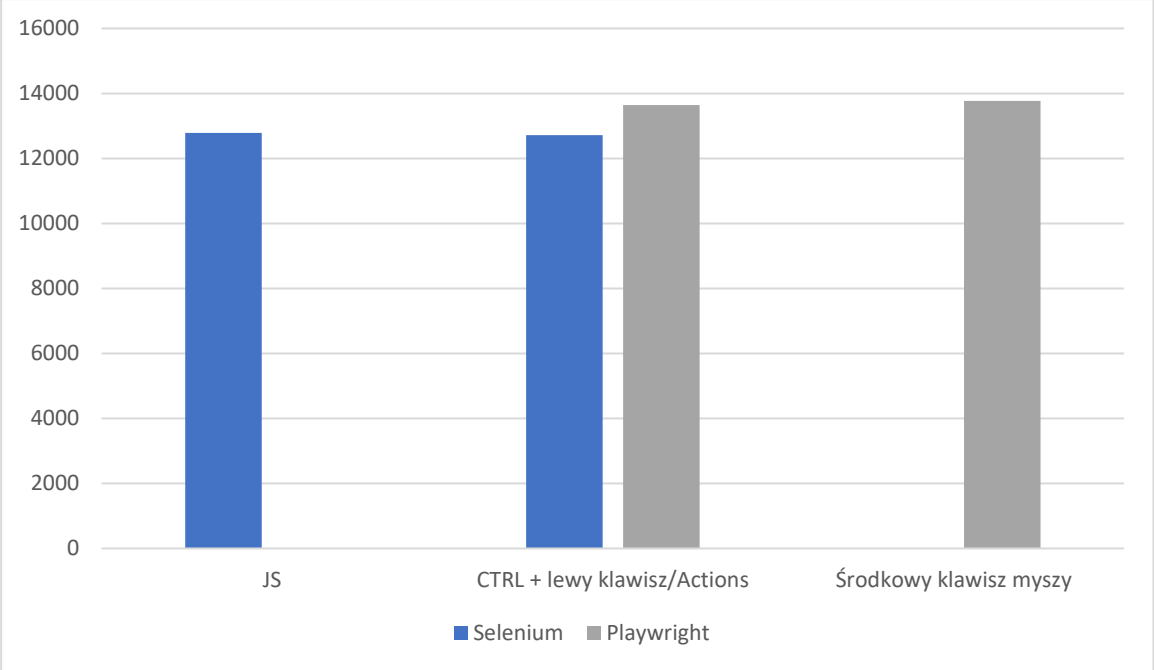
Na zakończenie analizy autorka pracy przeprowadziła zestawienie czasów wykonania, wyrażonych w milisekundach, porównując trzykrotne wykonanie każdego z analizowanych przypadków testowych (Wykres 9).

Szczegółowe badanie wskazuje, że w przypadku biblioteki Selenium średni czas realizacji działań za pomocą skryptów *JavaScript* nie różni się istotnie od czasów uzyskanych przy użyciu klasy `Actions`. Podobnie w przypadku biblioteki Playwright, okresy wykonania akcji związanych z otwieraniem nowych zakładek wykazują niewielkie różnice.

Dodatkowo podczas odpalania testów autorka pracy śledziła zużycie pamięcie *CPU*, w przeprowadzonych testach na komputerze z procesorem *Intel i7-8550U*. Dla Selenium maksymalne użycie procesora wynosiło 80%, podczas gdy dla Playwright osiągnęło *70%* w analogicznych scenariuszach testowych. Wyniki mogły być spowodowane działaniem w kontekście strony dynamicznej i komercyjnej, trudniejszej do opanowania przez testy automatyczne niż witryna statyczna lub stworzona na potrzeby testów.

Autorka pracy zauważyła, że proces przełączania się pomiędzy zakładkami był dużo lepszy dla frameworka Playwright pod względem :

- krótszego czasu wykonania,
- mniejszego zużycia pamięci,
- oraz łatwiejszej implementacji.



Wykres 9 Porównanie czasów wykonania w milisekundach przy testach przełączania się między zakładkami (mniej=lepiej). Źródło: opracowanie własne.

11. Testy symulacji zmian sieci

W tym rozdziale skupiono się na porównaniu, jak narzędzia Selenium oraz Playwright radzą sobie z implementacją symulacji warunków sieciowych (z ang. *throttling*). Zmiana rodzaju połączenia, które wykorzystuje przeglądarka jest kluczowym aspektem wpływającym na doznania użytkownika. Celem wykonywania testów ze zmienną infrastukturą jest pozyskanie informacji takich jak:

- analiza obsługi buforowania danych przez aplikację,
- czy czas oczekiwania na pobranie zasobów jest akceptowalny,
- ocena ciągłości działania funkcjonalności w aplikacji.

Testy przeprowadzono na stronie SauceDemo.com (rozdział 4.6), poddając ją zmianom prędkości połączenia na sieć 3G oraz 4G. Bazując na informacjach zawartych w artykule BandwidthPlace [68], autorka pracy postanowiła ustawić odpowiednie wartości dla każdej z sieci (Tabela 11).

Testom poddano przeglądarkę Chromium w wersji 130 [69], co wynikało z odmienności w implementacji. Autorka pracy świadomie zrezygnowała z przygotowania analogicznych implementacji dla innych rodzajów przeglądarek, gdyż analiza wykazała, iż wymagałoby to zastosowania zbliżonych konstrukcji programistycznych oraz powtarzalnych procedur.

Tabela 11 Tabela wartości podstawionych do symulacji sieci. Źródło: opracowanie własne.

Zmienna	Wartości dla 3G	Wartości dla 4G
zmienna odpowiadającą opóźnieniu w miliseknd ach (z ang.*latecy)*	100 ms	40 ms
zmienna do pobierania danych (z ang.*download*)	250 000 bajtów/s	2 500 000 bajtów/s
zmienna do przesyłania danych (z ang. *upload*)	50 000 bajtów/s	625 000 bajtów/s

Na zakończenie rozdziału przeprowadzono analizę napotkanych trudności oraz porównanie wydajności i funkcjonalności zastosowanych rozwiązań. Wspomniano również o sposobie implementacji rozwiązań dla innych przeglądarek i ich podobieństwach.

11.1. Scenariusz testowy

Opis scenariusza testowego symulacji zmian sieci.

Tabela 12 Scenariusz testowy symulacji zmian sieci. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna	Ustaw prędkość sieci na 3G / 4G Wprowadź adres https://www.saucedemo.com/ do przeglądarki i przejdź do strony głównej.

2.	Logowanie	W odpowiednie pola wpisz testową nazwę użytkownika: *standard_user* oraz testowe hasło *secret_sauce.* Zaloguj się na stonę wciskając przycik: *Login.*
3.	Strona z produktami	Wybierz przycisk dodawania do koszyka dedykowanego: • dla lampki rowerowej, • dla czerwonej bluzki. W prawym górnym rogu kliknij w ikonę koszyka. Zweryfikuj czy na stronie produktów dodanych do koszyka istenieją poprawne nazwy.
4.	Strona składania zamówienia	Przejdź do strony składania zamówienia. W odpowiednie miejsca wpisz dane potrzebne do złożenia zamówienia: *firstName – testName* *lastName - testLastName* *postalCode -03-333* Naciśnij przycisk *Continue.*
5.	Strona podsumowania	Zweryfikuj: • czy istnieje sekcja podsumowania, • czy sekcja płatności posiada tytuł *„Payment Information”,* informacje o karcie oraz tytuł „*Price Total”,* • czy dostawa posiada tytuł *„Shipping Information”* oraz informacje o sposobie dostawy, Naciśnij przycisk *Finish*
6.	Strona końcowa	Zweryfikuj czy na stronie końcowej pojawia się tytuł : „Thank you for your order!”

11.2. Implementacja w Selenium

W implementacji testów symulacji sieci dla każdej przeglądarki o której wspomniano w rozdziale 3.2.1, należało użyć stosownej konfiguracji. Głównym obiektem, do którego odnosi się Selenium np. przy wyszukiwaniu elementów na stronie jest obiekt `WebDriver` [9], jednak do symulacji zmiennych warunków sieci, narzędzie potrzebuje dodatkowych obiektów aby umożliwić testowanie z tą funkcjonalnością.

W przypadku przeglądarki Chrome i Edge z uwagi, że obie bazują na tym samym silniku renderującym, Selenium używa obiektu klasy `DevTools` [70]. Odpowiada on dokładnie takiej samej

sygnaturze jaka jest używana dla narzędzi programistycznych w fizycznych przeglądarkach. Dzięki tym rozwiązaniom użytkownik jest w stanie manipulować niektórymi funkcjonalnościami jak:

- symulacja sieci z poziomu konsoli,
- wyszukiwanie występujących problemów.

Zależnie od określonego typu, obiekt `WebDriver` musi zostać odpowiednio przemapowany na jedną z dwóch dedykowanych klas:

- `ChromeDiver` – dla przeglądarki Chrome,
- `EdgeDriver` – dla przeglądarki Edge.

Obie struktury dziedziczą po klasie `ChromiumDriver`, która udostępnia metodę umożliwiającą dostęp do instancji obiektu `DevTools`. Dopiero mając dostęp do narzędzi programistycznych, możliwe jest skonfigurowanie odpowiednich parametrów symulacji sieci.

Proces implementacji testów symulacji zmiennych warunków infrastruktury rozpoczęto od wywołania metody, która zwraca odpowiedni obiekt klasy `DevTools`, zależnie od przekazanej w parametrze nazwy przeglądarki (Listing 73). Funkcja ta została zintegrowana z konstruktorem klasy `Network`, zaprojektowanej przez autorkę pracy (Listing 72). Odpowiada ona za zarządzanie konfiguracją sieci, umożliwiając dynamiczne dostosowanie parametrów symulacji w zależności od wybranej przeglądarki.

Listing 72 Implementacja testu symulacji sieci 3G w Selenium.

```
Network network = new Network(driver,BROWSER);
network.set3GNetwork();
stepByStepTest();
```

- `Network` – klasa stworzona przez autorkę pracy na potrzeby testów,
- `BROWSER` – obiekt klasy `String` reprezentujący nazwę przeglądarki jak Chrome/Edge,
- `set3GNetwork()` – metoda do ustawiania konfiguracji dla sieci 3G,
- `stepByStepTest()` – funkcja z krokami scenariusza testowego symulacji sieci (Tabela 12).

Listing 73 Implementacja metody pomocniczej do pobierania obiektu *DevTools* dla przeglądarek Chrome i Edge w Selenium.

```
private DevTools createDevTools() {
    DevTools devTools;
    if (CHROME.equals(browser)) {
        devTools = ((ChromeDriver) driver).getDevTools();
    } else {
        devTools = ((EdgeDriver) driver).getDevTools();
    }
    devTools.createSession();
    return devTools;
}
```

- `browser` - lokalna zmienna reprezentująca nazwę przeglądarki jaka została przekazana w parametrze do konstruktora,

- `driver` – obiekt klasy `WebDriver`,
- `getDevTools()` – metoda z klasy `ChromiumDriver`, do pobierania obiektu `DevTools`,
- `createSeassion()` – rozpoczyna sesję *Chrome DevTools*

Listing 73, przedstawia metodę do ustawiania konkretnych parametrów przekazanych przeglądarce dla sieci typu 3G. Aby przekazać informację, na temat rodzaju sieci 4G, należało utworzyć analogiczną metodę z odpowiednimi dla tego rodzaju argumentami (Listing 74).

Listing 74 Implementacja metody ustawiającej sieć 3G dla Chrome i Edge w Selenium.

```
public void set3GNetwork() {
    emulateNetwork(devTools, 100, 250000, 50000,
      ConnectionType.CELLULAR3G);
}
```

- `ConnectionType.CELLULAR3G` - obiekt klasy `ConnectionType` wbudowanej w Selenium określający typ połączenia sieciowego 3G.

Listing 75 Implementacja metody ustawiającej sieć 4G dla Chrome i Edge w Selenium

```
public void set4GNetwork() {
    emulateNetwork(devTools, 40, 25000000, 625000,
      ConnectionType.CELLULAR4G);
}
```

- `ConnectionType.CELLULAR3G` – określa typ połączenia sieciowego 4G.

Z kolei Listing 75, przedstawia realizację funkcji wywołanej w poprzednich krokach, do której przekazywane jest kilka parametrów potrzebnych do symulacji warunków sieciowych. Pierwszym z etapów było włączenie funkcjonalności w `DevTools` wykonując metodę `enable()` na wbudowanym w Selenium module `Network`. Bez tej części kodu wszelkie manipulacje sieciowe byłby uniemożliwione.

Drugim etapem było wykonanie funkcji *`emulateNetworkConditions()`*, do której należało przekazać odpowiednie argumenty przekazane metodzie zależnie od typu testowanej sieci.

Listing 76 Implementacja metody włączającej funkcjonalności sieciowe oraz ustawienia siciowe w Selenium.

```
private void emulateNetwork(DevTools devTools, int latency, int download,
int upload, ConnectionType connectionType) {

devTools.send(org.openqa.selenium.devtools.v130.network.Network
   .enable(Optional.empty(), Optional.empty(), Optional.empty()));

devTools.send(org.openqa.selenium.devtools.v130.network.Network
   .emulateNetworkConditions(
            false,
            latency,
            download,
            upload,
            Optional.of(connectionType),
            Optional.of(0),
            Optional.of(0),
            Optional.of(true)
```

```
    ));
}
```

- `enable()` – metoda włączająca funkcjonalność sieciową w `DevTools`,
- `latecy` – obiekt typu int, reprezentujący opóźnienie sieciowe w milisekundach,
- `download` – reprezentuje prędkość pobierania danych w jednostce bajtów na sekundę,
- `upload` – prędkość wysyłania danych w jednostce bajtów na sekundę,
- `connectionType` - obiekt klasy `ConnectionType`, wbudowaną w Selenium określający typ połączenia sieciowego,
- `emulateNetworkConditions()` – metoda umożliwiająca konfigurację warunków sieciowych,
- Parametr `false` - parametr określający czy symulacja jest w trybie offline.
- Pierwszy parametr `Optional.of(0)`- odnosi się do ilości uraty pakietów,
- Drugi parametr `Optional.of(0)` – określa ilość pakietów,
- `Optional.of(true)` – informuje czy pakiety powinny posiadać zmienioną kolejność występowania.

11.3. Implementacja w Playwright

W przypadku biblioteki Playwright implementacja zmiennych warunków sieciowych sprowadzała się do ustawienia odpowiedniego sterownika przeglądarki przy użyciu wbudowanych metod w testowany framework. Dla każdej przeglądarki autorka pracy utworzyła odpowiednią funkcję, w której ustawiany jest rodzaj sterownika.

Listing 77, przedstawia implementację dla przeglądarki Chrome, który współdzieli kontroler Chromium z przeglądarką Edge.

Listing 77 Implementacja ustawień przeglądarki Chrome w Playwright.

```
setBrowser(getPlaywright().chromium()
   .launch(new BrowserType.LaunchOptions().setHeadless(headlessMode)));
```

- `getPlaywright()`- metoda utworzona przez autorkę pracy, zwracająca obiekt klasy Playwright przechowujący wszystkie sterowniki przeglądarek,
- `chromium()` – zwraca obiekt `BrowserType` odpowiadający *Chromium,*
- `launch()` – metoda odpowiadająca za odpalenie przeglądarki z podanymi opcjami w argumencie,
- `setHeadless()` – ustawienie parametru dla widoczności interfejsu graficznego.

Listing 78 implementuje przekazanie dodatkowej wartości dla przeglądarki Edge.

Listing 78 Implementacja ustawień przeglądarki Edge w Playwright.

```
setBrowser(getPlaywright().chromium()
      .launch(new BrowserType.LaunchOptions()
            .setChannel("msedge").setHeadless(false)));
```

W przypadku Safari, w Playwright stosowany jest silnik *Webkit* [71]. Jest to sterownik przeglądarki, na kórym bazuje dużo aplikacji używanych w systemie *macOS* (Listing 79).

Listing 79 Implementacja użycia sterownika *Webkit* w Playwright.

```
setBrowser(getPlaywright().webkit().launch(new
      BrowserType.LaunchOptions().setHeadless(false)));
```

- `webkit()` – zwraca obiekt *BrowserType* odpowiadający sterownikowi *Webkit.*

Listing 80 pokazuje analogiczny przykład dla przeglądarki FireFox.

Listing 80 Implementacja użycia sterownika FireFox w Playwright.

```
setBrowser(getPlaywright().firefox().launch(new
   BrowserType.LaunchOptions().setHeadless(false)));
```

- `firefox()` – zwraca obiekt `BrowserType` odpowiadający sterownikowi *FireFox.*

W dalszej części implementacji skupiono się na pokazaniu tylko części dla przeglądarki Chrome oraz Edge z uwagi na współdzielony sterownik Chromium.

Playwright przy użyciu języka programowania *Java*, nie ustawia warunków sieciowych względem kontekstu, jak przy innych wspieranych językach, ale na obiekcie `Page`.

Aby zaimplementować zmienne warunki sieciowe w Playwright, autorka pracy utworzyła pomocniczą klasę o nazwie `Network`, w której stworzona została metoda `setNetwork()`. Funkcja umożliwia podanie argumentów :

- opóźnienia (z ang. *latecy*),
- pobierania (z ang. *download*),
- przesyłania (z ang. *upload*).

Listing 81 Wywołanie klasy *Network* w Playwright.

```
Network network = new Network(page);
network.setNetwork(40,250000000,625000);

stepByStepTest();
```

- `setNetwork()` – posiada zaimplementowaną logikę zmienności warunków sieciowych,

- `stepByStepTest()` – wywołanie testu składania zamówienia.

Listing 82 przedstawia użycie klasy `CDPSession` [72] z biblioteki Playwright w celu symulacji zmienionej infrastruktury. Klasa kontaktuje się czystym protokołem *Chrome Devtools* [73]. Wykorzystując obiekt przeglądarki, ustawiane są odpowiednie parametry i wysyłane poprzez

wbudowaną metodę `send()` bezpośrednio do niej. Jest to najbardziej dokładne i najłatwiesze symulowanie warunków sieciowych w Playwright.

Listing 82 Implementacja logiki zmiany sieci w Playwright.

```
public void setNetwork(int latecy,int download,int upload){

    CDPSession cdpSession = BrowserContextUtils
      .getBrowserContext().newCDPSession(page);

    cdpSession.send("Network.enable");

    JsonObject params = new JsonObject();
    params.addProperty("offline",false);
    params.addProperty("latency",latecy);
    params.addProperty("downloadThroughput",download);
    params.addProperty("uploadThroughput",upload);

    cdpSession.send("Network.emulateNetworkConditions", params);

}
```

- `cdpSession.send("Network.enable")` – metoda służąca włączeniu protokołu *DevTools*,
- `params.addProperty()` – dodanie argumentów do obiektu typu *JsonObject*,
- `cdpSession.send("Network.emulateNetworkConditions", params)` – służy wysyłaniu zmiennych warunków sieciowych z ustawionymi parametrami.

11.4. Analiza porównawcza

W procesie realizacji testów symulacji zmiennych warunków sieciowych, zarówno Selenium, jak i Playwright oferują odmienny model implementacyjny, który zależny jest od wewnętrznej architektury oraz sposobu integracji z przeglądarkami internetowymi.

W przypadku Selenium, podstawowym obiektem umożliwiającym interakcję jest instancja klasy `WebDriver`. Jednakże do realizacji symulacji warunków sieciowych konieczne jest wykorzystanie dodatkowych narzędzi. Dla Chrome oraz Edge, które korzystają ze wspólnego silnika Chromium, Selenium umożliwia dostęp do narzędzi deweloperskich za pośrednictwem obiektu klasy `DevTools`. Jest to bezpośrednie nawiązanie do narzędzi programistycznych dostępnych w rzeczywistych przeglądarkach, co pozwala na manipulację parametrami sieciowymi z poziomu zautomatyzowanych testów.

W odróżnieniu od Selenium, Playwright nie wymaga przypisania dedykowanych sterowników przeglądarki zależnie od jej typu, a konfiguracja środowiska testowego dla różnych obiektów sprowadza się do użycia odpowiednich metod, takich jak :

- `chromium()`,
- `webkit()`,
- `czy firefox()`.

Dzięki temu proces konfiguracji jest bardziej ujednolicony i mniej zależny od specyficznych klas sterowników.

W przypadku implementacji innych przeglądarek, według [74], Selenium jak i Playwright stosuje podobne kontrukcje. W przypadku implementacji dla przeglądarki Firefox potrzebne jest użycie klasy `FirefoxMobBrowserProxy`, z biblioteki *LightBody* [75]. A w przypadku Safari [76] użycie zewnętrznej platformy BrowserStack. Połączenie z chmurą następuje poprzez dedykowaną dla każdego użytkownika nazwę oraz klucz wysyłany przy każdym zapytaniu API.

W pracy skupiono się na prównaniu różnic w implementacji obu narzędzi z tego względu zrezygnowano z utworzenia kodu dla inny przegądarek niż Chrome..

Pisząc kod autorka pracy zauważyła, że dla manipulacji sieci w języku *Java*, Selenium posiada dużo więcej materiałów dydaktycznych niż Playwright. Z tego względu implementacja w Selenium była dużo prostsza i szybsza. W przypadku drugiej biblioteki autorka pracy musiała poświęcić więcej czasu aby stworzyć poprawne rozwiązanie z uwagi na większy dostęp do publikacji dla technologii *Node.js*.

Na zakończenie badania przeprowadzono porównanie czasów wykonania testów automatycznych w dwóch trybach pracy:

- z włączonym interfejsem graficznym (*headless=false*)
- oraz w trybie bezinterfejsowym (*headless=true*).

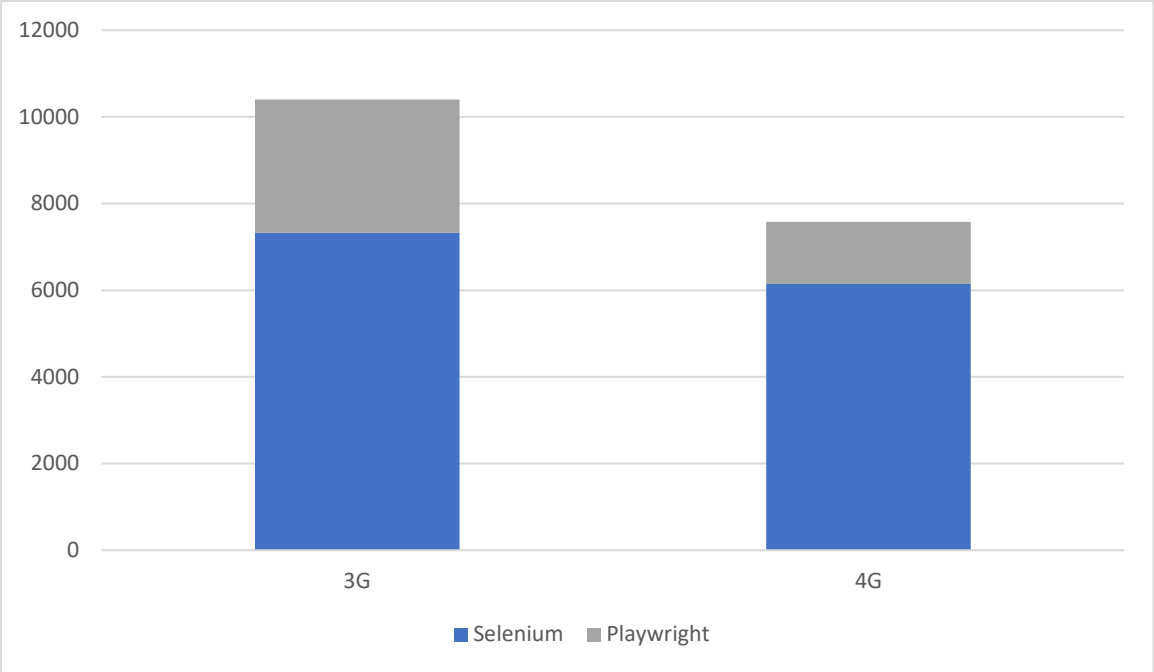
Analiza wykazała, że obecność interfejsu użytkownika miała marginalny wpływ na czas realizacji testów manipulacji sieciowej. Może to wynikać z charakteru strony internetowej, która została zaprojektowana specjalnie na potrzeby testów automatycznych, a nie była platformą skomercjalizowaną. Brak złożonych mechanizmów, takich jak zabezpieczenia antybotowe czy dynamiczne renderowanie, mógł ograniczyć różnice w wydajności między trybami.

Zgodnie z założeniami badania, testy przeprowadzone w symulowanych warunkach sieci 3G cechowały się dłuższym czasem wykonania w porównaniu do testów w warunkach sieci 4G, co jest zgodne z ich specyfikacją techniczną (niższa przepustowość i wyższe opóźnienia w przypadku 3G).

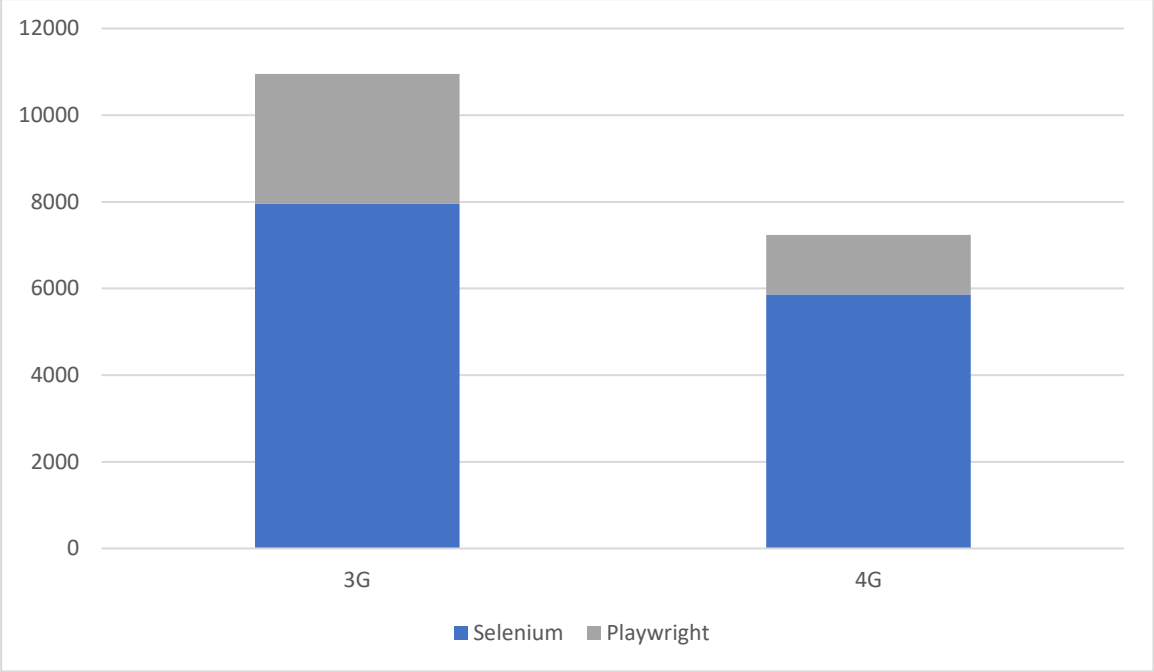
Wykres 10 oraz Wykres 11 zawierają wyniki, które wskazują, że Playwright osiągnął lepsze czasy wykonania w porównaniu do Selenium w obu scenariuszach sieciowych. Jednakże autorka badania zauważyła, że Selenium oferuje:

- prostszą implementację testów,
- oraz większe wsparcie materiałów,

co może czynić to narzędzie bardziej przystępnym w pewnych kontekstach, szczególnie dla mniej doświadczonych użytkowników lub w projektach wymagających szybkiego prototypowania.



Wykres 10 Porównanie czasów wykonania w testach zmienności sieci dla trybut *Headless=false* (mniej=lepiej). Źródło: opracowanie własne.



Wykres 11 Porównanie czasów wykonania w testach zmienności sieci dla trybu *Headless=true* (mniej=lepiej). Źródło: opracowanie własne.

12. Testy równoległego wykonywania

W niniejszym rozdziale przedstawiono implementację równoległego wykonywania testów z wykorzystaniem dwóch bibliotek. Do realizacji tego celu wykorzystano zestaw przypadków opracowanych w rozdziałach 7–10, które posłużyły do stworzenia kompletnego zbioru skryptów testowych.

Na zakończenie rozdziału dokonano analizy efektywności obu narzędzi pod kątem realizacji testów równoległych oraz porównano ich skuteczność.

Testy równoległego wykonywania (z ang. *Parallel testing*) [78], odnoszą się do procesu uruchamiania wielu testów jednocześnie. Ma na celu skrócenie całkowitego czasu ich wywołania, realizując je w oddzielnych instacjach wątków bądź przeglądarek. Wykorzystywane jest zazwyczaj w dużych zestawach skryptów testowych.

12.1. Implementacja w Selenium

Selenium oferuje wbudowany mechanizm *Selenium Grid* [11], którego szczegółowy opis znajduje się w rozdziale rozdziale 3.2.3. W celu implementacji konieczne było skorzystanie z aplikacji Docker [75], która umożliwia uruchamianie różnych środowisk w odizolowanych kontenerach.

Listing 83 przedstawia plik *docker-compose.yaml,* zawierający konfigurację kontenera [76] ułatwiającego komunikację w ramach *Selenium Grid*. Zmienna `chrome` odpowiada pojedynczemu węzłowi (z ang. *node*). Dla każdego z nich określono obraz, na którym działa oraz przypisane porty. W konfiguracji uwzględniono cztery węzły, na których będą uruchamiane testy.

Listing 83 Konfiguracja *docker-compose.yaml* w Selenium.

```
version: "3"
services:
  selenium-hub:
    image: selenium/hub:4.21.0
    container_name: selenium-hub
    ports:
      - "4442:4442"
      - "4443:4443"
      - "4444:4444"

  chrome:
    image: selenium/node-chrome:4.21.0
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
      - SE_NODE_MAX_SESSIONS=5

  chrome2:
      image: selenium/node-chrome:4.21.0
      depends_on:
        - selenium-hub
      environment:
        - SE_EVENT_BUS_HOST=selenium-hub
        - SE_EVENT_BUS_PUBLISH_PORT=4442
        - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
```

```
      - SE_NODE_MAX_SESSIONS=5

  chrome3:
    image: selenium/node-chrome:4.21.0
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443

  chrome4:
    image: selenium/node-chrome:4.21.0
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
      - SE_NODE_MAX_SESSIONS=5
```

- `Selenium-hub` – zmienna zawierająca konfigurację *Selenium Grid*,
- `chrome[i]` – ustawienia dla pojedynczego węzła,
- `SE_NODE_MAX_SESSIONS=5` – ilość maksymalnej ilości sesji w ramach jednego obiektu *node*.

W środowisku implementacyjnym konieczne było utworzenie pliku *testng.xml*, który umożliwia wykonywanie testów w trybie równoległym. *Selenium Grid* zapewnia wykonywanie testów rownoległych przy użyciu biblioteki *TestNG* [79].

Testy realizowane w pracy zostały zaimplementowane przy użyciu biblioteki JUnit5 [30], dlatego też cała konfiguracja *TestNG* została wyodrębniona do osobnego projektu w środowisku IntelliJ, co zapobiegło występowaniu błędów konfiguracyjnych. Listing 84 zawiera treść pliku konfiguracyjnego *testng.xml.*

Listing 84 Zawartość pliku *testng.xml*.

```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >
<suite name="Selenium Grid Suite" parallel="tests" thread-count="4">

    <test name="end2endTest">
        <classes>
            <class name="parallel.BaseTest"/>
            <class name="parallel.LoggingTest"/>
            <class name="parallel.NetworkTest"/>
            <class name="parallel.NewWindowHandlerTest"/>
            <class name="parallel.UploadDownloadTest"/>
        </classes>
    </test>

</suite>
```

- `thread-count` – ilość równocześnie uruchamianych wątków,
- `<class>` - tag zawierający nazwę klasy testowej.

Autorka pracy dodała do pliku *pom.xml* niezbędne zależności oraz wtyczkę dla *TestNG* (Listing 85). W związku z tym należało zmienić importy w testach na wersję nowego frameworka. Ponadto, w konfiguracji rozszerzenia `Maven Surefire` dodano tag `<suiteXmlFiles>`, wskazujący ścieżkę do pliku testów do wykonania.

Listing 85 Zależność dla *TestNG* w Selenium.

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.9.0</version>
    <scope>test</scope>
</dependency>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.5.2</version>
    <configuration>
        <suiteXmlFiles>
            <suiteXmlFile>src/test/resources/testng.xml</suiteXmlFile>
        </suiteXmlFiles>
    </configuration>
</plugin>
```

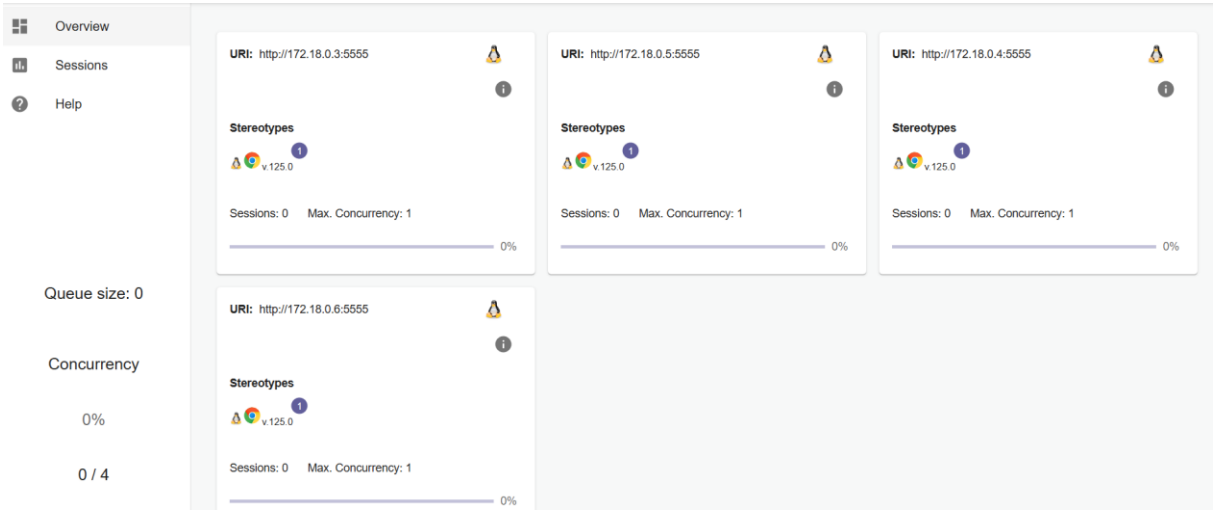
Aby uruchomić *Selenium Grid* w kontenerze, należało uruchomić aplikację Docker Desktop, a następnie wykonać komendę *docker-compose up* z poziomu terminala edytora Intellij, bazując na wcześniej przygotowanym pliku *docker-compose.yaml.* Rysunek 14 przedstawia widok konsoli po wykonaniu polecenia.

```
Run 'docker compose COMMAND --help' for more information on a command.
esources\\docker-compose.yaml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion
"
[+] Running 20/21
 ✓ selenium-hub Pulled                                                                                                    71.5s
 ✓ chrome Pulled                                                                                                          86.7s
[+] Running 3/3
 ✓ Network resources_default     Created                                                                                   0.1s
 ✓ Container selenium-hub        Created                                                                                   5.1s
 ✓ Container resources-chrome-1  Created                                                                                   0.1s
Attaching to chrome-1, selenium-hub
selenium-hub  | 2025-06-15 11:52:34,003 INFO Included extra file "/etc/supervisor/conf.d/selenium-grid-hub.conf" during parsing
selenium-hub  | 2025-06-15 11:52:34,008 INFO RPC interface 'supervisor' initialized
selenium-hub  | 2025-06-15 11:52:34,008 CRIT Server 'unix_http_server' running without any HTTP authentication checking
selenium-hub  | 2025-06-15 11:52:34,011 INFO supervisord started with pid 8
chrome-1      | 2025-06-15 11:52:34,304 INFO Included extra file "/etc/supervisor/conf.d/chrome-cleanup.conf" during parsing
chrome-1      | 2025-06-15 11:52:34,304 INFO Included extra file "/etc/supervisor/conf.d/selenium.conf" during parsing
chrome-1      | 2025-06-15 11:52:34,308 INFO RPC interface 'supervisor' initialized
chrome-1      | 2025-06-15 11:52:34,308 CRIT Server 'unix_http_server' running without any HTTP authentication checking
chrome-1      | 2025-06-15 11:52:34,310 INFO supervisord started with pid 8
```

Rysunek 14 Konsola po uruchomieniu *Selenium Grid* w kontenerze. Źródło: opracowanie własne.

Następnym etapem było uzyskanie dostępu do interfejsu *Selenium Grid* pod adresem:

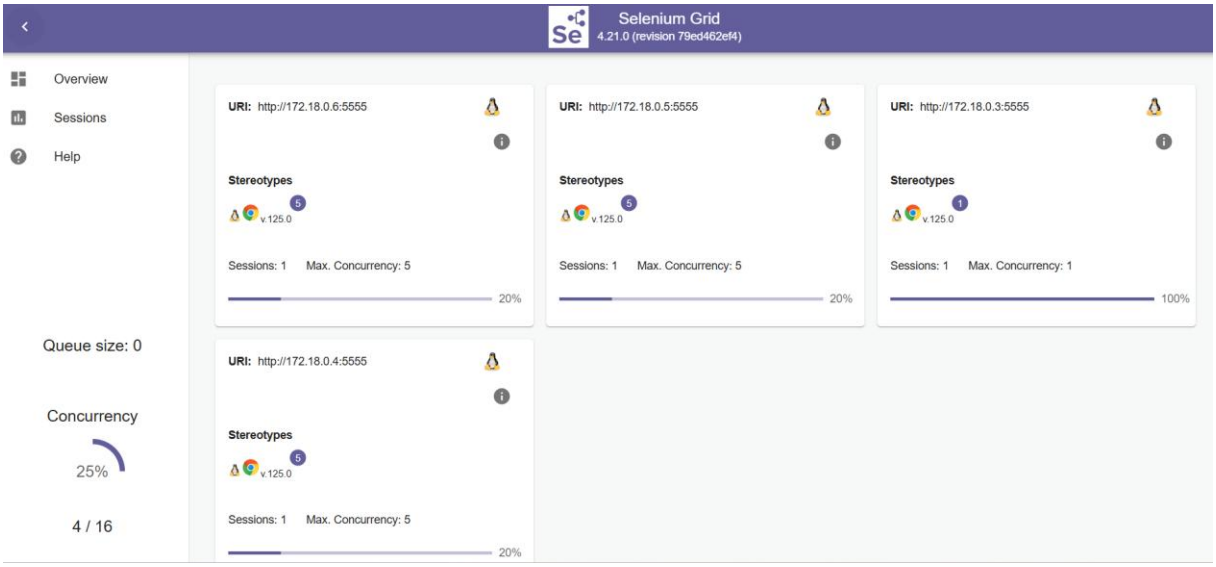
http://localhost:4444/ui/, odpowiadającym lokalnemu serwerowi (Rysunek 15).



Rysunek 15 Serwer *Selenium Grid*. Źródło: opracowanie własne.

Uruchomienie testów odbywało się poprzez komendę *mvn test*. W trakcie ich wykonywania możliwe było monitorowanie aktywności poszczególnych węzłów w interfejsie serwera.

Rysunek 16 przedstawia widok w trakcie działania testów. System automatycznie przydzielał wolne wątki oraz zasoby pamięci dla kolejno wykonywanych instacji, co potwierdziło prawidłowe działanie mechanizmu testów równoległych.



Rysunek 16 Aktywne węzeły w *Selenium Grid*. Źródło: opracowanie własne.

Podsumowując efektywność wykonywania testów, otrzymano raport wskazujący liczbę weryfikacji zakończonych pozytywnie oraz tych negatywnych. Analiza logów konsoli wykazała występowanie błędów związanych z wyszukiwaniem elementów oraz błędów typu *Timeout*. Listing 86 przedstawia raport z wykonania testów.

Listing 86 Raport podsumowujący wykonanie testów równoległych w Selenium.

```
[ERROR] Tests run: 10, Failures: 4, Errors: 0, Skipped: 0
```

12.2. Implementacja w Playwright

W Playwright do implementacji testów równoległych użyto wbudowanej annotacji `@UsePlaywright` [81]. Znacznik jest rozszerzeniem frameworka JUnit5 [30]. Dzięki niej Playwright umożliwia równoległe wykonywanie przypadków testowych w odseparowanych wątkach.

Biblioteka JUnit5 posiada domyślny mechanizm dostarczania odpowiedniego silnika do uruchamiania procedur kontrolnych (z ang. *runner*) [82].

W pierwszym etapie autorka pracy utworzyła kopię implementacji, o której wspomniano we wstępie tego rozdziału dodając je do odseparowanego pakietu - *parallelTest*, a następnie dodała annotację `@UsePlaywright` do każdego z nich. W kolejnym kroku nastąpiło wywołanie komendy z poziomu aplikacji Intellij aby wykonać wszystkie testy z jednego pakietu – *Run 'Tests' in 'ParallelTest'* (Listing 87)

Listing 87 Dodanie annotacji @UsePlaywright w testach równoległywch w Playwright.

```
@UsePlaywright
public class SnapshotTest {…}
```

Rysunek 17 przedstawia widok konsoli po próbie wywołania wszystkich testów równoległych. Błędy spowodowane były niewłaściwą konfiguracją obiektów :

- *Playwright*,
- *Browser*,
- oraz *BrowserContext*.

Rysunek 17 Kody błędów ze złą konfiguracją w Playwright dla testów równoległych. Źródło: opracowanie własne.

W przypadku testów wykonywanych sekwencyjnie, jak przedstawiono w rozdziałach 7 - 10, zastosowane ustawienia były w pełni poprawne. Jednakże w kontekście testów rónoległych

zaobserwowano istotny problem — wszystkie wątki współdzieliły pojedynczą instancję obiektów odpowiedzialnych za obsługę przeglądarki oraz kontekstu (Listing 88). Skutkowało to wzajemnymi konfliktami, szczególnie podczas zamykania zmiennych po zakończeniu działania testów w innych wątkach.

Listing 88 Błędna konfiguracja dla testów równoległych w Playwright.

```
@BeforeAll
public static void setupAll() {
    BrowserContextUtils.createPlaywrightAndBrowserChromium(true);
}

@BeforeEach
public void setUp() {
    page = BrowserContextUtils.getPage();
    BrowserContextUtils.setTestIdAttribute("data-test");
}

@AfterEach
public void closeContext() {
    BrowserContextUtils.closeContext();
}

@AfterAll
public static void close() {
    BrowserContextUtils.closeBrowser();
}
```

- `setUpAll()` – metoda wywołująca konfigurację do ustawienia obiektów Playwright, przeglądarki (z ang. *browser*) oraz kontekstu przeglądarki (z ang. *browser context*),
- `setUp()` – funkcja tworząca instancję klasy `Page`, oraz ustawiająca odpowiedni identyfikator dla danej zmiennej,
- `closeContext()` – zamyka kontekst przeglądarki,
- `close()` – zamyka obiekt przeglądarki.

W celu rozwiązania problemu konieczne było zastosowanie mechanizmu odizolowującego instancje testów na poziomie wątków, co umożliwiło niezależne wykonywanie. Do tego stworzono klasę przechowującą nową konfigurację użytą tylko do testów równoległych – `BrowserContextUtilsParallelTest`. W klasie wykorzystano specjalny wzorzec projektowy - *ThreadLocal* - opisany w artykule Mateusza Mazurka [83], umożliwiający współdzielenie kopii zmiennych zarządzanych przez pojedyncze wątki.

W `BrowserContextUtilsParallelTest` utworzono trzy odrębne obiekty odpowiadające trzem zmiennym, które były obsługiwane przez niezależne procesy w środowisku wykonawczym (Listing 89).

Listing 89 Implementacja klasy *BrowserContextUtilsParallelTest* w Playwright.

```
public class BrowserContextUtilsParallelTest {
    private static final ThreadLocal<Playwright> playwright = new
ThreadLocal<>();
    private static final ThreadLocal<Browser> browser = new
```

```
ThreadLocal<>();
    private static final ThreadLocal<BrowserContext> browserContext = new
ThreadLocal<>();

    public static void createPlaywrightAndBrowserChromium() {
        playwright.set(Playwright.create());
        browser.set(
            playwright.get().chromium()
            .launch(new BrowserType.LaunchOptions().setHeadless(false)));
          }

    public static Browser getBrowser() {
        return browser.get();
    }

    public static Playwright getPlaywright() {
        return playwright.get();
    }

    public static void setBrowserContext(BrowserContext context) {
        browserContext.set(context);
    }

    public static BrowserContext getBrowserContext() {
        return browserContext.get();
    }
    public static void setTestIdAttribute(String testId) {
        getPlaywright().selectors().setTestIdAttribute(testId);

    }

    public static void closeContext() {
            browserContext.get().close();
            browserContext.remove();
    }

    public static void closeBrowser() {
            browser.get().close();
            browser.remove();

            playwright.get().close();
            playwright.remove();
    }
}
```

- `ThreadLocal` – wzorzec projektowy do dzielenia kopii zmiennej potrzebnej w wielu wątkach,
- *playwright*`.set(Playwright.`*create*`())` - metoda inicjalizująca nową instancję obiektu *Playwright* i przypisującą ją w kontekście danego procesu,
- `browser.set()` - przydziela instancję przeglądarki (`Browser`) do zmiennej lokalnej wątku.,
- `getBrowser()` – zwraca obiekt przeglądarki powiązany z aktualnym procesem,
- `getPlaywright()` – zwraca obiekt klasy `Playwright`,

- `getBrowserContext()` – metoda dostarczająca kontekst przeglądarki (`BrowserContext`),
- `setTestIdAttribute()` – nadpisuje identyfikator elementu w interfejsie użytkownika,
- `closeContext()` – funkcja zamykająca kontekst przeglądarki po zakończeniu testu w danym wątku,
- `closeBrowser()` – zamyka instancję przeglądarki przypisanej do aktualnie wykonywanego procesu.

Listing 90, przedstawia użycie poprawnie stworzonej konfiguracji w ramach testów równoległych.

Listing 90 Implementacja zastosowania poprawnej konfiguracji testów róenoległych w Playwright.

```
@BeforeEach
public void setUp() {
    BrowserContextUtilsParallelTest.createPlaywrightAndBrowserChromium();
    BrowserContext context = BrowserContextUtilsParallelTest
      .getBrowser().newContext();

    BrowserContextUtilsParallelTest.setBrowserContext(context);
    page = context.newPage();
    BrowserContextUtilsParallelTest.setTestIdAttribute("data-test");
}

@AfterEach
public void tearDown() {
    BrowserContextUtilsParallelTest.closeContext();
    BrowserContextUtilsParallelTest.closeBrowser();
}
```

Ostatnim etapem implementacji rozwiązania dla biblioteki Playwright, było utworzenie pliku *junit-platfrom.properties*, zawierającego ustawienia niezbędne do uruchamiania testów równoległych przy wykorzystaniu frameworka JUnit5 [30] (Listing 91). Plik ten umożliwia definiowanie parametrów sterujących wykonywaniem testów, w tym sposobu i liczby współbieżnie uruchamianych wątków [84].

W konfiguracji zdecydowano się na zastosowanie ustawień nawiązujących do tych wykorzystywanych w środowisku *Selenium Grid*, co zapewnia spójność parametrów testowych pomiędzy obiema technologiami. Dzięki temu możliwe było zachowanie jednolitego podejścia do równoległego wykonywania testów zarówno w przypadku środowiska Playwright, jak i *Selenium Grid*.

Listing 91 Konfiguracja dla frameworka JUnit5 do wykonania testów równoległych w Playwright.

```
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = same_thread
junit.jupiter.execution.parallel.mode.classes.default = concurrent
junit.jupiter.execution.parallel.config.strategy=fixed
junit.jupiter.execution.parallel.config.fixed.parallelism=16
```

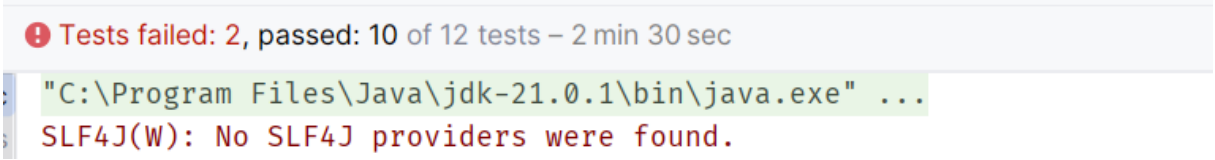
- `junit.jupiter.execution.parallel.enabled=true` – aktywuje możliwość wykonywania testów równolegle,

- `junit.jupiter.execution.parallel.mode.default=concurrent` - określa tryb współbieżnego wykonywania testów na poziomie metod,

- `junit.jupiter.execution.parallel.config.fixed.parallelism=16` - ustawia maksymalną liczbę równolegle uruchamianych wątków testowych na szesnaście, co odpowiada liczbie dostępnych sesji przeglądarek w środowisku *Selenium Grid*.

Po zakończeniu procesu wygenerowany został raport, w którym zawarto informacje dotyczące liczby przypadków testowych zakończonych powodzeniem oraz tych, które zakończyły się niepowodzeniem. Szczegółowa analiza logów wyświetlanych w konsoli ujawniła występowanie błędów związanych z nieprawidłowym odnajdywaniem elementów na stronie oraz przekroczeniem limitu czasu oczekiwania na określone akcje.

Rysunek 18 przedstawia podsumowanie wyników testów w formie zestawienia umożliwiającego dalszą analizę i identyfikację potencjalnych przyczyn niepowodzeń.



Rysunek 18 Raport wykonani testów równoległych w Playwright. Źródło: opracowanie własne.

12.3. Analiza porównawcza

W przypadku Selenium do implementacji testów równoległych wykorzystano wbudowany mechanizm *Selenium Grid*, wymagający zastosowania struktury kontenerowej. Konieczne było przygotowanie dodatkowego pliku konfiguracyjnego w formacie *YAML*, definiującego środowisko testowe. Dodatkowo ustawienia wymagały utworzenia osobnego projektu wykorzystującego framework *TestNG*, odmiennego od stosowanego przy sekwencyjnym uruchamianiu testów. Spowodowało to dodatkowy nakład pracy oraz wpłynęło na zmniejszenie płynności w procesie pisania kodu.

Playwright natomiast oferuje uproszczony mechanizm ustawień, niewymagający wykorzystania zewnętrznych narzędzi do zarządzania środowiskiem testowym. Dzięki natywnemu wsparciu dla wielowątkowości w środowisku JUnit5 oraz możliwości wykorzystania wzorca *ThreadLocal* dla instancji obiektów `Playwright`, `Browser` oraz `BrowserContext`, możliwe było osiągnięcie równoległego wykonywania testów w obrębie jednego projektu.

Selenium Grid zapewnia dużą skalowalność środowiska oraz elastyczność, umożliwiając dynamiczne uruchamianie wielu kontenerów z możliwością konfigurowania takich parametrów jak liczba wątków czy wersje przeglądarek. Playwright natomiast, choć charakteryzuje się mniejszą elastycznością w tym zakresie, oferuje prostszy mechanizm równoległego wykonywania testów. Mogą być one wykonywane jednocześnie w różnych przeglądarkach, jednak w ramach jednej instancji maszyny wirtualnej JVM.

Wdrożenie rozwiązania opartego na *Selenium Grid* wymaga od osoby implementującej dobrej znajomości infrastruktury kontenerowej oraz narzędzi takich jak Docker i Docker Compose, co może generować dodatkowy nakład pracy i stanowić barierę wejścia dla nowych użytkowników. Playwright z kolei oferuje mniej skomplikowane rozwiązanie, co obniża próg wejścia w technologię. Niemniej

jednak, także w tym przypadku konieczne jest odpowiednie skonfigurowanie środowiska testowego oraz zarządzanie cyklem życia instancji przeglądarki, co nie jest procesem całkowicie trywialnym.

Pod względem efektywności wykonywanych testów, *Selenium Grid* zapewnia wysoką wydajność w środowiskach rozproszonych, umożliwiając jednoczesne uruchamianie dużej liczby testów w wielu przeglądarkach i konfiguracjach systemowych. Wymaga to jednak odpowiedniego przygotowania infrastruktury oraz zapewnienia jej dostępności. Playwright natomiast osiąga bardzo dobre wyniki wydajnościowe w środowisku lokalnym, szczególnie w testach obejmujących jednorodne konfiguracje przeglądarek, dzięki natywnemu wsparciu dla wykonywania testów wielowątkowo w ramach jednej aplikacji.

Zauważono istotne różnice pomiędzy rozwiązaniami opartymi na Selenium oraz Playwright w zakresie:

- czasu wykonania testów,
- zużycia pamięci operacyjnej,
- oraz w liczbie poprawnie zakończonych przypadków testowych.

W przypadku Selenium, czas realizacji testów wyniósł 55 sekund i 398 milisekund, przy czym na dziesięć uruchomionych testów cztery zakończyły się niepowodzeniem. Obciążenie procesora w trakcie wykonywania testów osiągało wartości maksymalną na poziomie 78%.

Dla porównania, Playwright wykonał zestaw testów w czasie 2 minut i 30 sekund, uzyskując przy tym wyższą skuteczność — na dwanaście uruchomionych testów, dziesięć zakończyło się powodzeniem. W trakcie działania testów obciążenie jednostki centralnej było wyższe niż w przypadku Selenium, osiągając nawet 100%. Wzrost ten wynikał z faktu, iż Playwright wykonywał testy w środowisku lokalnym, natomiast Selenium korzystało z infrastruktury kontenerowej, co w naturalny sposób rozkładało część obciążeń systemowych.

Różnica w liczbie testów pomiędzy rozwiązaniami była związana z odmienną implementacją operacji na plikach. W przypadku Playwright zaimplementowano dwa dodatkowe przypadki testowe względem Selenium, co mogło wpłynąć na całkowity czas wykonania oraz zużycie zasobów systemowych.

Tabela 13 Podsumowanie porównania Selenium vs Playwright przy testach równoległych. Źródło: opracowanie własne.

Kryterium	Selenium	Playwright
Czas wykonania testów	55s 296 ms	2 min 30 s
Skuteczność testów	60%	83%
Obciążenie CPU	78%	100%
Środowisko uruchomieniowe	Kontener	Lokalnie
Liczba testów	10	12

Tabela 13 zawiera zbiór analizowanych danych, na jej podstawie stwierdzono, że Selenium przy użyciu *Selenium Grid* osiąga lepsze wyniki w przypadku gdy priorytetem jest:

- skalowalność,
- oraz szybkość wykonania testów.

Natomiast Playwright jest efektywniejsze w sytuacji, gdy kluczowe znaczenie mają:

- prostota implementacji,
- oraz wysoka skuteczność testów.

13. Kompleksowe testy End-to-End

W tym rozdziale przedstawiono dwa testy *End-to-End* wykonane na różych stronach internetowych: Booking.com (rozdział 4.3) oraz Orange.pl (rozdział 4.1). Wybór tych serwisów był spowodowany występowaniem odmiennych funkcjonalności, co miało znaczący wpływ na implementację skryptów.

Celem analizy było zbadanie różnic:

- w implementacji,
- w czasie wykonania,
- oraz w zużyciu zasobów systemowych, takich jak wartość *CPU*.

W każdej z sekcji opisujących implementację testów przedstawiono szczegółową konfigurację narzędzi, która została wcześniej wykorzystana w rozdziałach 5 - 12, ale nie była omawiana w celu zachowania zwięzłości wcześniejszych rozdziałów. Niniejsza sekcja koncentruje się na pełnym opisie procesu implementacji, obejmującym wszystkie etapy – od projektowania testów, przez ich realizację, aż po finalizację skryptów testowych.

Dodatkowo, w każdej sekcji zidentyfikowano i omówiono kluczowe problemy napotkane podczas tworzenia ostatecznych wersji skryptów, co pozwoliło na lepsze zrozumienie wyzwań związanych z automatyzacją testów na różnych platformach.

13.1. Testy aplikacji Booking.com

13.1.1.Scenariusz testowy

Opis scenariusza testowego dla *End-to-End* na stronie Booking.com.

Tabela 14 Scenariusz testowy dla testów *End-to-End* na stronie Booking.com. Źródło: opracowanie własne.

Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna	Wprowadź adres: https://www.booking.com/index.pl.html do przeglądarki i przejdź do strony głównej. Zaakceptuj ciasteczka. W pole „Dokąd się wybierasz” wpisz **Zakopane** Na kalendarzu wybierz miesiąc **lipiec** i jakąkolwiek datę przyjazdu. Jako datę wyjazdu wybierz tą za trzy dni. W polu wyboru ilości osób dodaj jedną dorosła osobę oraz jedno dziecko. Zaznacz, że dziecko ma dwa lata. Zweryfikuj czy w polu tekstowym wybour parametrów jest są trzy osoby dorosłe, jedno dziecko i jeden pokój. Wciśnij przycisk *Szukaj*.

2.	Sekcja filtrowania	Zamknij wyskakujące okno do zalogowania. Przefiltruj lokalizację pod względem: • dostępności parkingu • oceny lokalizacji: bardzo dobrej • bezpłatnego Wi-Fi W filtrowaniu ceny ustaw aby obiekty nie były droższe niż 700 zł.
3.	Strona z miejscami pobytu	Wybierz pierwszy zwrócony element klikając przycisk dostępności.
4.	Strona z wybranym miejscem pobytu	Wciśnij *Zarezerwuj,* u dołu strony. Zweryfikuj, że serwis, na którym się znajdujesz nie pozwala przejść dalej, a w miejscu wyboru ilości pokoi pojawia się komunikat błędu. Wybierz ilość – „1'', przy pierwszym typie pokoju. Naciśnij *Zarezerwuj*
5.	Rezerwacja	Zweryfikuj: • że w sekcji szczegółów znajduje się taka sama nazwa obiektu, która była rezerwowana, • w sekcji szczegółów znajduje się wybrany uprzednio rodzaj pokoju. Uzupełnij dane do rezerwacji : • Imię - Natalia • Nazwisko – Nazwisko • Emial - nataliaNazwisko@gmail.com • Telefon - 504444444 Jeżeli dostępne wpisz również: • Ulica – Zegarynki 22 • Miasto – Warszawa • Kod pocztowy – 03-020 Uzupełnij godzinę przyjazdu na 15 :00 Wciśnij przycisk *Zarezerwuj*
6.	Strona podsumowania	Zweryfikuj czy cena na ostatniej stronie jest taka sama jak kwota na stronie z rezerwacją.

13.1.2.Implementacja w Selenium

Aby rozpocząć pracę z frameworkiem Selenium, w pierwszej kolejności należało dodać odpowiednią zależność do pliku konfiguracyjnego *pom.xml* (Listing 92).

Listing 92 Zależność biblioteki Selenium.

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.27.0</version>
</dependency>
```

Następnie przystąpiono do implementacji klasy testowej. W jej obrębie zdefiniowano metodę oznaczoną adnotacją `@BeforeEach` z biblioteki Junit5 [30]. W taki sposób wskazana funkcja będzie wykonywać się przed każdym przypadkiem testowym (Listing 93).

Listing 93 Implementacja metody *@BeforeEach* w testach *End-to-End* strony Booking.com w Selenium.

```
@BeforeEach
public void setUp() {
    driver = DriverFactory.getDriver("chrome", false);
}
```

- `@BeforeEach` – annotacja z biblioteki JUnit5 oznaczająca, że metoda będzie wykonywana przed każdym testem jednostkowym,
- `DriverFactory` - klasa utworzona przez autorkę pracy, w celu przechowywania konfiguracji w oddzielnym obiekcie,
- `getDriver()` – funkcja zwracająca instację klasy `WebDriver` [9], dla typu przeglądarki oraz trybu interfeju graficzngo, przekazanych jako argumenty.

Listing 94, przedstawia implementację metody `getDriver()`, wywołanej w Listing 93.

Listing 94 Implementacja metody *getDriver()* w Selenium.

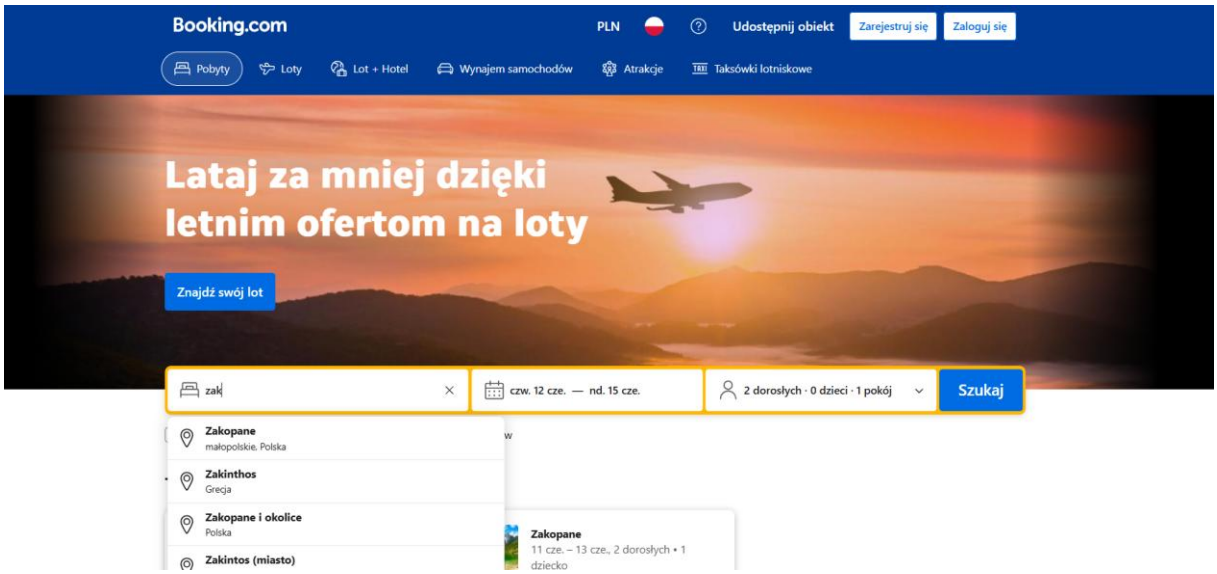
```
Public static WebDriver getDriver(String browserName, Boolean headless) {
    browserName = browserName.toLowerCase();

    switch (browserName) {
        case "chrome":
            ChromeOptions chromeOptions = new ChromeOptions();
            if (headless)
                  chromeOptions.addArguments("—headless");
            chromeOptions.addArguments("user-agent=Mozilla/5.0 (Windows NT
10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0
Safari/537.36");
            return DriverUtils.startChromeDriver(chromeOptions);

// reszta kodu
      }
```

- `toLowerCase()` – metoda zmieniająca rozmiar czcionki, z klasy `String`,

- `ChromeOptions` – obiekt reprezentujący konfigurację przeglądarki Chrome,
- `addArguments()` – funkcja dodająca parametry klucz-wartość do opcji przeglądarki,
- `"user-agent=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0" Safari/537.36"` – parameter ustawiający nagłówek *User-Agent* w przypadku jego braku - tzn. w przypadku braku interfejsu graficznego.

Po zdefiniowaniu konfiguracji środowiska testowego możliwe było przystąpienie do implementacji scenariuszy testowych (Tabela 14).

Pierwszym krokiem było przejście do serwisu Booking.com oraz wpisanie miejsca docelowego do wyszukiwarki na stronie głownej (Rysunek 19).



Rysunek 19 Wybór miejsca docelowego na stronie Booking.com. Źródło: [18].

Listing 95 został wykonany w celu realizacji etapu, w którym zaakceptowano pliki ciasteczek po załadowaniu strony. Po zamknięciu komunikatu, nastąpiło wprowadzenie nazwy miejscowości w polu wyszukiwarki oraz wybranie pierwszej pozycji z listy rozwijanej. Rysunek 19 przedstawia widok opisanej witryny.

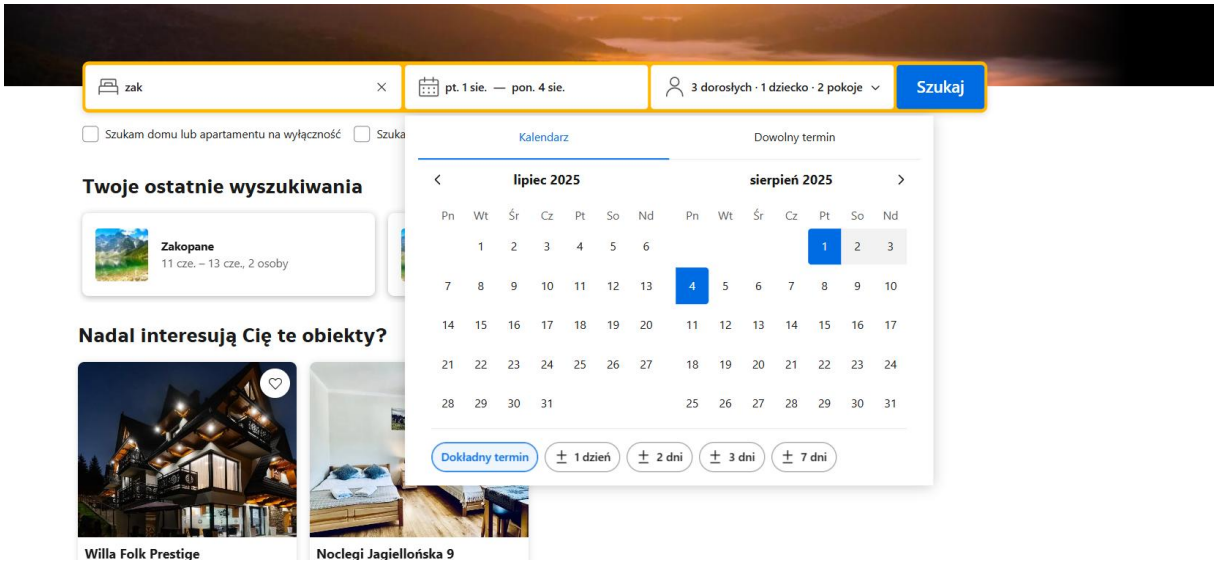
Jednym z bardziej czasochłonnych elementów implementacji było zagwarantowanie, że z listy wybrano poprawną miejscowość : **Zakopane**. W tym celu wykorzystano skrypt *JavaScript* oczekujący na zakończenie ładowania zasobów. Na koniec wprowadzono walidację, sprawdzającą czy wybrano prawidłową nazwę miejsca docelowego.

Listing 95 Implementacja akcji wykonywanych na stronie głównej Booking.com w Selenium.

```
driver.get(„https://www.booking.com/index.pl.html");

WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(15));

WebElement acceptButton = wait.until(ExpectedConditions
```

```
      .visibilityOfElementLocated(
            By.xpath("//button[contains(text(),'Akceptuj')]")));

acceptButton.click();

boolean invisible = wait.until(ExpectedConditions
            .invisibilityOfElementLocated(
                  By.xpath("//button[contains(text(),'Akceptuj')]")));

if(invisible) {
 WebElement destinationInput = wait.until(web ->
      driver.findElement(
      By.cssSelector("input[placeholder=\"Dokąd się wybierasz?\"]")));

 destinationInput.sendKeys("Zak");
 wait.until(
         webDriver -> ((JavascriptExecutor) webDriver)
                 .executeScript(
                  "return document.readyState").equals("complete")
             );


 WebElement option = wait.until(ExpectedConditions
            .visibilityOfElementLocated(
                  By.xpath("//li[@role='option'][div[@role='button']]")));

 option.click();

 assertTrue(destinationInput.getText().contains("Zakopane"));


}
```

- `get()` – metoda przekierowująca na adres podany w parametrze,
- `WebDriverWait` [43] – klasa umożliwiająca oczekiwanie na dalszą interakcję z elementami na stronie,
- *`visibilityOfElementLocated()`* – określa widoczność obiektór,
- *`invisibilityOfElementLocated()`* – funkcja weryfikująca niewidoczność obiektu,
- `"return document.readyState").equals("complete")` – skrypt *JavaScript* kontrolujący, pobranie zasobów na stronę. Zwraca `complete` w momencie ukończenia ładowania serwisu,
- `click()` – metoda wykonująca akcję kliknięcia na podanym elemencie,
- `sendKeys()` - wysyła ciąg znaków do obiektu, na którym jest wywoływana.



Rysunek 20 Wybór daty z kalendarza na stronie Booking.com. Źródło: [18]

Kolejnym etapem realizacji scenariusza testowego było wskazanie dat pobytu. Jako termin przyjazdu wybrano pierwszy dostępny w kalendarzu, natomiast datę wyjazdu ustalono na trzy dni później. Rysunek 20 przedstawia wygląd tej fazy testu.

Proces polegał na zidentyfikowaniu komponentu interfejsu użytkownika odpowiedzialnego za wybór dat. Procedura wymagała iteracyjnego aktywowania kontrolki nawigacyjnej, zmieniającej wyświetlany miesiąc w kalendarzu, aż do momentu pojawienia się nagłówka z napisem **„sierpień 2025”**. Po jego odnalezieniu, pobrano sekwencyjną listę dostępnych dat z danego miesiąca, a następnie wybrano pierwszą oraz trzecią z nich, zgodnie z pozycją w strukturze komponentu.

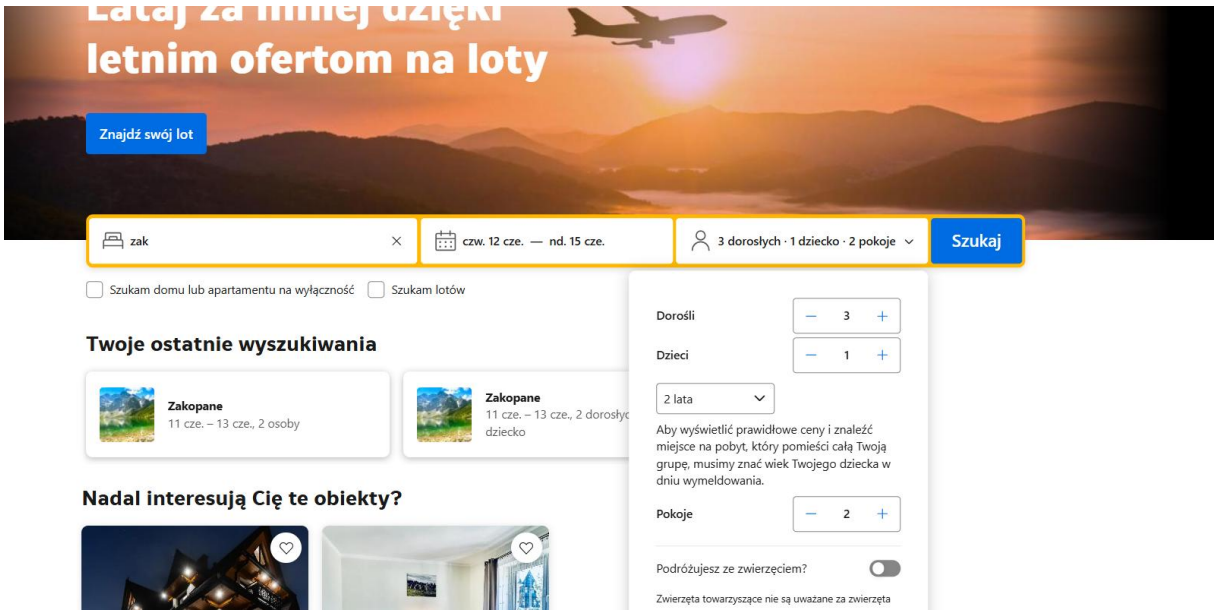
W trakcie implementacji napotkano wyzwanie wynikające z braku jednoznacznych atrybutów identyfikujących w kalendarzu, co znacząco utrudniało ich selekcję. W odpowiedzi na ten problem przyjęto strategię wyszukiwania wszystkich elementów potomnych, współdzielących tego samego rodzica w hierarchii *DOM*, co umożliwiło identyfikację wymaganych obiektów interfejsu (Listing 96).

Listing 96 Implementacja wyboru daty w Selenium.

```
while (driver.findElements(By.xpath("//h3[contains(text(),sierpień
2025')]")).isEmpty()) {
    WebElement nextMonthButton = driver.findElement(
      By.xpath("//button[@aria-label='Następny miesiąc']"));

    nextMonthButton.click();
}

wait.until(ExpectedConditions
      .visibilityOfElementLocated(By.xpath("//h3[contains(text(),'sierpień
      2025')]")));

List<WebElement> availableDates = wait.until(driver -> {
     List<WebElement> dates = driver.findElements(
      By.xpath("//h3[contains(text(),
      'sierpień 2025')]/followingsibling::table//td[not(@aria-
hidden)]//span[@data-date]"));
```

```
if (!dates.isEmpty()) {
  return dates;
} else {
  return null;
}
});

availableDates.get(0).click();
availableDates.get(2).click();
```

- `findElements()` – metoda wbudowana w klasę `WebDriver` [9], która zwraca listę elementów spełniających określony warunek w parametrze,
- `findElement()` – zwraca jeden obiekt zgodny z argumentem,
- `By.xpath()` – metoda wbudowana w obiekt `By` [85], która szuka elementu na podstawie ścieżki *XPath* w strukturze *DOM*,
- `h3[contains(text(),'sierpień 2025')]`- wyrażenie *XPath* identyfikujące nagłówek kalendarza dla **sierpień 2025**,
- `followingsibling::table` – wskazuje tabelę kalendarza będącą bezpośrednim następnikiem nagłówka miesiąca,
- `td[not(@aria-hidden)`- filtruje elementy odpowiadające elementom możliwym do wyboru,
- `span[@data-date]` – atrybut obiektu reprezentującego datę.



Rysunek 21 Wybór podróżujących na stronie Booking.com. Źródło: [18]

Trzecim etapem realizacji scenariusza testowego (Tabela 14) była implementacja wyboru liczby podróżujących. Rysunek 21 przedstawia wartości oczekiwane.

W tym celu autorka pracy zdecydowała się zastosować konstrukcję opartą na wyszukiwaniu elementów przy użyciu atrybutu *data-testid*, który został dodany do głównego komponentu wyboru ilości turystów. Procedura polegała na:

- kliknięciu w element rozwijający sekcję,
- zlokalizowaniu wyświetlonego komponentu popup,
- oraz odnalezieniu przycisków służących do inkrementacji wartości.

Znalezienie obiektów nie należało do trywialnych, z uwagi na brak jakiegokolwiek odnośnika ułatwiającego lokalizację guzików. W związku z tym przyjęto rozwiązanie polegające na wyszukiwaniu ich na podstawie kolejności występowania w obrębie wspólnego kontenera. Znaleziono elementy według dwóch kryteriów:

- obiekt typu `div` z klasą „e301a14002”,
- podrzędne przyciski z klasą „dc8366caa6”.

Wszystkie kontrolki do inkrementowania liczby dorosłych oraz dzieci posiadały jednakowe selektory, co wymusiło odwoływanie się do ich pozycji w liście elementów. Należy zaznaczyć, że takie rozwiązanie jest wrażliwe na zmiany w strukturze strony i w przyszłości może prowadzić do awarii testów. Listing 97 realizuje opisaną strategię, którą zastosowano z uwagi na brak bardziej stabilnej metody identyfikacji komponentów.

Listing 97 Implementacja wyboru podróżujących na stronie Booking.com w Selenium.

```
driver.findElement(
      By.cssSelector("[data-testid='occupancy-config']")).click();

WebElement occupancyPopup = driver
      .findElement(By.cssSelector("[data-testid='occupancy-popup']"));

WebElement secondAdultButton = occupancyPopup
      .findElements(
      By.cssSelector("div.e301a14002 button.dc8366caa6")).get(0);

secondAdultButton.click();
```

Dodatkowym problemem była obsługa rozwijanej listy wyboru wieku dziecka. Selenium posiada dedykowaną klasę `Select` służącą do pracy z tego typu obiektami, jednak możliwą do wykorzystania jedynie w przypadku prawidłowo zaimplementowanych elementów *HTML* typu `<select>`. W tym scenariuszu obiekt wyboru wieku został poprawnie zakodowany, co umożliwiło jego obsługę przez Selenium (Listing 98).

Listing 98 Wybór z listy rozwijanej na stronie Booking.com w Selenium.

```
      WebElement ageDropdown = wait.until(ExpectedConditions
      .visibilityOfElementLocated(By.name("age")));

Select selectAge = new Select(ageDropdown);
selectAge.selectByValue("2");
```

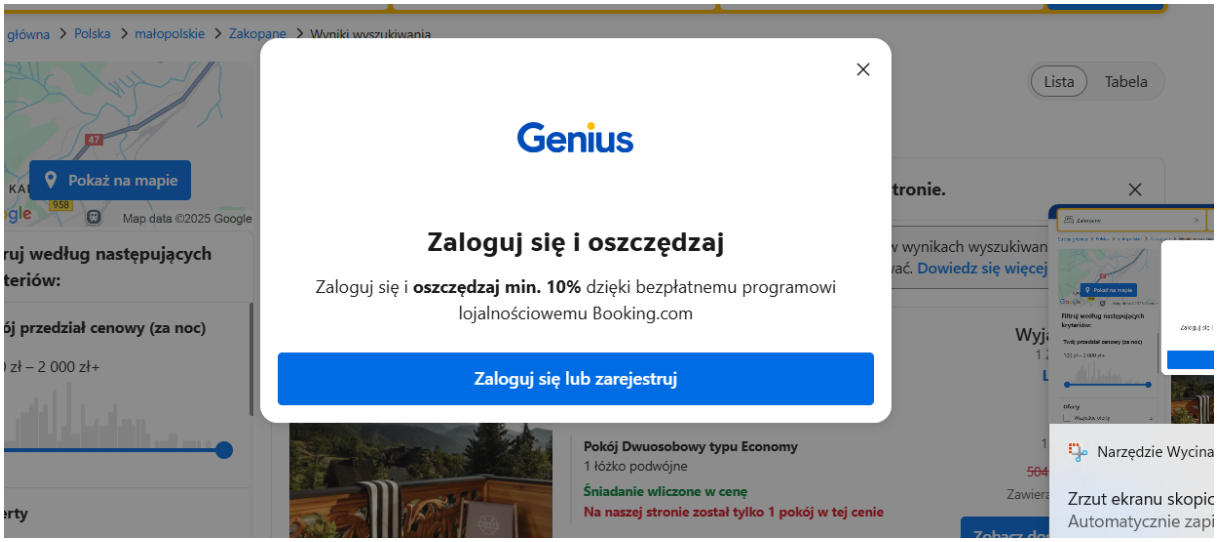
```
WebElement occupancyText = driver
      .findElement(By.cssSelector("[data-testid='occupancy-config']"));

assertEquals(
      "3 dorosłych · 1 dziecko · 1 pokój", occupancyText.getText());


WebElement searchButton = driver
      .findElement(By.cssSelector("[type='submit']"));

searchButton.click();
```

- `By.cssSelector()` – metoda wbudowana w klasę `By`, do wyszukiwania elementów po sektorach CSS,
- `Select` – umożliwia obsługę rozwijanych list `<select>`,
- `selectByValue()` – funkcja klasy `Select`. Zezwala na obsługę wartości otrzymanych z obiektu `<select>`.

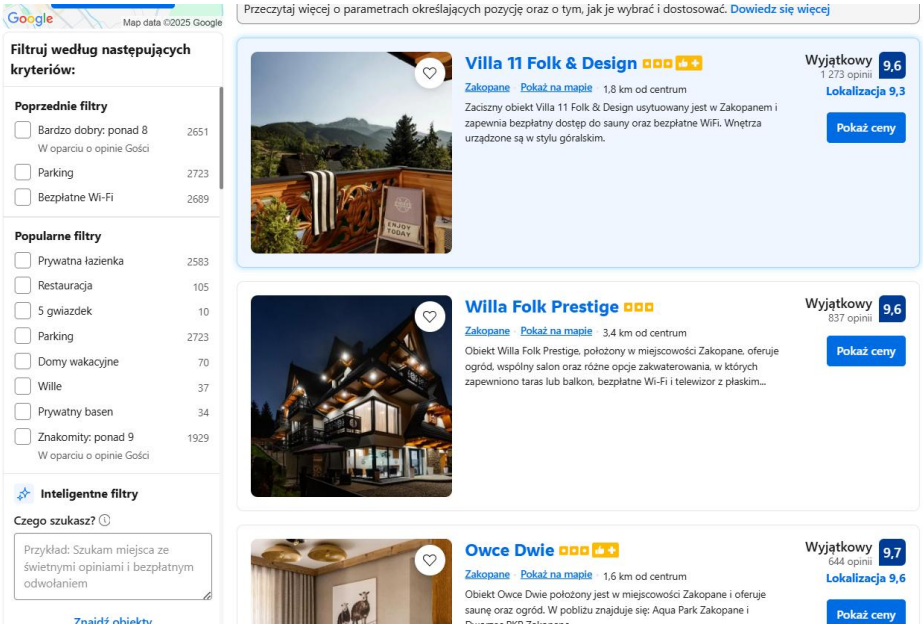
Na zakończenie pierwszego etapu scenariusza testowego dokonano weryfikacji poprawności ustawienia liczby podróżujących oraz zatwierdzono wybór, przechodząc na stronę z wynikami wyszukiwania miejsc noclegowych.

W ramach drugiego punktu należało :

- zamknąć wyskakujące okno do logowania użytkownika (Rysunek 22),
- zaznaczyć odpowiednie filtry, znajdujące się po lewej stronie witryny internetowej Booking.com (Rysunek 23).



Rysunek 22 Wyskakujące okno do logowania użytkownika na stronie Booking.com. Źródło: [18]



Rysunek 23 Filtry na stronie Booking.com. Źródło: [18].

W przypadku elementów, które nie są widoczne w aktualnym widoku strony, zastosowano mechanizm przewijania okna do wskazanego komponentu. W tym celu wykorzystano klasę `JavascriptExecutor` [64] oraz metodę `executeScript()`, w której wywoływano funkcję *`scrollIntoView()`*.

Dzięki temu możliwe było przewinięcie interfejsu do elementów:

- filtru dostępności parkingu,
- filtru oceny „Bardzo dobry”,
- przycisku „Pokaż wszystkie”,
- oraz dostępności Bezpłatnego Wi-Fi.

Każdy z nich był wyszukiwany za pomocą selektorów *CSS* lub *XPath*, a następnie odpowiednio aktywowany (Listing 99).

Listing 99 Implementacja filtrowania na stronie Booking.com w Selenium.

```
WebElement closeLoginWindowButton = wait.until(ExpectedConditions
.visibilityOfElementLocated(
      By.cssSelector("[aria-label='Zamknij okno logowania.']")));

closeLoginWindowButton.click();

((JavascriptExecutor)
driver).executeScript("arguments[0].scrollIntoView(true);",
driver.findElement(By.xpath("//input[contains(@aria-label, 'Parking')]")));


WebElement parkingCheckbox = driver
.findElement(By.xpath("//input[contains(@aria-label, 'Parking')]"));
parkingCheckbox.click();
```

- `[aria-label='Zamknij okno logowania.']` – element z parametrem *aria-label*,
- `executeScript()` – metoda wykonująca przekazaną w parametrze funkcję *JavaScript*,
- `scrollIntoView()` – powoduje przewinięcie strony do odpowiedniego miejsca,
- `input[contains(@aria-label, 'Parking')]")));` - wyszukanie obiektu typu `input` z *JavaScript*, o argumencie *aria-label* i wartości Parking,

Kolejnym z filtrów jakie wzięto pod uwagę była cena. W serwisie Booking.com zaimplementowano go w postaci suwaka (z ang. *slider*) (Rysunek 23).

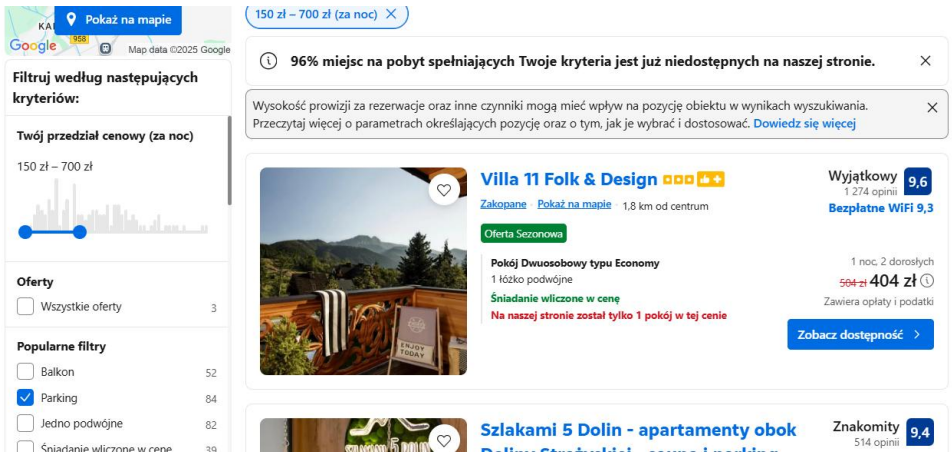
Proces rozpoczęto od identyfikacji części obiektu odpowiedzialnego za ustawienie maksymalnej kwoty. Następnie element wraz z wartością do skonfigurowania został przekazany do pomocniczej metody `adjustSlider()`. W rozwiązaniu zastosowano klasę `JavascriptExecutor`, gdzie w `executeScript()`, zasymulowano działania użytkownika poprzez wykonanie zdarzeń [87] (Listing 100):

- `change` – aktualizacja widoku,
- *`input`* – zmiana wartości suwaka.

Listing 100 Implementacja obsługi suwaka na stronie Booking.com w Selenium.

```
WebElement maxSliderInput = driver
      .findElement(
      By.cssSelector("input[type='range'][aria-label='Maks.']"));

adjustSlider(maxSliderInput, 700);


private void adjustSlider(WebElement slider, int targetValue) {

  JavascriptExecutor js = (JavascriptExecutor) driver;
  js.executeScript(
                "arguments[0].value = arguments[1];" +
                        "arguments[0].dispatchEvent(new Event('input'));" +
                        "arguments[0].dispatchEvent(new Event('change'));",
                maxSliderInput, targetValue
        );

    }
```

Trzeci etap scenariusza testowego (Tabela 14), dotyczył wybrania miejsca pobytu spośród listy zwróconych wyników. W ramach tego kroku, zdecydowano się na wybór pierwszego z dostępnych elementów. Implementacja polegała na odnalezieniu i kliknięciu przycisku „Zobacz dostępność" odpowiadającego wybranemu miejscu. Aby zapewnić poprawne działanie, zastosowano pomocniczą metodę przewijania strony, wykorzystując klasę `JavascriptExecutor`, co umożliwiło pokazanie się w widoku, przycisku zanim został on kliknięty. (Listing 101).



Rysunek 24 Wybór pierwszego elementu na liscie zwracanych miejsc pobytu. Źródło: [18].

Listing 101 Implementacja przewijania strony wyboru miejsca docelowego na stronie Booking.com w Selenium.

```
WebElement availabilityButton = wait.until(webDriver -> driver
      .findElement(
      By.cssSelector("[data-testid='availability-cta-btn']")));

((JavascriptExecutor) driver)
.executeScript("arguments[0].scrollIntoView(true);", availabilityButton);

wait.until(ExpectedConditions
      .visibilityOfElementLocated(
      By.cssSelector("[data-testid='availability-cta-btn']")));

availabilityButton.click();
```

- `wait` – obiekt oczekujący na pojawienie się elementu na stronie.

W czwartym etapie testu należało zasymulować kliknięcie przycisku „Zarezerwuj” znajdującego się na dole strony wybranego miejsca pobytu. Akcja ta powoduje otwarcie nowego okna przeglądarki z aktualnym adresem *URL*, na którym Selenium będzie dalej wykonywać operacje.

W związku z tym konieczne było przejście do nowo otwartej zakładki. Zastosowano tu technikę opisaną w rozdziale 10, dotyczącą zarządzania wieloma oknami przeglądarki w Selenium.

Przy pomocy metody `getWindowHandle()`, pobrano pierwszy identyfikator, odpowiadający stronie głównej wyszukiwania obiektów noclegowych.

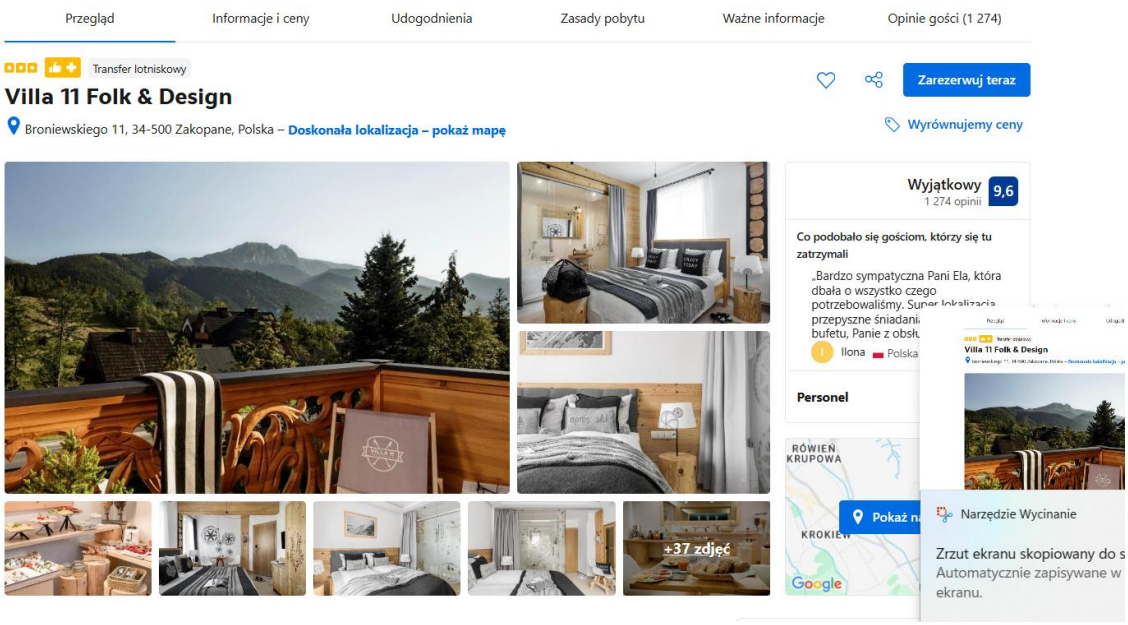
Następnie, z wykorzystaniem `wait`, kod oczekiwał, aż liczba otwartych okien wzrośnie, co świadczy o pojawieniu się nowej zakładki.

Na końcu dokonano iteracji po wszystkich identyfikatorach, wybierając i przełączając kontekst sterowania Selenium do pierwszego w kolejności różniącego się numeru identyfikacyjnego (Listing 102).

Listing 102 Implementacja nawigacji między zakładkami na stronie Booking.com w Selenium.

```
String firstPage = driver.getWindowHandle();

wait.until(driver -> driver.getWindowHandles().size() > 1);

for (String windowHandle : driver.getWindowHandles()) {
    if (!windowHandle.equals(firstPage)) {
        driver.switchTo().window(windowHandle);
        break;
    }
}
```

- `getWindowHandle()` – metoda zwraca identyfikator aktualnie aktywnej zakładki,
- `getWindowHandles()` – zwraca listę identyfikatorów wszystkich stron otwartych w nowych oknach.



Rysunek 25 Witryna wybranego obiektu na stronie Booking.com. Źródło: [18].

Po przełączeniu na nowo otwartą zakładkę, autorka pracy zaimplementowała dodatkowy mechanizm synchronizacji, który opierał się na oczekiwaniu, aż kluczowy element - tytuł obiektu noclegowego - stanie się widoczny. Bez wprowadzenia tej strategii test często kończył się błędami, ponieważ Selenium próbowało odwoływać się do elementów jeszcze niewidocznych w *DOM*.

Ponadto, aby przycisk „Zarezerwuj" znalazł się w widoku, konieczne było przewinięcie strony do samego dołu. W tym celu ponownie wykorzystano klasę `JavascriptExecutor`, która pozwala wykonać skrypt *JavaScript* bezpośrednio w kontekście interfejsu użytkownika (Listing 103).

Listing 103 Implementacja rezerwacji na stronie booking.com w Selenium.

```
wait.until(webDriver -> driver.findElement(
   By.className("pp-header__title")));

WebElement bookButton = wait.until(ExpectedConditions
   .presenceOfElementLocated(
   By.className("js-reservation-button__text")));

((JavascriptExecutor)
driver).executeScript("arguments[0].scrollIntoView(true);", bookButton);

bookButton.click();

WebElement errorMessage = driver
   .findElement(By.xpath("//*[contains(text(), 'Błąd:')]"));

assertTrue(errorMessage.isDisplayed());

String roomType = driver
   .findElement(By.className("hprt-roomtype-icon-link")).getText();

Select rooms = new Select(driver
   .findElement(By.cssSelector("[data-testid='select-room-trigger']")));

rooms.selectByValue("1");

invisible = wait.until(ExpectedConditions
   .invisibilityOfElementLocated(
        By.xpath("//*[contains(text(), 'Błąd:')]")
));

if (invisible) {
    WebElement bookButton2 =
wait.until(ExpectedConditions.presenceOfElementLocated(
            By.xpath("(//div[@class='hprt-reservation-cta'])[2]")));

    bookButton2.click();
}
```

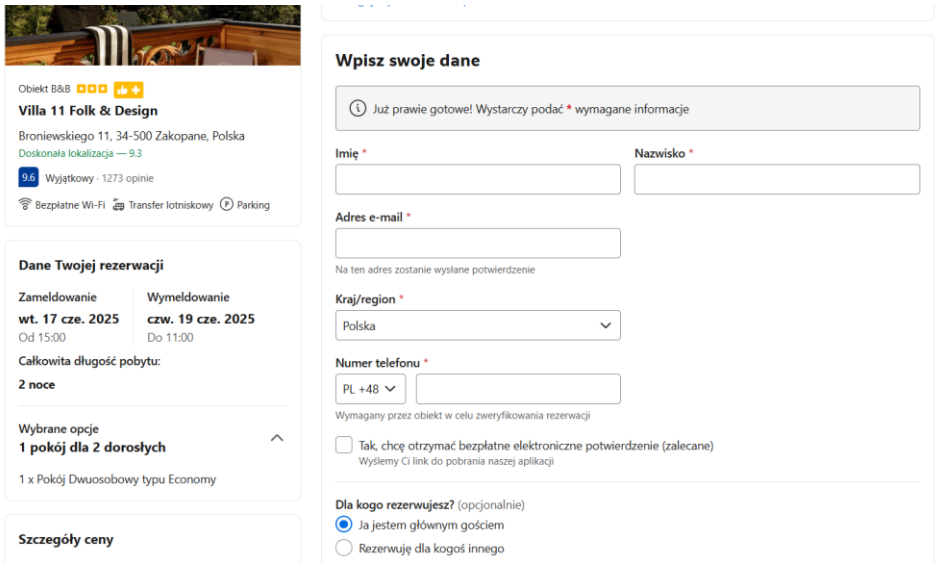
- `By.className("hprt-roomtype-icon-link")` - metoda wbudowana w klasę `By` do wyszukiwania elementów posiadający atrybut `class`,
- *`invisibilityOfElementLocated`* – oczekuje na zniknięcie obiektu na stronie.

Na etapie piątym scenariusza testowego (Tabela 14) użytkownik ma za zadanie wypełnić formularz rezerwacji. Rysunek 26 przedstawia ten dostępny na stronie.

Implementacja kroku przebiegła bez trudności, gdyż opierała się na wcześniej opisanych konstrukcjach wyszukiwania elementów i interakcji z nimi.

Należy jednak podkreślić, że Booking.com w zależności od obiektu noclegowego może wymagać uzupełnienia dodatkowych pól, takich jak dane adresowe osoby rezerwującej. W związku z tym, w implementacji zastosowano prostą metodologię warunkową, która sprawdza obecność tych pól w *DOM* przed próbą ich wypełnienia. Takie podejście zwiększa odporność testu na różnice w formularzach między obiektami (Listing 104).

W Selenium, jeśli element nie zostanie znaleziony, natychmiast rzucany jest wyjątek *NoSuchElementException*, co uniemożliwia dalsze odwołanie się do niego. Listing 105 przedstawia utworzoną przez autorkę pracy metodę pomocniczą, która podczas napotkania tego wyjątku zwraca *false*, sygnalizując brak elementu na stronie. Dzięki temu możliwe jest bezpieczne sprawdzenie istnienia obiektów bez przerywania działania testu.



Rysunek 26 Rezerwacja na stronie Booking.com. Źródło:[18].

Listing 104 Uzupełnienie danych do rezerwacji na stronie Booking.com w Selenium.

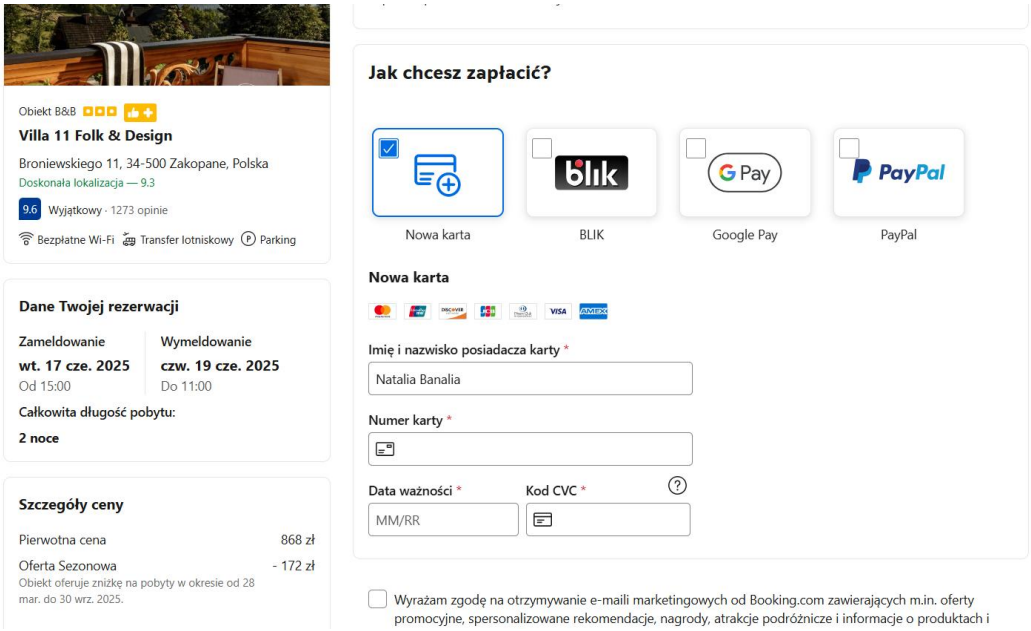
```
WebElement header = driver.findElement(By.className("pp-header__title"));
assertTrue(header.isDisplayed());

WebElement roomUnit = wait.until(ExpectedConditions
      .visibilityOfElementLocated(
      By.cssSelector("[data-testid='booking-details-unit-selection']")));

((JavascriptExecutor) driver)
      .executeScript("arguments[0].scrollIntoView(true);", roomUnit);

roomUnit.click();

WebElement openSummarySection = driver.findElement(
      By.cssSelector("[data-testid=\"booking-details-unit-selection\"]"));

openSummarySection.click();

WebElement roomText = driver
      .findElement(By.cssSelector("li div.b99b6ef58f"));

String roomTypeText = roomType.trim();
String roomTextValue = roomText.getText().trim();
assertTrue(roomTextValue.contains(roomTypeText));

driver.findElement(
      By.cssSelector("[data-testid='user-details-firstname']"))
```

```
      .sendKeys("Natalia");

if (isElementVisible(driver, By.cssSelector("[data-testid='user-details-
address1']"))) {

    driver
      .findElement(By.cssSelector("[data-testid='user-details-address1']"))
      .sendKeys("Zegarynki 22");

}

Select selectArrivalTime = new Select(driver
      .findElement(By.name("checkin_eta_hour")));

selectArrivalTime.selectByValue("15");

String totalPrice1 = driver.findElement(
      By.cssSelector(
      "[data-animate-price-group name='bp_user_total_price']"))
      .getText();

WebElement submitButton = driver
   .findElement(By.cssSelector("[name='book']"));

submitButton.click();
```

- `sendKeys()` – wysyła tekst do elementu, na którym jest wywoływana,
- `isElementVisible()` – metoda utworzona na potrzeby testów.

Listing 105 Implementacja metody pomocniczej do określenia widoczności elementów na stronie Booking.com w Selenium.

```
   private boolean isElementVisible(WebDriver driver, By by) {
       try {
           driver.findElement(by);
           return true;
       } catch (NoSuchElementException e) {
           return false;
       }
   }
```



Rysunek 27 Podsumowanie rezerwacji Booking.com [18].

Listing 106 implementuje ostatni krok, porównania wyświetlanej ceny na stronie podsumowania z tą znajdująca się w zakładce rezerwacji poprzez zwykłe odczytanie wartości tekstu z elementu.

Listing 106 Podsumowanie na stronie Booking.com w Selenium.

```
String totalPrice2 = driver.findElement(
By.cssSelector("[data-animate-price-group-name='bp_user_total_price']"))
      .getText();
assertEquals(totalPrice1, totalPrice2);
```

13.1.3.Implementacja w Playwright

Pierwszym etapem implementacji rozwiązania w środowisku Playwright było dodanie zależności biblioteki Playwright do pliku konfiguracyjnego *pom.xml* w systemie zarządzania budową Maven [31].

Listing 107 Zależność Playwright w pliku pom.xml.

```
<dependency>
    <groupId>com.microsoft.playwright</groupId>
    <artifactId>playwright</artifactId>
    <version>1.49.0</version>
</dependency>
```

Początkowa konfiguracja przeglądarki w środowisku Playwright wymagała utworzenia kilku kluczowych obiektów niezbędnych do prawidłowego działania testów [88]. Pierwszym z nich była instancja klasy `Playwright`, inicjalizowana za pomocą metody `create()`, która generowała obiekt z domyślnymi ustawieniami przeglądarki. Następnie zdefiniowano typ `Browser`, odpowiadający za określony rodzaj klienta sieciowego, zależnie od zastosowanej metody inicjalizacyjnej.

Kolejnym istotnym elementem była klasa `BrowserContext`, reprezentująca kontekst przeglądarki, w tym między innymi profil użytkownika oraz jego sesję. Na utworzonym obiekcie `Playwright` wywoływano metodę inicjującą przeglądarkę, przekazując w jej parametrze informację o trybie uruchomienia — z aktywnym interfejsem graficznym bądź bez (*headless*).

Dodatkowo została dodana opcja ustawiania własnych identyfikatorów elementów według, których były one potem wyszukiwane w obrębie strony. Zrealizowano to poprzez pobranie dostępnych opcji z obiektu `Playwright`, a następnie nadpisanie ich przekazaną wartością (Listing 108).

Listing 108 Implementacja konfiguracji dla testów w Playwright.

```
setPlaywright(create());

setBrowser(getPlaywright().chromium()
        .launch(new
            BrowserType.LaunchOptions().setHeadless(headlessMode)));

setBrowserContext(browser.newContext());
setTestIdAttribute("data-testId");
```

- `setPlaywright()` – metoda stworzona na potrzeby testów do ustawiania zmiennej obiektu `Playwright`,
- `setBrowser()` – ustawia zmienną reprezentującą typ przeglądarki (z ang. *browser),*
- `setBrowserContext()` - metoda utworzona przez autorkę pracy do ustawienia zmiennej kontekstu przeglądarki,
- `setTestIdAttribute()` – charakteryzuje wartość identyfikatora elementów – domyślnie *testId.* Listing 109, przedstawia ciało tej metody.

Listing 109 Ciało metody *setTestIdAttribute()* w Playwright.

```
getPlaywright().selectors().setTestIdAttribute(testId);
```

W klasie testowej konieczne było utworzenie instancji obiektu klasy `Page`, stanowiącego centralny komponent środowiska Playwright, odpowiedzialny za realizację wszystkich interakcji z interfejsem strony internetowej podczas automatyzacji [89].

Dodatkowo, w celu zapewnienia prawidłowego cyklu życia testów oraz odpowiedniego zarządzania zasobami, zaimplementowano cztery metody opatrzone adnotacjami frameworka JUnit5:

- metoda z annotacją `@BeforeAll`. - Listing 108 przedstawia konfigurację, którą należało uruchomić w metodze z tym znacznikiem,
- metoda z annotacją `@BeforeEach`, dla każdego testu inicjalizowano nowy obiekt `page`,
- metoda z annotacją `@AfterEach`, zamykająca kontekst przeglądarki,
- metoda z annotacją `@AfterAll`, zamykająca przeglądarkę.

Mając poprawnie skonfigurowane środowisko autorka pracy przystąpiła do realizacji kodu dla scenariusza testowego na stronie Booking.com (Tabela 14).

Rysunek 19 przedstawia stronę główną serwisu Booking.com, do której należało przejść w pierwszym etapie scenariusza testowego. Następnie zaakceptowano pliki ciasteczkowe oraz wprowadzono nazwę miejsca docelowego do pola tekstowego reprezentowanego przez element klasy `input`. Każda z akcji została wykonana przy użyciu odpowiednich konstrukcji wbudowanych w bibliotekę Playwright, bazujących na obiekcie klasy `Locator` [32]. Przechowuje on każdy wyszukany element na stronie internetowej.

Aby przeczekać doładowanie zasobów, użyto metody `waitForLoadState()`, z argumentem przekazującym informację, że obiekt klasy `Page`, oczekuje na zmianę stanu w strukturze *DOM*. W momencie uzyskania pozytywnej weryfikacji, Playwright kontynuował wykonywanie dalszych kroków implementacji.

W trakcie realizacji kodu autorka napotkała pewne trudności techniczne. Pierwszy problem dotyczył obsługi obiektu odpowiedzialnego za wybór miejsca docelowego. W związku z tym zdecydowano wykorzystać hierarchiczną strukturę elementów na stronie — rozpoczynając od obiektu typu `option`, przechodząc następnie do powiązanego przycisku, a finalnie wybierając pierwszy z dostępnych wyników (Listing 110).

Listing 110 Implementacja wyboru miejsca docelowego na stronie Booking.com w Playwright.

```
page.navigate("https://www.booking.com/index.pl.html");

page
.getByRole(AriaRole.BUTTON, new Page.GetByRoleOptions()
            .setName("Akceptuj")).click();

Locator destination = page
      .locator("input[placeholder=\"Dokąd się wybierasz?\"]");

destination.fill("Zak");

page.waitForLoadState(LoadState.NETWORKIDLE);
page.locator("li[role='option']:has(div[role='button'])")
        .nth(0)
        .click();

assertTrue(destination.getAttribute("value").contains("Zakopane"));
```

- `AriaRole.Button` - rola elementu, do którego będzie odnosił się kod, w tym przypadku wyszukiwany jest obiekt o charakterze przycisku (z ang. *button*),
- `new Page.GetByRoleOptions().setName("Akceptuj")` - wyszukiwanie elementu według posiadanej nazwy,
- `fill()` – metoda przesyłająca do obiektu wartość tekstową,
- `waitForLoadState(LoadState.NETWORKIDLE)` – oczekuje na załadowanie całej zawartości strony,
- `locator()` – zwraca obiekt klasy `Locator`, spełniający warunki podane w argumencie,
- `nth()` – metoda określająca pozycję wybieranego elementu,
- `click()` – symulacja kliknięcia w przycisk przez użytkownika.

Drugi problem napotkany podczas prac, związany był z wyborem daty podróży. W tym przypadku zastosowano analogiczną metodę jak w implementacji przy użyciu Selenium, polegającą na wyszukaniu jednego z elementów pokrewnych oraz odczytaniu pierwszego z rodzeństwa, które spełniało określone warunki (Listing 111).

Listing 111 Implementacja wyboru daty na stronie Booking.com w Playwright.

```
while (!page.getByRole(AriaRole.HEADING, new Page.GetByRoleOptions()
      .setName("sierpień 2025")).isVisible()) {

  page.locator("button[aria-label=\"Następny miesiąc\"]").click();

  Locator calendar = page.getByRole(AriaRole.HEADING, new
            Page.GetByRoleOptions()
            .setName("sierpień 2025 "));

  Locator availableDates = calendar
      .locator("xpath=following-sibling::table")
            .first().locator("td[role='gridcell']
                  :not([aria-hidden='true'])
      :has(span[data-date])");

  availableDates.nth(0).waitFor(new
      Locator.WaitForOptions()
      .setState(WaitForSelectorState.VISIBLE));

  availableDates.nth(0).click();

  availableDates.nth(2).click();
}
```

Playwright posiada wbudowaną metodę `getByTestId()`, która domyślnie ma ustawioną wartość identyfikatora na *date-testid*. W momencie potrzeby użycia innego, domyślny można nadpisać własną zmienną (Listing 108). Przykładem wywołania funkcji było pobranie komponentu wyboru ilości podróżnych. Poprzez odniesienie się do wbudowanej metody dzięki temu autorka pracy w prosty sposób była w stanie otrzymać porządany element (Listing 112).

Listing 112 Użycie metody *getBytestId()* przy wyszukiwaniu elementów na stronie Booking.com w Playwright.

```
page.getByTestId("occupancy-config").click();
Locator occupancyPopup = page.getByTestId("occupancy-popup");

occupancyPopup.locator(".aaf9b6e287  .dc8366caa6 button").nth(0).click();
occupancyPopup.locator(".aaf9b6e287  .dc8366caa6 button").nth(1).click();
```

- `AriaRole.HEADING` – rola elementu, do którego będzie odnosił się kod, w tym przypadku wyszukiwany jest obiekt o charakterze nagłówka (z ang. *heading*),
- `waitFor()`- metoda oczekująca na pojawienie się elementu z przekazanymi dodatkowymi opcjami,
- `setState(WaitForSelectorState.VISIBLE)` – opcja widoczności obiektu,

- `getByTestId()` – wbudowana funkcja wyszukująca elementy o domyślnym lub podanym przez użytkownika identyfikatorze,

W celu zaimplementowania wyboru opcji z obiektu typu `Select`, z wykorzystaniem biblioteki Playwright, zastosowano metodę `selectOption()`. Jako argument przekazywana jest wartość odpowiadająca opcji, która powinna zostać wybrana w rozwijanym polu wyboru (Listing 113).

Na koniec kroku pierwszego wykonano weryfikację, czy wybór ilości osób podróżujących zgadza się z założeniami biznesowymi. Tabela 14 przedstawia wymagania dla testu *End-to-End* na stronie Booking.com

Listing 113 Wybór z listy rozwijanej na stronie Booking.com w Playwright.

```
page.locator(".ebf4591c8e").selectOption("2");

Locator divSearch = page.getByTestId("searchbox-layout-wide");
divSearch.locator("button[type=\"submit\"]").click();

String occupancyText = page
      .getByTestId("occupancy-config").textContent();

Assertions
   .assertEquals("3 dorosłych · 1 dziecko · 1 pokój", occupancyText);
```

- `selectOption()` – metoda do wybierania opcji z klasy `Select`,

- `textContent()` – zwraca tekst zawarty w elemencie.

Drugi krok scenariusza testowego, obejmował zamknięcie wyskakującego okna logowania oraz wybór odpowiednich filtrów dostępnych na stronie (Rysunek 22). Obiekty zostały zaimplementowane jako typ pola wyboru (z ang. *checkbox*). Dzięki wbudowanej w bibliotekę Playwright obsłudze odniesień do ról elementów na stronie, możliwe było w prosty i czytelny sposób wyszukanie oraz zaznaczenie odpowiednich pól [90]. Mechanizm ten istotnie uprościł implementację, zwiększając jednocześnie przejrzystość oraz odporność testów na zmiany struktury strony (Listing 114).

Listing 114 Implementacja filtrów na stronie Booking.com w Playwright.

```
page.locator("button[aria-label=\"Zamknij okno logowania.\"]").click();

Locator parkingCheckbox = page
      .getByRole(AriaRole.CHECKBOX, new Page.GetByRoleOptions()
      .setName(Pattern.compile(".*parking.*", Pattern.CASE_INSENSITIVE)))
      .nth(0);

parkingCheckbox.click();
```

- `AriaRole.CHECKBOX` - rola elementu, do którego będzie odnosił się kod, w tym przypadku wyszukiwany jest obiekt o charakterze pola wyboru (z ang. *checkbox*),

- `Pattern.compile()` – metoda umożliwiająca wyszukiwanie według podanego w argumencie wyrażenia regularnego,

- `Pattern.CASE_INSENSITIVE` – argument przekazujący informację aby ignorować wielkość liter,

Do filtrowania według cen na stronie użyto elementu suwaka (z ang. *Slider*), w Playwright jedną z ról elementów jest obiekt `Slider` odpowiadający danemu komponentowi (Listing 115).

Listing 115 Implementacja obsługi suwaka na stronie Booking.com w Playwright.

```
Locator maxSliderInput = page
      .getByRole(AriaRole.SLIDER, new Page.GetByRoleOptions()
      .setName("Maks."));

maxSliderInput.scrollIntoViewIfNeeded();
maxSliderInput.waitFor(new Locator.WaitForOptions()
      .setState(WaitForSelectorState.VISIBLE));

maxSliderInput.fill("700");

Page newPage = page.context()
      .waitForPage(() -> {
      page.locator("[data-testid='availability-cta-btn']")
      .nth(0).click();
});
```

- `AriaRole.SLIDER` - charakter elementu, do którego będzie odnosił się kod, w tym przypadku wyszukiwany jest obiekt o roli suwaka (z ang. *slider*),
- `scrollIntoViewIfNeeded()` – wbudowana metoda do przesuwania strony,
- `waitForPage()` – uruchamia nową zakładkę przy użyciu akcji wykonanej w ciele funkcji lambda,

W kolejnym etapie realizacji scenariusza testowego użytkownik został przekierowany na stronę szczegółów wybranego miejsca noclegowego (Rysunek 24). W celu zapewnienia, że wszystkie zasoby oraz elementy witryny zostały poprawnie załadowane, zastosowano metodę `waitForLoadState()`. Playwright nie wymaga dodatkowych instrukcji przewijania strony w celu uzyskania dostępu do niewidocznych w widoku elementów. Dzięki temu możliwe było bezpośrednie wyszukanie obiektu odpowiadającego za przejście do podsumowania rezerwacji. Automat bazuje na strukturze *DOM*.

Zgodnie z założeniami scenariusza testowego, przed przejściem do kolejnego kroku należało zweryfikować, czy aplikacja poprawnie sygnalizuje brak wyboru liczby pokoi poprzez wyświetlenie odpowiedniego komunikatu błędu. Po spełnieniu tego warunku z rozwijanej listy należało wybrać wartość „1” przy pomocy metody `selectOption()` działającej na obiekcie typu `Select`. Następnie ponownie wykonano akcję kliknięcia przycisku umożliwiającego przejście do podsumowania rezerwacji (Listing 116).

Listing 116 Implementacja rezerwacji obiektu noclegowego na stronie Booking.com w Playwright.

```
newPage.waitForLoadState();

String header = newPage.locator(".pp-header__title").nth(0).textContent();

Locator bookButton = newPage.locator(".hprt-reservation-cta");
bookButton.click();

Locator errorLabel = newPage
```

```
      .locator("span.invisible_spoken", new Page.LocatorOptions()
      .setHasText(Pattern.compile("Błąd")));

assertThat(errorLabel).isVisible();

String roomType = newPage
      .locator(".hprt-roomtype-icon-link ")
      .nth(0)
      .textContent();

newPage.getByLabel("Wybierz pokoje").nth(0).selectOption("1");

Locator bookButton2 = newPage.locator(".hprt-reservation-cta");
bookButton2.click();

assertTrue(newPage.getByRole(AriaRole.HEADING, new Page.GetByRoleOptions()
      .setName(header))
      .nth(0).isVisible());

Locator targetButton = newPage
      .getByTestId("booking-details-unit-selection");

targetButton.nth(0).click();
```

- `new Page.LocatorOptions().setHasText()` – wyszukiwanie element według posiadania zdefiniowanego tekstu,
- `getByLabel()` – metoda zwracająca obiekt według argumentu labelu (z ang. *label*)

Rysunek 26 przedstawia stronę rezerwacji, na której należało uzupełnić dane użytkownika, oraz pobrać nazwę wybranego pokoju i porównać z etykietą z poprzedniej strony. W przypadku implementacji tego kroku autorka pracy napotkała problem występowania białych znaków. Należało usunąć występujące znaki formatowania aby otrzymać czyste zdania zawierające dokładnie takie same obiekty klasy `String` (Listing 117).

Listing 117 Usunięcie białych znakow z tekstu na stronie Booking.com w Playwright.

```
String roomText = newPage
      .locator("li div.b99b6ef58f")
      .nth(0)
      .textContent();

roomText = roomText.replace("\u00A0", " ").trim();
roomType = roomType.replace("\u00A0", " ").trim();

roomText = roomText.replaceAll("\\s+", " ").trim();
roomType = roomType.replaceAll("\\s+", " ").trim();

assertTrue(roomText.contains(roomType));
```

W finalnym etapie scenariusza testowego dokonano pobrania wartości ceny wyświetlanej na stronie rezerwacji. Następnie, dla zapewnienia spójności danych oraz poprawności działania aplikacji, przeprowadzono porównanie jej z kwotą prezentowaną na stronie podsumowania (Rysunek 27). Po uzyskaniu pozytywnej weryfikacji zgodności obu zmiennych, test został zakończony, a kontekst przeglądarki zamknięty (Listing 118).

Listing 118 Podsumowanie na stronie Booking.com w Playwright.

```
String totalPrice1 = newPage
      .locator("[data-animate-price-group-name=\"bp_user_total_price\"]")
      .textContent();

newPage.locator("[name='book']").click();


String totalPrice2 = newPage
      .locator("[data-animate-price-group-name=\"bp_user_total_price\"]")
      .textContent();

assertEquals(totalPrice1, totalPrice2);
```

13.1.4.Analiza Porównawcza

Analiza objętości kodu wskazuje, że implementacja testów w Selenium zajęła niemal o 100 linii więcej niż odpowiednik w Playwright. Znaczna część dodatkowego skryptu wynikała z konieczności użycia zaawansowanych konstrukcji do wywoływania funkcji *JavaScript*, które były niezbędne do realizacji funkcjonalności niewspieranych natywnie przez Selenium, takich jak:

- przewijanie zakładki,

Playwright automatycznie obsługuje nawigację po stronie, wyszukując wymagane elementy bez konieczności ręcznej ingerencji w hierarchię *DOM*. Natomiast oczekiwanie na pełne załadowanie zasobów strony realizowane jest za pomocą wygodnej i intuicyjnej metody `page.waitForLoadState(LoadState.NETWORKIDLE)` [49], która zatrzymuje wykonywanie testu do momentu ustabilizowania się aktywności sieciowej.

W przypadku Selenium każdy wyszukiwany element wymagał zastosowania klasy `WebDriverWait` [43], która wspomagała oczekiwanie na pojawienie się obiektów lub ich gotowość do interakcji. Konstrukcja ta jest dość rozbudowana i wymuszała od użytkownika dobrej znajomości mechanizmów synchronizacji, co może utrudniać czytelność kodu. Natomiast Playwright integruje sposoby oczekiwania bezpośrednio w klasie `Locator` [32], co przekłada się na znacznie mniejszą ilość kodu oraz bardziej przejrzystą strukturę.

Dodatkowo, w Playwright dostępna jest metoda `getByTestId()`, która umożliwia wyszukiwanie elementów po ich identyfikatorach testowych, dzięki czemu kod jest bardziej intuicyjny i czytelny dla programisty. W przypadku Selenium odczyt identyfikatorów realizowany jest poprzez klasę `By` z metodą `CSSSelector()`, co skutkuje mniej eleganckim i trudniejszym do interpretacji zapisem.

W kontekście obsługi sytuacji, gdy element nie jest widoczny na stronie, autorka pracy implementując kod w Selenium musiała stworzyć dodatkową metodę pomocniczą, pozwalającą na pominięcie błędów związanych z opcjonalnymi polami, aby test mógł przebiegać bez zakłóceń. Z kolei Playwright oferuje wbudowaną metodę `isVisible()` działającą na obiekcie `Locator`, która umożliwia sprawne i wygodne zarządzanie widocznością elementów, redukując nakład pracy programistycznej.

Pomimo zastosowania mechanizmów oczekiwania na załadowanie strony Selenium czasem kończył testy niepowodzeniem przez brak odpowiedniego elementu z rozwijanej listy na głównej stronie

Booking.com (Rysunek 19). Playwright nie miał tego rodzaju problemów od momentu użycia konstrukcji oczekiwania na doładowanie zasobów.

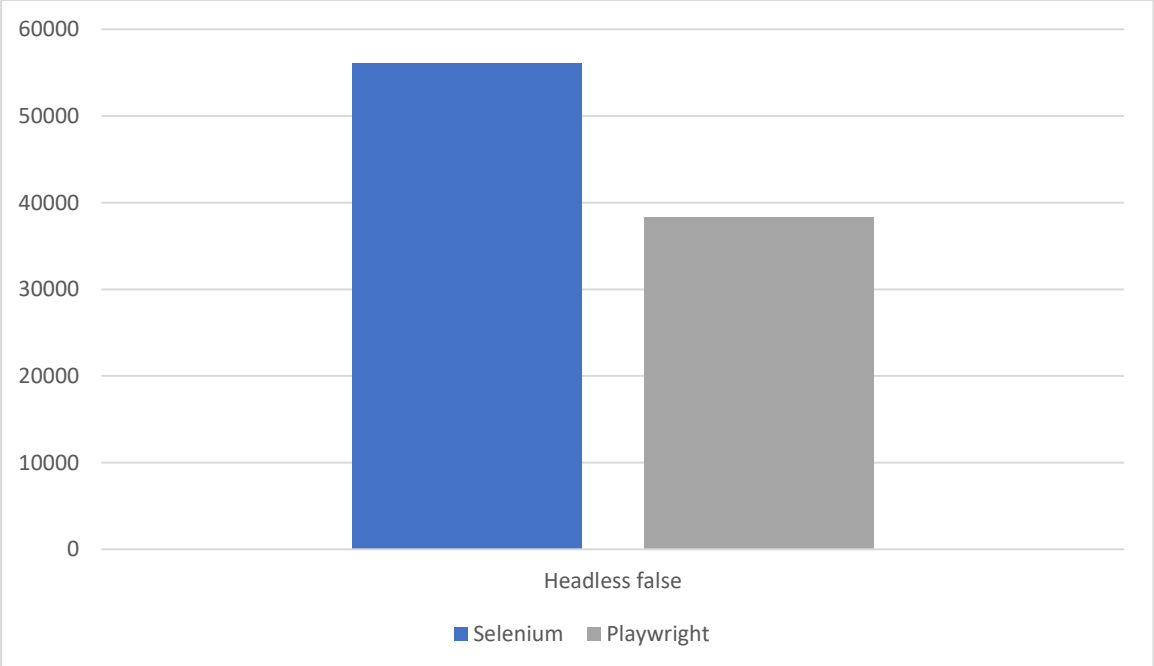
Analizując zużycie procesora podczas wykonywania testów *End-to-End* na stronie Booking.com, zaobserwowano, że Playwright wykazywał średnie zużycie na poziomie około 61%, podczas gdy Selenium oscylowało wokół 66%. Czasy wykonywania testów zostały przedstawione na załączonym wykresie (Wykres 12).

Testy przeprowadzono wyłącznie w trybie z interfejsem graficznym, ze względu na mechanizmy zabezpieczające po stronie Booking.com, które modyfikują strukturę elementów *DOM* i uniemożliwiają prawidłowe testowanie w trybie bez interfejsu graficznego. Listing 34 (Selenium) oraz Listing 40 (Playwright) przedstawia próbę oszukania takich mechanizmów, jednak w tym przypadku nieskuteczną.

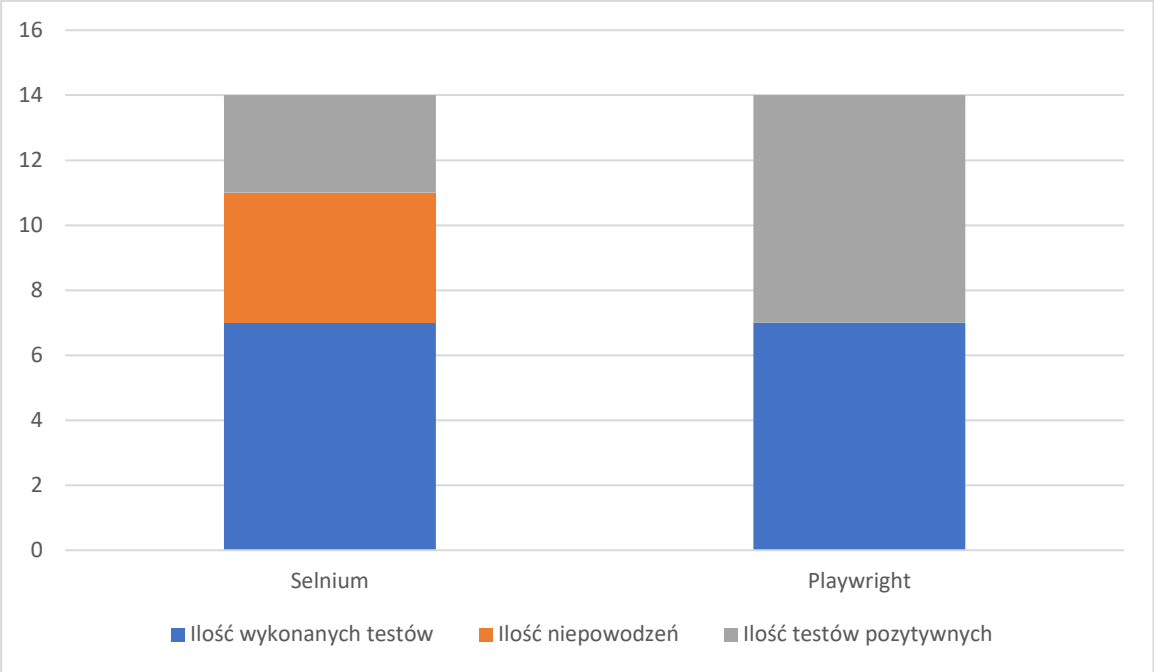
Ponadto, na drugim wykresie zaprezentowano porównanie obu narzędzi pod kątem liczby testów zakończonych niepowodzeniem, spowodowanych niedociągnięciami zasobów lub losowymi błędami związanymi z niemożnością wyszukania elementów (Wykres 13).

Na zakończenie implementacji autorka pracy stwierdziła, że Playwright sprawdził się lepiej pod względem:

- czytelności kodu,
- krótszej długości implementacji,
- szybszego czasu wykonywania testów,
- niższego zużycia procesora,
- mniejszej podatności na błędy.



Wykres 12 Porównanie czasów wykonania przy teście *End-to-End* na stronie Booking.com (mniej=lepiej). Źródło: opracowanie własne.



Wykres 13 Porównanie ilości testów zakończonych niepowodzeniem przy kilkukrotnym wywołaniu testów jeden po drugim przy testach *End-to-End* na stronie Booking.com (mniej=więcej). Źródło: opracowanie własne.

13.2. Testy aplikacji Orange.pl

W ramach rozdziału opisano konstrukcje wykorzystane przy testowaniu strony komercyjnej Orange.pl opisanej w rozdziale 4.1. Witryna stosuje wiekszą liczbę list rozwijanych oraz elementów dynamicznych, co zwiększa trudność jej weryfikacji automatycznej.

Z uwagi, że niektóre sposoby implementacji kodu powtarzały się przy testach *End-to-End* dla strony Booking.com, opisanych w rozdziale 13.1, autorka pracy postanowiła zawrzeć tylko te znacznie różniące się aby nie powielać kodu.

Na koniec sekcji zawarto analizę porównawczą oraz wykresy czasów wykonania i ilości testów zakonczonych niepowodzeniem.

13.2.1.Scenariusz testowy

Opis scenariusza testowego dla *End-to-End* na stronie Orange.pl.

Tabela 15 Scenariusz testowy dla testów *End-to-End* na stronie Orange.pl Źródło: opracowanie własne.

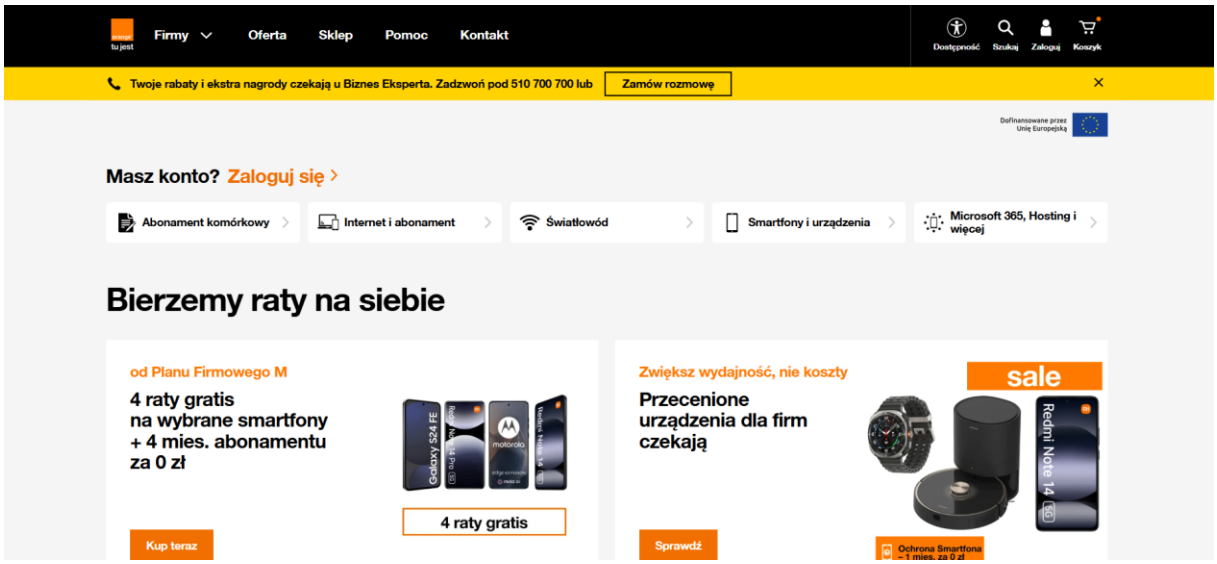
Lp.	Nazwa transakcji	Opis kroku
1.	Strona główna	Wprowadź adres https://www.orange.pl/ do przeglądarki i przejdź do strony głównej.

		Kliknij w menu rozwijane po lewej stronie na górze ,,Klient indywidualny", a następnie wybierz ,,Firmy".
2.	Strona z ofertami dla firm	Wciśnij na ukazanym *popup* przycisk ,,Kontynuuj bez wyrażania zgody →". Z menu z paska nawigacyjnego na górze wybierz ; Oferty ->,,Abonament komórkowy''. Zaznacz drugą ofertę od lewej bez dodatkowych urządzeń.
3.	Strona dobierania dodatkowych ofert	Zweryfikuj czy adresem strony jest ,,https://www.orange.pl/male-srednie-firmy/abonament-dla-firm/nowy-numer". Dołącz do koszyka ofertę ,,Ochrona Smartfona. Premium'' klikając przycisk ,,Dodaj". Z menu nawigacyjnego kliknij w Oferta->,,nowy numer''.
4.	Strona z ofertami dla firm	Z dostępnych ofert wybierz pierwszą wraz z opcją dodania urządzenia do koszyka.
6.	Strona z urządzeniami	Na pasku filtrów po prawej stronie wybierz i wciśnij przycisk: • ,,Marka", • ,,Standard karty SIM" • ,,Dual SIM" - ,,Tak". • ,,System operacyjny" - ,,Android". Z przefiltrowany urządzeń wybierz pierwszy z listy i kliknij w przycisk ,,Zmień liczbę rat" - wartość 6. Dodaj urządzenie do koszyka.
7.	Strona z dodatkowymi ofertami	Z dostępnych ofert wybierz ,,Połączenia międzynarodowe". Kliknij przycisk ,,Przejdź do koszyka".
8.	Koszyk	Rozwiń sekcję ,,Od 1. do 6. miesięca" oraz płatności. Wybierz przycisk na górze podsumowania ,,Wyczyść koszyk" i zaakceptuj akcję. Zweryfikuj, że koszyk został wyczyszczony i pojawił się tytuł ,,Twój koszyk jest pusty".

13.2.2.Implementacja w Selenium

Pierwsze dwa kroki scenariusza testowego (Tabela 15) nie spowodowały dodatkowego nakładu pracy. Konstrukcje wykorzystywane w implementacji zależały od właściwości elementów na stronie. Wiele elementów należało odnaleźć poprzez `By.Xpath` [45] odnoszącego się do *JavaScript*.

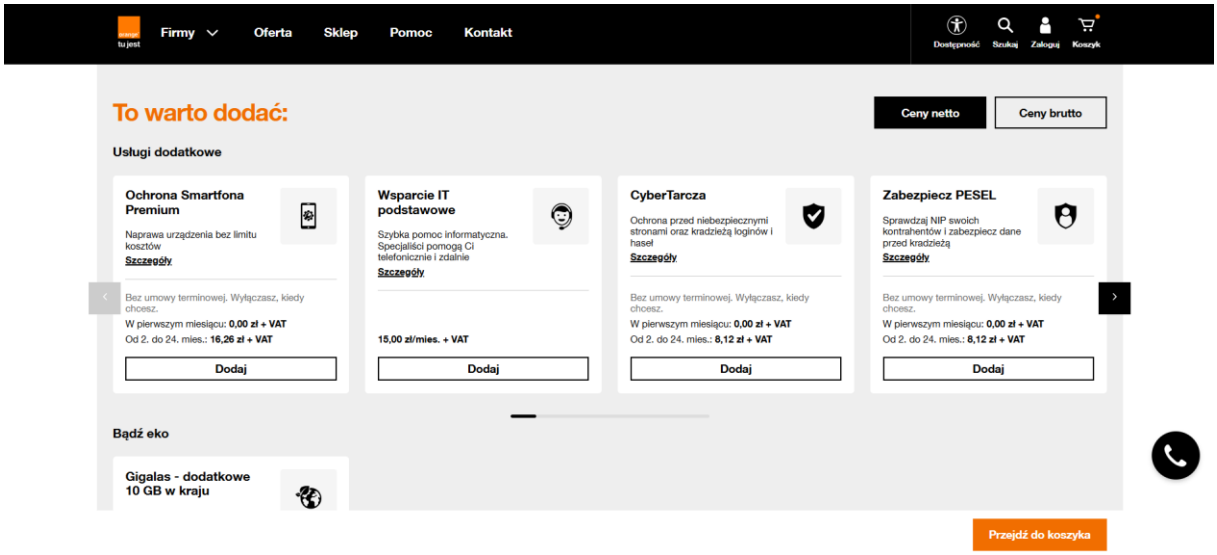
Listing 119 przedstawia przykład takiego wywołania jakim jest odnalezienie zakładki „Abonament komórkowy”, na głównej witrynie Orange.pl. (Rysunek 28).



Rysunek 28 Główna strona Orange.pl. Źródło: [16].

Listing 119 Implementacja wyszukania zakładki „Abonament komórkowy" na stronie Orange.pl w Selenium.

```
WebElement mobileSub =
driver.findElement(By.xpath(".//span[contains(text(),'Abonament
komórkowy')]"));
mobileSub.click();
```



Rysunek 29 Wybór pakietu „Ochrona Smartfona'' na stronie Orange.pl. Źródło: [16].

Problem implementacyjny pojawił się w momencie wyboru dodatkowego pakietu „Ochrona Smartfona Premium". Rysunek 29 przedstawia zakładkę wyboru pakietów.

Wymagało to użycia mechanizmu oczekiwania na pełne załadowanie strony z dostępnymi modułami, zanim możliwe stanie się wykonanie jakichkolwiek interakcji na elementach. W tym celu wykorzystano obiekt klasy `WebDriverWait` [43], który blokował wykonywanie dalszych operacji do momentu pojawienia się na stronie komponentów odpowiadających za prezentację pakietów.

Dodatkowym utrudnieniem była implementacja wszystkich dostępnych kontenerów jako instancja tej samej klasy z identycznym identyfikatorem *data-test-id="package-container"*. Takie rozwiązanie strukturalne utrudniało jednoznaczne zlokalizowanie wybranego w scenariuszu testowym pakietu. W związku z tym, autorka pracy zdecydowała się na odczytanie wszystkich komponentów z podanym atrybutem, a następnie, iterując po ich zawartości przy użyciu pętli `for-each`, wyszukała element zawierający tekst „Ochrona Smartfona Premium".

W momencie odnalezienia właściwego obiektu wykonano symulację kliknięcia w przycisk „Dodaj", będący częścią struktury danego kontenera. Po zatwierdzeniu wyboru, konieczne było ponowne oczekiwanie na aktualizację koszyka i pojawienie się w nim wybranego pakietu. Ze względu na dynamiczny charakter ładowania komponentów, czas oczekiwania często przekraczał domyślne 10 sekund określone w obiekcie `WebDriverWait`. W związku z tym, w celu zwiększenia niezawodności testu, zastosowano dodatkowe wstrzymanie wątku na 1,5 sekundy (Listing 120).

Jeśli element nie został załadowany w określonym czasie, weryfikacja kończyła się niepowodzeniem z powodu przekroczenia limitu czasu (z ang. *TimeoutException*). Jednak w większości przypadków przyjęta implementacja była wystarczająca do poprawnego przeprowadzenia scenariusza testowego.

Listing 120 Implementacja wyboru pakietu na stronie Orange.pl w Selenium.

```
wait.until(ExpectedConditions
      .visibilityOfElementLocated(
      By.xpath(".//div[@data-test-id='package-container']")));
```

```
List<WebElement> packageContainers = driver.findElements(
        By.xpath(".//div[@data-test-id='package-container']"));

for (WebElement container : packageContainers) {
    if (container.getText().contains("Ochrona Smartfona Premium")) {
        WebElement divAdd = container.findElement(
           By.xpath(".//div[contains(text(),'Dodaj')]"));
              divAdd.findElement(By.xpath("..")).click();
              break;
         }
}

Thread.sleep(1500);
```

Trzeci krok (Tabela 15), zakłada konieczność wyboru oferty z listy rozwijanej. Charakteryzuje się ona wysoką wrażliwością na ruchy kursora myszy — w sytuacji, gdy wskaźnik opuści obszar elementu, obiekt ulega natychmiastowemu zamknięciu. Tego rodzaju konstrukcja interfejsu użytkownika stanowiła istotne wyzwanie podczas implementacji testu automatycznego, znacząco wydłużając czas opracowania skutecznego rozwiązania.

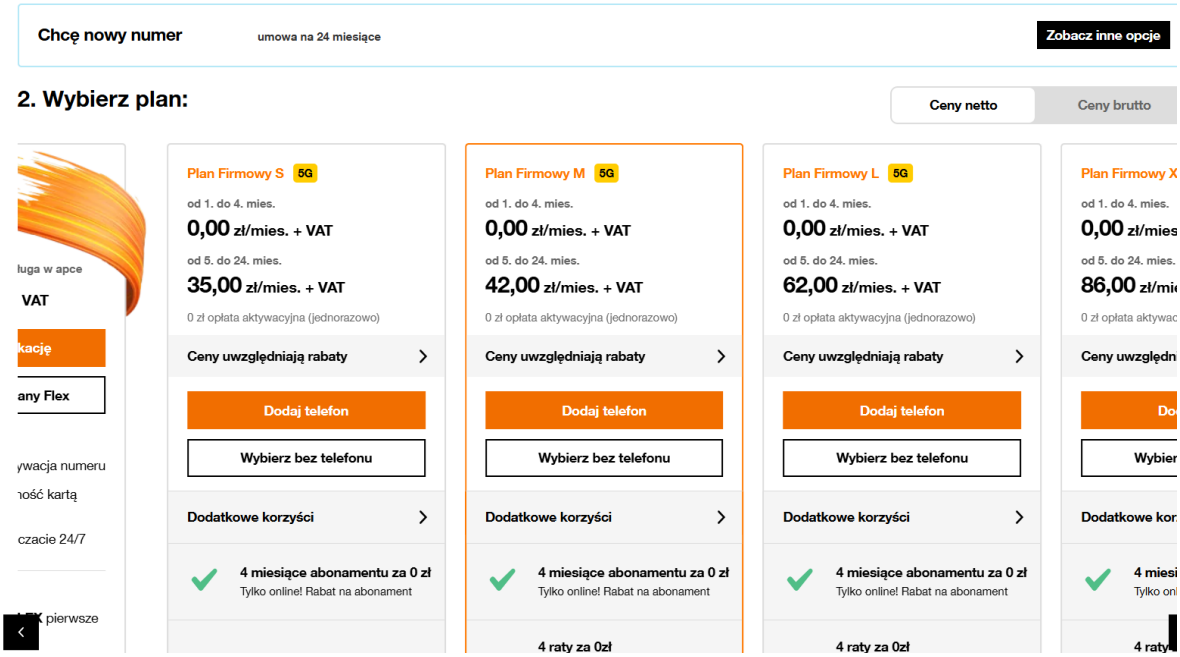
Ostatecznie autorka pracy zidentyfikowała sposób, który w zdecydowanej większości przypadków umożliwił poprawne przeprowadzenie testu. W tym celu wykorzystano klasę `Actions` [60] z biblioteki Selenium, która umożliwia bardziej zaawansowaną symulację interakcji użytkownika z interfejsem aplikacji. Klasa ta pozwala m.in. na wykonywanie akcji związanych z użyciem przycisków myszy lub klawiatury.

W opisywanym przypadku zastosowano metodę `moveToElement()` [92], która powoduje przesunięcie wskaźnika w centralny punkt bytu. Warunkiem poprawnego działania tej funkcji jest uprzednie zapewnienie widoczności danego obiektu w obrębie widoku przeglądarki. Listing 121 przedstawia implementację rozwiązania.

Listing 121 Implementacja rozwijanej listy ofert na stronie Orange.pl w Selenium.

```
Actions actions = new Actions(driver);

WebElement oferta = new WebDriverWait(driver,Duration.ofSeconds(10))
      .until(web->driver
            .findElement(
                  By.cssSelector("[data-test-id='menu_lvl1_Oferta']")));

actions.moveToElement(oferta).perform();
oferta.click();
```

- `moveToElement().perform()` – metoda symulująca najeżdżanie na widoczny element.



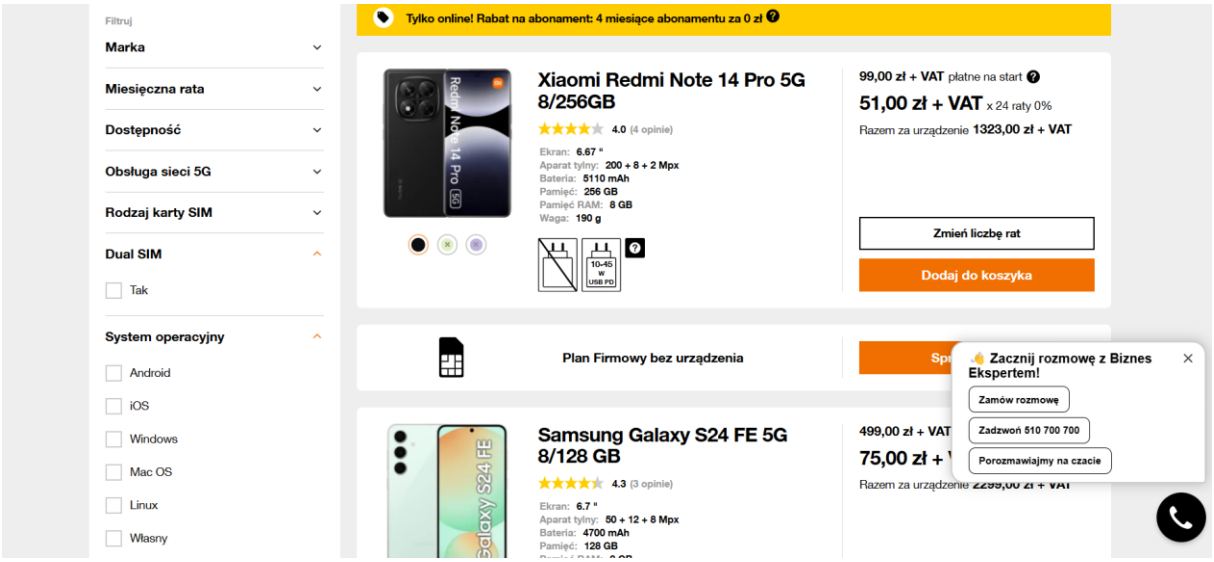
Rysunek 30 Wybór oferty na stronie Orange.pl. Źródło:[16]

Rysunek 30, przedstawia widok wyboru oferty na stronie internetowej Orange.pl. W implementacji interfejsu zastosowano, podobnie jak w przypadku dostępnych pakietów, wielokrotnie powielany komponent o identycznej strukturze i tym samym wbudowanym identyfikatorze.

Autorka pracy podjęła decyzję wyboru elementów według konkretnej pozycji, jaką zajmuje w strukturze strony. W tym celu, odczytano wszystkie wystąpienia danego obiektu w liście, a następnie dokonano wyboru elementu znajdującego się na trzecim indeksie. Taki sposób implementacji pozwolił na obejście ograniczeń wynikających z powtarzających się identyfikatorów i umożliwił skuteczne wykonanie kroku testowego (Listing 122).

Listing 122 Implementacja wyboru oferty Abonament Komórkowy na stronie Orange.pl w Selenium.

```
wait.until(ExpectedConditions
      .visibilityOfElementLocated(
      By.cssSelector("[data-test-id='atc-with-device']")));

List<WebElement> atcWithDevice = driver
      .findElements(By.cssSelector("[data-test-id='atc-with-device']"));

if (atcWithDevice.size() > 2) {
    atcWithDevice.get(2).click();
}
```



Rysunek 31 Filtry urządzeń na stronie Orange.pl. Źródło: [16].

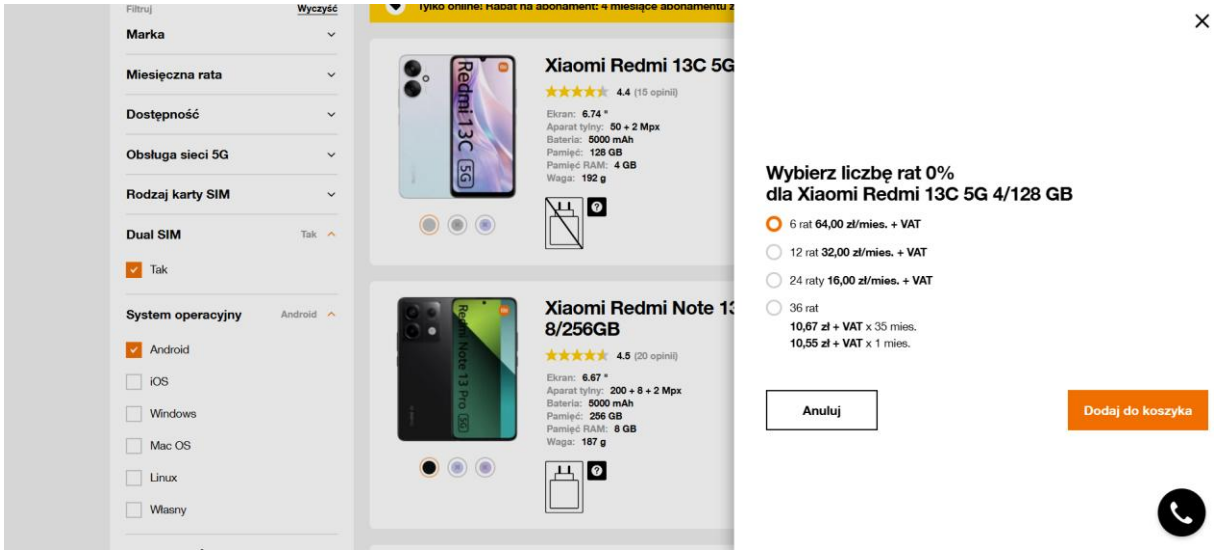
W kolejnym kroku scenariusza testowego (Tabela 15) konieczne było zastosowanie filtrów w celu zawężenia listy urządzeń do interesujących wartości. Każdy z nich został wybrany w analogiczny sposób, poprzez odwołanie się do zawartości tekstowej danego elementu przy użyciu lokalizatora *XPath* z klasy `By` [45]. Taka metoda umożliwiła jednoznaczne wskazanie odpowiednich opcji filtrujących w interfejsie strony i ich automatyczne zaznaczenie w ramach wykonywanego testu (Listing 123).

Listing 123 Implementacja filtrowania urządzeń na stronie Orange.pl w Selenium.

```
wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("//span[c
ontains(text(),'Marka')]")));
driver.findElement(By.xpath("//span[contains(text(),'Marka')]")).click();
```

W trakcie implementacji przy użyciu biblioteki Selenium, konieczne było wielokrotne zastosowanie konstrukcji umożliwiających oczekiwanie na załadowanie dynamicznych elementów strony. Wynikało to z faktu, iż liczne komponenty, takie jak np. dodawanie obiektu do koszyka w formie dynamicznego okna typu *popup*, wymagały dodatkowego czasu na pełne wyrenderowanie i udostępnienie do interakcji (Rysunek 32).

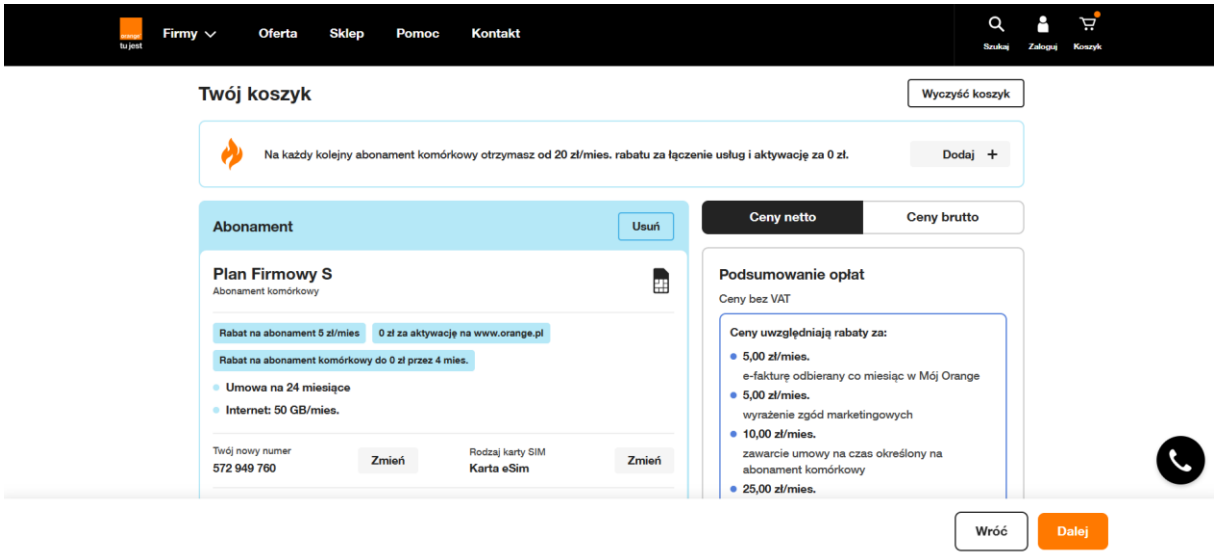
W opisywanych przypadkach zastosowano proste wywołanie metody `Thread.sleep()`, która wstrzymywała wykonanie wątku na określony czas. Pomimo że rozwiązanie to skutecznie eliminowało problemy związane z przedwczesnym wykonywaniem operacji na niedostępnych jeszcze elementach, jego sposób implementacji nie różnił się znacząco od innych zaprezentowanych wcześniej konstrukcji. Z tego względu zrezygnowano ze szegółowego omówienia wykonania.



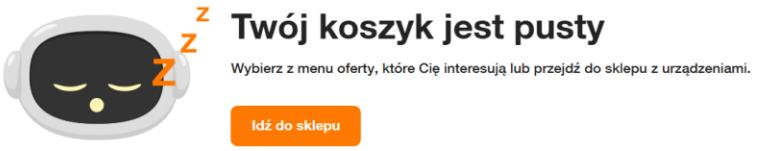
Rysunek 32 Wybór urządzenia na stronie Orange.pl. Źródło: [16].

Ostatnimi krokami wykonanymi w ramach testu typu *End-to-End* na stronie Orange.pl była weryfikacja poprawności wyświetlanych informacji na podsumowaniu koszyka. Rysunek 33 przedstawia stronę, na której w kolejnym kroku należało usunąć cały obiekt przyciskiem „Wyczyść koszyk".

Po wykonaniu tych operacji przeprowadzono prostą weryfikację, mającą na celu sprawdzenie, czy tytuł strony (Rysunek 34) jest zgodny z oczekiwanym. Listing 124 przezentuje proces realizacji kroku.



Rysunek 33 Widok koszyka na stronie Orange.pl. Źródło: [16].



Rysunek 34 Komunikat o pustym koszyku na stronie Orange.pl. Źródło: [16].

Listing 124 Implementacja weryfikacji poprawności komunikatu po usunięciu koszyka na stronie Orange.pl.

```
Assertions.assertEquals("Twój koszyk jest pusty", driver.findElement(
By.cssSelector("[data-test-id=\"info-panel-title\"]")).getText());
```

13.2.3.Implementacja w Playwright

W przypadku implementacji testów z wykorzystaniem biblioteki Playwright, wiele konstrukcji związanych z wyszukiwaniem elementów na stronie, opierało się na ponownym wykorzystaniu doświadczeń i rozwiązań wypracowanych podczas tworzenia wcześniejszych skryptów. Większość bardziej złożonych, interesujących i powtarzających się konstrukcji została szczegółowo omówiona przez autorkę w rozdziale 13.1.3. Z tego względu, niektóre nie będą powielane.

Ze względu na dynamiczny charakter działania strony Orange.pl oraz wydłużony czas ładowania poszczególnych zasobów, w implementacji testów konieczne było częste korzystanie z metod służących do oczekiwania na pełne załadowanie treści. W tym celu wykorzystano funkcjonalność wbudowaną w bibliotekę Playwright, wywoływaną na obiekcie klasy `Page` [42]. Rozwiązanie to pozwoliło na wstrzymanie wykonywania dalszych operacji testowych do momentu, gdy wszystkie wymagane elementy były gotowe do dalszych interakcji (Listing 125).

Listing 125 Metoda oczekująca na doładowanie zasobów w Playwright.

```
page.waitForLoadState(LoadState.NETWORKIDLE);
```

Implementacja trzeciego kroku scenariusza testowego – dodawanie pakietów do koszyka - została zrealizowana z wykorzystaniem konstrukcji wbudowanych w bibliotekę Playwright, które charakteryzują się wysoką czytelnością, nawet dla użytkowników dysponujących jedynie podstawową znajomością programowania. Rysunek 29 przedstawia stronę Orange.pl z pakietami.

W pierwszej kolejności odnaleziono wszystkie obiekty posiadające ten sam identyfikator. W Playwright tego typu elementy są przechowywane zbiorczo w jednej instancji klasy `Locator` [32], co w niektórych przypadkach może wpływać na ograniczenie przejrzystości kodu. Jednocześnie jednak

biblioteka udostępnia wygodną metodę umożliwiającą filtrowanie odnalezionych elementów na podstawie zawartości nagłówka. Po zidentyfikowaniu właściwego obiektu możliwe było pobranie przypisanego do niego przycisku odpowiedzialnego za dodanie produktu do koszyka i wykonanie na nim odpowiedniej akcji. Listing 126 prezentuje opisaną implementację.

Listing 126 Implementacja wyboru dodatkowego pakietu na stronie Orange.pl w Playwright.

```
Locator packageContainerList = page.getByTestId("package-container");

packageContainerList.filter(new Locator.FilterOptions()
      .setHas(page.getByRole(AriaRole.HEADING, new Page.GetByRoleOptions()
            .setName("Ochrona Smartfona Premium"))))
            .getByRole(AriaRole.BUTTON, new Locator.GetByRoleOptions()
                  .setName("Dodaj"))
      .click();
```

W bibliotece Playwright wybór oferty z rozwijanej listy (Rysunek 30), został zaimplementowany w sposób prosty i czytelny. Operacja ta realizowana jest poprzez wykorzystanie metody `hover()` z klasy `Locator`, która odpowiada za przesunięcie kursora nad wskazany element, co w przypadku dynamicznie rozwijanego menu umożliwia jego wyświetlenie (Listing 127). Następnie wykonywane jest kliknięcie na widoczny obiekt, aby zatwierdzić wybór i przejść do następnej strony z ofertami abonamentów komórkowych.

Listing 127 Implementacja obsługi dynamicznie rozwijanej listy na stronie Orange.pl w Playwright.

```
Locator oferta = page.getByTestId("menu_lvl1_Oferta");
oferta.hover();
oferta.click();
page.getByTestId("menu_lvl2_choice_Nowy_numer").click();
```

- `hover()` – metoda symulująca przesunięcie wskaźnika myszy na element.

Listing 128 realizuje wybór komponentu z przedstawionej listy ofert, poprzez użycie wbudowanej funkcji do pobierania identyfikatorów. W tym przypadku wskazano trzeci obiekt, a następnie wykonano symulację kliknięcia przycisku myszy. Spowodowało to przekierowanie na następną stronę filtrowania dostępnych urządzeń (Rysunek 31).

Listing 128 Implementacja wyboru ofert z telefonem na stronie Orange.pl w Playwright.

```
page.getByTestId("atc-with-device").nth(2).click();
```

Komponent kryteriów wyboru produktów został zaimplementowany jako obiekty typu `Button`, co umożliwiło ich intuicyjne i przejrzyste wyszukiwanie w kodzie testów. Dzięki temu autorka pracy mogła w prosty sposób odwołać się do każdego z nich, wykorzystując przypisaną do elementów rolę przycisków oraz nazwę widoczną na stronie. Takie podejście zapewniło wysoką czytelność oraz minimalizowało ryzyko błędów wynikających z potencjalnych zmian struktury *HTML*. Listing 129 przedstawia przykład realizacji tej operacji.

Listing 129 Implementacja wyboru filtrów na stronie Orange.pl w Playwright.

```
page.getByRole(AriaRole.BUTTON, new
Page.GetByRoleOptions().setName("Marka")).click();
```

Podczas dodawania urządzenia do koszyka użytkownik zobowiązany jest do wyboru liczby rat, w jakich zamierza spłacić zakup (Rysunek 32). W implementacji testu rozwiązanie to zrealizowano przy pomocy przycisków typu `RadioButton`. Niestety, ze względu na brak dedykowanych identyfikatorów pomocniczych w strukturze *HTML*, konieczne było zastosowanie konstrukcji *XPath*.

W praktyce, na obiekcie `page` wywołano metodę `locator()`, której parametrem był selektor w składni *JavaScript*, wskazujący przycisk typu *radio* z określoną wartością odpowiadającą liczbie rat. Podczas wykonywania testów wystąpiły jednak incydentalne niepowodzenia, których przyczyn autorka pracy nie zdołała jednoznacznie zidentyfikować. W związku z tym wprowadzono dodatkowy mechanizm oczekiwania na widoczność wskazanego elementu w strukturze *DOM* przed próbą wykonania na nim operacji (Listing 130).

Listing 130 Implementacja zaznaczania *RadioButton* w Playwright.

```
Locator radio = page.locator("label:has(input[type='radio'][value='6'])");
radio.waitFor(new
Locator.WaitForOptions().setState(WaitForSelectorState.VISIBLE));
radio.click();
```

Rysunek 33 przedstawia jeden z ostatnich kroków wykonywania akcji na koszyku, w którym należało zweryfikować poprawność wybranej liczby rat w podsumowaniu zamówienia. Dokonano sprawdzenia obecności rozwijanej listy z tekstem „od 1. do 6. miesięcy", który potwierdzał poprawny wybór wcześniej wskazanej opcji (Listing 130).

W implementacji testu wykorzystano w tym celu konstrukcję Playwright opartą na roli elementu. Dzięki metodzie selekcji po `AriaRole` możliwe było precyzyjne odnalezienie listy rozwijanej. Listing 131 realizuje operację wyszukiwania obiektu.

Listing 131 Weryfikacja poprawności podsumowania na stronie Orange.pl w Playwright.

```
page.getByRole(AriaRole.LISTITEM)
              .filter(new Locator.FilterOptions()
      .setHasText(Pattern.compile("Od 1. do 6. miesiąca")))
            .getByTestId("list-item-button").click();
```

- `AriaRole.LISTITEM` – rola elementu, do którego będzie odnosił się kod w tym przypadku wyszukiwany jest obiekt o charakterze składnika listy (z ang. *list item*),

Weryfikacja poprawności wyświetlanego tytułu strony po usunięciu zawartości koszyka (Rysunek 34), została zrealizowana z wykorzystaniem mechanizmu asercji dostępnego w bibliotece JUnit5 [30]. W tym celu pobrano treść elementu odpowiadającego nagłówkowi strony, korzystając z jego unikalnego identyfikatora. Następnie porównano uzyskaną wartość z oczekiwanym tekstem, co umożliwiło potwierdzenie poprawności przeprowadzonej operacji oraz finalnego stanu aplikacji po wykonaniu scenariusza testowego. Tabela 15 przedstawia wspomniany scenariusz. (Listing 132).

Listing 132 Weryfikacja wyświetlania poprawnego tytułu strony Orange.pl w Playwright.

```
Assertions
.assertEquals("Twój koszyk jest pusty",page.getByTestId("info-panel-title")
.textContent());
```

13.2.4.Analiza Porównawcza

W implementacji testu *End-to-End* dla strony Orange.pl, długość kodu dla biblioteki Selenium okazała się jedynie nieznacznie większa w porównaniu do rozwiązania stworzonego przy użyciu Playwright. W obu przypadkach zastosowano szeroki wachlarz opcji wbudowanych w wybrane narzędzia testowe.

W przypadku Selenium wykorzystano klasę `Actions` [60] do symulacji działań użytkownika, takich jak najeżdżanie kursorem czy wykonanie kliknięcia, co wymagało dodatkowej konfiguracji. Natomiast w bibliotece Playwright operacje te zrealizowano przy pomocy metod wbudowanych bezpośrednio w obiekt `Page`, który stanowi centralny punkt zarządzania elementami na stronie i nie wymaga dodatkowej implementacji.

Ze względu na dużą dynamiczność strony Orange.pl, w obu przypadkach konieczne było wielokrotne zastosowanie konstrukcji służących do oczekiwania na załadowanie zasobów przed wykonaniem kolejnych akcji. W Selenium wykorzystano klasę `WebDriverWait` [43], a w sytuacjach wymagających dodatkowego zabezpieczenia — również wywołania `Thread.sleep()`, w celu zagwarantowania dostępności wszystkich niezbędnych elementów w strukturze *DOM*.

Z kolei w Playwright zastosowano metodę `waitForLoadState(LoadState.NETWORKIDLE)`, która w sposób automatyczny wstrzymywała dalsze operacje do momentu, aż strona zakończy pobieranie zasobów, a żądania sieciowe będą nieaktywne. Rozwiązanie to okazało się w testach na stronie Orange.pl wysoce niezawodne i eliminowało konieczność ręcznego definiowania czasu oczekiwania.

Warto również zaznaczyć, że w przypadku elementów występujących wielokrotnie z tym samym identyfikatorem, biblioteka Playwright oferuje możliwość ich przefiltrowania oraz selekcji po zawartości tekstowej lub innych atrybutach przy pomocy dedykowanych metod na obiekcie `Locator` [32]. W przypadku Selenium konieczne było natomiast pobranie listy wszystkich wystąpień elementów i przeszukiwanie jej przy użyciu konstrukcji iteracyjnych, takich jak pętla *`for-each`*, w celu odnalezienia właściwego obiektu.

Porównując czasy wykonania testów *End-to-End* na stronie Orange.pl przy użyciu obu narzędzi, zaobserwowano, że w obu przypadkach realizacja scenariuszy wykonywała się dość długo. Zarówno w przypadku Selenium, jak i Playwright, pojedynczy test kończył się średnio po upływie około jednej minuty, przy czym w przypadku Selenium czas ten był nieco dłuższy.

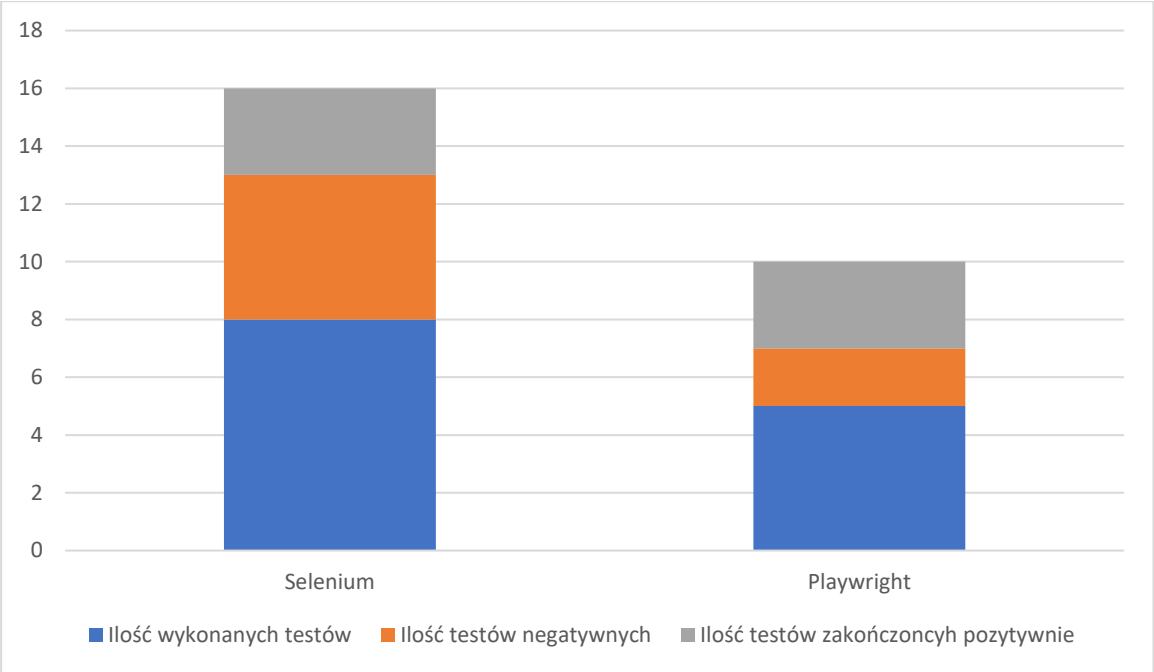
Wykres 14 ilustruje liczbę wykonanych prób testowych, niezbędnych do uzyskania co najmniej trzech pozytywnych wyników. Na podstawie przeprowadzonych pomiarów autorka pracy zaobserwowała, że żadne z analizowanych narzędzi nie osiągnęło pełnej niezawodności podczas weryfikacji testów na stronie Orange.pl. Aby uzyskać trzy pomyślnie weryfikacje, konieczne było wykonanie ośmiu prób z wykorzystaniem Selenium oraz pięciu prób przy użyciu Playwright, bez dokonywania modyfikacji w implementacji.

Zaobserwowane wahania dotyczące liczby koniecznych testów do wykonania oraz zmienność czasu ich realizacji były zjawiskami spodziewanymi i trudnymi do całkowitego wyeliminowania. Potwierdzają one, że zarówno dynamika testowanej aplikacji, jak i specyfika działania poszczególnych narzędzi testowych, mają istotny wpływ na stabilność oraz powtarzalność wyników.

Jeśli chodzi o średnie zużycie zasobów, w przypadku Selenium wynosiło w granicach 67%, a w przypadku Playwright około 62%.

Autorka pracy doszła do wniosków, że Playwright jest lepszym rozwiązaniem do implementowania testów *End-to-End* niż Selenium pod takimi względami jak:

- średni czas wykonania,
- zużycie pamięci,
- łatwiejsza implementacja,
- stabilność testów



Wykres 14 Porównanie skuteczności testów *End-to-End* na stronie Orange.pl (mniej=lepiej). Źródło: opracowanie własne.

Podsumowanie

Efektem przeprowadzonych badań jest kompleksowa analiza dwóch popularnych narzędzi do automatyzacji testów — Selenium oraz Playwright. Na podstawie przeprowadzonej implementacji opracowano szczegółowe wnioski, które stanowią obszerne źródło wiedzy na temat dostępnych funkcjonalności tych narzędzi oraz sposobów ich realizacji. Zebrane informacje umożliwiają świadomy i odpowiedni wybór rozwiązania w zależności od wymagań projektowych i jego specyfiki.

Przeprowadzone pomiary i obserwacje wykazały, że narzędzie Playwright przewyższa Selenium pod względem:

- krótszego czasu wykonywania testów,
- mniejszego zużycia pamięci
- oraz czytelności i przejrzystości kodu.

Dodatkowo, autorka pracy zaobserwowała, że Playwright charakteryzuje się prostszym sposobem implementacji oraz niższym progiem wejścia dla nowych użytkowników. Z kolei Selenium, pozostaje narzędziem niezwykle precyzyjnym, elastycznym i skalowalnym. Pozwala ono na implementację rozbudowanych scenariuszy testowych, dając większą kontrolę nad poszczególnymi elementami procesu automatyzacji.

Warto również zauważyć, że wiele funkcjonalności dostępnych w Playwright jest zaimplementowanych w sposób natywny i nie wymaga dodatkowych bibliotek. W przypadku Selenium, często zachodzi konieczność importowania zewnętrznych bibliotek w celu realizacji bardziej złożonych operacji.

Pewnym ograniczeniem, które napotkano podczas pracy z Playwright, był jednak okrojony zakres oficjalnego wsparcia dla języka *Java* w porównaniu z innymi językami programowania, takimi jak *JavaScript* czy *Python*. Choć w większości przypadków możliwe było zaimplementowanie wymaganych funkcji, ich składnia i dostępność różniły się od rozwiązań stosowanych w innych językach, co niekiedy wymagało dodatkowych poszukiwań.

Selenium posiada większą i bardziej aktywną społeczność użytkowników w kontekście języka *Java*, co przekłada się na większą ilość dostępnych materiałów edukacyjnych dedykowanych temu językowi programistycznemu. Natomiast Playwright wyróżnia się bardziej przejrzystą i przyjazną dokumentacją, zawierającą liczne, szczegółowe przykłady implementacji. Należy jednak zaznaczyć, że dokumentacja Playwright koncentruje się przede wszystkim na środowisku *Node.js*, co stanowi wyzwanie dla użytkowników korzystających z innych języków programowania, w tym *Java*.

Autorka pracy zauważyła wiele za oraz przeciw obu technologiom w zakresie implementacji testów automatycznych w języku *Java*. Pomimo napotkanych trudności uważa, że wiele porównań w niniejszej pracy wskazuje na korzyść Playwright. Tabela 16 zawiera dopełnienie analizy w postaci przedstawienia punktacji przydzielonej przez autorkę pracy na podstawie wniosków zawartych w rozdziałach 5 - 13.

Ocena w postaci 63 punktów jednoznacznie pokazuje, że Playwright poradził sobie lepiej i to właśnie to narzędzie autorka pracy wybrałaby pracując jako tester automatyczny.

Niniejszą pracę można w przyszłości rozszerzyć o analizę kolejnych narzędzi do automatyzacji testów oraz porównać ich funkcjonalności i efektywność z narzędziami opisanymi w tej pracy, co pozwoli na uzyskanie bardziej kompleksowego obrazu dostępnych rozwiązań w tym obszarze.

Tabela 16 Ocena efektywności Playwright vs Selenium w postaci punktacji.

Rodzaj testów	Selenium	Playwright
Wbudowany framework testowy (rozdział 5).	0	5
Obsługa zapytań *REST API* (rozdział 6)	3	5
Logowanie z użyciem *Http Basic Athentication* (rozdział 7.1)	3	5
Logwanie z użyciem *OAuth* (rozdział 7.2).	3	5
Logowanie z wykorzystaniem sesji (rozdział 7.3).	3	3
Przesyłanie pliku na stronę (rozdział 8.1).	1	5
Pobieranie pliku ze strony (rozdział 8.2).	4	5
Wykonywanie akcji *Drag-and-Drop* (rozdział 8.3).	3	5
Obsługa *Snapshot Aria* (rozdział 9)	4	5
Przełączanie między zakładkami (rozdział 10).	3	5
Symulacja zmiennych warunków sieci (rozdział 11)	5	3
Równoległe wykonywanie testów (rozdział 12).	4	4
Testy *End-to-End* (rozdział 13.1).	3	4
Testy *End-to-End* na w pełni dynamicznej stronie (rozdział 13.2).	4	4
Suma	43	63

Sposób przydzielania ocen:

- 0 – brak wsparcia,
- 1 – minimalne wsparcie,
- 2 – biblioteka posiada wsparcie ale problematyczne w implementacji,
- 3 – funkcjonalność obsługiwana. Wysoki czas wykonania, duża implementacja do osiągnięcia małego celu,
- 4 – możliwe wykonanie kosztem czasochłonnej implementacji, wysokim czasem wykonania, dużego zużycia pamięci,
- 5 – w pełni kompatybilne rozwiązanie, wyróżniające się niskim zużyciem pamięci oraz krótkim czasem wykonania testów.

Bibliografia

[1] Phill Powell, Ian Smalley, „Functional testing,” [Online]. Available: https://www.ibm.com/think/topics/functional-testing. [Data uzyskania dostępu: czerwiec 2025].

[2] W. Szafraniec, „Testy End-to-End,” [Online]. Available: https://www.wyszkolewas.com.pl/testy-end-to-end-e2e/. [Data uzyskania dostępu: czerwiec 2025].

[3] T. K. W. G. K. Agnieszka Dorota Wac, „Analiza porównawcza rozwiązań wykorzystywanych w testowaniu automatycznym,” czerwiec 2025.

[4] K. Divya Kumar, „The Impacts of Test Automation on Software's Cost, Quality and Time to Market,” 9 April 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050916001277. [Data uzyskania dostępu: 9 Kwiecień 2019].

[5] C. R. Institute, „World Quality Report (WQR) 2020-2021,” Capgemeni, Worldwide, 2020.

[6] A. Thorsen, 28 October 2024. [Online]. Available: https://www.leapwork.com/blog/top-20-test-automation-tools. [Data uzyskania dostępu: 28 Październik 2024].

[7] „Selenium,” [Online]. Available: https://www.selenium.dev/history/. [Data uzyskania dostępu: Maj 2025].

[8] M. Trzaska, Modelowanie i implementacja systemów informatycznych 2.0; ISBN: 978-83-976442-0-5, Warszawa: Mariusz Trzaska, 2025.

[9] „Selenium/WebDriver,” [Online]. Available: https://www.selenium.dev/documentation/webdriver/. [Data uzyskania dostępu: Maj 2025].

[10] „Selenium/SeleniumIde,” [Online]. Available: https://www.selenium.dev/selenium-ide/. [Data uzyskania dostępu: Maj 2025].

[11] „Selenium/SeleniumGrid,” [Online]. Available: https://www.selenium.dev/documentation/grid/. [Data uzyskania dostępu: Maj 2025].

[12] „SauceLabs,” [Online]. Available: https://saucelabs.com/. [Data uzyskania dostępu: czerwiec 2025].

[13] „BrowserStack,” [Online]. Available: https://www.browserstack.com/. [Data uzyskania dostępu: czerwiec 2025].

[14] „LambdaTest,” [Online]. Available: https://www.lambdatest.com/. [Data uzyskania dostępu: czerwiec 2025].

[15] „Playwright docs,” [Online]. Available: https://playwright.dev/java/. [Data uzyskania dostępu: 2025].

[16] „Single Page Application,” [Online]. Available: https://kissdigital.com/pl/blog/single-page-application-jak-dziala-spa-i-czym-sie-rozni-od-mpa. [Data uzyskania dostępu: czerwiec 2025].

[17] „Playwright Auto-waiting,” [Online]. Available: https://playwright.dev/docs/actionability.

[18] „Playwright Browser-Context,” [Online]. Available: https://playwright.dev/docs/browser-contexts. [Data uzyskania dostępu: 2025].

[19] „Playwright Emulation,” [Online]. Available: https://playwright.dev/docs/emulation. [Data uzyskania dostępu: 2025].

[20] „Playwrigth TestAssertions,” [Online]. Available: https://playwright.dev/java/docs/test-assertions. [Data uzyskania dostępu: 2025].

[21] „Strona Orange.pl," [Online]. Available: https://www.orange.pl/. [Data uzyskania dostępu: maj 2025].

[22] „Strona Github.com," [Online]. Available: https://github.com/. [Data uzyskania dostępu: czerwiec 2025].

[23] „Strona Booking.com," [Online]. Available: https://www.booking.com/index.pl.html?label=gen173nr-1BCAEoggI46AdIM1gEaLYBiAEBmAEeuAEXyAEM2AEB6AEBiAIBqAIDuALN_cvCBsACAdICJGNhZmM1MjQ1LTI5Y2ItNDFlNS05OTU2LWY4YWIwODU5MTEzY9gCBeACAQ&sid=fce1746c5a290a91e39816c23523cc13&keep_landing=1&sb_price_type=total&.[Data uzyskania dostępu: luty 2025].

[24] „Strona HackerRank.com," [Online]. Available: https://www.hackerrank.com/. [Data uzyskania dostępu: czerwiec 2025].

[25] „Strona StackOverflow," [Online]. Available: https://stackoverflow.com/. [Data uzyskania dostępu: czerwiec 2025]

[26] „Strona SauceDemo.com," [Online]. Available: https://www.saucedemo.com/. [Data uzyskania dostępu: czerwiec 2025].

[27] „Strona the-internet.herokuapp.com," [Online]. Available: https://the-internet.herokuapp.com/. [Data uzyskania dostępu: czerwiec 2025].

[28] „Strona jsonplaceholder.typicode.com," [Online]. Available: https://jsonplaceholder.typicode.com/. [Data uzyskania dostępu: czerwiec 2025].

[29] „Strona commitquality.com," [Online]. Available: https://commitquality.com/. [Data uzyskania dostępu: czerwiec 2025].

[30] Stefan Bechtold ,Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rencourt, Christian Stein, „Junit5 user guide," [Online]. Available: https://junit.org/junit5/docs/current/user-guide/. [Data uzyskania dostępu: 21 Maj 2025].

[31] „Maven docs," [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html. [Data uzyskania dostępu: 19 Maj 2025].

[32] „Locator Playwright," [Online]. Available: https://playwright.dev/docs/api/class-locator. [Data uzyskania dostępu: czerwiec 2025].

[33] „LocatorAssertions," [Online]. Available: https://playwright.dev/java/docs/api/class-locatorassertions. [Data uzyskania dostępu: czerwiec 2025].

[34] P. Bykowski, 2021. [Online]. Available: https://bykowski.pl/rest-api-efektywna-droga-do-zrozumienia/. [Data uzyskania dostępu: 28 Maj 2025].

[35] „Rest Assured," [Online]. Available: https://rest-assured.io/. [Data uzyskania dostępu: czerwiec 2025].

[36] V. Balasubramaniam, „Bealdung.com," [Online]. Available: https://www.baeldung.com/junit-testinstance-annotation. [Data uzyskania dostępu: Czerwiec 2025].

[37] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/docs/api/class-apirequestcontext. [Data uzyskania dostępu: Czerwiec 2025].

[38] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/api-testing. [Data uzyskania dostępu: Czerwiec 2025].

[39] A. Machado, „Simple HTTP Authentication: A Beginner's Guide," [Online]. Available: https://zuplo.com/blog/2024/07/31/simple-api-authentication. [Data uzyskania dostępu: 31 July 2024].

[40] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/api/class-browser#browser-new-context. [Data uzyskania dostępu: czerwiec 2025].

[41] „Dokumentacja Java- HttpCredentials," [Online]. Available: https://doc.akka.io/japi/akka-http/current/akka/http/scaladsl/model/headers/HttpCredentials.html. [Data uzyskania dostępu: czerwiec 2025].

[42] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/api/class-page. [Data uzyskania dostępu: czerwiec 2025].

[43] „Dokumentacja Selenium -WebDriverWait," [Online]. Available: https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/support/ui/WebDriverWait.html. [Data uzyskania dostępu: czerwiec 2025].

[44] R. Boid, Getting Started with OAuth2, O'Reilly Media, 2012. ISBN: 9781449311605

[45] „Dokumentacja Selenium," [Online]. Available: https://www.selenium.dev/documentation/webdriver/elements/locators/#xpath. [Data uzyskania dostępu: czerwiec 2025].

[46] „ChromeOptions," [Online]. Available: https://www.browserstack.com/guide/selenium-chrome-options. [Data uzyskania dostępu: czerwiec 2025].

[47] J. Overson, „Medium," [Online]. Available: https://jsoverson.medium.com/how-to-bypass-access-denied-pages-with-headless-chrome-87ddd5f3413c. [Data uzyskania dostępu: marzec 2025].

[48] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/locators#filtering-locators. [Data uzyskania dostępu: czerwiec 2025].

[49] P. Bhat, „WaitForLoadState," [Online]. Available: https://www.browserstack.com/guide/playwright-waitforloadstate. [Data uzyskania dostępu: czerwiec 2025].

[50] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/api/class-browser#browser-new-context.

[51] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/screenshots.

[52] K. Ahmed, „Medium," [Online]. Available: https://roadmap.sh/guides/session-based-authentication. [Data uzyskania dostępu: czerwiec 2025].

[53] „Dokumentacja Selenium," [Online]. Available: https://www.selenium.dev/documentation/webdriver/interactions/cookies/. [Data uzyskania dostępu: Styczeń 2025].

[54] „Session Based Login Playwright," [Online]. Available: https://playwright.dev/java/docs/auth#session-storage. [Data uzyskania dostępu: czerwiec 2025].

[55] „WebElement," [Online]. Available: https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/WebElement.html. [Data uzyskania dostępu: czerwiec 2025].

[56] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/input. [Data uzyskania dostępu: Luty 2025].

[57] „SWTestAcademy," [Online]. Available: https://www.swtestacademy.com/how-to-download-file-in-selenium//. [Data uzyskania dostępu: Czerwiec 2025].

[58] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/downloads. [Data uzyskania dostępu: luty 2025].

[59] „Dokumentacja Playwright - Download," [Online]. Available: https://playwright.dev/java/docs/api/class-download. [Data uzyskania dostępu: czerwiec 2025].

[60] „Selenium.dev," [Online]. Available: https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/interactions/Actions.html. [Data uzyskania dostępu: luty 2025].

[61] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/input. [Data uzyskania dostępu: luty 2025].

[62] S. Bharadwaj, „Aria Snapshot," [Online]. Available: https://dzone.com/articles/aria-snapshot-testing-playwright. [Data uzyskania dostępu: 2025 Marzec 05].

[63] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/aria-snapshots. [Data uzyskania dostępu: luty 2025].

[64] „JavaScriptExecutor," [Online]. Available: https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/JavascriptExecutor.html. [Data uzyskania dostępu: czerwiec 2025].

[65] „Dokumentacja Selenium Window Navigation," [Online]. Available: https://www.selenium.dev/documentation/webdriver/interactions/windows/. [Data uzyskania dostępu: Czerwiec 2025].

[66] „Dokumentacja Playwrgith - Actions," [Online]. Available: https://playwright.dev/java/docs/input#mouse-click. [Data uzyskania dostępu: Czerwiec 2025].

[67] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/api/class-page. [Data uzyskania dostępu: Czerwiec 2025].

[68] BandWidthPlace, „Speed Comparison: 5G, 4G, LTE, and 3G," [Online]. Available: https://www.bandwidthplace.com/article/speed-comparison-5g-4g-lte-3g. [Data uzyskania dostępu: luty 2025].

[69] S. Ugale, „Selenium 4: Chrome DevTools Protocol," [Online]. Available: https://applitools.com/blog/selenium-4-chrome-devtools/. [Data uzyskania dostępu: luty 2025].

[70] P. Durał, „Analiza strony internetowej narzędziem DevTools (Chrome)," [Online]. Available: https://ks.pl/blog/analiza-strony-internetowej-narzedziem-devtools. [Data uzyskania dostępu: czerwiec 2025].

[71] „Webkit," [Online]. Available: https://webkit.org/. [Data uzyskania dostępu: czerwiec 2025].

[72] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/api/class-cdpsession. [Data uzyskania dostępu: czerwiec 2025].

[73] „Dokumentacja Chrome DevTools," [Online]. Available: https://chromedevtools.github.io/devtools-protocol/. [Data uzyskania dostępu: czerwiec 2025].

[74] „BrowserStack," [Online]. Available: https://www.browserstack.com/docs/automate/playwright/browsers-and-os. [Data uzyskania dostępu: luty 2025].

[75] S. Radzymiński, „Awesome Testing," [Online]. Available: https://www.awesome-testing.com/2019/06/throttling-network-in-selenium-tests.html. [Data uzyskania dostępu: luty 2025].

[76] „BrowserStack," [Online]. Available: https://www.browserstack.com/guide/how-to-perform-network-throttling-in-safari. [Data uzyskania dostępu: czerwiec 2025].

[77] „Docker," [Online]. Available: https://docs.docker.com/desktop/setup/install/windows-install/. [Data uzyskania dostępu: marzec 2025].

[78] H. S. Nair, „Lambda Test," [Online]. Available: https://www.lambdatest.com/blog/what-is-parallel-testing-and-why-to-adopt-it/. [Data uzyskania dostępu: marzec 2025].

[79] „TestNG," [Online]. Available: https://testng.org/. [Data uzyskania dostępu: czerwiec 2025].

[80] „Dokumentacja Selenium Github," [Online]. Available: https://github.com/SeleniumHQ/docker-selenium/blob/trunk/README.md. [Data uzyskania dostępu: marzec 2025].

[81] „Dokumentacja Playwright," [Online]. Available: https://playwright.dev/java/docs/junit. [Data uzyskania dostępu: marzec 2025].

[82] M. L. Gonzals, „Bealdung,” [Online]. Available: https://www.baeldung.com/junit-5-runwith. [Data uzyskania dostępu: marzec 2025].

[83] M. Mazurek, „MMazurek,” [Online]. Available: https://mmazurek.dev/wzorzec-threadlocal-w-praktyce/. [Data uzyskania dostępu: marzec 2025].

[84] „Dokumentacja Playwright,” [Online]. Available: https://playwright.dev/java/docs/junit.

[85] „By,” [Online]. Available: https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/By.html. [Data uzyskania dostępu: czerwiec 2025].

[86] Ł. Tkaczyk, „BulldogJob,” [Online]. Available: https://bulldogjob.pl/readme/mozliwosci-junit-5. [Data uzyskania dostępu: czerwiec 2025].

[87] S. Dwievdi, „TestGrid,” [Online]. Available: https://testgrid.io/blog/javascriptexecutor-in-selenium/. [Data uzyskania dostępu: czerwiec 2025].

[88] „Dokumentacja Playwright,” [Online]. Available: https://playwright.dev/java/docs/writing-tests#test-isolation. [Data uzyskania dostępu: grudzień 2025].

[89] „Dokumentacja Playwright,” [Online]. Available: https://playwright.dev/java/docs/api/class-page. [Data uzyskania dostępu: grudzień 2025].

[90] „Dokumentacja Playwright,” [Online]. Available: https://playwright.dev/java/docs/locators#locate-by-role. [Data uzyskania dostępu: luty 2025].

[91] „Dokumentacja Selenium,” [Online]. Available: https://www.selenium.dev/documentation/webdriver/actions_api/. [Data uzyskania dostępu: luty 2025].

[92] „Dokumentacja Selenium,” [Online]. Available: https://www.selenium.dev/documentation/webdriver/actions_api/mouse/. [Data uzyskania dostępu: luty 2025].

Wykaz Wykresów

Wykaz Rysunków

Wykaz Tabel

Wykaz Listingów