JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 31 (2024) 145–150
Received: 20 May 2024
Accepted: 4 June 2024

Comparison of selected tools for automation testing of Web applications

Porównanie wybranych narzędzi do automatyzacji testów aplikacji internetowych

Piotr Pasławski*, Maciej Pańczyk

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

The article is as an introduction to software testing, considering its definition and reasons for errors appearing in code. Three selected libraries available in Python, used for automating tests of web applications, are presented: Selenium, Playwright, and Splinter. Subsequently, a detailed comparison of these tools is made based on specific comparative criteria.

Keywords: test automation; Web applications

Streszczenie

Artykuł jest wprowadzeniem do problematyki testowania oprogramowania, rozważając jego definicję i przyczyny pojawiania się błędów w kodzie. Przedstawiono trzy wybrane biblioteki dostępne w języku Python, wykorzystywane do automatyzacji testów aplikacji internetowych: Selenium, Playwright i Splinter. Następnie dokonano szczegółowego porównania tych narzędzi na podstawie określonych kryteriów porównawczych.

Słowa kluczowe: automatyzacja testów; aplikacje internetowe

*Corresponding author

Email address: piotr.paslawski@pollub.edu.pl (P. Pasławski)

Published under Creative Common License (CC BY 4.0 Int.)

1. Wstęp

Ewolucja oprogramowania jest napędzana postępem związanym ze sprzętem komputerowym. Rosnące moce obliczeniowe oraz pojemności pamięci masowych, a także postęp w technologiach graficznych i pozostałych usług pozwalają na tworzenie aplikacji, które są nowoczesne i znacznie wydajniejsze. Konsumenci i entuzjaści chętnie eksplorują potencjał szeroko rozumianej sztucznej inteligencji, Internetu Rzeczy, rozwiązań chmurowych lub podejścia do tworzenia wieloplatformowych aplikacji. Ostatni aspekt nabiera szczególnego znaczenia dla przedsiębiorców, którzy dążą do maksymalizacji swojej obecności na rynku. Jednak, żeby to osiągnąć, należy zapewnić, że oprogramowanie jest w stanie poprawnie działać na różnorodnych urządzeniach [1]. Dodatkowym, przydatnym aspektem, jest też możliwość korzystania z oprogramowania z dowolnego miejsca za pośrednictwem Internetu.

Niezależnie od aspektów technicznych czy specyfiki danej aplikacji, koniecznym etapem w cyklu wytwarzania każdego rodzaju oprogramowania jest proces testowania. Szacuje się, że aż 70% czasu poświęconego na tworzenie aplikacji jest pochłaniane przez proces testowania [2]. Osoby zaangażowane w ten proces mają jednocześnie możliwość dokonywania subiektywnej oceny produktu lub identyfikacji potencjalnych obszarów do udoskonalenia o nowe funkcjonalności, co w rezultacie podnosi jego wartość rynkową. W czasach, w których rynek jest nasycony różnymi produktami, rośnie potrzeba tworzenia niezawodnych i bezpiecznych aplikacji – czyli takich, w których liczba defektów jest minimalna, a najlepiej w ogóle niezauważalna przez użytkowników. Wiadome jest, że błędy mogą być popełniane na każdym etapie, ale wraz z rozwojem projektu stają się one coraz bardziej kosztowne i czasochłonne, dlatego należy je ograniczać jak najszybciej [3].

Występowanie błędów w oprogramowaniu może wynikać z różnych przyczyn. Jednakże, we wszystkich źródłach literaturowych wskazuje się, że jednym z potencjalnych powodów jest błąd ludzki. Choć środowiska programistyczne chronią przed błędami syntaktycznymi, niestety błędy semantyczne w kodzie mogą nadal występować. Deweloperzy mogą też niepoprawnie rozumieć dokumentację systemu czy specyfikację aplikacji. Dodatkowo, presja czasu lub problemy związane z komunikacją, mogą prowadzić do niedostatecznego zapoznania się z wymaganiami projektu i oczekiwaniami klienta.

Inne przyczyny usterek mogą dotyczyć różnic między środowiskiem deweloperskim i produkcyjnym, szczególnie jeśli chodzi o zależności lub konfiguracje systemowe. Takie błędy mogą być trudniejsze do wykrycia na etapie tworzenia oprogramowania. Ponadto, aktualizacje systemów czy migracje środowiskowe mogą również wywoływać nieoczekiwane problemy. Zrozumienie potencjalnych zagrożeń w tym obszarze może mieć istotny wpływ na skuteczne zarządzanie ryzykiem i zwiększenie jakości oprogramowania.

Celem niniejszego artykułu jest zestawienie i przeprowadzenie szczegółowego porównania narzędzi Selenium, Playwright i Splinter, służących do automatyzacji testów aplikacji internetowych, dostępnych jako biblioteki języka Python.

2. Automatyzacja testów oprogramowania

Testowanie oprogramowania polega na dokonaniu oceny w celu znalezienia usterek lub błędów [4]. Jako integralny proces deweloperski, weryfikuje czy aplikacja jest

bezpieczna, niezawodna i spełnia oczekiwania konsumentów oraz interesariuszy. To oznacza, że nie jest to tylko proces wykrywania defektów aplikacji, ale także pełen zestaw działań, który zapewnia, że dane oprogramowanie jest zgodne ze specyfikacją, wydajne, konkurencyjne i gotowe do wprowadzenia na rynek. W przypadku testowania aplikacji internetowych, cele testowania pozostają takie same jak w przypadku testowania aplikacji klasycznych. Różnice w wyborze narzędzi lub w podjętych strategiach wynikają jedynie z odmiennych środowisk uruchomieniowych aplikacji.

Dzięki automatyzacji, cały proces testowania może odbywać się bez interakcji człowieka lub w ograniczonym stopniu zaangażowania. Wykonywanie przypadków testowych i porównywanie wyników z oczekiwanymi rezultatami w pełni automatyczny sposób umożliwia programistom szybsze podejmowanie odpowiednich działań w celu poprawy jakości oprogramowania. Testy automatyczne można wielokrotnie uruchamiać, na przykład po wdrożeniu zmian lub po dodaniu nowej funkcjonalności i za każdym razem zostaną wykonane dokładnie te same czynności jednakowo. Dodatkowo, takie testy mogą wykonywać zadania trudne dla testerów manualnych, na przykład symulacja obciążenia.

W związku z powyższym, wykonywanie badań i eksperymentów w obszarze testowania oprogramowania, między innymi we wspomniany wcześniej sposób automatyczny, jest imperatywem w środowisku programistycznym i na pewno wpłynie pozytywnie na proces testowania w celu zapewnienia jak najwyższej jakości oprogramowania.

3. Przegląd literatury

W pierwszym omawianym artykule, pod tytułem „Testing Web-based applications: The state of the art and future trends" [5], autorzy przedstawiają wyzwania związane z testowaniem aplikacji internetowych z powodu ich skomplikowanej architektury, rozproszonej struktury oraz różnorodności komponentów, często generowanych dynamicznie na podstawie danych wprowadzanych przez użytkowników albo stanu serwera. Słusznie podkreślają, iż aplikacja internetowa powinna być testowana z dwóch perspektyw – po pierwsze, powinno się weryfikować zgodność z wymaganiami funkcjonalnymi, a po drugie, należy badać wydajność, stabilność czy responsywność środowiska uruchomieniowego aplikacji, co jest równie kluczowe.

Drugą pozycją literaturową jest obszerna książka „Software testing and quality assurance: Theory and practice" [4], zawierająca solidne fundamenty teoretyczne, jednocześnie koncentrująca się na praktyce w zakresie testowania oprogramowania. Dodatkowym atutem są liczne przykłady oraz zadania do samodzielnego rozwiązania, a także opisy norm ISO związanych z testowaniem. Autorzy we wstępie stwierdzają, że ta książka wypełnia lukę wśród pozycji literaturowych na temat testowania oprogramowania oraz zapewniania jakości i rzeczywiście jest cennym źródłem wiedzy dla deweloperów na każdym etapie doświadczenia zawodowego.

Kolejny rozważany artykuł to „A study of automated software testing: Automation tools and frameworks" [3], w którym autorzy już na wstępie podkreślają, że szybsze dostarczenie wysokiej jakości oprogramowania wiąże się z przeprowadzaniem skutecznych testów. Przedstawiają statystyki, z których wynika, że błędy programistów generują rocznie spore wydatki, których można uniknąć. Z tego powodu w artykule kładziony jest nacisk na tworzenie efektywnych testów, a szczególnie na korzystanie z gotowych szkieletów programistycznych służących do ich automatyzacji. W przytaczanej publikacji są pokazane zalety i wady testów automatycznych, kategorie narzędzi do automatyzacji, a także porównanie wybranych narzędzi,

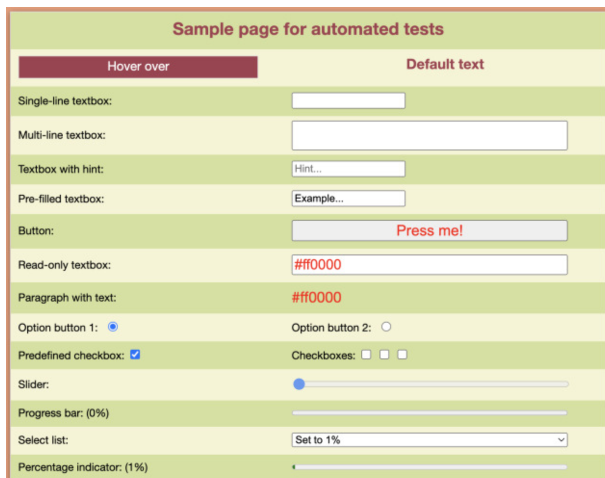
Analizując dany temat warto też zwrócić uwagę na artykuł „A survey on Web application penetration testing" [6], który uzupełnia powyższe zagadnienia o aspekty związane z testowaniem bezpieczeństwa aplikacji internetowych. Zdaniem autorów, wzrost liczby luk w zabezpieczeniach oprogramowania podkreśla istotność przeprowadzania testów penetracyjnych. Te testy są niezbędnym krokiem w ocenie podatności systemu na ataki oraz sprawdzeniu skuteczności jego zabezpieczeń. Biorąc pod uwagę to, że ręczne testy penetracyjne są czasochłonne i wymagają dużego wysiłku, autorzy skupili się w badaniach na automatycznej alternatywie takich testów. W artykule opisują metody ich implementacji, dostępne narzędzia i programy, a także środki zabezpieczające, które należy stosować podczas tworzenia oprogramowania.

W trakcie prowadzenia badań literaturowych na temat wytwarzania oprogramowania i jego testowania, artykuł „The humble programmer" autorstwa Edsgera Dijkstry może stanowić źródło inspiracji dla współczesnych programistów. Autor opisuje różne aspekty kodowania, dokonując refleksji na temat środowiska technologicznego lat siedemdziesiątych i ówczesnych informatycznych osiągnięć. W kontekście testowania oprogramowania Dijkstra napisał, że testowanie może wykazać obecność błędów, lecz nie powinno się najpierw implementować programu, a dopiero później go sprawdzać. Wręcz przeciwnie – najpierw trzeba opracować dowód poprawności, a następnie stworzyć działający na tej podstawie program [7], co jest zgodne z powszechnie stosowanym modelem tworzenia oprogramowania, w którym oprogramowanie jest tworzone na podstawie zdefiniowanych najpierw testów.

4. Metoda badań

W celu przeprowadzenia porównania narzędzi Selenium, Playwright i Splinter stworzono witrynę w języku HTML, wyposażoną w podstawowe elementy do interakcji z narzędziami automatyzującymi testy. Są to między innymi pola tekstowe jedno- i wielolinijkowe, paragrafy z tekstem, przyciski opcji, pola wyboru czy suwaki. Zawartość witryny symbolizuje to, z czym można spotkać się w Internecie. Przykładowo, pola tekstowe reprezentują formularze, a paragrafy z tekstem to sekcje z treścią strony. Elementy zostały zaprojektowane w taki sposób, żeby odzwierciedlać możliwe interakcje użytkownika z

rzeczywistą aplikacją internetową. W ten sposób stworzono środowisko, które pozwala sprawdzić, jak wybrane narzędzia do automatyzacji testów radzą sobie z różnymi aspektami aplikacji internetowych.



Rysunek 1: Wygląd strony przygotowanej na potrzeby badania.

Po przygotowaniu strony, przystąpiono do formułowania przypadków testowych, obejmując następujące aspekty:

1. Weryfikacja poprawności tytułu strony w karcie przeglądarki.
2. Sprawdzenie, czy układ elementów jest w strukturze tabelarycznej.
3. Sprawdzenie zgodności tekstu z nagłówka.
4. Testowanie funkcjonalności rozwijanego menu, zarówno jego wyświetlania, jak i ukrywania.
5. Weryfikacja zmiany domyślnego tekstu po wyborze opcji z listy rozwijanej.
6. Weryfikacja poprawności wprowadzenia tekstu w pola tekstowe.
7. Sprawdzenie treści domyślnej w polu tekstowym.
8. Weryfikacja zmiany treści i koloru tekstu po kliknięciu przycisku, zarówno w polu tekstowym, jak i akapicie.
9. Weryfikacja niemożliwości edycji pola tekstowego tylko do odczytu.
10. Testowanie zachowania predefiniowanego tekstu w polu tekstowym po edycji innych pól.
11. Sprawdzenie zaznaczenia przycisku po jego kliknięciu.
12. Sprawdzenie, czy predefiniowane pole wyboru jest domyślnie zaznaczone.
13. Weryfikacja zaznaczenia pola wyboru po jego kliknięciu.
14. Weryfikacja odznaczenia pola wyboru po dwukrotnym kliknięciu.
15. Testowanie jednoczesnego zaznaczenia wielu pól wyboru.
16. Weryfikacja zmiany paska postępu po przesunięciu suwaka.
17. Sprawdzenie aktualizacji wartości paska postępu po kliknięciu suwaka.
18. Test zmiany wskaźnika procentowego po wyborze opcji w rozwijanym menu.
19. Sprawdzenie braku aktualizacji wartości wskaźnika procentowego po kliknięciu na niego.
20. Weryfikacja poprawności wyświetlenia strony testowej po odświeżeniu.

Powyższe przypadki testowe zostały zaimplementowane w trzech odrębnych skryptach języka Python. Pierwszy z nich wykorzystuje bibliotekę Selenium, drugi opiera się na narzędziu Playwright, a trzeci korzysta z biblioteki Splinter. Skrypty są opracowane tak, aby wykonywać te same operacje testowe, zapewniając maksymalną wiarygodność badania.

Po zaimplementowaniu testów automatycznych, stworzono skrypt porównawczy, który uruchamiał każdy z nich w dwóch trybach – *headless* i *no headless*. Pierwszy z nich oznacza, że przypadki testowe są wykonywane w przeglądarce bez wyświetlania interfejsu graficznego dla użytkownika, podczas gdy drugi tryb umożliwia widoczność przeglądarki podczas działania. W trakcie wykonywania tych testów, skrypt porównawczy monitorował wydajność podzespołów komputera i zapisywał dane do plików *csv*. Monitorowanie wydajności obejmowało następujące aspekty:

1. Czas trwania skryptu w sekundach.
2. Procentowe obciążenie procesora przed uruchomieniem skryptu.
3. Procentowe obciążenie procesora mierzone co sekundę.
4. Liczba przełączeń kontekstu procesora mierzona co sekundę.
5. Liczba przerwań procesora mierzona co sekundę.
6. Procentowe zużycie pamięci operacyjnej przed uruchomieniem skryptu.
7. Procentowe zużycie pamięci operacyjnej mierzone co sekundę.
8. Rozmiar pamięci operacyjnej w użyciu przez proces w bajtach mierzony co sekundę.
9. Liczba odczytanych bajtów z dysku w trakcie trwania skryptu.
10. Liczba zapisanych bajtów na dysk w trakcie trwania skryptu.

Badanie zostało przeprowadzone na laptopie *MSI GL63 8SC* z systemem operacyjnym *Linux Ubuntu* (wersja 22.04.3), o następującej specyfikacji:

Procesor: *Intel i7-8750H, 6-rdzeniowy*
Pamięć operacyjna: *32 GB DDR4*
Karta graficzna: *NVIDIA GeForce GTX 1650*
Dysk twardy: *Patriot Burst SBFMK1.3, SSD 1 TB*

Przed eksperymentem system operacyjny został przywrócony do ustawień fabrycznych z opcją usunięcia plików użytkownika i zainstalowanego dodatkowego oprogramowania. Bezpośrednio przed uruchomieniem skryptów zadbano o to, żeby wyłączyć działające w tle programy, aby zminimalizować czynniki, które mogłyby negatywnie wpłynąć na wyniki badania.

5. Charakterystyka wybranych narzędzi

Do badania wybrano Selenium, Playwright oraz Splinter, spośród dostępnych bibliotek języka Python do automatycznego testowania aplikacji internetowych. Dokonano wyboru ze względu na ich funkcjonalność, dostępność

dokumentacji i zainteresowanie wśród programistów w sieci. Są to narzędzia typu *Web Scrapping*, czyli służą do interakcji z elementami na stronie na podstawie jej struktury HTML i symulację działania użytkownika. Znajomość najważniejszych funkcjonalności i dostarczanych dodatkowych narzędzi jest nieodzowna do efektywnego pisania testów automatycznych. Poniżej przedstawiono cechy charakterystyczne wybranych do badania bibliotek w celu porównania ich możliwości.

Selenium jest dostępne w językach Python, Java, C#, Ruby, JavaScript i Kotlin i wspiera przeglądarki takie jak Google Chrome, Mozilla Firefox, Microsoft Edge, Safari i Opera. To narzędzie wyróżnia się między innymi możliwością uruchamiania skryptów równolegle w wielu przeglądarkach i na różnych maszynach jednocześnie, z użyciem narzędzia Selenium Grid, co może być przydatne podczas automatycznego testowania kompatybilności aplikacji z różnymi przeglądarkami lub systemami operacyjnymi [8].

Playwright pobiera wskazany silnik przeglądarki – Chromium, WebKit lub Firefox – i automatyzuje w niej czynności. Jest dostępny w językach Python, Java, C# i JavaScript. Playwright wyróżnia się wbudowanymi narzędziami deweloperskimi, takimi jak Codegen, który automatycznie generuje kod testu automatycznego na podstawie interakcji programisty ze stroną, Playwright Inspector do debugowania kodu w czasie rzeczywistym oraz Trace Viewer, dzięki któremu można prześledzić działania i reakcję witryny na podstawie zrzutów ekranu i zapisanego kodu HTML po każdej akcji [9].

Splinter to szkielet programistyczny oparty na Selenium, co oznacza, że dziedziczy jego cechy techniczne. Zapewnia wyższy poziom abstrakcji, ukrywając szczegóły implementacyjne Selenium, co ułatwia pisanie testów automatycznych poprzez intuicyjne API. W porównaniu z czystym Selenium, Splinter oferuje liczne ulepszenia i poprawki, usprawniając proces tworzenia testów automatycznych [10].

6. Wyniki badania

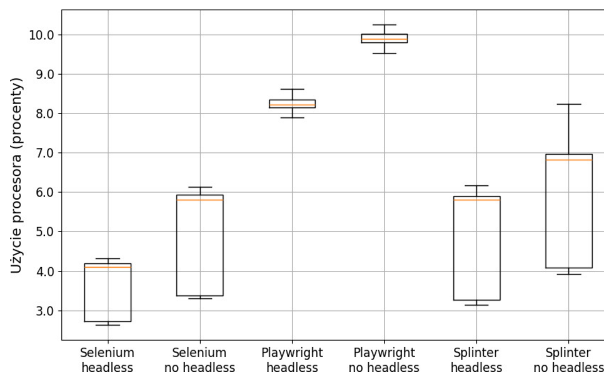
W poniższej tabeli umieszczono porównanie średniego czasu działania skryptów w obydwu trybach.

Tabela 1: Średni czas trwania skryptów

Narzędzie	Tryb	Średni czas trwania (s)
Selenium	*headless*	11,2
	no headless	12,3
Playwright	*headless*	4,0
	no headless	4,3
Splinter	*headless*	8,5
	no headless	9,6

Na podstawie tabeli można stwierdzić, że testy automatyczne napisane z użyciem biblioteki Playwright wykonywały się najkrócej, zarówno w trybie *headless* i *no headless*. Analizując dane na podstawie trybu pracy, można zauważyć, że tryb *headless* osiągał krótsze czasy wykonania niż *no headless*, co potwierdza korzyści wynikające z braku wyświetlonego interfejsu graficznego. Różnice między trybami *headless* i *no headless* były bardziej widoczne dla Selenium i Splinter niż dla Playwright. Może to wynikać z różnic w implementacji tych trybów w poszczególnych bibliotekach lub różnic w strategiach optymalizacji.

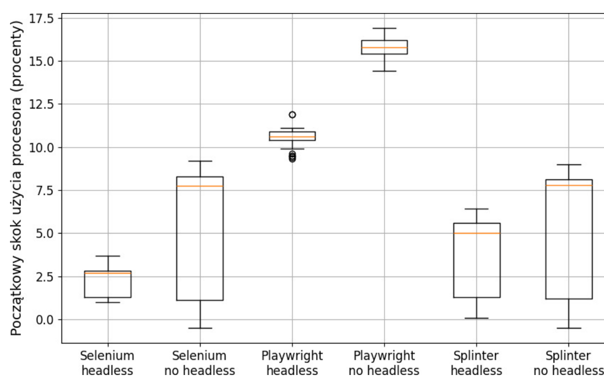
Na następnym wykresie przedstawiono porównanie średniego obciążenia procesora podczas działania skryptów w obydwu trybach. Obciążenie procesora było sprawdzane co sekundę w trakcie wykonywania testów i na podstawie wartości obliczono średnie arytmetyczne.



Rysunek 2: Wykres średniego obciążenia procesora podczas działania skryptów.

Playwright, mimo iż wykazywał krótszy czas wykonania testów, generował znacznie wyższe obciążenie procesora. Rozpatrując tryb działania testów, w trybie *headless* obciążenie procesora było zazwyczaj niższe niż w trybie *no headless*, co sugeruje mniejsze obciążenie związane z renderowaniem interfejsu graficznego. Generalnie, jeśli chodzi o użycie procesora, testy z użyciem Selenium najmniej obciążały procesor, następnie w kolejności znajdowały się testy korzystające z biblioteki Splinter, a na końcu Playwright, które generowały najwyższe obciążenie.

Uzupełniając to porównanie, należy rozpatrzeć różnicę między obciążenia procesora przed testem a sekundę po jego rozpoczęciu, co obrazuje poniższy wykres.

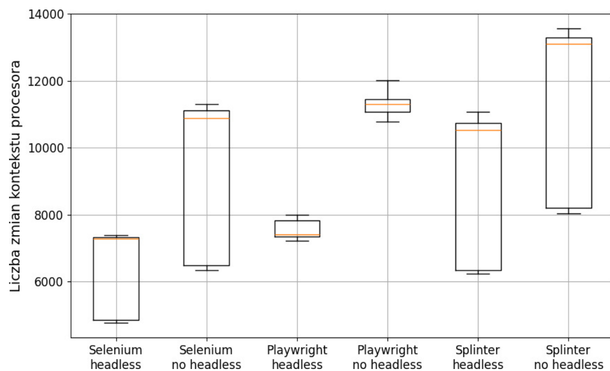


Rysunek 3: Wykres początkowego wzrostu obciążenia procesora po uruchomieniu skryptów.

Te dane sugerują, że Selenium i Splinter wciąż wykazywały tendencję do mniejszego wzrostu obciążenia procesora w porównaniu do Playwright. Zgodnie z oczekiwaniami, testy w trybie *no headless* wykazywały się większym skokiem obciążenia procesora po ich

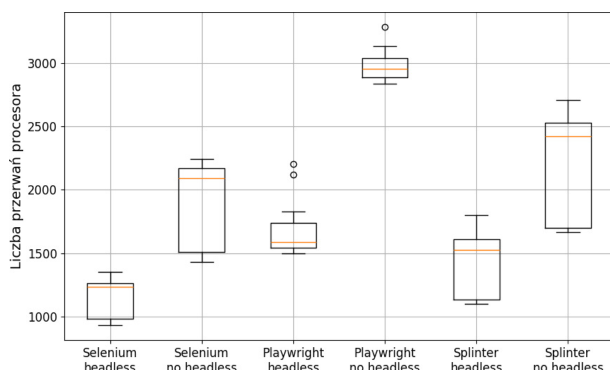
rozpoczęciu, ze względu na renderowanie interfejsu graficznego.

Na następnym wykresie przedstawiono porównanie średniej liczby zmian kontekstu procesora podczas działania skryptów. Liczba zmian kontekstu procesora była sprawdzana co sekundę w trakcie wykonywania testów i na podstawie tych wartości zostały obliczone średnie arytmetyczne. Należy zaznaczyć, że co sekundę odnotowywano różnicę w liczbie zmian kontekstu procesora od momentu uruchomienia systemu.



Rysunek 4: Wykres średniej liczby zmian kontekstu procesora podczas działania skryptów.

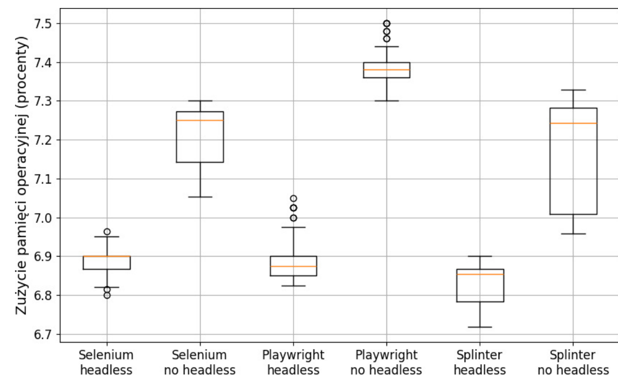
Następnie pokazano porównanie średniej liczby przerwań procesora podczas działania skryptów. Liczba przerwań procesora była sprawdzana co sekundę w trakcie wykonywania testów i na podstawie tych wartości zostały obliczone średnie arytmetyczne. Należy zaznaczyć, że co sekundę odnotowywano różnicę w liczbie przerwań procesora od momentu uruchomienia systemu.



Rysunek 5: Wykres średniej liczby przerwań procesora podczas działania skryptów.

Na podstawie wykresów na rysunkach 4 i 5 można wyciągnąć następujące wnioski – po pierwsze, biblioteka Selenium była najwydajniejsza, w szczególności w trybie *headless*. Te testy generowały mniejszą liczbę zmian kontekstu procesora i liczbę przerwań. Co więcej, tryb *headless* w ogóle generował mniej zmian kontekstu i mniejszą liczbę przerwań niż tryb *no headless*.

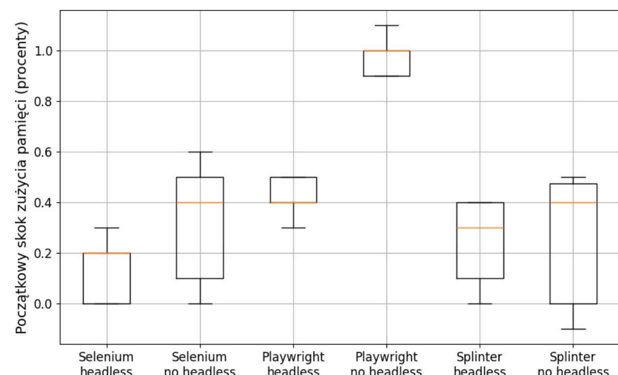
Na poniższym wykresie przedstawiono porównanie średniego zużycia pamięci operacyjnej podczas działania skryptów. Zużycie pamięci operacyjnej było sprawdzane co sekundę w trakcie wykonywania testów i na podstawie wartości obliczono średnie arytmetyczne.



Rysunek 6: Wykres średniego zużycia pamięci operacyjnej podczas działania skryptów.

Rozpatrując powyższy wykres można stwierdzić, że w obrębie danego trybu działania, różnice między bibliotekami w średnim zużyciu pamięci operacyjnej były nieznaczne. Niemniej jednak, widać, iż tryb *no headless* obciążał pamięć operacyjną najbardziej.

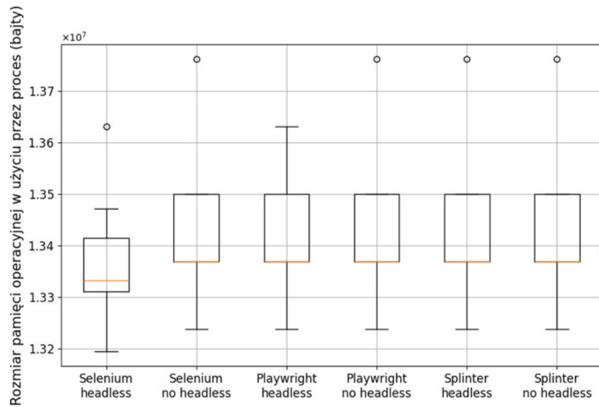
Uzupełniając to porównanie, należy rozpatrzeć różnicę między obciążenia pamięci operacyjnej przed testem a sekundę po jego rozpoczęciu, co obrazuje poniższy wykres.



Rysunek 7: Wykres początkowego wzrostu zużycia pamięci operacyjnej podczas działania skryptów.

Playwright wykazywał znacznie większy wzrost zużycia pamięci operacyjnej w porównaniu do Splinter i Selenium. Splinter zazwyczaj wykazywał podobieństwo wzrostu pamięci co Selenium. Wzrost zużycia pamięci operacyjnej mógł być związany między innymi z implementacją biblioteki i sposobem optymalizacji w działaniu.

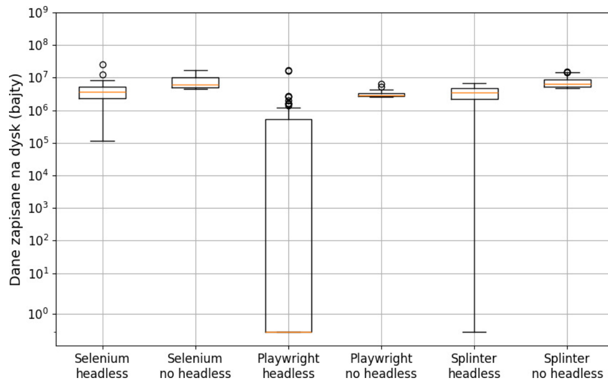
Dokładniejszym porównaniem w przypadku pamięci operacyjnej jest porównanie średniego rozmiaru pamięci operacyjnej używanej przez uruchomiony proces z testem podczas działania. Rozmiar wykorzystywanej pamięci operacyjnej przez skrypt był sprawdzany co sekundę w trakcie wykonywania testów i na podstawie tych wartości zostały obliczone średnie arytmetyczne.



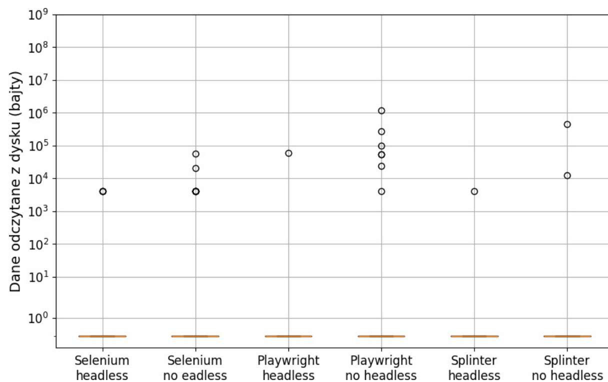
Rysunek 8: Wykres rozmiaru używanej pamięci operacyjnej przez proces.

Biblioteki wykazują tendencję do podobnego zużycia pamięci operacyjnej, ale spośród uzyskanych wyników najmniejsze obciążenie miały procesy z Selenium.

Na następnych wykresach przedstawiono porównanie danych zapisanych na dysk twardy oraz odczytanych z dysku przez skrypty podczas działania.



Rysunek 9: Wykres rozmiaru danych zapisanych na dysk twardy podczas działania skryptów.



Rysunek 10: Wykres rozmiaru danych odczytanych z dysku twardego podczas działania skryptów.

Wyniki zapisu danych na dysk są zbliżone. Tryb *headless* wykazywał nieznacznie mniejszą liczbę danych zapisanych na dysk niż tryb *no headless*. Z kolei liczba danych odczytanych z dysku była minimalna.

7. Wnioski i podsumowanie

Przeprowadzone badanie pozwoliło na dokonanie analizy trzech wybranych narzędzi do automatyzacji testów aplikacji internetowych.

Wyciągnięto następujące wnioski:

1. Testy z użyciem Playwright charakteryzowały się najkrótszym czasem działania.
2. Testy z użyciem Selenium i Splinter miały podobny czas trwania, ale jednak częściej Selenium było nieco dłuższe od testów z użyciem Splinter.
3. Mimo krótszego czasu wykonania, biblioteka Playwright obciążała procesor bardziej niż Selenium i Splinter.
4. Selenium wykazało najmniejsze obciążenie procesora, w porównaniu do Playwright i Splinter.
5. Tryb *no headless* zawsze generował większe obciążenie procesora niż tryb *headless*, zarówno w testach z użyciem Playwright, Selenium, jak i Splinter. To sugeruje, że testowanie aplikacji internetowych bez renderowania grafiki jest najbardziej optymalne.
6. Zużycie pamięci operacyjnej oraz dysku było zbliżone dla wszystkich narzędzi.
7. Pomimo nieco dłuższego czasu wykonania testów, Selenium charakteryzowało się najlepszą stabilnością działania.

Podsumowując, wybrane biblioteki spełniają swoje zadania, lecz jeśli chodzi o stabilność i efektywne zarządzanie zasobami komputera, Selenium wyróżnia się jako najlepsza opcja. Niemniej jednak, jeśli chodzi o czas wykonywania testów, to Playwright okazał się najkorzystniejszy.

Literatura

[1] L. Delia, N. Galdamez, P. Thomas, L. Corbalan, P. Pesado, Multi-platform mobile application development analysis, In 2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS) (2015) 181–186.

[2] K. Ali, X. Xia, A reliable and an efficient Web testing system, International Journal of Software Engineering & Applications (IJSEA) 10 (2019) 1–16.

[3] M. A. Umar, Z. Chen, A study of automated software testing: Automation tools and frameworks, International Journal of Computer Science Engineering (IJCSE) 6 (2019) 217–225.

[4] K. Naik, P. Tripathy, Software testing and quality assurance: Theory and practice, A John Wiley & Sons, New Jersey, 2008.

[5] G. A. Di Lucca, Testing Web-based applications: The state of the art and future trends, Information and Software Technology 48 (2006) 1172–1186.

[6] E. A. Altulaihan, A. Alismail, M. Frikha, A survey on Web application penetration testing, Electronics 12 (2023) 1229–1252.

[7] E. W. Dijkstra, The humble programmer, Communications of the ACM 15 (1972) 859–866.

[8] Selenium documentation, https://www.selenium.dev [01.05.2024].

[9] Playwright documentation, https://www.playwright.dev [01.05.2024].

[10] Splinter documentation, https://splinter.readthedocs.io, [01.05.2024].