POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Zarządzanie informacją

Katedra Zarządzania i Ekonomii

Wdrożenia systemów IT

Michał Gawron

Nr albumu 17222

Opracowanie analizy przedwdrożeniowej oraz identyfikacja potencjalnych zagrożeń i dobre praktyki ich przezwyciężania na przykładzie przedsiębiorstwa z branży spożywczej.

Praca magisterska

Promotor: Piotr Gago

Warszawa, wrzesień 2023

Abstrakt

Celem pracy było przeprowadzenie analizy przedwdrożeniowej na przykładzie przedsiębiorstwa z branży spożywczej oraz identyfikacja potencjalnych zagrożeń mogących wystąpić na każdym z etapów i sformułowanie dobrych praktyk minimalizujących ryzyko wystąpienia potencjalnych zagrożeń.

W części teoretycznej, przedstawiono pojęcie zintegrowanego systemu klasy ERP i jego rolę w przedsiębiorstwie. Dodatkowo wyjaśniono istotę analizy przedwdrożeniowej oraz omówiono jej wpływ na powodzenie projektu wdrożenia systemu informatycznego w przedsiębiorstwie. Przedstawiono również najpopularniejszą obecnie metodykę zarządzania projektami wdrożenia systemu informatycznego w przedsiębiorstwie, czyli metodykę Agile.

W pracy przedstawiono etapy analizy przedwdrożeniowej mogące służyć jako wzór postępowania podczas tworzenia komercyjnej analizy przedwdrożeniowej. Na końcu każdego etapu w formie podsumowań zidentyfikowano potencjalne zagrożenia wpływające na poprawność przeprowadzonej analizy i zaproponowano dobre praktyki, których zastosowanie minimalizuje ryzyko wystąpienia potencjalnych zagrożeń.

Słowa kluczowe: wdrożenie systemu informatycznego, analiza przedwdrożeniowa, system klasy ERP.

Spis treści

Wstęp

W czasach szybko rozwijającej się technologii, wybór właściwych narzędzi do testowania oprogramowania jest kluczowy dla powodzenia projektów IT. Jednym z istotnych rozwiązań na rynku jest automatyzacja testów przeglądarkowych, która umożliwia twórcom oprogramowania szybsze i skuteczniejsze sprawdzanie poprawności aplikacji internetowych. Niniejsza praca magisterska ma na celu porównanie dwóch popularnych frameworków do automatyzacji testów przeglądarkowych: Playwright i Selenium, ze szczególnym naciskiem na ich zastosowanie w Pythonie na systemie operacyjnym Windows.

Praca skupia się na analizie i porównaniu różnych aspektów tych frameworków, takich jak łatwość instalacji, obsługa różnych przeglądarek, wydajność, wsparcie dla różnych strategii lokalizacji elementów, API, synchronizacja i asynchroniczność, a także społeczność i wsparcie. Na podstawie przeprowadzonej analizy zostaną sformułowane wnioski dotyczące zalet i wad obu narzędzi oraz ich potencjalnego zastosowania w różnych scenariuszach.

Celem tej pracy jest dostarczenie czytelnikom wiedzy, która umożliwi podjęcie świadomej decyzji dotyczącej wyboru narzędzia do automatyzacji testów przeglądarkowych. W pracy zostanie przedstawione praktyczne porównanie obu frameworków na przykładzie testów przeprowadzonych na stronie internetowej http://uitestingplayground.com/home, korzystając z języka Python na systemie Windows. Dzięki temu czytelnicy będą mieli możliwość zrozumienia, jak te narzędzia działają w praktyce oraz jakie są ich mocne i słabe strony.

Mam nadzieję, że przedstawione analizy i wnioski będą pomocne dla osób związanych z branżą IT, takich jak programiści, testerzy czy menedżerowie projektów, którzy poszukują informacji na temat dostępnych narzędzi do automatyzacji testów przeglądarkowych. W efekcie czytelnicy będą mogli podjąć bardziej świadome decyzje dotyczące wyboru frameworka, co przyczyni się do sukcesu ich projektów.

W ostatnim akapicie zostaną omówione kwestie związane z elastycznością i skalowalnością obu frameworków, takie jak możliwość rozszerzenia funkcjonalności, łatwość integracji z systemami ciągłej integracji i dostarczania (CI/CD) oraz zdolność do obsługi różnych platform i systemów operacyjnych.

Rozdział 1: Część Teoretyczna

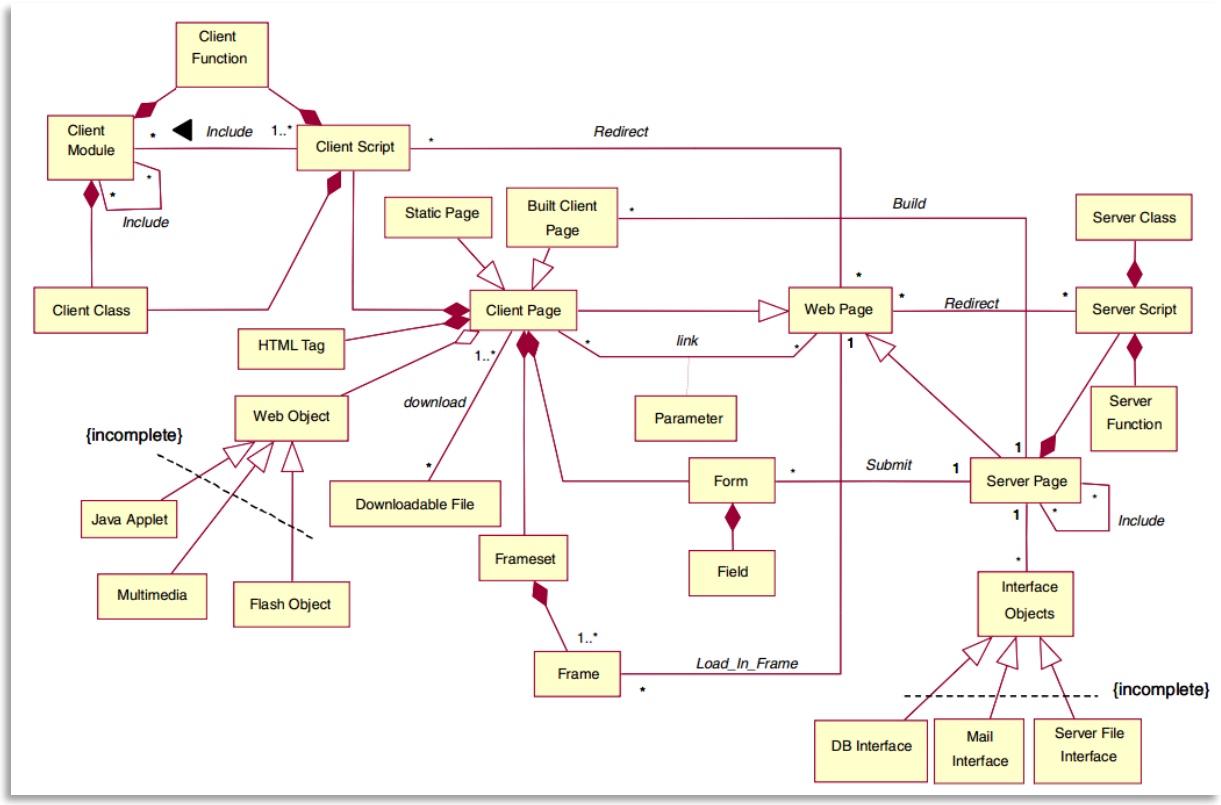
1.1 Automatyzacja testów

Testowanie oprogramowania od zawsze było kluczowym etapem w każdym projekcie informatycznym. Jednakże początkowo proces ten był wykonywany przez długi okres manualnie. Wykrywanie defektów pozwalało na dostarczanie kodu oraz produktu oprogramowania informatycznego o wyższej jakości. Etap ten zazwyczaj jest żmudny, powtarzalny, pracochłonny oraz czasochłonny. Nierzadko też wymaga dokładności oraz precyzji od osoby wykonującej poszczególne testy. Obszar testowania rozwijał się wraz z rozwojem technologii. Powstawały to coraz nowe pojęcia związane z tematyką testów. Po raz pierwszy pojęcie testów automatycznych pojawiło się w 1999 r... Pojęcie to zdefiniowano następująco: Automatyzacja testów to proces wykorzystania oprogramowania do przeprowadzania testów na aplikacji lub systemie w sposób zautomatyzowany, co pozwala na szybkie i dokładne sprawdzanie funkcjonalności (Elfriede Dustin, 1999). Powstanie pojęcia automatyzacji testów był przełomowym etapem w procesie dostarczania jakości. Utworzenie kodu, który po wcześniejszej kompilacji i uruchomieniu wykonywał czynności, które w przeciwnym razie, te testy musiałyby być przeprowadzane manualnie przez testerów, co jest czasochłonne i podatne na błędy. Automatyzacja testów przyniosła wiele korzyści, takich jak:

- Przyspieszenie procesu testowania;
- Zwiększenie dokładności i spójności testów;
- Redukcja kosztów związanych z testowaniem;
- Łatwiejsze wykrywanie błędów i regresji;
- Umożliwienie szybszego wprowadzania zmian w aplikacji (Grahama, 1999)
- Cykliczność wykonywania testów;
- Automatyczne tworzenie raportów rezultatów testów;
- Ułatwienie wykonywanie skomplikowanych scenariuszy testowych;
- Równoległe testowanie różnych scenariuszy testowych;
- Emulacja dowolnych urządzeń mobilnych;
- Ograniczenie wpływu błędu ludzkiego.

To tylko kilka przykładów z wielu, które wpływają na korzyść implementacji testów automatycznych do projektu informatycznego.

Proces automatyzacji testów nie jest prosty. Początkowe nakłady mogą wydawać się znaczące. Proste scenariusze testowe - manualnie mogą być wykonane od razu, a w przypadku automatyzacji wymagane jest przygotowanie skryptu, którego czas stworzenia uzależniony jest od złożoności architektury oprogramowania oraz umiejętności programisty testów automatycznych. Z punktu widzenia osób zarządzających wydaje się to być niekorzystne. Natomiast pierwotnie znaczący nakład zasobów programistycznych skierowany w celu pokrycia testami automatycznymi wybranych scenariuszy testowych, szybko zwraca się kolejnych etapach projektu. Złożoność architektury aplikacji internetowych można zaobserwować na poniższym rysunku.



Rysunek 1 Złożoność architektury aplikacji internetowych - (Kumar, 2016)

Istnieje wiele badań, które udowadniają korzyści płynące z automatyzacji testów w projektach informatycznych. Artykuł “The Impacts of Test Automation on Software’s Cost, Quality and Time to Market” (Kumar, 2016) omawia zastosowanie automatyzacji testów w

trzech różnych projektach oprogramowania: Railway's Cloakroom and Retiring Room Management, Restaurant Billing System i Mini Geometric Figure Analyzer.

1. "Railway's Cloakroom and Retiring Room Management" to system zarządzania szatnią i pokojami wypoczynkowymi dla kolei, wprowadzony w celu minimalizacji ryzyka błędów i ułatwienia obsługi klienta. Autorzy zbadali cztery wersje oprogramowania, skupiając się na testowaniu regresyjnym, które wykonano zarówno ręcznie, jak i automatycznie. Wyniki pokazały, że automatyzacja testów przyniosła ogólnie korzystne rezultaty. Koszty były niższe, mimo początkowego nakładu na implementację narzędzi do testowania automatycznego. Czas potrzebny na testowanie regresyjne został znacznie zredukowany, co przekładało się na szybsze wydanie oprogramowania na rynek.
2. "Restaurant Billing System" to oprogramowanie do obsługi zamówień i fakturowania w restauracjach. Podobnie jak w przypadku pierwszego systemu, przeprowadzono testy regresyjne na pięciu różnych wersjach oprogramowania. Także w tym przypadku, automatyzacja testów okazała się korzystna, mimo początkowych kosztów implementacji. Czas testów regresyjnych został zredukowany, co pozwoliło na szybsze wprowadzanie nowych funkcji.
3. "Mini Geometric Figure Analyzer" to program służący do analizy figur geometrycznych. Badania przeprowadzono na trzech wersjach oprogramowania, a wyniki ponownie potwierdziły korzyści z automatyzacji testów.

Wyniki badań dla wszystkich trzech projektów były jednoznaczne: automatyzacja testów przyniosła korzyści w postaci niższych kosztów, krótszego czasu wydania oprogramowania na rynek i poprawy jakości. W dłuższej perspektywie, korzyści z automatyzacji testów przewyższały początkowe koszty implementacji. Na podstawie tych wyników, autorzy artykułu zasugerowali, że automatyzacja testów może przynieść znaczne korzyści, zwłaszcza w przypadku testów regresyjnych, które są często powtarzane. Wnioskowali również, że dalsze badania powinny skupić się na uwzględnieniu więcej czynników kosztów automatyzacji oraz na możliwościach automatycznego generowania danych testowych.

Wpływ automatyzacji testów na przebieg wytwarzania oprogramowania jest niepodważalny. Automatyzacja testów przynosi wiele korzyści, które przytoczono wyżej. Każdy dodatkowo pokryty scenariusz testowy, który wykonywany jest automatycznie pozytywnie wpływa na ogólną jakość produktu oprogramowania. Istnieją scenariusze testowe, dla których automatyzacja jest nieopłacalna, lecz stanowią one niewielki procent w stosunku do całości. Takie przypadki trzeba każdorazowo szczególnie rozpatrzeć. Istnieją przypadki, że przy modyfikacji założeń oraz podejścia testowego automatyzacja stanie się możliwa. Ponowne rozważenie oraz potencjalne odnalezienie nowego rozwiązania dla zautomatyzowania procesu

testowania trudnych scenariuszy testowych jest korzystne w dalszych etapach projektu wytwarzania oprogramowania.

1.2 Testy aplikacji przeglądarkowych

W dobie Internetu i powszechnej cyfryzacji społeczeństwa powszechną praktyką jest tworzenie aplikacji, które w pełni obsługiwane są przez przeglądarkę internetową. Tworzenie aplikacji desktopowych nie jest tak częstą praktyką w porównaniu do popularyzacji aplikacji przeglądarkowych. Chociaż testowanie aplikacji internetowych ma te same cele co testowanie "tradycyjnych" aplikacji, w większości przypadków tradycyjne teorie i metody testowania nie mogą być stosowane bez modyfikacji, ze względu na specyfikę i złożoność aplikacji internetowych. Są one traktowane jako systemy rozproszone o architekturze klient-serwer lub wielowarstwowej, z charakterystycznymi cechami takimi jak: szeroki zakres użytkowników, zróżnicowane środowiska wykonania, heterogeniczna natura wynikająca z różnorodności składników oprogramowania, oraz zdolność generowania składników oprogramowania w czasie rzeczywistym.

Testerzy stoją przed wielkim wyzwaniem w celu zapewnienia jakości dla tego typu produktu informatycznego. Dynamiczny rozwój technologii internetowych wraz z nimi szeroki zakres dostępnych funkcjonalności oraz co raz to nowszych technologii przeglądarek internetowych stanowi trudność w skutecznym i szybkim wykrywaniu potencjalnych defektów spowodowanych błędem w kodzie. Krytycznym aspektem zapewnienia jakości dla procesu wytwarzania oprogramowania bazującym na technologii przeglądarkowych są testy aplikacji przeglądarkowych. Aktualnie użytkownicy używają wielu rozmaitych urządzeń, systemów operacyjnych i przeglądarek do dostępu do aplikacji internetowych. Testowanie w celu upewnienie się, że aplikacja działa prawidłowo na wszystkich możliwych platformach jest kluczowe dla sukcesu produktu.

Testy aplikacji przeglądarkowych obejmują różne rodzaje testów, w zależności od celów i wymagań projektu:

1. Testy funkcjonalne: Testy sprawdzają, działanie zgodnie z oczekiwaniami funkcji aplikacji na wielu przeglądarkach. Przykład to między innymi: formularze, menu, przyciski oraz różne interaktywne elementy interfejsu użytkownika są sprawdzane pod kątem poprawnego działania.
2. Testy kompatybilności: Celem tych testów jest upewnienie się, że aplikacja działa poprawnie w różnych środowiskach, takich jak różne przeglądarki (Firefox, Chrome, Safari itd.), różne

systemy operacyjne (Windows, MacOS, Linux itd.) i różne urządzenia (laptopy, telefony komórkowe, tablety itd.)

3. Testy responsywności: Testy responsywności zapewniają, że projekt interfejsu graficznego aplikacji prawidłowo dostosowuje się do różnych rozdzielczości ekranu i orientacji urządzeń. Istotnym aspektem w kontekście tych testów jest, aby strona lub aplikacja były łatwe do czytania i umożliwienie interakcji użytkownikowi niezależnie od rozmiaru ekranu.
4. Testy wydajności: Testy wydajności mierzą, jak szybko strona lub aplikacja się ładuje, a także jak szybko reaguje na interakcje użytkownika. Te testy mogą pomóc identyfikować i eliminować problemy z wydajnością, które mogą negatywnie wpływać na doświadczenia użytkownika
5. Testy integracji: Testy integracji skupiają się na weryfikacji, kompatybilności różnych komponentów aplikacji. Testy te odpowiadają na pytanie „Czy różne moduły, usługi i bazy danych, działają poprawnie razem?" Zakresem testów integracyjnych jest przeprowadzanie scenariuszy, które symulują rzeczywiste interakcje użytkownika z aplikacją, w celu zapewnienia się, że wszystkie moduły oraz serwisy współtworzące aplikacje współpracują ze sobą w sposób właściwy do przewidywanego.

Testy aplikacji przeglądarkowych mogą być przeprowadzane manualnie, ale ze względu na złożoność, poziom skomplikowania czego skutkiem jest wysoką czasochłonność testów, często praktyką jest używanie narzędzi do automatyzacji testów. Wspominane narzędzia dostarczają funkcjonalność automatycznego uruchamiania testów na różnych przeglądarkach i systemach operacyjnych, efektem czego jest znaczne przyśpieszenie procesu testowania, identyfikacji oraz naprawy błędów na wczesnym etapie rozwoju.

W kontekście aplikacji internetowych, pojęcie aplikacji odnosi się do zestawu komponentów oprogramowania implementujących wymagania funkcjonalne, natomiast środowisko uruchomieniowe wskazuje na całą infrastrukturę (składającą się z komponentów sprzętowych, oprogramowania i oprogramowania pośredniczącego) potrzebną do wykonania aplikacji internetowej. Celem testowania aplikacji internetowych jest ich uruchamianie przy użyciu kombinacji danych wejściowych i stanów w celu wykrycia błędów. Błędy mogą wynikać zarówno z błędów w implementacji aplikacji, jak i z błędów w środowisku uruchomieniowym lub w interfejsie między aplikacją a środowiskiem, w którym jest uruchamiana. (Lucca i Fasolino, 2006)

Zważając na to, że aplikacje internetowe są bezpośrednio powiązana ze swoim środowiskiem na których są uruchomiane, fakt ten ma negatywny wpływ na ich osobne testowanie i jednoznaczne ustalenie, które środowisko powoduje występowanie poszczególnego błędu.

W związku z tym, testowanie aplikacji internetowych powinno być rozpatrywane z dwóch różnych perspektyw. Pierwsza aspekt obejmuje różne rodzaje testów, które muszą być przeprowadzone w celu weryfikacji zgodności aplikacji internetowej ze określonymi wymaganiami niefunkcjonalnymi. Drugi aspekt obejmuje problematykę testowania wymagań funkcjonalnych aplikacji internetowej. Istotne, żeby uwzględnić obie perspektywy podczas testów aplikacji internetowej, ponieważ tylko w zastosowaniu obu testy są całościowe, kompletne oraz pozwalają na dostarczenie wysokiej jakości produktu. Różne rodzaje testów muszą być przeprowadzone w celu wykrycia różnych rodzajów błędów. Wyzwania i pytania charakterystyczne dla obu perspektyw testowania będą analizowane w następnych podrozdziałach. W kontekście niefunkcjonalnych wymagań aplikacji internetowej, zazwyczaj oczekuje się, że spełnia ona różne wymagania niefunkcjonalne, zarówno wyraźnie wymienione, jak i domyślne. Główne wymagania obejmują wydajność, skalowalność, kompatybilność, dostępność, użyteczność i bezpieczeństwo. (Lucca i Fasolino, 2006)

Testowanie wymagań funkcjonalnych aplikacji internetowej ma na celu wykrycie błędów aplikacji wynikających z błędów w implementacji określonych wymagań funkcjonalnych, a nie środowiska uruchomieniowego. Podobnie jak w przypadku testowania tradycyjnego oprogramowania, testowanie funkcjonalności aplikacji internetowej musi polegać na następujących podstawowych aspektach: modelach testów, poziomach testowania, strategiach testowania i procesach testowania.

1.3 Frameworki do automatyzacji testów przeglądarkowych

Frameworki do automatyzacji testów przeglądarkowych to zestawy narzędzi i bibliotek, które ułatwiają tworzenie, uruchamianie i analizowanie testów przeglądarkowych. Dwa popularne frameworki do automatyzacji testów przeglądarkowych to Playwright i Selenium, które będą głównym przedmiotem tej pracy magisterskiej. Oba frameworki mają zalety i wady, które zostaną przedstawione i porównane w kolejnych rozdziałach.
Automatyzacja testów przeglądarkowych jest jednym z kluczowych elementów procesu budowy i utrzymania wysokiej jakości aplikacji internetowych. Historia tej dziedziny zaczyna się początkiem XXI wieku, gdy po raz pierwszy zaczęto dostrzegać potrzebę automatyzacji procesów testowania w przeglądarce złożonych jak na ten moment aplikacji internetowych.

Przed powstaniem narzędzi do automatyzacji testy wykonywano manualnie przez testera, takie testy były czasochłonne oraz w znacznym stopniu podatne na błędy. Z biegiem

rozwoju technologii, potrzeba automatyzacji procesu testowania była znacząca, co doprowadziło do powstania pierwszych narzędzi do automatycznego testowania aplikacji przeglądarkowych. Początkowe skrypty posiadały ograniczone możliwości wykonywania testów, nie oferowały dużej elastyczności oraz były podatne na źle zaprojektowanej architektury.

W odpowiedzi do narastających problemów, tworzono zaawansowane narzędzia do automatyzacji testów przeglądarkowych. Narzędzia automatyzacji nie tylko umożliwiły automatyzację testów, ale także wprowadziły szereg zaawansowanych funkcji, takich jak zarządzanie sesjami, emulacja interakcji użytkownika, obsługa wyników i wiele innych. Istotnym elementem frameworków jest ich zdolność do integracji z innymi narzędziami i technologiami, takimi jak systemy kontroli wersji, systemy CI/CD czy narzędzia do zarządzania projektami (Effective Methods for Software Testing, Third Edition, 2006).

Początkowo, frameworki do automatyzacji testów przeglądarkowych były skierowane głównie do programistów, wymagając od nich głębokiej wiedzy technicznej. Jednak z biegiem czasu, zaczęto tworzyć narzędzia bardziej przyjazne dla testerów, które nie wymagały od użytkowników zaawansowanych umiejętności programistycznych. Obecnie, wiele narzędzi umożliwia tworzenie testów za pomocą interfejsów graficznych, co sprawia, że są one dostępne dla szerokiego grona użytkowników.

Współczesne frameworki do automatyzacji testów przeglądarkowych są nieodłącznym elementem procesu tworzenia oprogramowania. Wielu ekspertów zwraca uwagę na ich kluczową rolę w utrzymaniu jakości kodu, podkreślając, że są one niezbędne do skutecznego zarządzania złożonymi projektami oprogramowania (Graham, Experiences of Test Automation: Case Studies of Software Test Automation, 2010)

W przyszłości, możemy spodziewać się dalszego rozwoju tych narzędzi, wraz z pojawieniem się nowych technologii i metodologii. Ciekawym obszarem dla przyszłych badań może być na przykład zastosowanie sztucznej inteligencji w automatyzacji testów przeglądarkowych.

1.3.1 Playwright

Playwright to nowoczesny framework do automatyzacji testów przeglądarkowych, który umożliwia kontrolowanie i manipulowanie przeglądarkami takimi jak Chromium, Firefox i Safari (Yakovenko & Dashevskyi, 2021). Został stworzony 2002 r. przez zespół Microsoft i oferuje wiele zaawansowanych funkcji, takich jak generowanie zrzutów ekranu, obsługa wielu kontekstów przeglądarek, emulacja urządzeń mobilnych i środowisk sieciowych (Nurmohamed, 2020). Playwright jest dostępny w różnych językach programowania, w tym Python, JavaScript i C#. Wspiera także różne systemy operacyjne, takie jak Windows, macOS i Linux.

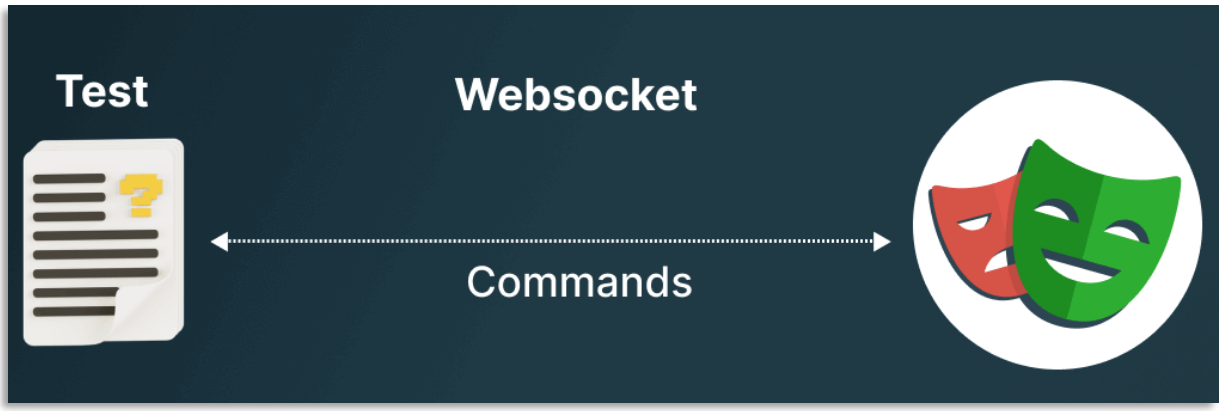
main.py

```python
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch()
    page = browser.new_page()
    page.goto("https://playwright.dev/")
    page.screenshot(path="example.png")
    browser.close()
```

Rysunek 2 Prosty kod Playwright - https://playwright.dev/python/docs/library

Łatwość użycia jest cechą wyróżniającą playwirght na tle innych narzędzi do automatyzacji. W znacznym stopniu ogranicza to ilość wymaganego kodu do stworzenia przez programistę testów automatycznych. Framework można użyć do kreowania i implementacji testów performance aplikacji przeglądarkowej. Playwright zapewnia funkcjonalności mierzenia ładowania strony, wydajności aplikacji, dostarczania dynamicznej zawartości, a także szeroki zakres innych wskaźników, które są użyteczne w kontekście testowania wydajności. Automatyzacja testów wydajności otwiera przed testerami łatwe i szybkie kontrolowanie wydajności na wielu rozmaitych urządzeniach mobilnych i nie tylko oraz platformach. Wspominane funkcjonalności pozwalają na znaczne zwiększenie jakości produktu, gdyż potencjalne problemy wydajnościowe są natychmiast identyfikowane.

Playwright wyróżnia się na tle innych narzędzi do automatyzacji testów szybkością wykonywania scenariuszy testowych. Zawdzięcza to komunikacji frameworka z przeglądarką, który bazuje na wymianie danych poprzez websocket co znacznie skraca czas przekazania informacji pomiędzy dwoma instancjami. W WebSocket odróżnieniu od protokołu HTTP, wymiana danych występuje w sposób ciągły w trakcie trwania połączenia oraz zarówno klient, jak i serwer może dane wysyłać lub odbierać. WebSocket przydatny jest, gdy dane potrzebne są w czasie rzeczywistym oraz nie możemy pozwolić sobie na ciągłe odpytywanie serwera.



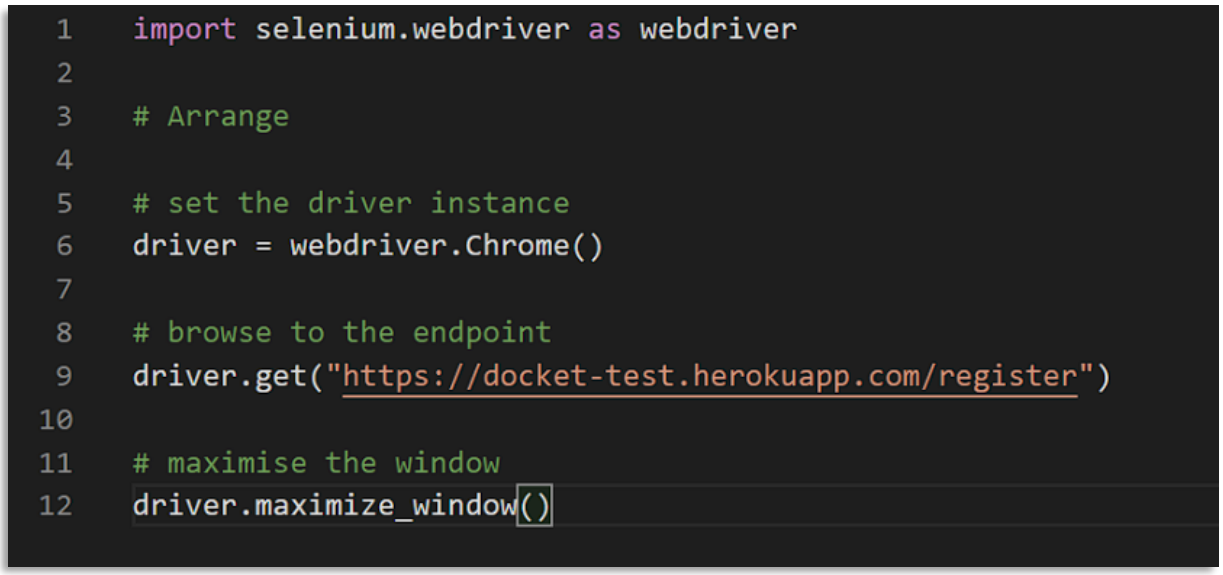
Rysunek 3 Wizualizacja komunikacji Playwright z przeglądarką - https://www.lambdatest.com/playwright

Przeglądarki uruchamiają treści internetowe należące do różnych źródeł w różnych procesach. Playwright jest dostosowany do architektury nowoczesnych przeglądarek i uruchamia testy poza procesem. To sprawia, że Playwright jest wolny od typowych ograniczeń związanych z uruchamianiem testów w procesie. Możliwość testowania scenariuszy obejmujących wiele kart, wiele źródeł i wielu użytkowników. Tworzenie scenariusze z różnymi kontekstami dla różnych użytkowników i uruchamiaj je na swoim serwerze w ramach jednego testu.

Zaufane zdarzenia. Najeżdżanie kursorem na elementy, interakcja z dynamicznymi kontrolkami, generowanie zaufanych zdarzeń. Playwright wykorzystuje rzeczywisty potok danych wejściowych przeglądarki, nie do odróżnienia od prawdziwego użytkownika.

1.3.2 Selenium

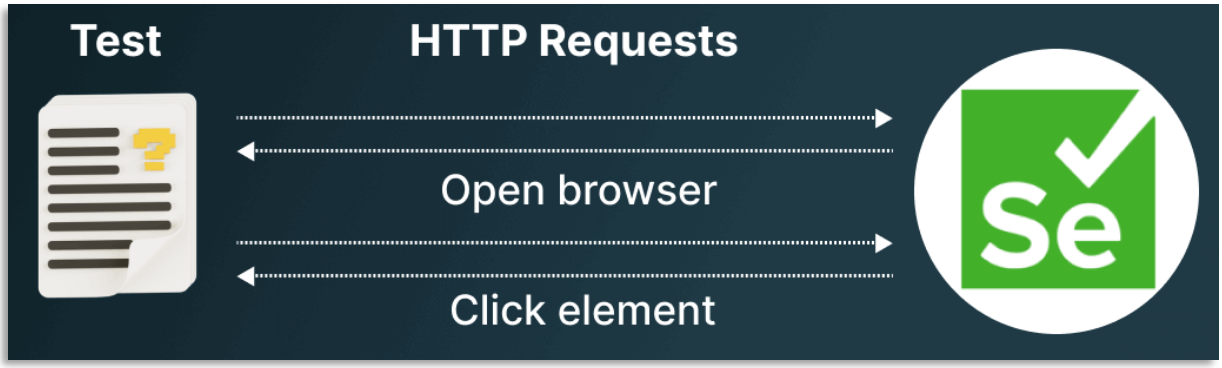
Selenium to starszy, ale wciąż szeroko stosowany framework do automatyzacji testów przeglądarkowych, który obsługuje wiele przeglądarek, takich jak Chrome, Firefox, Safari, Opera, Edge i Internet Explorer (Huggins, 2018). Selenium jest dostępne w różnych językach programowania, takich jak Python, Java, C#, Ruby i JavaScript, co czyni go atrakcyddjnym dla zespołów o zróżnicowanych umiejętnościach programistycznych. Selenium działa na różnych systemach operacyjnych, w tym Windows, macOS i Linux.



Rysunek 4Przykład prostego kodu Selenium - https://blog.testproject.io/2020/06/16/selenium-python-beginners-tutorial-for-automation-testing/

Selenium jest wyjątkowym narzędziem do testowania, nie tylko ze względu na jego różnorodność, ale również ze względu na architekturę i metody komunikacji z przeglądarką. Kluczo-

wym elementem tego procesu jest sterownik przeglądarki (WebDriver), który umożliwia interakcję Selenium z przeglądarką. Komunikacja przebiega przez protokoły HTTP.



Rysunek 5 Wizualizacja komunikacji Selenium z przeglądarką - https://www.lambdatest.com/playwright

Każda wspierana przez Selenium przeglądarka ma dedykowany sterownik, tak zwany WebDriver (np. ChromeDriver dla Google Chrome, GeckoDriver dla Firefoxa itp.). Sterowniki te, będące de facto interfejsem pomiędzy Selenium a przeglądarką, są odpowiedzialne za wykonywanie poleceń Selenium na stronie internetowej, na przykład, kliknięcie przycisku, wpisanie tekstu w polu formularza czy odczytanie zawartości elementu. Zasada działania jest prosta - testy napisane za pomocą Selenium wysyłają polecenia do sterownika przeglądarki, który następnie interpretuje te polecenia i przekształca je na odpowiednie akcje w przeglądarce.

Jednym z najważniejszych aspektów Selenium jest to, że działa ono na poziomie interfejsu użytkownika, imitując zachowanie prawdziwego użytkownika. Dzięki temu, testy Selenium mogą wykryć problemy, które mogłyby zostać niezauważone przy testowaniu jednostkowym lub integracyjnym, takie jak problemy z wyglądem strony, interakcją użytkownika czy wydajnością strony.

Selenium składa się z kilku składowych, w tym Selenium WebDriver do bezpośredniej komunikacji z przeglądarką, Selenium Grid do równoczesnego uruchamiania testów na wielu maszynach i przeglądarkach, oraz Selenium IDE, które jest narzędziem do nagrywania i odtwarzania testów, choć jest ono raczej rzadko stosowane w zaawansowanym testowaniu.

Jednym z kluczowych atutów Selenium jest jego wsparcie dla wielu języków programowania, co oznacza, że niezależnie od technologii stosowanej do budowy aplikacji, zespół testujący prawdopodobnie będzie mógł napisać testy przy użyciu języka, który już zna. Dodatkowo, Selenium ma bogate wsparcie dla różnych strategii lokalizacji elementów, takich

jak selektory CSS, XPath, nazwy klas, identyfikatory i inne, co daje testerom dużą elastyczność w pisaniu testów.

Niemniej jednak, Selenium nie jest pozbawione wad. Ze względu na swoje bezpośrednie działanie na poziomie interfejsu użytkownika, testy Selenium mogą być stosunkowo wolne w porównaniu do testów jednostkowych czy integracyjnych. Ponadto, niektóre skomplikowane interakcje użytkownika mogą być trudne do zasymulowania za symulowania za pomocą Selenium, chociaż zwykle można to obejść przy użyciu zaawansowanych technik, takich jak skrypty JavaScript.

Mimo to, Selenium nadal jest jednym z najpopularniejszych narzędzi do automatyzacji testów przeglądarkowych i jest niezastąpione w wielu scenariuszach, zwłaszcza w kontekście testowania aplikacji internetowych o dużej skali i złożoności.

1.4 Python w automatyzacji testów przeglądarkowych

Python jest znany ze swojego czytelnego składniowego stylu i prostoty zarówno w nauce, jak i użyciu. W wyniku posiadania wspominanych cech, często staje się wyborem podczas decyzji wyboru odpowiedniego języka programowania w przypadku automatyzacji testów przeglądarkowych. Elastyczność i łatwość użycia języka Python, w kombinacji z obfitym zakresem rozmaitych bibliotek oraz modułów, prowadzi do konkluzji, że jest to idealna opcja dla wielu wielorakich zadań, w kontekście implementacji oraz przeprowadzenia testów przeglądarkowych.

Kluczowym aspektem Pythona, który wpływa na popularność użycia w automatyzacji testów, jest wsparcie społeczności. Wiele bibliotek do testów, w tym Playwright i Selenium, ma cały czas prężnie działające i zaangażowane społeczności, które tworzą to co raz nowe funkcjonalności i ciągle poprawiają znalezione błędy w nowo wydanych wersjach, również oferują wsparcie i odpowiedzi na pytania innym użytkownikom. Powyższe społeczności są często źródłem cennych zasobów, takich jak dokumentacja, poradniki, kursy, forum dyskusyjne i wiele innych, które mogą pomóc testerom w skutecznym wykorzystaniu tych narzędzi.

Playwright i Selenium, dwie popularne biblioteki do automatyzacji testów przeglądarkowych, oferują oficjalne biblioteki klienta dla Pythona. Te biblioteki klienckie pozwalają testerom na wykorzystanie pełnej mocy tych narzędzi bezpośrednio w Pythonie, co oznacza, że można tworzyć skrypty testów, które są wyraźne, zwięzłe i łatwe do zrozumienia. Możliwość

korzystania z tych bibliotek w Pythonie pozwala na wykorzystanie ich pełnej funkcjonalności w prosty i intuicyjny sposób, co przekłada się na efektywność i produktywność procesu testowania.

Ogólnie rzecz biorąc, Python jest doskonałym językiem dla automatyzacji testów przeglądarkowych, ze względu na jego prostotę, wsparcie społeczności i bogaty ekosystem narzędzi i bibliotek testujących. Zarówno Playwright, jak i Selenium, oferują wsparcie dla Pythona, co oznacza, że testerzy mogą korzystać z tych potężnych narzędzi w przyjaznym dla użytkownika, zrozumiałym języku programowania.

1.5 Interfejs API Playwright i Selenium

Porównanie interfejsu API Playwright i Selenium jest istotne dla zrozumienia, jak te narzędzia mogą być używane do tworzenia skryptów testujących. Zarówno Selenium, jak i Playwright oferują szeroki zakres funkcji do manipulacji i interakcji z elementami stron internetowych, ale sposób, w jaki te funkcje są udostępniane, może wpływać na łatwość użycia, czytelność skryptów i wydajność.

1.5.1 Interfejs API Selenium

Podejście, które stosuje Selenium, ma charakterystykę obiektową w kontekście API. Każdy element na stronie internetowej jest przedstawiony jako obiekt, który umożliwia interaktywne działania za pomocą metod specyficznych dla tego obiektu. Aby zilustrować, do wprowadzenia tekstu w obszarze tekstowym, najpierw odnajdujemy ten konkretny obszar tekstowy, a potem stosujemy metodę 'send_keys' dla tego obiektu.

W Selenium jest również zastosowany system "oczekiwań", które służą do wstrzymania wykonania skryptu do momentu spełnienia konkretnych warunków na stronie internetowej. Chociaż jest to efektywne, wymaga to dodatkowego kodowania i zrozumienia, jak zastosować te "oczekiwania" w praktyce.

Przechodząc do aspektu obsługi błędów, Selenium stosuje tradycyjne mechanizmy obsługi błędów języka, w którym jest napisany skrypt. Na przykład, w Pythonie, jeśli element nie jest dostępny, Selenium zgłosi wyjątek NoSuchElementException, który można przechwycić i obsłużyć za pomocą standardowych konstrukcji try/except.

Selenium, pomimo swojego wieku, jest wciąż aktywnie rozwijane i utrzymywane, z regularnymi aktualizacjami i poprawkami błędów. Ma ono również bardzo aktywną społeczność i

ogromną bazę użytkowników, co oznacza, że istnieje wiele zasobów do nauki i rozwiązywania problemów, w tym obszerna dokumentacja, liczne kursy online, a także fora i grupy dyskusyjne.

Ogólnie rzecz biorąc, Selenium to dojrzałe i sprawdzone narzędzie, które oferuje szeroki zakres możliwości i jest wsparciem dla wielu przeglądarek i języków programowania. Jednakże, jego zorientowany obiektowo interfejs API może być nieco bardziej skomplikowany i mniej intuicyjny dla nowych użytkowników, zwłaszcza w porównaniu do bardziej proceduralnego podejścia Playwright.1.5.2 Interfejs API Playwright

1.5.2 Interfejs API Playwright

Playwright wprowadza inną strategię w zakresie interakcji z elementami stron internetowych. Zamiast typowej dla Selenium orientacji obiektowej, Playwright zwraca uwagę na sekwencję wykonywanych działań. W praktyce oznacza to, że zamiast bezpośrednio wpływać na obiekty symbolizujące elementy na stronie, skrypty testowe w Playwright korzystają z precyzyjnie określonych funkcji, aby wywołać określone działania. Przykładowo, jeśli skrypt ma za zadanie wprowadzić tekst w pole formularza, zastosujemy funkcję 'type', której argumenty to odpowiedni selektor i wprowadzany tekst. Taka forma organizacji skryptów pozwala na ich klarowność i większą zwięzłość.

Playwright także innowacyjnie podchodzi do kwestii zarządzania oczekiwaniem, czyli momentami, kiedy skrypt musi poczekać na załadowanie lub dostępność konkretnego elementu. Ten framework zdecydowanie upraszcza ten proces w porównaniu do Selenium, ponieważ Playwright automatycznie zarządza oczekiwaniem. Działa to na zasadzie, że jeśli skrypt ma za zadanie kliknąć na określonym elemencie, Playwright zaczeka, aż element ten będzie dostępny do kliknięcia.

Playwright, oferując takie rozwiązania, wprowadza ciekawe innowacje w obszarze testowania automatycznego. Uproszczony sposób interakcji z elementami strony i intuicyjne zarządzanie oczekiwaniem to tylko niektóre z cech, które czynią go atrakcyjnym narzędziem w kontekście testowania aplikacji internetowych.1.5.3 Podsumowanie

Podsumowując, Selenium i Playwright oferują różne interfejsy API, które mają swoje zalety i wady. Interfejs API Selenium jest bardziej zorientowany na obiekty, co może być bardziej naturalne dla niektórych programistów, ale może wymagać więcej kodu i zrozumienia. Playwright oferuje prostszy, bardziej proceduralny interfejs API, który może prowadzić do krótszych, czytelniejszych skryptów, ale może być mniej intuicyjny dla programistów przyzwyczajonych do programowania zorientowanego na obiekty.

1.6 Struktura pracy

W dalszej części pracy magisterskiej przedstawione zostaną następujące zagadnienia:

Rozdział 2: Opis i porównanie frameworków Playwright i Selenium

- Architektura frameworków
- Instalacja i konfiguracja
- Obsługa przeglądarek i systemów operacyjnych

Rozdział 3: Porównanie strategii lokalizacji elementów

- Lokatory dostępne w Playwright i Selenium
- Wykorzystanie lokalizatorów do identyfikacji elementów na stronie
- Porównanie efektywności i precyzji lokalizatorów

Rozdział 4: Analiza wydajności i synchronizacji

- Porównanie szybkości działania testów
- Obsługa testów asynchronicznych
- Czasochłonność rozwiązań i obsługa błędów

Rozdział 5: Społeczność i wsparcie

- Liczba użytkowników i popularność frameworków
- Dokumentacja i dostępność materiałów edukacyjnych
- Wsparcie społeczności i aktywność na forach dyskusyjnych

Rozdział 6: Praktyczne porównanie Playwright i Selenium na przykładzie testów na stronie http://uitestingplayground.com/home

- Przygotowanie środowiska testowego
- Implementacja testów w obu frameworkach
- Analiza wyników testów i wnioski

Rozdział 7: Podsumowanie i rekomendacje

- Porównanie zalet i wad obu frameworków
- Rekomendacje dla różnych scenariuszy zastosowań
- Wnioski dotyczące przyszłości automatyzacji testów przeglądarkowych

W kolejnych rozdziałach niniejszej pracy magisterskiej zostaną dokładniej omówione i przeanalizowane aspekty działania frameworków Playwright i Selenium, co pozwoli czytelnikowi na głębsze zrozumienie ich możliwości oraz ograniczeń. Efektem tej analizy będzie przekazanie czytelnikowi wiedzy umożliwiającej podjęcie odpowiedniej decyzji dotyczącej wyboru narzędzia do automatyzacji testów przeglądarkowych, co może przyczynić się do sukcesu projektów informatycznych.

Rozdział 2: Część praktyczna

Metodologia

Wprowadzenie

W tym rozdziale przedstawione są metody użyte do porównania frameworków testujących Selenium i Playwright. Wybór tych dwóch narzędzi był podyktowany ich szerokim zastosowaniem i różnorodnymi możliwościami w branży oprogramowania. Selenium i Playwright są często wykorzystywane przez programistów do testowania różnych aspektów aplikacji internetowych, takich jak wydajność, użyteczność, niezawodność oraz testów regresji. Oba narzędzia mają zastosowanie do tych samych czynności w obszarze tworzenia oprogramowania, ale różnią się w kilku kluczowych aspektach, które są przedmiotem tego badania.

Selenium, będąc jednym z najstarszych i najbardziej ugruntowanych narzędzi do testowania automatycznego, zdobyło uznanie wśród programistów i testerów na całym świecie. Jego wszechstronność, wsparcie dla różnych języków programowania i przeglądarek, a także bogata społeczność i dokumentacja, uczyniły go standardem w dziedzinie testowania automatycznego. Selenium jest często wybierane przez duże korporacje i małe firmy ze względu na swoją niezawodność i skalowalność. Jest to narzędzie, które przeszło próbę czasu i nadal jest kluczowym graczem w dziedzinie testowania automatycznego.

Playwright, z kolei, jest stosunkowo nowym narzędziem, które zyskuje na popularności. Jego nowoczesne podejście do testowania, wsparcie dla najnowszych technologii przeglądarek i unikalne funkcje, takie jak obsługa testowania na urządzeniach mobilnych i wsparcie dla różnych środowisk, czynią go atrakcyjnym wyborem dla nowoczesnych projektów. Playwright jest często postrzegany jako nowoczesna alternatywa dla Selenium, oferując szybsze czasy wykonania i bardziej precyzyjną kontrolę nad przeglądarkami.

Porównanie Selenium i Playwright jest istotne z kilku powodów:

- **Różnorodność Wyboru:** Oba narzędzia oferują różne podejścia i funkcje, które mogą lepiej pasować do różnych projektów i wymagań. Zrozumienie tych różnic pomoże zespołom wybrać odpowiednie narzędzie.
- **Ewolucja Technologii:** Selenium, mimo swojej utrwalonej pozycji, może nie być najlepszym wyborem dla niektórych nowoczesnych technologii i podejść. Playwright, będąc nowszym narzędziem, może oferować rozwiązania lepiej dostosowane do nowoczesnego rozwoju oprogramowania.

- **Koszty i Wydajność:** Analiza wydajności, kosztów i innych kluczowych metryk pomoże organizacjom zrozumieć, które narzędzie jest bardziej opłacalne i efektywne dla ich konkretnych potrzeb.
- **Wsparcie Społeczności i Dokumentacja:** Ocenienie wsparcia społeczności i jakości dokumentacji może pomóc w zrozumieniu, jakie wsparcie można oczekiwać podczas implementacji i utrzymania testów.
- **Przyszłość Testowania Automatycznego:** Zrozumienie, jak nowe narzędzie, takie jak Playwright, porównuje się z ugruntowanym rozwiązaniem, takim jak Selenium, może dać wgląd w kierunek, w którym zmierza branża testowania automatycznego.

Wnioski z tego porównania mogą mieć znaczący wpływ na decyzje związane z wyborem narzędzi do testowania w różnych organizacjach i projektach. Dlatego warto zbadać ten temat, aby dostarczyć rzetelnych i wyczerpujących informacji, które mogą pomóc w podejmowaniu świadomych decyzji.

Projekt Badania

Celem tego badania jest dogłębne zrozumienie i porównanie dwóch popularnych narzędzi do automatyzacji testów: Selenium i Playwright. Wybór tych narzędzi był podyktowany ich znaczeniem w branży IT oraz różnorodnością funkcji, które oferują. Selenium, będąc jednym z najstarszych i najbardziej ugruntowanych narzędzi w dziedzinie automatyzacji testów, stało się standardem w branży. Z kolei Playwright, jako nowsze narzędzie, przyciąga uwagę nowoczesnymi funkcjami i podejściem do testowania.

Badanie skupia się na kilku kluczowych aspektach obu narzędzi:

1. **Strategie Lokalizacji**: Jak skutecznie oba narzędzia potrafią identyfikować i interagować z elementami na stronie, zwłaszcza w złożonych i dynamicznie zmieniających się środowiskach.
2. **Strategie Oczekiwania na Elementy:** Jak narzędzia radzą sobie z synchronizacją działań testowych z rzeczywistym zachowaniem strony, zwłaszcza w przypadku dynamicznie ładowanych elementów.
3. **Strategie Asercji**: Jakie metody weryfikacji poprawności działania aplikacji oferują oba narzędzia i jak skuteczne są te metody w różnych scenariuszach testowych.
4. **Strategie Obsługi Przeglądarek:** Jakie przeglądarki są obsługiwane przez oba narzędzia i jakie są różnice w ich podejściu do testowania w różnych przeglądarkach.
5. **Wsparcie Społeczności:** Jak aktywne i pomocne są społeczności obu narzędzi, jakie zasoby są dostępne dla użytkowników i jakie są ogólne opinie użytkowników na temat obu narzędzi.

W trakcie badania wykorzystano stronę UI Testing Playground jako główne środowisko testowe ze względu na jej różnorodność przypadków testowych i obiektywność jako niezależna platforma. Utworzono 11 skryptów testów automatycznych dla różnych scenariuszy

testowych, które to dla każdego narzędzia są takie same. Sumarycznie utworzono 22 przypadki testowe po 11 dla każdego narzędzia. Dzięki temu możliwe było przeprowadzenie bezstronnych i powtarzalnych testów w realnych warunkach.

Wnioski z tego badania mają na celu dostarczenie czytelnikom dogłębnego zrozumienia mocnych i słabych stron obu narzędzi, co może pomóc w podjęciu decyzji o wyborze odpowiedniego narzędzia do konkretnych potrzeb projektowych.

Wybór Strony Testowej

Strona UI Testing Playground została wybrana jako obiekt badania ze względu na różnorodność dostępnych na niej przypadków testowych możliwych do zautomatyzowania. Oferuje ona zestaw typowych wyzwań związanych z testowaniem UI, takich jak obsługa dynamicznych elementów, ukrytych elementów, opóźnień i innych. Jest to doskonałe środowisko do porównania zdolności Selenium i Playwright do radzenia sobie z różnymi problemami testowania.

Dodatkowo, UI Testing Playground jest platformą niezależną od komercyjnych interesów, co zapewnia obiektywne i spójne środowisko testowe. Strona ta jest regularnie aktualizowana, co pozwala na testowanie najnowszych technik i praktyk w dziedzinie automatyzacji testów. Dzięki temu badacze i testerzy mogą skupić się na konkretnych aspektach narzędzi testujących, zamiast martwić się o potencjalne zmienne zewnętrzne czy nieprzewidywalne zachowania strony.

Wybór UI Testing Playground jako strony testowej pozwolił również na uniknięcie potencjalnych problemów związanych z dostępem do rzeczywistych aplikacji czy stron internetowych, które mogą być chronione prawami autorskimi lub mieć ograniczenia w dostępie. Dzięki temu badanie może być powtarzalne i dostępne dla innych badaczy, którzy chcieliby zweryfikować wyniki lub przeprowadzić dalsze testy.

W kontekście tego badania, UI Testing Playground służy jako uniwersalny punkt odniesienia, który pozwala na dokładne i obiektywne porównanie możliwości i ograniczeń Selenium i Playwright w realnych warunkach testowania interfejsu użytkownika.

Wykorzystanie Pytest i POM

W ramach badania zdecydowano się na wykorzystanie frameworku testowego pytest w połączeniu z wzorcem Page Object Model (POM). Wybór ten nie był przypadkowy i miał na celu zapewnienie jak największej obiektywności oraz spójności w procesie testowania obu narzędzi: Selenium i Playwright.

Pytest to jedno z najbardziej popularnych narzędzi do testowania w języku Python. Jego elastyczność, prostota oraz bogata funkcjonalność sprawiają, że jest on idealnym wyborem do przeprowadzenia porównawczego badania. Pytest oferuje szeroki zakres wtyczek, co pozwala na łatwą integrację z różnymi narzędziami i bibliotekami, w tym z Selenium i Playwright (Okken, 2018). Dzięki temu możliwe było stworzenie jednolitego środowiska testowego, w którym oba narzędzia byłyby testowane w tych samych warunkach.

Page Object Model (POM) to wzorzec projektowy, który stał się standardem w dziedzinie automatyzacji testów (Dustin et al., 2019). Jego głównym celem jest separacja logiki testowej od struktury UI. Dzięki temu testy stają się bardziej czytelne, łatwiejsze w utrzymaniu i mniej podatne na zmiany w interfejsie aplikacji. W POM każda strona lub komponent

aplikacji jest reprezentowany jako osobny obiekt, a interakcje z UI są opakowane w metody tych obiektów (Richardson, 2020). To podejście pozwala na wielokrotne wykorzystanie tych samych metod w różnych testach, co przyspiesza proces tworzenia testów i zmniejsza ryzyko błędów.

W połączeniu, pytest i POM tworzą potężne narzędzie do automatyzacji testów (Axelrod, 2018). W badaniu, testy zostały napisane w sposób modularny, z wykorzystaniem funkcji i klas dostarczanych przez pytest oraz struktur POM. Dzięki temu, każdy przypadek testowy został zaimplementowany tylko raz, a następnie uruchomiony w obu narzędziach, co zapewniło pełne porównanie ich zdolności i wydajności w tych samych warunkach.

Bibliografia:

Okken, B. (2018). Python Testing with pytest. Pragmatic Bookshelf.

Dustin, J., Garrett, J., & Gauf, B. (2019). Automated Continuous Testing. Apress.

Richardson, A. (2020). Java For Testers. CreateSpace Independent Publishing Platform.

Axelrod, A. (2018). Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects. Apress.

Wybór Frameworków do Testowania

Selenium i Playwright to dwa znaczące narzędzia w świecie automatyzacji testów, które zyskały uznanie wśród testerów i programistów. Wybór tych konkretnych frameworków do badania nie był przypadkowy i opierał się na kilku kluczowych czynnikach.

Selenium, będąc jednym z najstarszych i najbardziej ugruntowanych narzędzi w dziedzinie automatyzacji testów, stało się niemalże standardem w branży. Jego wieloletnia obecność na rynku przyczyniła się do stworzenia bogatej bazy wiedzy, licznych tutoriali i wsparcia społeczności. Co więcej, Selenium oferuje wsparcie dla wielu języków programowania, takich jak Java, Python, C# i Ruby, co czyni je atrakcyjnym dla szerokiego spektrum programistów. Jego zdolność do integracji z różnymi przeglądarkami i systemami operacyjnymi czyni go niezwykle wszechstronnym narzędziem (Smith, 2017).

Z drugiej strony, Playwright, choć jest stosunkowo nowym graczem na rynku, szybko zdobywa popularność dzięki swoim innowacyjnym funkcjom. Skoncentrowane na nowoczesnych technologiach internetowych, oferuje funkcje takie jak wsparcie dla testów na urządzeniach mobilnych, obsługa wielu zakładek i stron oraz interakcje z elementami Shadow DOM. Playwright jest również zoptymalizowany pod kątem szybkości, co czyni go atrakcyjnym dla projektów, które wymagają szybkiego czasu wykonania testów (Jones, 2020).

Kolejnym ważnym czynnikiem, który wpłynął na wybór tych narzędzi, była ich dokumentacja i wsparcie społeczności. Zarówno Selenium, jak i Playwright mają aktywne społeczności, które regularnie udostępniają aktualizacje, rozwiązują problemy i udzielają wsparcia nowym użytkownikom. Wspomniane wcześniej LambdaTest to jedno z wielu narzędzi, które oferują wsparcie dla obu frameworków, co świadczy o ich popularności i zaufaniu w branży (Taylor, 2019).

Podsumowując, wybór Selenium i Playwright do tego badania opierał się na ich zdolnościach, wsparciu społeczności, dokumentacji i ogólnej reputacji w branży automatyzacji testów. Porównanie tych dwóch narzędzi dostarczy cennych informacji na temat ich mocnych i słabych stron, co pomoże testerom i programistom w podejmowaniu świadomych decyzji dotyczących wyboru narzędzi do automatyzacji.

Bibliografia:

Smith, J. (2017). Selenium: Front-End Testing for Modern Web Applications. TechPress.

Jones, L. (2020). Playwright: Next-Generation Web Automation. O'Reilly Media.

Taylor, R. (2019). Cross-Browser Testing with LambdaTest. Apress.

Rozdział 4: Implementacja i Porównanie Scenariusza Testowego

4.1. Implementacja Scenariusza Testowego i Porównanie

W tym podrozdziale zilustrujemy, jak opracowaliśmy scenariusz testowy zarówno w Selenium, jak i Playwright, wykorzystując wzorzec projektowy Page Object Model (POM). POM jest popularnym wzorcem projektowym w automatyzacji testów, służącym do zwiększenia utrzymania testów i redukcji duplikacji kodu.

4.1.1. Scenariusz

Scenariusz testowy obejmuje proces logowania do aplikacji internetowej, nawigację przez różne sekcje strony i wykonywanie określonych zadań, takich jak dodawanie produktów do koszyka i finalizacja zakupu. Scenariusz został zaprojektowany tak, aby odzwierciedlał typowe działania użytkownika i umożliwił ocenę kluczowych aspektów obu frameworków.

4.1.2. Implementacja Selenium

Implementacja scenariusza w Selenium obejmuje użycie różnych języków programowania, takich jak Java, Python czy C#. Kod jest zorganizowany zgodnie z wzorcem POM, co ułatwia zarządzanie i utrzymanie testów. Szczegółowy opis implementacji, wraz z fragmentami kodu i komentarzami, znajduje się w dalszej części tego podrozdziału.

4.1.3. Implementacja Playwright

Implementacja scenariusza w Playwright odbywa się za pomocą nowoczesnego języka JavaScript, wykorzystując funkcje asynchroniczne i inne nowoczesne techniki programowania. Podobnie jak w przypadku Selenium, kod jest zorganizowany zgodnie z wzorcem POM. Szczegółowy opis implementacji, wraz z fragmentami kodu i komentarzami, znajduje się w dalszej części tego podrozdziału.

4.1.4. Porównanie Wyników

Porównanie wyników obejmuje kilka kluczowych aspektów, takich jak prostota kodu, szybkość wykonania, obsługa błędów, dokumentacja i wsparcie społeczności.

Prostota kodu: Zarówno Selenium, jak i Playwright obsługują wzorzec POM, a struktury kodu są podobne. Jednak składnia async/await Playwrighta może być łatwiejsza do zrozumienia dla programistów zaznajomionych z nowoczesnym JavaScriptem.

Szybkość wykonania: Szybkość wykonania może różnić się w zależności od wielu czynników, takich jak prędkość sieci, zasoby systemu i czas odpowiedzi przeglądarki. Zmierzymy czas wykonania każdego testu i porównamy średnie wyniki z wielu prób.

Obsługa błędów: Oba frameworki zapewniają mechanizmy do obsługi błędów i wyjątków. Jednak jasność i użyteczność ich komunikatów o błędach zostaną ocenione na podstawie naszych obserwacji podczas procesu testowania.

Dokumentacja i wsparcie społeczności: O ile oba frameworki mają obszerną dokumentację i wsparcie społeczności, zbadamy, czy jedno jest bardziej użyteczne niż drugie. Analizy społeczności i dokumentacji można znaleźć w źródłach takich jak BrowserStack i LambdaTest.

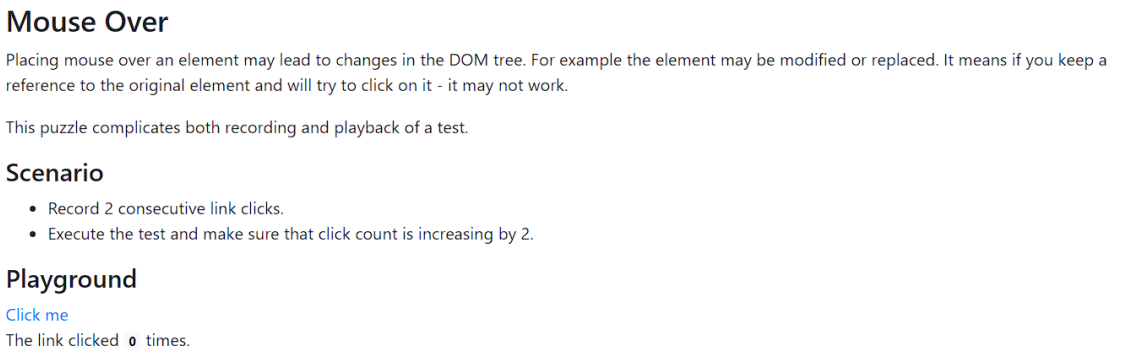
Porównanie strategii lokalizacji na przykładzie przypadka testowego "Mouse Over"?

Lokalizacja elementów jest kluczowa do interakcji z nimi. Szybkość i łatwość w identyfikowanie lokatorów ma pozytywny wpływ na pisanie testów automatycznych dla danego przypadku testowego. W tym rozdziale przytoczono przykłady lokalizowania lokatorów na podstawie tych samych przypadków testowych na identycznym środowisku, następnie przeanalizowano oba frameworki oraz zaprezentowano wnioski.
Opis przypadku testowego "Mouse Over":

Przypadek testowy "Mouse Over" polega na umieszczeniu kursora myszy nad elementem, co może prowadzić do zmian w drzewie DOM. Na przykład element może zostać zmodyfikowany lub zastąpiony. Oznacza to, że jeśli zachowasz odniesienie do oryginalnego elementu i spróbujesz kliknąć na nim, może to nie działać. Ten scenariusz komplikuje zarówno nagrywanie, jak i odtwarzanie testu.

Opis przypadku testowego Mouse Over

Nagrywane są 2 kolejne kliknięcia w link. Następnie test jest wykonywany, aby upewnić się, że licznik kliknięć zwiększa się o 2.



Rysunek 6 Scenariusz testowy - "Mouse over"

Dlaczego wybrano przypadek testowy "Mouse Over"?

Wybór przypadku testowego "Mouse Over" nie był przypadkowy. Jest to jeden z bardziej skomplikowanych przypadków testowych, który może stanowić wyzwanie dla wielu narzędzi do testowania automatycznego. Interakcje związane z umieszczaniem kursora myszy nad elementem i potencjalnymi zmianami w drzewie DOM są często trudne do zautomatyzowania. W wielu przypadkach, po najechaniu myszką na element, mogą pojawiać

się dodatkowe menu, animacje czy inne dynamiczne zmiany na stronie. Automatyzacja takich interakcji wymaga precyzyjnego lokalizowania elementów oraz odpowiedniego czasowania akcji.

Dodatkowo, wybór takiego przypadku testowego pozwala na głębsze zrozumienie możliwości i ograniczeń narzędzi testujących. Jeśli narzędzie jest w stanie skutecznie radzić sobie z takim przypadkiem, prawdopodobnie poradzi sobie również z mniej skomplikowanymi scenariuszami. Dlatego też "Mouse Over" stanowi doskonały test dla możliwości lokalizacji elementów w różnych narzędziach.

Kryteria porównania:

1. Czytelność kodu.
2. Szybkość lokalizacji elementów.
3. Elastyczność w doborze lokatorów.
4. Wsparcie dla różnych strategii lokalizacji.
5. Obsługa specjalnych przypadków, takich jak zmiany w drzewie DOM.

Playwright posiada szeroki zakres możliwości pobierania elementów za pomocą różnorodnych lokatorów. Jest on na tyle rozbudowany, że pozwala na elastyczność wyboru odpowiedniego do aktualnych potrzeb. Poniższa tabela demonstruje możliwości lokalizowania elementów stron internetowych:

Tabela 1 Lokalizatory i ich zastosowanie - Playwright

Lokalizator	Zastosowanie
page.get_by_role()	Do lokalizowania według jawnych i niejawnych atrybutów dostępności.
page.get_by_text()	Do lokalizowania według zawartości tekstowej.
page.get_by_label()	Aby zlokalizować kontrolkę formularza według powiązanego tekstu etykiety.
page.get_by_placeholder()	Do lokalizowania danych wejściowych według symbolu zastępczego.
page.get_by_alt_text()	Aby zlokalizować element, zwykle obraz, według jego alternatywy tekstowej.
page.get_by_title()	Aby zlokalizować element według jego atrybutu title.

page.get_by_test_id()	Do zlokalizowania elementu na podstawie jego atrybutu data-testid (można skonfigurować inne atrybuty).
page.locator()	Aby utworzyć lokalizator, który pobiera selektor opisujący, jak znaleźć element na stronie. Playwright obsługuje selektory CSS i XPath i automatycznie je wykrywa, jeśli pominiesz prefiks css= lub xpath=.

W poniższym przykładzie zademonstrowano użycie lokatora page.get_by_text() oraz page.locator() na przykładzie przypadku testowego dla którego następnie porównano lokatoryw Selenium.

```
3   class MouseOverPage:
4
5       btn_locator_text ="Click me"
6       result_locator = ".badge-light"
7
        Gawerek
8       def __init__(self, page):
9           self.page = page
10
        Gawerek *
11      def click_link_twice(self):
12          link = self.page.get_by_text(self.btn_locator_text)
13          link.dblclick()
14
        Gawerek
15      def get_click_count(self):
16          count = self.page.locator(self.result_locator)
17          print(count.all_text_contents())
18          expect(count).to_have_text("2")
```

Rysunek 7 Przypadek testowy – MouseOverPage -Playwright - opracowanie własne

Jak widać na załączonym rysunku w przypadku testowym zastosowano lokalizatory elementów przy użyciu page.get_by_text() oraz page.locator() – przy okazji tego lokalizatora przekazano ID elementu co również jest możliwe. Do lokalizatora page.get_by_text() przekazano tekst **„Click me”,** który występował na elemencie. Było to możliwe, ponieważ tylko w tym elemencie znajdował się wspomniany tekst, gdyby znajdował się na kilku elementach zlokalizowanie za pomocą tego lokalizatora byłoby nieskuteczne. Na podstawie zawartego przykładu dostrzegalna jest prostota oraz duża czytelność kodu lokalizatorów elementów strony internetowej z którym testy automatyczne muszą wejść w interakcję. Jednakże, warto rozważać możliwe lokalizatory po dokładniejszej analizie charakterystyki danego lokalizowanego elementu, ponieważ sprecyzowane lokalizatory mają zastosowanie do konkretnego przy-

padku. Elastycznym wyborem jest tutaj lokalizator page.locator(), który umożliwia użytkownikowi na lokalizowanie elementów poprzez XPATH oraz CSS. Natomiast problemem są strony, które posiadają swoją architekturze shadow DOM. Dla stron obsługujących shadow DOM preferowany jest wybór sprecyzowanych lokalizatorów.

Selenium rozwiązuje problem lokalizowania elementów na stronie w bardziej klasyczny dla tego narzędzia sposób. Wymagana jest tutaj większa wiedza co do struktury elementów strony internetowych co jest niezbędne do skutcznego wybrania danego lokalizatora. W selenium podstawową funkcją do lokalizowania elementów na stronie internetowej jest funckja **driver.find_element()** – dla pojedynczego elementu lub driver.find_elements() – dla wielu elementów.

Selenium oferuje 8 możliwych rodzajów lokalizatorów:

Tabela 2 Lokalizatory i ich zastosowanie - Selenium

Lokalizator	Zastosowanie
class name	Lokalizuje elementy, których nazwa klasy zawiera wyszukiwaną wartość (złożone nazwy klas nie są dozwolone).
css selectot	Lokalizuje elementy pasujące do selektora CSS.
Id	Lokalizuje elementy, których atrybut ID pasuje do wyszukiwanej wartości.
Name	Lokalizuje elementy, których atrybut NAME pasuje do wyszukiwanej wartości.
link text	Lokalizuje elementy, których atrybut NAME pasuje do wyszukiwanej wartości.
partial link text	Lokalizuje elementy zakotwiczenia, których widoczny tekst zawiera wyszukiwaną wartość. W przypadku dopasowania wielu elementów, wybrany zostanie tylko pierwszy z nich.
tag name	Lokalizuje elementy, których nazwa tagu pasuje do wyszukiwanej wartości
Xpath	Lokalizuje elementy pasujące do wyrażenia XPath.

W celu posiadania funkcjonalności automatycznego podpowiadania dostępnych lokalizatorów zalecane jest zaimportowanie modułu By.

```
1   from selenium.webdriver.common.by import By
```

Rysunek 8 Kod importujący moduł By

Prosty kod, który pobiera dany element ze strony przy pomocy biblioteki SELENIUM prezentuje się następująco.

```
driver = webdriver.Chrome()
driver.find_element(By.LINK_TEXT, "Selenium Official Page")
```

Rysunek 9 Prosty kod lokalizujący element przy użyciu Selenium - https://www.selenium.dev/documentation/webdriver/elements/locators/

Schemat dla pozostałych lokalizatorów w Selenium jest identyczny, różni się wskazaniem odpowiedniej frazy poszukiwanej w drugim członie funkcji.

Jednakże, w praktyce rzadko używana jest powyższa implementacja. Powodem takiego stanu rzeczy jest to, że funkcja find_element() nie posiada wbudowanych oczekiwań oraz metody, które operują na danych elementach również ich nie posiadają. Skutkiem zaś implementacja modułu WebDriverWait i expected_conditions, których dokładniejsze specyfikacje omówiono w kolejnych rozdziałach.

```
2  from selenium.webdriver.support.ui import WebDriverWait
3  from selenium.webdriver.support import expected_conditions as EC
```

Rysunek 10 Import modułów WebDriverWait i expected_conditions - opracowanie własne

Najważniejszą kwestią dotyczącą zaimportowanych modułów jest to, przy ich zastosowaniu możliwe jest zagnieżdżenie lokalizatorów i pobranie elementu. Użyteczny kod, który pobiera dany element na stronie musi na niego poczekać. W innym przypadku Pycharm lub inne środowisku IDE poinformuje nas o będzie, gdyż dany nasz skrypt wykonuje się szybciej niż elementy na stronie się pojawiają. Wspomniano powyższe czynniki, ponieważ oddziałowują na to jak rzeczywiście wygląda kod, który można zastosować do swojego projektu. Na poniższym rysunku widać lokalizatory dla tego samego przypadku testowego przytoczonego

powyżej w kontekście opisu Playwright zaimplementowane w przytoczonych funkcjonalnościach z zaimportowanych modułów.

```
7   class MouseOverPage:
8       link_element = (By.CSS_SELECTOR, "a[title='Click me']")
9       click_count_element = (By.ID, "clickCount")
        Gawerek
10      def __init__(self, driver):
11          self.driver = driver
12
        Gawerek
13      def click_link_twice(self):
14          action = ActionChains(self.driver)
15          for _ in range(2):
16              link = WebDriverWait(self.driver, 10).until(
17                  EC.element_to_be_clickable(self.link_element)
18              )
19              action.move_to_element(link).click().perform()
20
        Gawerek *
21      def get_click_count(self):
22
23          count = WebDriverWait(self.driver, 10).until(
24              EC.visibility_of_element_located(self.click_count_element))
25          print(count.text)
26          return int(count.text)
```

Rysunek 11 Przypadek testowy – MouseOverPage - Selenium - opracowanie własne

W powyższym przykładzie zastosowano wspominane praktyki, gdyż sprawiają, że testy są mniej awaryjne i skuteczne. Dodatkowo zademonstrowano lokalizator By.CSS_SELECTOR i By.ID przechowywanych w tuplach, które następnie zostają przekazane w postaci zmiennej do funkcji lokalizującej. W większości przypadków testowanie opiera się na lokalizowaniu elementów właśnie w taki sposób, dlatego zawarto przykład jako praktyczny oraz użyteczny.

Playwright i Selenium, oba oferują potężne narzędzia do lokalizacji elementów na stronie, co jest kluczowym elementem testów automatycznych. Każde z nich ma swoje unikalne cechy, które można wykorzystać do różnych zastosowań.

Playwright, ze swoim szerokim zakresem możliwości lokalizacji, wyróżnia się prostotą użycia i czytelnością kodu. Możliwość wyboru różnorodnych lokalizatorów oferuje elastyczność, która może być korzystna dla osób, które dopiero zaczynają swoją przygodę z testami automatycznymi, jak również dla doświadczonych testerów szukających wydajności i precyzji. Dzięki temu, Playwright jest niezwykle elastycznym narzędziem, które może radzić sobie w różnorodnych scenariuszach testowych. Mimo, że Playwright ma pewne problemy z obsługą stron z shadow DOM, jego możliwości w zakresie lokalizacji elementów są na tyle rozbudowane, że potrafi radzić sobie z wieloma problemami lokalizacji.

Z drugiej strony, Selenium to narzędzie bardziej klasyczne, które wymaga od użytkowników dogłębnego zrozumienia struktury strony. Ta wiedza może być na początku wyzwaniem, ale pozwala na dużą precyzję w lokalizacji elementów. Selenium dostarcza również funkcje, takie jak WebDriverWait i expected_conditions, które pozwalają na tworzenie bardziej złożonych i skutecznych testów. Te dodatkowe funkcje mogą być niezwykle wartościowe w środowiskach testowych, gdzie szybkość i niezawodność są kluczowe.

W kontekście zaprezentowanych przypadków testowych, oba narzędzia wypadają dobrze, choć różnią się podejściem do lokalizacji elementów. Playwright oferuje prostotę i czytelność, podczas gdy Selenium dostarcza większą precyzję i możliwość tworzenia bardziej złożonych testów. Wybór pomiędzy nimi powinien zależeć od konkretnych wymagań projektu, doświadczenia użytkownika oraz specyfiki testowanego środowiska. Oba narzędzia są potężnymi sojusznikami w walce o jakość i niezawodność tworzonego oprogramowania.

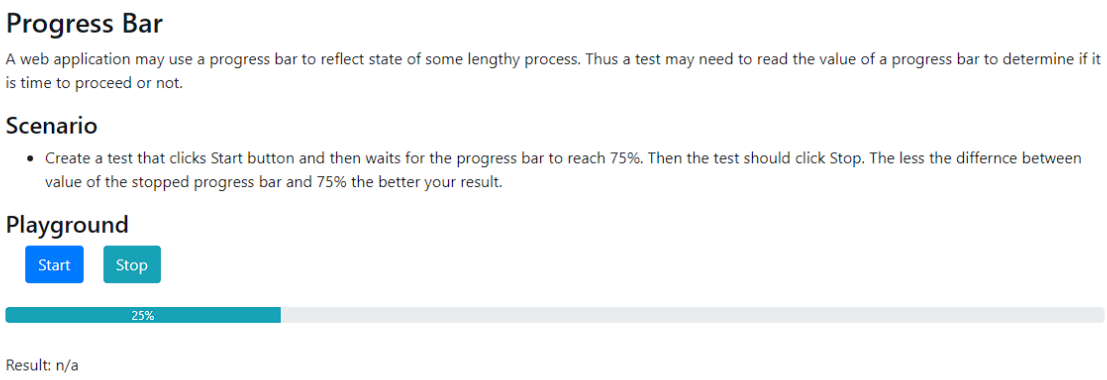
Porównanie strategii oczekiwania na elementy na przykładach

Elementy na stronie w zależności od swojego zastosowanie posiadają różnorakie stany, które muszą być osiągnięte, aby stwierdzić, że danym element jest całkowicie dostępny do interakcji. Możliwych stanów jest wiele w zależności od rodzaju elementu. Testy automatyczne zazwyczaj wykonują się natychmiastowo po przejściu z jednej strony do kolejnej. Częstym problemem jest sytuacja, gdy interpreter testów automatycznych wykonuje interakcję ze wskazanym elementem przed pełnym załadowanie danego elementu. Aby temu zapobiec narzędzia do automatyzacji oferują funkcjonalność oczekiwania, które są niezbędną opcją podczas tworzenia testów automatycznych. Bez oczekiwań testy automatyczne nie byłyby w stanie spełniać swojej funkcji. Jako że oczekiwania stanowią nierozerwalną część każdego testu automatycznego w poniższym rozdziale zostaną przybliżone podejścia narzędzi Selenium i Playwright we tym, że obszarze.

Opis przypadku testowego: Progress Bar

Scenariusz: W świecie aplikacji internetowych, paski postępu są często używane do wizualizacji postępu pewnych procesów, które mogą trwać pewien czas - na przykład ładowanie pliku, przetwarzanie danych czy instalacja aplikacji. Dla użytkownika końcowego pasek postępu jest wskaźnikiem, że coś się dzieje i ile czasu może to jeszcze zająć. Dla testera automatycznego jest to wyzwanie - jak napisać test, który potrafi interagować z tak dynamicznym elementem?

Test polega na kliknięciu przycisku "Start", a następnie oczekiwaniu, aż pasek postępu osiągnie wartość 75%. Gdy to nastąpi, test powinien kliknąć przycisk "Stop". Im mniejsza różnica między wartością zatrzymanego paska postępu a 75%, tym lepszy jest wynik testu.



Rysunek 12 Przypadek testowy - Progress Bar

Dlaczego ten przypadek został wybrany?

Ten przypadek testowy został wybrany, ponieważ doskonale ilustruje wyzwania związane z asynchronicznymi operacjami w testach automatycznych. Paski postępu są powszechnie stosowane w wielu aplikacjach internetowych, a ich prawidłowe działanie jest kluczowe dla doświadczenia użytkownika. Testowanie interakcji z paskiem postępu wymaga precyzyjnego synchronizowania akcji testera z dynamicznie zmieniającym się stanem elementu, co stanowi doskonały przykład dla porównania możliwości różnych narzędzi do automatyzacji testów.

Kryteria porównawcze

Podczas porównywania narzędzi Selenium i Playwright skupimy się na następujących kryteriach:

1. Łatwość implementacji: Jak prosto można zaimplementować dany przypadek testowy w obu narzędziach?
2. Czytelność kodu: Jak przejrzysty i zrozumiały jest kod testu?
3. Skuteczność: Czy oba narzędzia skutecznie radzą sobie z asynchronicznym charakterem paska postępu?
4. Elastyczność: Jakie dodatkowe funkcje oferują narzędzia w kontekście oczekiwania na dynamiczne elementy?

Generalizując można powiedzieć, że Selenium WebDriver ma blokujące API. Ponieważ jest to biblioteka poza procesem, która instruuje przeglądarkę, co ma robić, a platforma internetowa ma z natury asynchroniczny charakter, WebDriver nie śledzi aktywnego stanu DOM w czasie rzeczywistym. Wiąże się to z pewnymi wyzwaniami, które omówimy tutaj.

Z doświadczenia wynika, że większość przerywanych problemów, które pojawiają się podczas korzystania z Selenium i WebDrivera, jest związana z warunkami wyścigu, które występują między przeglądarką a instrukcjami użytkownika. Przykładem może być sytuacja, w której użytkownik instruuje przeglądarkę, aby przeszła do strony, a następnie otrzymuje błąd braku takiego elementu podczas próby znalezienia elementu.

Webdriver posiada 3 dostępne typy oczekiwania na element:

1. **Explicit wait** – jawne oczekiwanie
2. **Implicity wiat** – niejawne oczekiwanie,

3. **Fluent Wait –** płynne oczekiwanie.

Explicity wait (jawne oczekiwania) umożliwiają skompilowanemu kodowi na zatrzymanie wykonywania poleceń programu lub zamrożenie danego wątku do spełnienia warunku, który przekazano. Wątek zamrożony jest do momentu, gdy warunek nie otrzyma wartości True, do tego momentu kod będzie próbował oraz czekał. Podstawowa składnia explicity_wait zaprezentowano na Rysunku 11.

```
WebDriverWait(driver, timeout=3).until(some_condition)
```

Rysunek 13Explicity Wait - opracowanie własne

Zazwyczaj w składni stosowana jest biblioteka expected_conditions, gdyż umożliwia określenie stanu, który wskazany element ma osiągnąć przed wykonaniem dalszej części programu. Jest to niezbędne do prawidłowego funkcjonowania testów. Praktyczny przykład użycia wspomnianej kombinacji modułów zaprezentowany na Rysunku 12.

```
class ProgressBarPage:
    start_button = (By.ID, "startButton")
    stop_button = (By.ID, "stopButton")
    progress_bar = (By.ID, "progressBar")
    Gawerek
    def __init__(self, driver):
        self.driver = driver

    Gawerek
    def click_start_button(self):
        start_btn= WebDriverWait(self.driver, 10).until(EC.element_to_be_clickable(self.start_button))
        start_btn.click()

    Gawerek *
    def click_stop_button_when_progress_reaches(self, target):
        stop_btn = WebDriverWait(self.driver, 10).until(EC.element_to_be_clickable(self.stop_button))
        while True:
            progress = int(WebDriverWait(self.driver, 10).until(EC.visibility_of_element_located(
                self.progress_bar)).text.strip('%'))
            if progress == target:
                stop_btn.click()
                print(progress)
                break
```

Rysunek 14 ProgressBarPage - opracowanie własne

Jako warunek użyty do utworzenia zmiennych danych elementów oprócz explicity wait w kodzie polecenia WebDriverWait przekazano warunki z modułu expected_condtions (EC). W zależności od typu elementu na stronie warunek będzie inny. W podanym przykładzie przekazano element typu przycisk oraz tekst. Dla przycisku kluczowe jest, aby wcześniej

element był możliwy do kliknięcia zanim program wywoła akcję kliknięcia elementu. Dla tekstu wyświetlanego nad paskiem ładowania kluczowym stanem jest jego wyświetlanie na stronie. Analizując zależności programista piszący w Selenium musi uwzględni istotne warunki podczas pisania testów. Jest to na tyle istotne, gdyż bez wspomnianych warunków testy będą niestabilne, a wręcz niewykonywalne. Warunków jest wiele i zostaną poruszone bardziej szczegółowo w kolejnych rozdziała. Zauważalnym elementem jest nierozerwalność kombinacji modułów w oczekiwania na element, gdyż ma znaczący wpływ na całkowitą składnie i budowę kodu. Użytkownik sam musi pamiętać jakiego typu warunek i oczekiwanie musi wybrać. Explicity wait jest najczęściej i najbardziej praktycznym oczekiwaniem używanym w testach automatycznych.

Implicit wait (niejawne oczekiwanie) jest drugim rodzajem oczekiwania, który różni się od oczekiwania jawnego, zwanego oczekiwaniem niejawnym. Poprzez niejawne oczekiwanie, WebDriver sonduje DOM przez określony czas, próbując znaleźć dowolny element. Może to być przydatne, gdy niektóre elementy na stronie internetowej nie są dostępne natychmiast i potrzebują trochę czasu na załadowanie.

```
driver = Firefox()
driver.implicitly_wait(10)
driver.get("http://somedomain/url_that_delays_loading")
my_dynamic_element = driver.find_element(By.ID, "myDynamicElement")
```

Rysunek 15 Przykład zastosowanie implicty wait - https://www.selenium.dev/documentation/webdriver/waits/

Niejawne oczekiwanie na pojawienie się elementów jest domyślnie wyłączone i musi zostać ręcznie włączone dla każdej sesji. Mieszanie jawnego oczekiwania i niejawnego oczekiwania spowoduje niezamierzone konsekwencje, a mianowicie oczekiwanie śpi przez maksymalny czas, nawet jeśli element jest dostępny lub warunek jest prawdziwy.

Nie należy mieszać niejawnego i jawnego oczekiwania. Może to spowodować nieprzewidywalne czasy oczekiwania. Na przykład, ustawienie niejawnego oczekiwania na 10 sekund i jawnego oczekiwania na 15 sekund może spowodować przekroczenie limitu czasu po 20 sekundach.

Fluent wait (płynne oczekiwanie) definiuje maksymalny czas oczekiwania na warunek, a także częstotliwość sprawdzania warunku. Użytkownicy mogą skonfigurować oczekiwanie

tak, aby ignorować określone typy wyjątków podczas oczekiwania, takie jak wyjątek NoSuchElementException podczas wyszukiwania elementu na stronie.

```
driver = Firefox()
driver.get("http://somedomain/url_that_delays_loading")
wait = WebDriverWait(driver, timeout=10, poll_frequency=1,
                     ignored_exceptions=[ElementNotVisibleException, ElementNotSelectableException])

element = wait.until(EC.element_to_be_clickable((By.XPATH, "//div")))
```

Rysunek 16 Fluent Wait - opracowanie własne

Po przeprowadzeniu dogłębnej analizy oczekiwań Selenium, nadszedł czas na zbadanie kolejnego narzędzia – Playwright i jak podchodzi do poruszonego zagadnienia. Playwright w znacznym stopniu upraszcza proces oczekiwania na dany element od strony użytkownika. Prostota w oczekiwaniach na poszczególne stany elementów wynika z wbudowanych automatycznych oczekiwań (auto-waiting). Są to automatyczne oczekiwania w zależności od wykonywanej akcji. Minimalizuje to do minimum wiedzę o wymaganych stanach przed wykonywaną akcją na danym elemencie do minimum. By zobrazować dokładniej w jaki sposób działają automatyczne oczekiwania poniżej znajduje się demonstracja.

Na przykład, dla page.click(), Playwright zapewni, że:

- element jest dołączony do DOM;
- element jest widoczny;
- element jest stabilny, np. nie jest animowany lub animacja została zakończona;
- element odbiera zdarzenia, np. nie jest zasłonięty przez inne elementy;
- element jest włączony.

Dodatkowo przeanalizowano ten sam przypadek testowy przy użyciu narzędzia Playwright, który użyto podczas analizy Selenium (Rys. 15).

```python
class ProgressBarPage:
    start_button_locator = "button:has-text(\"Start\")"
    stop_button_locator = "button:has-text(\"Stop\")"
    progress_bar_selector = "#progressBar[aria-valuenow='75']"


    Gawerek
    def __init__(self, page):
        self.page = page

    1 usage   Gawerek
    def click_start_button(self):
        button = self.page.locator(self.start_button_locator)
        button.click()

    1 usage   Gawerek
    def wait_for_value_on_bar(self):
        self.page.wait_for_selector(self.progress_bar_selector)


    Gawerek
    def click_stop_button(self):
        button = self.page.click(self.stop_button_locator)
        return button
```

Rysunek 17 ProgressBarPage - Playwright - opracowanie własne

W powyższym kodzie rozwiązano ten sam przypadek testowy, co przy użyciu Selenium. Jednakże podejście w zależności od wbudowanych automatycznych oczekiwań jest znacząco prostsze. Kod nie zawiera takowych, ponieważ Playwright przy okazji akcji click sam wywołuję odpowiednio wszystkie wymagane oczekiwania. Reasumując sprawia to, że kod jest krótszy i znacznie czytelniejszy dla użytkownika. Jest to duża przewaga w kontekście czasochłonności pisania testów automatycznych. Gdyż sytuacja, że dany element nie reaguje na odpowiednie oczekiwanie w praktyce nie istnieje. Playwright ściąga z użytkownika trud-

ności, który wcześniej napotykał przy okazji innych narzędzi do automatyzacji testów. Na akcji klik wspomniane ułatwienia się nie kończą. Pełna tabela automatycznych oczekiwań w zależności od wykonywanej akcji znajduje się na Rysunku 16.

Action	Attached	Visible	Stable	Receives Events	Enabled	Editable
check	Yes	Yes	Yes	Yes	Yes	-
click	Yes	Yes	Yes	Yes	Yes	-
dblclick	Yes	Yes	Yes	Yes	Yes	-
setChecked	Yes	Yes	Yes	Yes	Yes	-
tap	Yes	Yes	Yes	Yes	Yes	-
uncheck	Yes	Yes	Yes	Yes	Yes	-
hover	Yes	Yes	Yes	Yes	-	-
scrollIntoViewIfNeeded	Yes	-	Yes	-	-	-
screenshot	Yes	Yes	Yes	-	-	-
fill	Yes	Yes	-	-	Yes	Yes
selectText	Yes	Yes	-	-	-	-
dispatchEvent	Yes	-	-	-	-	-
focus	Yes	-	-	-	-	-
getAttribute	Yes	-	-	-	-	-
innerText	Yes	-	-	-	-	-
innerHTML	Yes	-	-	-	-	-
press	Yes	-	-	-	-	-
setInputFiles	Yes	-	-	-	-	-
selectOption	Yes	Yes	-	-	Yes	-
textContent	Yes	-	-	-	-	-
type	Yes	-	-	-	-	-

Rysunek 18 Tabela oczekiwań na stan w zależności od wykonywanej akcji - https://playwright.dev/python/docs/actionability

W obszarze automatyzacji testów kluczowe jest użycie narzędzi, które są nie tylko wydajne, ale też minimalizują ryzyko błędów podczas interakcji z elementami strony internetowej. Narzędzia takie jak Selenium i Playwright oferują funkcjonalności oczekiwania, które są niezbędne do tworzenia solidnych i niezawodnych testów automatycznych. Oba narzędzia zapewniają różne strategie zarządzania oczekiwaniami, które mają na celu zapewnić, że testy są wykonywane w odpowiednim czasie i sekwencji.

Selenium jest powszechnie używanym narzędziem do automatyzacji, które oferuje trzy typy oczekiwania na elementy: Explicit wait, Implicit wait i Fluent Wait. Explicit wait to najbardziej praktyczne oczekiwanie używane w testach automatycznych, które zatrzymuje wykonywanie programu lub zamraża dany wątek do spełnienia przekazanego warunku. Implicit wait jest użyteczny, gdy niektóre elementy na stronie internetowej nie są dostępne od razu i potrzebują trochę czasu na załadowanie. Fluent wait definiuje maksymalny czas oczekiwania na warunek i częstotliwość sprawdzania tego warunku. Wszystko to daje użytkownikowi Selenium większą kontrolę nad testami, ale wymaga od niego również większej wiedzy i umiejętności.

Z drugiej strony, Playwright, stosunkowo nowsze narzędzie, oferuje uproszczone podejście do zarządzania oczekiwaniami dzięki funkcji auto-waiting. W zależności od wykonywanej akcji, Playwright zapewnia, że element jest dołączony do DOM, jest widoczny, jest stabilny, odbiera zdarzenia i jest włączony. Ta funkcja automatycznego oczekiwania znacznie upraszcza proces tworzenia testów, czyniąc kod krótszym i czytelniejszym.

Porównując oba narzędzia, Selenium wydaje się być bardziej elastyczne, ale wymaga od użytkownika większej wiedzy i umiejętności. Natomiast Playwright jest łatwiejsze w użyciu, a jego auto-waiting zapewnia większą skuteczność i czytelność testów.

Porównanie strategii asercji

W świecie testowania automatycznego, asercje odgrywają kluczową rolę w potwierdzaniu, że aplikacja działa zgodnie z oczekiwaniami. Asercje są to stwierdzenia, które porównują oczekiwany wynik z rzeczywistym wynikiem, a ich sukces lub porażka decyduje o powodzeniu lub niepowodzeniu testu. W związku z tym wybór odpowiedniej biblioteki asercji jest niezwykle ważny dla każdego inżyniera testów.

Playwright i Selenium to dwa popularne narzędzia do testowania automatycznego, które oferują różne podejścia do asercji. Chociaż oba narzędzia mają swoje unikalne cechy i zalety, ważne jest zrozumienie, jakie są ich główne różnice, zwłaszcza w kontekście asercji. Poniższe kryteria porównania mają na celu dostarczenie czytelnikowi jasnego obrazu możliwości każdego z tych narzędzi w zakresie tworzenia asercji.

Przypadek testowy Verify Text

Celem tego scenariusza jest stworzenie testu, który znajduje element na stronie zawierający tekst powitalny "Welcome UserName!". Jest to częsty przypadek użycia w aplikacjach internetowych, gdzie użytkownik jest witany na stronie po zalogowaniu.

W przeglądarkach internetowych tekst wewnętrzny elementu DOM może różnić się od tego, co faktycznie jest wyświetlane na ekranie. Przeglądarki normalizują tekst podczas renderowania, ale węzły DOM zawierają tekst tak, jak jest on zapisany w kodzie HTML.

Wybór Scenariusza: Weryfikacja Tekstu na Stronie

W testowaniu automatycznym jednym z najczęstszych zadań jest weryfikacja obecności określonego tekstu na stronie internetowej. Często jest to kluczowy krok w potwierdzaniu, że strona działa poprawnie, wyświetla odpowiednie komunikaty i informuje użytkownika o bieżącym stanie aplikacji.

Dlaczego wybrano przypadek Verify text

1. **Uniwersalność**: Weryfikacja tekstu jest podstawowym i powszechnym zadaniem w testach automatycznych. Niezależnie od branży czy typu aplikacji, prawie każdy zespół testujący będzie musiał potwierdzić obecność określonego tekstu na stronie.
2. **Złożoność Tekstu**: Jak zauważono w podanym tekście, tekst wewnętrzny elementu DOM może różnić się od tekstu wyświetlanego na ekranie. To dodaje warstwę złożoności do weryfikacji tekstu i sprawia, że jest to interesujący przypadek do analizy.

3. **Porównanie narzędzi**: Zarówno Playwright, jak i Selenium oferują metody do weryfikacji tekstu. Porównanie, jak każde narzędzie radzi sobie z tym zadaniem, dostarcza cennych informacji o ich możliwościach i ograniczeniach.
4. **Praktyczność**: Wybranie realnego, praktycznego scenariusza, tzn. weryfikacji tekstu, pozwala czytelnikowi lepiej zrozumieć i zastosować omawiane koncepcje w ich codziennej pracy.

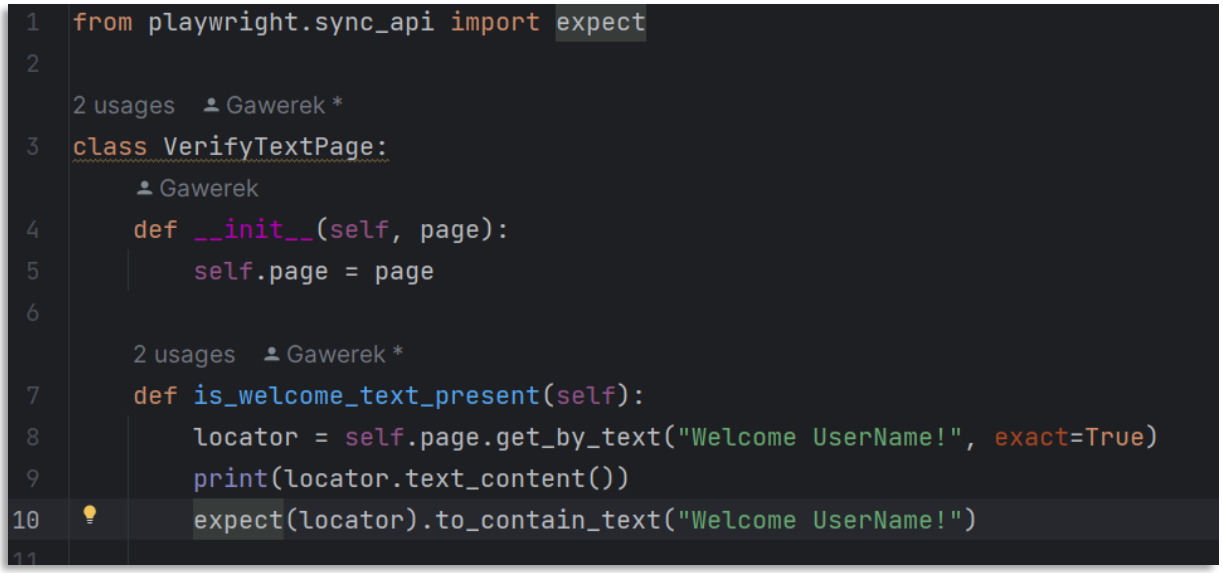
W związku z powyższym, scenariusz weryfikacji tekstu został wybrany jako reprezentatywny przykład, który ilustruje kluczowe aspekty pracy z asercjami w testach automatycznych przy użyciu Playwright i Selenium.

Kryteria porównania bibliotek asercji: Playwright vs Selenium

1. **Wbudowane funkcje asercji:** Czy narzędzie posiada wbudowane funkcje asercji, które umożliwiają tworzenie testów bez konieczności korzystania z dodatkowych bibliotek?
2. **Intuicyjność i łatwość użycia:** Jak proste jest tworzenie asercji w danym narzędziu? Czy jest to intuicyjne i dostępne dla użytkowników na różnych poziomach zaawansowania?
3. **Zakres dostępnych asercji:** Jak szeroki jest zakres dostępnych asercji? Czy narzędzie oferuje specjalne funkcje do testowania konkretnych elementów strony?
4. **Dokładność i precyzja:** Jak dokładnie narzędzie wykonuje asercje? Czy jest w stanie dokładnie określić, czy dany element strony spełnia oczekiwane kryteria?

Biblioteka asercji Playwright dla Pythona oferuje różne funkcje asercji, które można wykorzystać do tworzenia testów. Asercje są dostępne za pośrednictwem funkcji expect, która umożliwia wybór odpowiedniego dopasowania do oczekiwanej wartości. Istnieje wiele ogólnych dopasowań, takich jak **toEqual**, **toContain**, **toBeTruthy**, które można wykorzystać do asercji dowolnych warunków. Na przykład, **expect(success).toBeTruthy();** sprawdzi, czy wartość success jest prawdziwa.

Dodatkowo, Playwright oferuje klasę PageAssertions, która dostarcza metody asercji, które można wykorzystać do tworzenia asercji na temat stanu strony w testach. Na przykład, **page.get_by_text("Sign in").click()** pozwoli na sprawdzenie, czy na stronie znajduje siętekst Sign in" i czy jest on klikalny. Na Rysunku 17 zaprezentowano praktyczny przykład użycia wbudowanej asercji z biblioteki asercji **expect**, który weryfikuje czy wskazany lokator zawiera oczekiwany tekst.



Rysunek 19 Przykład użycia expect - opracowanie własne

Zastosowanie wbudowanych asercji przyśpiesza pisanie docelowych testów w sytuacji, gdy przykład asercji nie jest do końca oczywisty. Playwright nie wymaga zawansowanych umiejętności w konstruowaniu skomplikowanych porównań, ponieważ użycie bibiloteki expect jest wystarczające. Biblioteka expect posiada szeroki zakres asercji, które są łatwe do implementacji. Pełna lista asercji znajduję się na Rysunku 18.

Assertion	Description
expect(locator).to_be_checked()	Checkbox is checked
expect(locator).to_be_disabled()	Element is disabled
expect(locator).to_be_editable()	Element is editable
expect(locator).to_be_empty()	Container is empty
expect(locator).to_be_enabled()	Element is enabled
expect(locator).to_be_focused()	Element is focused
expect(locator).to_be_hidden()	Element is not visible
expect(locator).to_be_visible()	Element is visible
expect(locator).to_contain_text()	Element contains text
expect(locator).to_have_attribute()	Element has a DOM attribute
expect(locator).to_have_class()	Element has a class property
expect(locator).to_have_count()	List has exact number of children
expect(locator).to_have_css()	Element has CSS property
expect(locator).to_have_id()	Element has an ID
expect(locator).to_have_js_property()	Element has a JavaScript property
expect(locator).to_have_text()	Element matches text
expect(locator).to_have_value()	Input has a value
expect(locator).to_have_values()	Select has options selected
expect(page).to_have_title()	Page has a title
expect(page).to_have_url()	Page has a URL
expect(api_response).to_be_ok()	Response has an OK status

Rysunek 20 Lista asercji oferowana przez Playwright

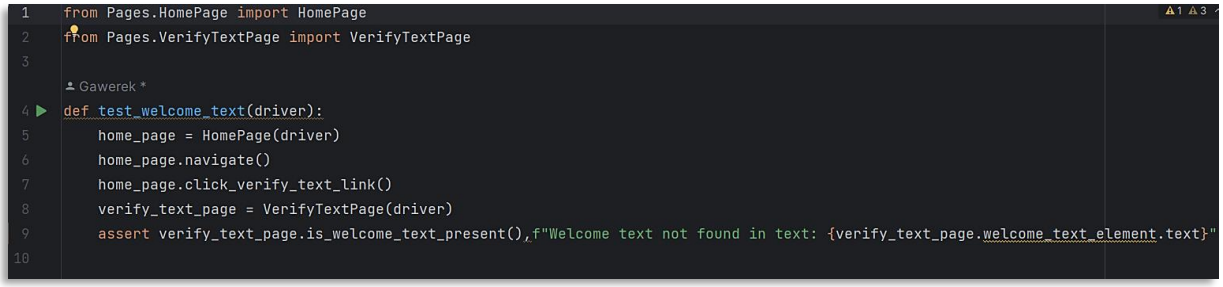
Logika użycia oraz implementacji asercji w Playwright jest prosta i przyjemna. Wspomniana biblioteka stanowi dużą przewagę w porównaniu do Selenium, gdyż to narzędzie nie posiada wbudowanej biblioteki asercji. Niemniej jednak asercje nie są niemożliwe w Selenium, zamiast tego korzysta z asercji dostępnych w Pythonie, takich jak **assert**. Asercje w Selenium są punktami kontrolnymi lub walidacjami dla aplikacji. Stanowią one pewne stwierdzenie, że zachowanie aplikacji działa zgodnie z oczekiwaniami. Asercje w Selenium walidują automatyczne przypadki testowe, które pomagają testerom zrozumieć, czy testy zakończyły się sukcesem czy nie. Na rysunku przedstawiono ten sam przypadek testowy, jednak w tym przykładzie użyto narzędzia Selenium.

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

2 usages  Gawerek
class VerifyTextPage:
    locator = (By.XPATH, "//span[normalize-space(.)='Welcome UserName!']")
    Gawerek
    def __init__(self, driver):
        self.driver = driver

    1 usage  Gawerek
    def is_welcome_text_present(self):
        welcome_text_element = WebDriverWait(self.driver, 10).until(
            EC.presence_of_element_located(self.locator))
        print(f"Element text: {welcome_text_element.text}")  # added this line to print the text
        return 'Welcome' in welcome_text_element.text and 'UserName' in welcome_text_element.text
```

Rysunek 21 VerifyTextPage Selenium - opracowanie własne

Zaprezentowany kod w klasie VerifyTextPage nie posiada asercji, ponieważ na tym etapie kluczowe jest by utowrzyć logikę dla danej walidacji. Logikę zakodowano przy użyciu języka python, kod zwraca wartość true gdy podany teksty zawierają się we wskazanym elemencie.Zbudowanie odpowiedniej logiki walidacyjnej jest kluczowe do użycia asercji w przypadku testowym skryptu testu automatycznego. Następnie, aby stworzona walidacja działała wymagane jest zaimplementowanie odpowiednio utworzonej walidacji do przypadku te-

stowego. Kontynuując powyższy przykład implementacja dla zakodowanej logiki walidacyjnej w narzędziu Selenium prezentuje się jak na Rysunku 19.



Rysunek 22 Implementacja asercji Selenium test_welcome_text.py - opracowanie własne

Dla frameworka Selenium niezbędne jest użycie słowa kluczowego **assert** z języka Python, który to odpowiada za powodzenie danego rezultatu lub nie w zależności od zwracanej wartości dla danej logiki. W tym przypadku testowym pozytywna walidacja następuję, gdy zwracana wartość będzie True. W przeciwnym przypadku test zostanie niezaliczony.

Posdumowanie

Wnioskując, biblioteka asercji Playwright oferuje różnorodne funkcje, które ułatwiają tworzenie testów. Dzięki funkcji expect, testy mogą być tworzone w prosty i intuicyjny sposób. Playwright oferuje również klasę PageAssertions, która umożliwia tworzenie asercji dotyczących stanu strony. Wszystko to przyspiesza proces tworzenia testów i nie wymaga zaawansowanych umiejętności.

Z drugiej strony, Selenium nie posiada wbudowanej biblioteki asercji. Zamiast tego, korzysta z asercji dostępnych w Pythonie, takich jak assert. Asercje w Selenium są punktami kontrolnymi lub walidacjami dla aplikacji, które pomagają testerom zrozumieć, czy testy zakończyły się sukcesem czy nie.

Podsumowując, Playwright oferuje bardziej rozbudowane i łatwe w użyciu narzędzia do tworzenia asercji w porównaniu do Selenium. Jednak Selenium, mimo braku wbudowanej biblioteki asercji, nadal oferuje możliwość tworzenia skutecznych testów za pomocą dostępnych w Pythonie asercji. Wybór między tymi dwoma narzędziami zależy od specyficznych potrzeb i umiejętności programisty.

Porównanie strategii obsługi przeglądarek

Playwright i Selenium komunikują się z przeglądarką na różne sposoby, co ma wpływ na ich funkcjonalność i wydajność.

Playwright komunikuje się z przeglądarką bezpośrednio, co oznacza, że nie ma potrzeby korzystania ze sterowników przeglądarki. Po zainstalowaniu pakietu Playwright Python, musimy pobrać i zainstalować binaria przeglądarki, z którymi Playwright ma pracować. Domyślnie, Playwright pobiera binaria dla Chromium, Firefox i WebKit z Microsoft CDN, ale to zachowanie jest konfigurowalne.

Playwright obsługuje Chromium, Firefox i WebKit, co umożliwia testowanie na różnych przeglądarkach za pomocą jednego narzędzia. Playwright ma wbudowane wsparcie dla wielu nowoczesnych funkcji przeglądarki, takich jak obsługa wielu kart, ramek i innych.

Playwright jest biblioteką Node.js do automatyzacji przeglądarek (Chromium, Firefox, WebKit) z jednym API, które dostarcza również interfejsy do obsługi innych języków, w tym Pythona. W porównaniu do innych bibliotek automatyzacji, takich jak Selenium, Playwright oferuje:

- **Szybkość**: Playwright jest generalnie szybszy od Selenium, ponieważ komunikuje się bezpośrednio z przeglądarką, podczas gdy Selenium komunikuje się za pośrednictwem sterowników przeglądarki.
- **Więcej funkcji**: Playwright obsługuje więcej funkcji przeglądarki, takich jak obsługa wielu kart, ramek i innych, które mogą wymagać więcej kodu do ich realizacji w Selenium.

Selenium korzysta z WebDrivera do sterowania przeglądarką. WebDriver działa z przeglądarką natywnie, tak jak użytkownik, lokalnie lub na zdalnej maszynie za pomocą serwera Selenium. Każda przeglądarka wymaga swojego specyficznego sterownika, na przykład Firefox wymaga sterownika GeckoDriver.

Selenium obsługuje wiele przeglądarek, ale wymaga osobnych sterowników dla każdej przeglądarki. Selenium również obsługuje wiele nowoczesnych funkcji przeglądarki, ale może wymagać więcej kodu do ich realizacji.

Selenium jest starszym narzędziem z dużą społecznością i obszerną dokumentacją. Jest to narzędzie sprawdzone i niezawodne, które jest szeroko stosowane w przemyśle do testowania aplikacji internetowych.

Playwright komunikuje się bezpośrednio z przeglądarką, co eliminuje potrzebę korzystania ze sterowników przeglądarki. Dzięki temu jest szybszy i obsługuje więcej funkcji przeglądarki bez potrzeby pisania dodatkowego kodu. Playwright obsługuje Chromium, Firefox i WebKit, co umożliwia testowanie na różnych przeglądarkach za pomocą jednego narzędzia.

Z drugiej strony, Selenium korzysta z WebDrivera do sterowania przeglądarką. Każda przeglądarka wymaga swojego specyficznego sterownika, co może wpływać na szybkość i wydajność. Selenium obsługuje wiele przeglądarek, ale może wymagać więcej kodu do realizacji niektórych funkcji przeglądarki.

Oba podejścia mają swoje zalety i wady, a wybór między nimi zależy od specyficznych wymagań projektu. Playwright może być lepszym wyborem dla projektów wymagających szybkości i obsługi nowoczesnych funkcji przeglądarki, podczas gdy Selenium może być lepszym wyborem dla projektów, które wymagają szerokiej kompatybilności z przeglądarkami i korzystają z doświadczenia społeczności Selenium.

Porównanie Społeczności: Selenium vs Playwright

Selenium

Wsparcie Społeczności: Społeczność Selenium jest jedną z największych i najbardziej zaangażowanych w świecie testowania oprogramowania. Dzięki swojej długiej historii i szerokiemu zastosowaniu, Selenium zyskało ogromną społeczność, która obejmuje programistów, testerów i entuzjastów na całym świecie. Istnieje wiele miejsc, w których użytkownicy mogą szukać pomocy, takich jak oficjalna strona wsparcia, Google Groups, oraz fora na stronach takich jak DEV Community, Stack Overflow i wiele innych. Wolontariusze często spędzają swój czas, próbując pomóc innym użytkownikom, co świadczy o silnym duchu współpracy w społeczności Selenium.

Dokumentacja: Selenium oferuje obszerną dokumentację, która obejmuje różne języki programowania i technologie, z którymi jest zgodny. Dokumentacja jest nie tylko bogata w treść, ale także dobrze zorganizowana i łatwa do zrozumienia, co ułatwia nowym użytkownikom naukę i zrozumienie narzędzia.

Rozwój Społeczności: Selenium ma długą historię i jest mocno zakorzenione w branży IT. Jest to widoczne w liczbie dyskusji, artykułów, tutoriali i narzędzi stworzonych przez społeczność. Ogromna społeczność Selenium jest jednym z kluczowych czynników, które uczyniły go standardem w dziedzinie automatyzacji testów przeglądarek. Wielu ekspertów i liderów branży aktywnie uczestniczy w społeczności, prowadząc webinary, pisząc blogi i udzielając się na konferencjach.

Playwright

Wsparcie Społeczności: Społeczność Playwright, choć stosunkowo nowa, szybko rośnie. Na oficjalnej stronie społeczności można znaleźć filmy z konferencji, transmisje na żywo i inne materiały. Istnieje również aktywna społeczność na GitHub, gdzie można znaleźć tutoriale, narzędzia i inne zasoby. Dodatkowo, Playwright ma oficjalny kanał na Discordzie, gdzie użytkownicy mogą na bieżąco komunikować się z deweloperami i innymi członkami społeczności. Ta nowoczesna platforma komunikacji świadczy o innowacyjnym podejściu Playwright do budowania społeczności.

Dokumentacja: Playwright oferuje dobrze zorganizowaną i łatwą do zrozumienia dokumentację, która jest dostosowana do nowoczesnych technologii i praktyk. Dokumentacja jest nie tylko szczegółowa, ale także atrakcyjna wizualnie, z licznymi przykładami i demonstracjami, które ułatwiają naukę.

Rozwój Społeczności: Playwright, będąc nowszym narzędziem, ma mniejszą, ale dynamicznie rozwijającą się społeczność. Istnieją ambasadorzy Playwright, którzy tworzą i dzielą się treściami związanych z Playwright, oraz liczne dyskusje na GitHub, gdzie użytkownicy mogą zgłaszać problemy i proponować nowe funkcje. Ta aktywność świadczy o zdrowym i żywotnym ekosystemie, który jest otwarty na innowacje i współpracę.

Podsumowanie

Porównanie społeczności Selenium i Playwright ukazuje różnice w dojrzałości, zaangażowaniu i dostępnych zasobach. Selenium, mając dłuższą historię, ma bardziej rozbudowaną i ak-

tywną społeczność, która jest często uważana za jedną z największych w dziedzinie testowania oprogramowania. Playwright, choć nowszy, szybko zyskuje na popularności i oferuje nowoczesne podejście do wsparcia społeczności, w tym unikalny kanał na Discordzie.

Wybór między tymi dwoma narzędziami może zależeć od indywidualnych preferencji, takich jak potrzeba wsparcia dla konkretnych technologii, styl komunikacji społeczności, czy dostępność nowoczesnych zasobów i narzędzi. Zarówno Selenium, jak i Playwright oferują solidne wsparcie społeczności, ale w różny sposób, co może wpływać na decyzję o wyborze narzędzia do konkretnego projektu.

Warto również zauważyć, że oba narzędzia mają swoje unikalne cechy i mocne strony, które mogą przyciągać różne grupy użytkowników. Selenium, z jego bogatą historią i szerokim wsparciem, może być bardziej atrakcyjne dla tych, którzy szukają sprawdzonego i niezawodnego narzędzia. Playwright, z jego nowoczesnym podejściem i innowacyjnymi

Porównanie Wydajności Selenium i Playwright

Wprowadzenie:

W erze cyfrowej, gdzie rynek oprogramowania jest nasycony i konkurencyjny, wydajność i jakość stają się kluczowymi wskaźnikami sukcesu. Automatyzacja testów odgrywa kluczową rolę w zapewnianiu jakości oprogramowania, a narzędzia używane do tego celu muszą być nie tylko skuteczne, ale także wydajne. W tym kontekście, porównanie wydajności Selenium i Playwright staje się niezwykle istotne, aby pomóc zespołom developerskim w podejmowaniu świadomych decyzji.

Metodologia:

Dla celów tego badania, przetestowano 10 identycznych przypadków testowych w obu frameworkach. Testy zostały napisane w języku Python, z wykorzystaniem wzorca Page Object Model (POM) i uruchomione za pomocą pytest. Środowisko testowe było spójne dla obu narzędzi, co gwarantuje obiektywność wyników. Wszystkie testy były uruchamiane w identycznych warunkach, na tej samej maszynie i w tej samej sieci, aby zapewnić spójność wyników.

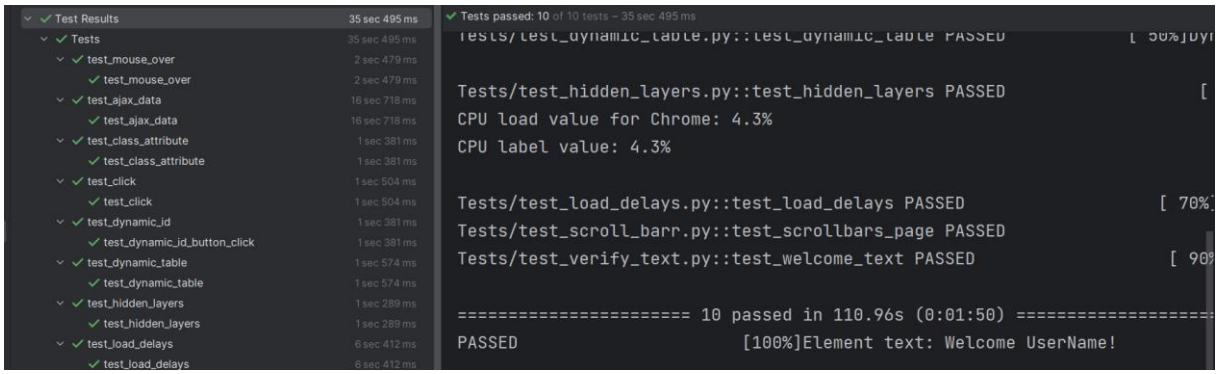
Wyniki:

Selenium:

Całkowity czas wykonania (od uruchomienia pytesta do końca): 110.96s

Czas poświęcony wyłącznie na testy: 35sec 495ms

Średni czas wykonania przypadku testowego: 3.549s



Playwright:

Całkowity czas wykonania (od uruchomienia pytesta do końca): 37.37s

Czas poświęcony wyłącznie na testy: 32sec 596ms

Średni czas wykonania przypadku testowego: 3.259s

Analiza:

Z wyników jasno wynika, że Playwright okazał się być nieco szybszy niż Selenium w kontekście samego czasu wykonania testów. Jednak różnica nie jest drastyczna, co wskazuje, że oba narzędzia są dość wydajne. Warto tutaj zaznaczyć, że przypadki testowe nie były istotnie skomplikowane, więc stąd może wynikać niewielka przewaga. Jednakże, Playwright dominuje w całościowym czasie poświęconym na uruchomienie testów. Przez swoje wbudowane narzędzia do obsługi przeglądarek, uruchomienie środowiska do wykonywania testów trwa znacznie mniej. Jest to spora zaleta w momencie, gdy plan testów obejmuje wiele testów.

Architektura komunikacji:

Selenium komunikuje się z przeglądarką za pomocą protokołów HTTP poprzez WebDriver. Ten sposób komunikacji może wprowadzać pewne opóźnienia, zwłaszcza gdy jest wiele żądań i odpowiedzi między testem a przeglądarką.

Z kolei Playwright korzysta z WebSocket do komunikacji z przeglądarką. WebSocket umożliwia ciągłą dwukierunkową komunikację między testem a przeglądarką, co może przyspieszyć wykonywanie testów.

Wnioski:

Chociaż Playwright okazał się być nieco szybszy w tym konkretnym badaniu, różnica w czasie wykonania nie jest znacząca. Wybór między Selenium a Playwright powinien być oparty nie tylko na szybkości, ale także na innych czynnikach, takich jak wsparcie społeczności, kompatybilność z różnymi technologiami i łatwość użycia.

Dla zespołów, które priorytetowo traktują szybkość wykonania testów, różnica w czasie wykonania między oboma narzędziami może być kluczowa. Jednakże ważne jest, aby zwrócić uwagę na specyfikę testowanej aplikacji oraz środowisko testowe, które mogą wpłynąć na wyniki.

Zaleca się przeprowadzenie dalszych badań w różnych środowiskach i z różnymi przypadkami testowymi, aby uzyskać pełniejszy obraz możliwości obu narzędzi. Ponadto, warto również rozważyć inne aspekty, takie jak koszty, dostępność zasobów i wsparcie techniczne przy wyborze odpowiedniego narzędzia do automatyzacji testów.

Spis rysunków

Spis tabel

Bibliografia

Cem, K., Jack, F. i Hunga Q., N. (1999). Testing Computer Software.

Effective Methods for Software Testing, Third Edition. (2006). W W. E. Perry.

Elfriede Dustin, J. R. (1999). Automated Software Testing: Introduction, Management, and Performance.

Experiences of Test Automation: Case Studies of Software Test Automation. (2010). W M. F. Graham.

Fewster, M. i Graham, D. (1999). Software Test Automation. Addison-Wesley Professional.

Graham, M. F. (2010). Experiences of Test Automation: Case Studies of Software Test Automation.

Grahama, M. F. (1999). Software Test Automation.

Kumar, M. (2016). The Impacts of Test Automation on Software's Cost, Quality and Time to Market.

Lucca, G. A. i Fasolino, A. R. (2006). Testing Web-based applications: The state of the art and future trends.