**WYŻSZA SZKOŁA INFORMATYKI I ZARZĄDZANIA**

**„COPERNICUS” WE WROCŁAWIU**

**WYDZIAŁ INFORMATYKI, ADMINISTRACJI I FIZJOTERAPII**

Kierunek: **Informatyka (INF)**

Specjalność: **Inżynieria Systemów Informatycznych (ISI)**

**PRACA DYPLOMOWA INŻYNIERSKA**

Marta Korzeniowska

**Projekt i wykonanie testów kluczowych funkcjonalności wybranej aplikacji webowej za pomocą Selenium WebDriver**

**Project and execution of automated tests for key functionalities in a selected web application using Selenium WebDriver**

**Ocena pracy**:………….…………....……………………………………

(ocena pracy dyplomowej, data, podpis promotora)

…..……………………………………………

(pieczątka uczelni)

**Promotor:**

**mg inż. Marek Michalski**

**WROCŁAW 2022**

**Spis treści**

1. **Wstęp** 3
   1. Motywacja 3
   2. Cel i zakres pracy 3
   3. Struktura pracy 4
2. **Specyfikacja wymagań 4**
   1. Opis rzeczywistości 4
   2. Słownik pojęć 5
   3. Wymagania funkcjonalne 6
   4. Scenariusze testowe 7
   5. Plan testów 17
3. **Zastosowane technologie 25**
   1. Platforma i środowisko 25
   2. Selenium WebDriver 25
   3. Python 29
   4. Pytest 30
   5. Narzędzia programistyczne 31
   6. Uzasadnienie wyboru technologii 31
4. **Projekt testów 3**2
   1. Przygotowanie środowiska 32
   2. Struktura projektu….……………………………..………….………………33
   3. Page Object Model 36
   4. Napotkane problemy 42
   5. Wybrane rozwiązania programistyczne 49
   6. Uruchamianie testów 53
   7. Raportowanie 55
5. **Zakończenie** 56
   1. Rozwój testów 56
   2. Wnioski 57
6. **Bibliografia** 57
7. **Dodatkowe elementy pracy** 58
8. **Wstęp**
   1. **Motywacja**

Obecnie e-commerce jest jednym z najbardziej dynamicznie rozwijających się rynków. W dobie pandemii i globalnego lockdownu konsumenci wybierali internetowe wersje sklepów w miejsce ich stacjonarnych odpowiedników. Część z nich przełamała swoje obawy niejako z przymusu w czasie kiedy dostęp do sklepów stacjonarnych był mocno ograniczony, natomiast część miała okazję tylko ugruntować swoje pozytywne nastawienie do zakupów online. Wszystko to złożyło się na fakt, iż w 2020 roku światowy rynek e-commerce doświadczył ogromnej ekspansji. Według szacunków IBM pandemia przyspieszyła o ok.5 lat przechodzenie klientów ze strefy stacjonarnej do strefy online. Dla przykładu w Polsce sprzedaż dóbr online zwiększyła się o 35% w stosunku do roku poprzedniego (źródło: PwC „Strategie, które wygrywają. Liderzy e-commerce o rozwoju handlu cyfrowego”). Dzięki temu polski rynek e-commerce jest jednym z najbardziej dynamicznie rozwijających się w Europie. Konsumenci cenią sobie wygodę oraz prędkość zakupów, łatwość uzyskania informacji o produktach, metody płatności pozwalające na użycie dowolnych mobilnych urządzeń oraz bezproblemowość dostaw. Składa się to na rosnące zapotrzebowanie aplikacji webowych dla sklepów internetowych, a co za tym idzie również na projekty testów, oceniających poprawność działania takich aplikacji. Niniejsza praca skupia się na testowaniu ponieważ jest to bardzo ciekawe i niezwykle potrzebne zagadnienie. Niewłaściwie wyświetlona strona sklepu internetowego, niedziałający przycisk, problem z założeniem konta czy dodaniem produktu do koszyka, wszystko to może sprawić, że klient zniechęci się do zakupów, a to tylko niektóre z potencjalnych problemów jakie mogą wystąpić. Każdy błąd, niepoprawne bądź niezgodne z oczekiwaniem zachowanie elementów aplikacji webowej będzie podważać zaufanie jakim kupujący obdarza sprzedającego. Dlatego przetestowanie gotowej już aplikacji w realnym środowisku jest równie ważne co wcześniejsze testowanie jej poszczególnych elementów na etapie wytwarzania oprogramowania.

* 1. **Cel i zakres pracy**

Celem pracy jest stworzenie planu, opisu i projektu testów automatycznych aplikacji webowej My Store ([www.automationpractice.com](http://www.automationpractice.com/)). Skupia się ona na podejściu do testowania opartym o możliwe przykłady użycia aplikacji przez zwykłego użytkownika. Z tego względu plan testów zawiera scenariusze grupujące przypadki użycia konkretnych funkcjonalności poprzez GUI (Graphical User Interface) dzięki zastosowaniu Selenium Webdriver.

* 1. **Struktura pracy**

Praca została podzielona na 3 główne części. W części pierwszej (rozdział 2) znajduje się specyfikacja wymagań, opis rzeczywistości, wymagania funkcjonalne oraz scenariusze testowe wraz z planem testów. Część druga (rozdział 3) opisuje wybrane technologie ze szczególnym naciskiem położonym na Selenium WebDriver. Część trzecia (rozdział 4) stanowi opis struktury projektu testów wraz z wybranymi rozwiązaniami oraz napotkaną problematyką.

1. **Specyfikacja wymagań**

W tym rozdziale znajduje się opis rzeczywistości, spis wymagań funkcjonalnych wziętych pod uwagę przy projektowaniu testów oraz scenariusze testowe wraz z planem testów.

* 1. **Opis rzeczywistości.**

My Store jest niezależną od systemu operacyjnego responsywną aplikacją webową sklepu internetowego stworzoną przez Selenium Framework umożliwiającą naukę i ćwiczenie testów automatycznych w realnym środowisku webowym. Posiada wszystkie kluczowe funkcjonalności sklepu w tym m.in: podział produktów na kategorie, wyświetlanie karty produktu, zakładanie konta użytkownika, logowanie, dodawanie produktów do koszyka, aż po finalizację zakupów z wyłączeniem prawdziwych płatności. Stanowi zatem bezpieczne środowisko do nauki automatyzacji testów oraz wdrażania przykładowych scenariuszy testowych na wybranych przeglądarkach internetowych, odzwierciedlając przy tym realne zagadnienia i problemy. Z uwagi na ogólną dostępność twórcy My Store zastrzegli jedynie, by nie spamować strony harmonogramem testów uruchamianych w sposób ciągły. Dlatego w tej pracy nie korzystam z podobnego rozwiązania.

* 1. **Słownik pojęć**

Wybrane terminy testowe zostały zaczerpnięte ze „Słownika terminów testowych ISTQB®”:

**Plan testów (test plan):** Dokumentacja opisująca cele testowe do osiągnięcia oraz środki i harmonogram ich realizacji, zorganizowane tak, by koordynować czynności testowe.

**Przypadek testowy (test case):** Zestaw warunków wstępnych, danych wejściowych, akcji (w stosownych przypadkach), oczekiwanych rezultatów i warunków końcowych opracowany w oparciu o warunki testowe.

**Test (test):** Zestaw jednego lub więcej przypadków testowych.

**Test dymny (smoke test):** Zestaw testów obejmujący główną funkcjonalność modułu lub systemu sprawdzający, czy działa ona poprawnie przed rozpoczęciem planowanych testów.

**Testowanie funkcjonalne:** Testowanie wykonywane by ocenić czy moduł lub system spełnia wymagania funkcjonalne.

**Testowanie GUI (od ang. Graphical User Interface):** Testowanie wykonywane przez współdziałanie z testowanym oprogramowaniem poprzez graficzny interfejs użytkownika.

**Testowanie oparte na przypadkach użycia:** Czarnoskrzynkowa technika testowania, w której przypadki testowe są projektowane w celu wykonywania zachowań przypadków użycia.

**User agent:** aplikacja kliencka. Nagłówek zawierający tzw. *user agent string (UA String)* służy serwisom internetowym do rozpoznania typu programu klienckiego, również do budowania statystyk odwiedzin witryn WWW przez różne przeglądarki bądź [roboty](https://pl.wikipedia.org/wiki/Robot_internetowy).

**Użytkownik (user):** Użytkownik aplikacji webowej My Store. Niezalogowany użytkownik ma dostęp do strony, przeglądania produktów oraz dodawania ich do koszyka. Finalizacji zakupu wymaga zalogowania.

**Warunek wstępny (precondition):** Wymagany stan elementu testowego i jego środowiska przed wykonaniem przypadku testowego.

**Warunek wyjściowy (postcondition):** Oczekiwany stan elementu testowego i jego środowiska po zakończeniu wykonywania przypadku testowego.

* 1. **Wymagania funkcjonalne**

Wymagania funkcjonalne aplikacji webowej My Store wzięte pod uwagę przy projektowaniu testów.

**WF1**

Strona główna My Store powinna się otwierać w dowolnej przeglądarce internetowej. Wszystkie elementy strony muszą widoczne, a linki i przyciski klikalne.

**WF2**

Możliwość rejestracji konta przez użytkownika.

**WF3**

Użytkownik ma możliwość zalogowania się.

**WF4**

Zalogowany użytkownik ma możliwość wylogowania się.

**WF5**

Każdy użytkownik (zalogowany i niezalogowany) ma możliwość dodania wybranych przez siebie produktów do koszyka.

**WF6**

Każdy użytkownik (zalogowany i niezalogowany) może usunąć produkty ze swojego koszyka.

**WF7**

Każdy użytkownik musi mieć możliwość zakupu produktu.

**WF8**

Każdy użytkownik ma możliwość wysłania wiadomości poprzez formularz kontaktowy.

* 1. **Scenariusze testowe**

Scenariusze testowe zawierają zbiór przypadków testowych potrzebnych do sprawdzenia poprawności aplikacji webowej My Store w określonym zakresie. Każdy scenariusz jest zatem odzwierciedleniem dokładnie określonej funkcjonalności. Posiada cel wyrażony nieformalnym zdaniem, warunki wstępne niezbędne do wykonania testu, kroki wykonania oraz oczekiwane rezultaty, czyli warunki wyjściowe. Ponadto może zawierać czynności przygotowawcze oraz czynności końcowe.

**Scenariusz 1**

**Cel:** Ładowanie strony głównej <http://automationpractice.com/index.php>

**Warunki wstępne:** brak

| **Przypadki testowe #1** | | | | |
| --- | --- | --- | --- | --- |
| **Lp** | **Nazwa** | **Warunki wstępne** | **Kroki wykonania** | **Oczekiwane rezultaty** |
| 1 | Ładowanie strony głównej. | Adres strony:  <http://automationpractice.com/index.php> | 1. Otworzyć wybraną przeglądarkę.  2. Wpisać adres strony i zatwierdzić. | Strona załadowana ze wszystkimi elementami.  Sprawdzić kluczowe elementy: tytuł strony, podpis, menu, przyciski: contact us, sign in, zakładki: popular, bestsellers) |

Tabela 1: Przypadki testowe dla scenariusza nr 1

**Scenariusz 2**

**Cel:** Sprawdzenie funkcjonalności rejestrowania konta użytkownika.

**Warunki wstępne:** Adres e-mail i dane użytkownika do celów testowych.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Przypadki testowe #2** | | | | |
| **Lp.** | **Nazwa** | **Warunki wstępne** | **Kroki wykonania** | **Oczekiwane rezultaty** |
| 1 | Rejestracja konta użytkownika | 1.Adres strony:  <http://automationpractice.com/index.php>  2. Adres e-mail i dane użytkownika do celów testowych. | 1. Otworzyć stronę.  2. Kliknąć przycisk „Sign in” (prawy górny róg).  3. W polu „Create an account” wpisać poprawny adres e-mail.  4. Kliknąć przycisk „Create an account”.  5. Wypełnić formularz danymi testowymi.  6. Kliknąć przycisk „Register”. | 1. Otwiera się podstrona Authentication  2. Otwiera się podstrona Authentication z formularzem danych  3. Konto zostało utworzone. Otwiera się podstrona My Account. |
| 2 | Rejestracja konta użytkownika używanym adresem e-mail. | 1. Adres strony:  <http://automationpractice.com/index.php>  2. Adres e-mail zarejestrowanego użytkownika. | 1. Otworzyć stronę.  2. Kliknąć przycisk „Sign in” (prawy górny róg).  3. W polu „Create an account” wpisać e-mail adres.  4. Kliknąć przycisk „Create an account”. | Otwiera się podstrona Authentication.  Pojawia się komunikat: „An account using this email address has already been registered. Please enter a valid password or request a new one. „ |

Tabela 2: Przypadki testowe dla scenariusza nr 2

**Scenariusz 3**

**Cel:** Sprawdzenie funkcjonalności logowania się na stronę.

**Czynności przygotowujące:** Zarejestrowanie użytkownika w systemie (jednorazowo przed pierwszym uruchomieniem testów).

**Warunki wstępne:** Dane logowania dla użytkownika (adres e-mail i hasło).

**Czynności kończące:** Wylogowanie użytkownika z systemu.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Przypadki testowe #3** | | | | |
| **Lp.** | **Nazwa** | **Warunki wstępne** | **Kroki wykonania** | **Oczekiwane rezultaty** |
| 1 | Logowanie poprawnymi danymi. | 1. Adres strony:  <http://automationpractice.com/index.php?controller=authentication>  2. Poprawne dane logowania: adres e-mail i hasło. | 1. Otworzyć stronę.  2. Wpisz adres e-mail polu „Email address” formularza Already registered?  3. Wpisz hasło w polu „Password”.  4. Kliknij przycisk „Sign in” | Poprawne zalogowanie użytkownika.  Przejście do podstrony My Account |
| 2 | Logowanie niepoprawnym adresem e-mail. | 1. Adres strony:  <http://automationpractice.com/index.php?controller=authentication>  2. Dane logowania: niepoprawny adres e-mail i poprawne hasło. | 1. Otworzyć stronę.  2. Wpisać niepoprawny email.  3. Wpisać poprawne hasło | Pojawienie się komunikatu „There is 1 error. Authentication failed.” |

| **Przypadki testowe #3** | | | | |
| --- | --- | --- | --- | --- |
| **Lp** | **Nazwa** | **Warunki wstępne** | **Kroki wykonania** | **Oczekiwane rezultaty** |
| 3 | Logowanie niepoprawnym hasłem. | 1. Adres strony:  <http://automationpractice.com/index.php?controller=authentication>  2. Dane logowania: poprawny adres e-mail i niepoprawne hasło. | 1. Otworzyć stronę.  2. Wpisać email.  3. Wpisać niepoprawne hasło. | Pojawienie się komunikatu „There is 1 error. Authentication failed.” |
| 4 | Odzyskiwanie hasła do logowania | 1. Adres strony:  <http://automationpractice.com/index.php?controller=authentication>  2. Dane logowania: adres e-mail | 1. Otworzyć stronę.  2. Kliknąć link „Forgot your password?”.  3. Wpisać adres e-mail.  4. Kliknąć „Retrieve password”. | Przejście do podstrony „Forgot your password”  Wyświetla się komunikat: „A confirmation email has been sent to your address: {podany\_email}” |

Tabela 3: Przypadki testowe dla scenariusza nr 3

**Scenariusz 4**

**Cel:** Sprawdzenie funkcjonalności wylogowania użytkownika.

**Czynności przygotowujące:** Zarejestrowanie użytkownika w systemie (jednorazowo przed pierwszym uruchomieniem testów). Zalogowanie użytkownika.

**Warunki wstępne:** Dane logowania dla użytkownika (adres e-mail i hasło).

| **Przypadki testowe #4** | | | | |
| --- | --- | --- | --- | --- |
| **Lp** | **Nazwa** | **Warunki wstępne** | **Kroki wykonania** | **Oczekiwane rezultaty** |
| 1 | Wylogowanie użytkownika | 1. Adres strony:  <http://automationpractice.com/index.php?controller=authentication>  2. Zalogowany użytkownik | 1. Otworzyć stronę.  2. Kliknąć przycisk „Sign out” (prawy górny róg). | (Przed testem upewnić się, że użytkownik jest zalogowany.)  Wylogowanie użytkownika i powrót do strony logowania. |

Tabela 4: Przypadki testowe dla scenariusza nr 4

**Scenariusz 5**

**Cel:** Sprawdzenie funkcjonalności dodawania produktu do koszyka dla niezalogowanego użytkownika

**Warunki wstępne:** brak

| **Przypadki testowe #5** | | | | |
| --- | --- | --- | --- | --- |
| **Lp** | **Nazwa** | **Warunki wstępne** | **Kroki wykonania** | **Oczekiwane rezultaty** |
| 1 | Dodanie pierwszego produktu do koszyka (niezalogowany użytkownik) | Adres strony:  [http://automationpractice.com/index.phphttp://automationpractice.com/index.php?id\_category=3&controller=category](http://automationpractice.com/index.php?id_category=3&controller=category) | 1. Otworzyć stronę.  2. Wybrać pierwszy produkt z listy i kliknąć.  3. Wybrać ilość – 1 szt. (default)  4. Wybrać rozmiar z listy – S (default).  5. Wybrać kolor – default.  6. Kliknąć przycisk „Add to cart”.  7. Zamknąć okno („Continue shopping” lub „x”) | Pojawia się okno „Product successfully added to your shopping cart” z możliwościami: „Continue shopping” i „Proceed to Checkout”.  Po zamknięciu okna produkt znajduje się w koszyku. |
| 2 | Dodanie kolejnego produktu do koszyka (niezalogowany użytkownik) | Adres strony:  <http://automationpractice.com/index.phphttp://automationpractice.com/index.php?id_category=3&controller=category>  2. Przynajmniej jeden produkt w koszyku. | 1. Otworzyć stronę.  2. Wybrać drugi produkt z listy produktów i kliknąć.  3. Powtórzyć kroki 3-5 z przypadku nr 1.  4. Kliknąć przycisk „Add to cart”.  5. Zamknąć okno przez „Continue shopping” | (Upewnić się, że w koszyku znajduje się przynajmniej 1 produkt.)  Pojawia się okno „Product successfully added to your shopping cart” z możliwościami: „Continue shopping” i „Proceed to Checkout”.  Po zamknięciu okna produkt znajduje się w koszyku. |

Tabela 5: Przypadki testowe dla scenariusza nr 5

**Scenariusz 6**

**Cel:** Sprawdzenie funkcjonalności usuwania produktu z koszyka dla niezalogowanego użytkownika.

**Warunki wstępne:** Przynajmniej jeden produkt znajduje się w koszyku.

| **Przypadki testowe #6** | | | | |
| --- | --- | --- | --- | --- |
| **Lp** | **Nazwa** | **Warunki wstępne** | **Kroki wykonania** | **Oczekiwane rezultaty** |
| 1 | Usunięcie jednego produktu z koszyka (niezalogowany użytkownik) | Adres strony:  <http://automationpractice.com/index.php?controller=order>  1. Jeden produkt w koszyku. | 1. Otworzyć stronę koszyka.  2. Kliknąć ikonę kosza „Delete”. | Produkt został usunięty z koszyka.  Pojawia się komunikat „Your shopping card is empty”. |

Tabela 6: Przypadki testowe dla scenariusza nr 6

**Scenariusz 7**

**Cel:** Sprawdzenie funkcjonalności zakupu produktu

**Warunki wstępne:** Przynajmniej jeden produkt w koszyku. Dane logowania użytkownika (adres e-mail i hasło).

| **Przypadki testowe #7** | | | | |
| --- | --- | --- | --- | --- |
| **Lp** | **Nazwa** | **Warunki wstępne** | **Kroki wykonania** | **Oczekiwane rezultaty** |
| 1 | Zakup jednego produktu. | Adres strony:  <http://automationpractice.com/index.php?controller=order> | 1. Otworzyć stronę koszyka.  2. Kliknąć „Proceed checkout”.  3. Wpisać poprawne dane logowania.  4. Kliknąć „Sign in”  5. Zaznaczyć checkbox „Use the delivery address as the billing address.„  6. Kliknąć „Proceed to checkout”.  7. Zaznaczyć „Terms of service.”  8. Kliknąć: „Proceed to checkout”.  9. Wybrać „Pay by bank wire”  10. Kliknąć „I confirm my order”. | W koszyku jest przynajmniej jeden produkt.  Zamówienie zostaje złożone. Pojawia się informacja:  „Your order in My Store is complete”.  Koszyk jest pusty. |

Tabela 7: Przypadki testowe dla scenariusza nr 7

**Scenariusz 8**

**Cel:** Sprawdzenie funkcjonalności wysyłania wiadomości przez formularz kontaktowy.

**Warunki wstępne:** Dane testowe (adres e-mail).

| **Przypadki testowe #8** | | | | |
| --- | --- | --- | --- | --- |
| **Lp** | **Nazwa** | **Warunki wstępne** | **Kroki wykonania** | **Oczekiwane rezultaty** |
| 1 | Przejście ze strony głównej na stronę Contact us. | Adres strony:  <http://automationpractice.com/index.php> | 1. Otworzyć stronę.  2. Kliknąć przycisk „Contact us”. | Otwiera się podstrona z formularzem kontaktowym. |
| 2 | Wysyłanie wiadomości bez załącznika | Adres strony:  <http://automationpractice.com/index.php?controller=contact> | 1. Otworzyć stronę.  2. Wypełnić pola: subject heading, email address, order reference, message.  3. Kliknąć przycisk „Send” | Wiadomość została wysłana.  Pojawia się komunikat „Your message has been successfully send to our team. |
| 3 | Wysyłanie wiadomości bez tematu i załącznika | Adres strony:  <http://automationpractice.com/index.php?controller=contact> | 1. Otworzyć stronę.  2. Wypełnić pola: email address, order reference, message.  3. Kliknąć przycisk „Send”. | Wiadomość nie zostaje wysłana.  Pojawia się komunikat „Please select the subject from the list provided”. |
| 4 | Wysyłanie wiadomości bez adresu email | Adres strony:  <http://automationpractice.com/index.php?controller=contact> | 1. Otworzyć stronę.  2. Wypełnić pola: subject heading, order reference, message.  3. Kliknąć przycisk „Send”. | Wiadomość nie zostaje wysłana.  Pojawia się komunikat „Invalid email address”. |
| 5 | Wysyłanie pustej wiadomości | Adres strony:  <http://automationpractice.com/index.php?controller=contact> | 1. Otworzyć stronę.  2. Wypełnić pola: subject heading, email address, order reference.  3. Kliknąć przycisk „Send”. | Wiadomość nie zostaje wysłana.  Pojawia się komunikat „The message cannot be blank”. |

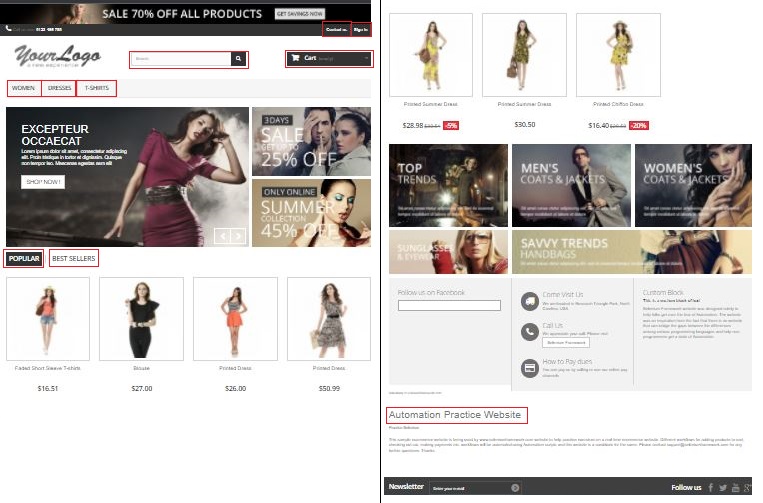
Tabela 8: Przypadki testowe dla scenariusza nr 8

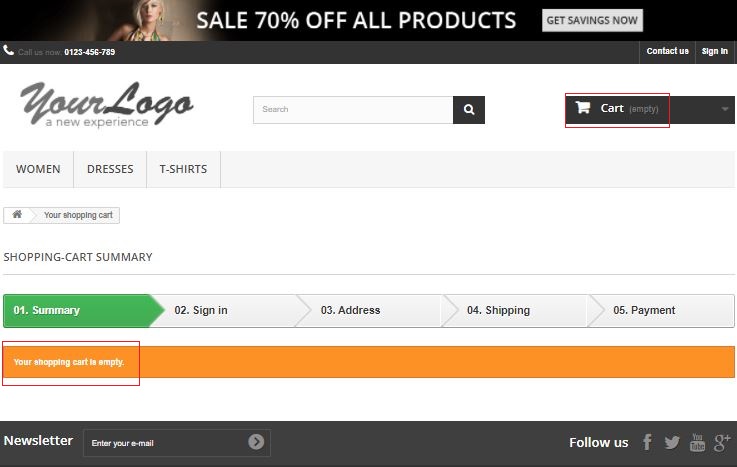
* 1. **Plan testów**

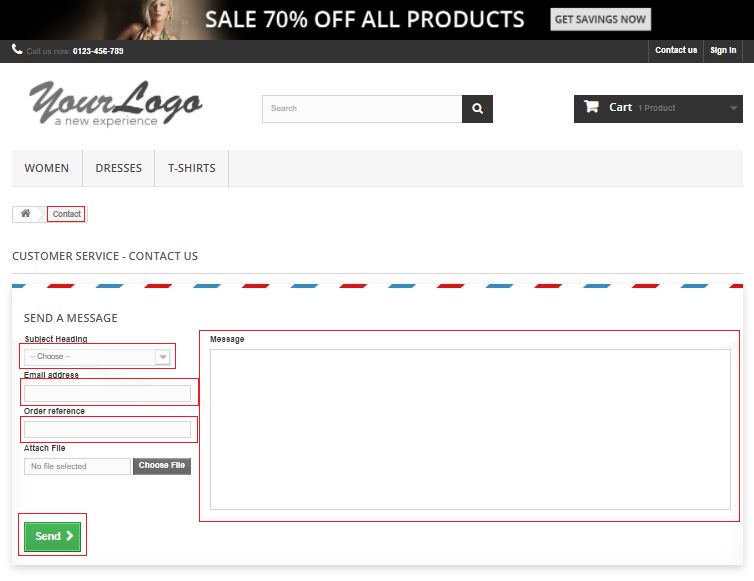
Podstawą do testów jest aplikacja webowa My Store. W jej obrębie na podstawie wyszczególnionych wymagań funkcjonalnych w części 2.3 niniejszej pracy testom podlegać będą następujące podstrony: strona główna (home page), logowania (authentication page), rejestrowania konta (registration page), kategoria women (women page) wraz z szybkim podglądem produktu oraz formularz kontaktowy (contact us page) i koszyk (cart page). Scenariusze testowe z części 2.4 zostały napisane w oparciu o przypadki użycia aplikacji przez zwykłego użytkownika. Numeracja scenariuszy odpowiada wymaganiom funkcjonalnym, w ten sposób scenariusz nr 1 sprawdza funkcjonalność WF1, scenariusz nr 2 – WF2, itd.

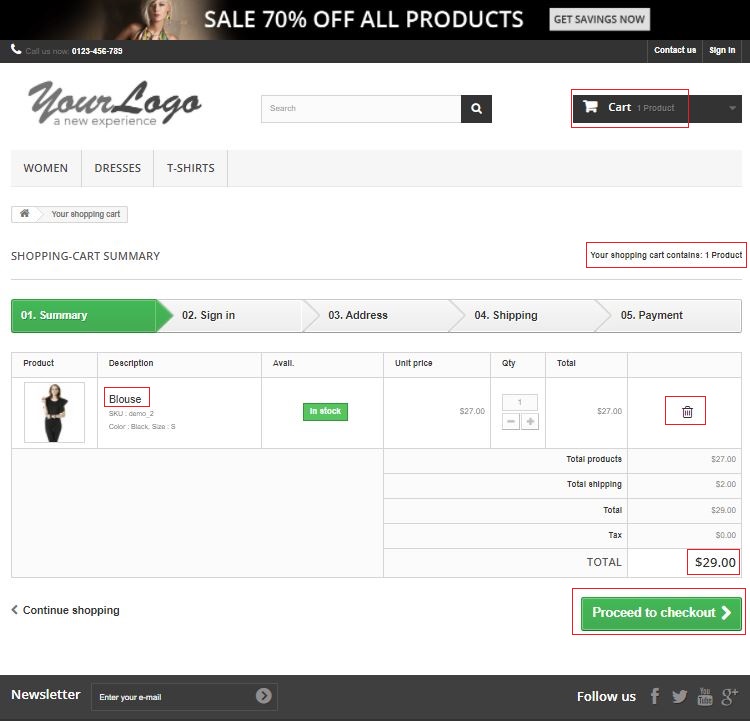
Testy zostaną przeprowadzone na trzech popularnych przeglądarkach: Google Chrome, Mozilla Firefox oraz Microsoft Edge.

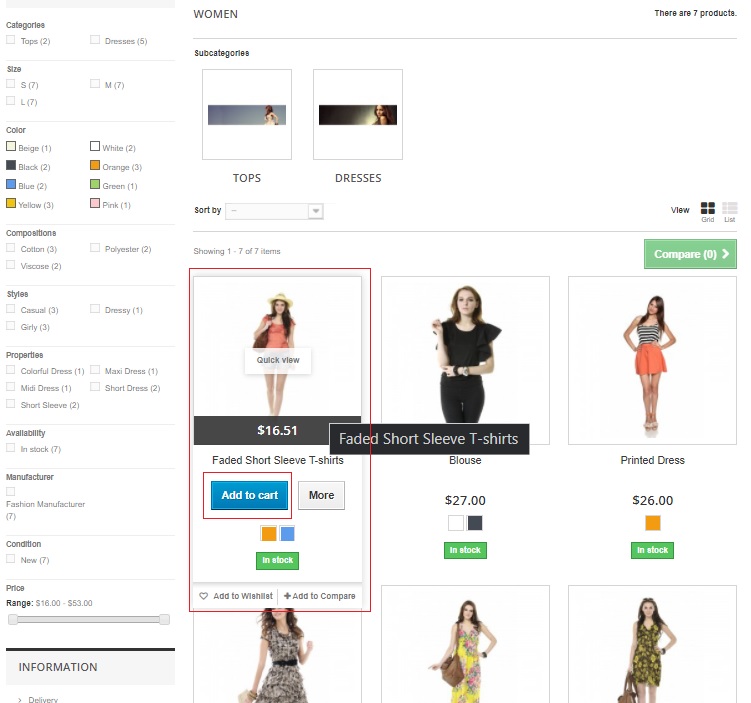
Poniżej znajdują się screenshot’y wyszczególnionych stron My Store. Elementy, których działanie zostanie poddane sprawdzeniu w trakcie testów zostały zaznaczone czerwoną ramką.

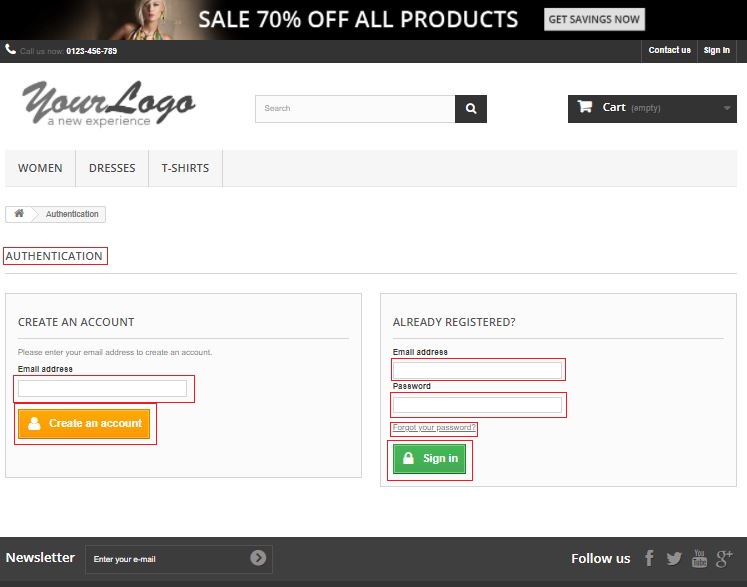
Rysunek 1: Strona główna

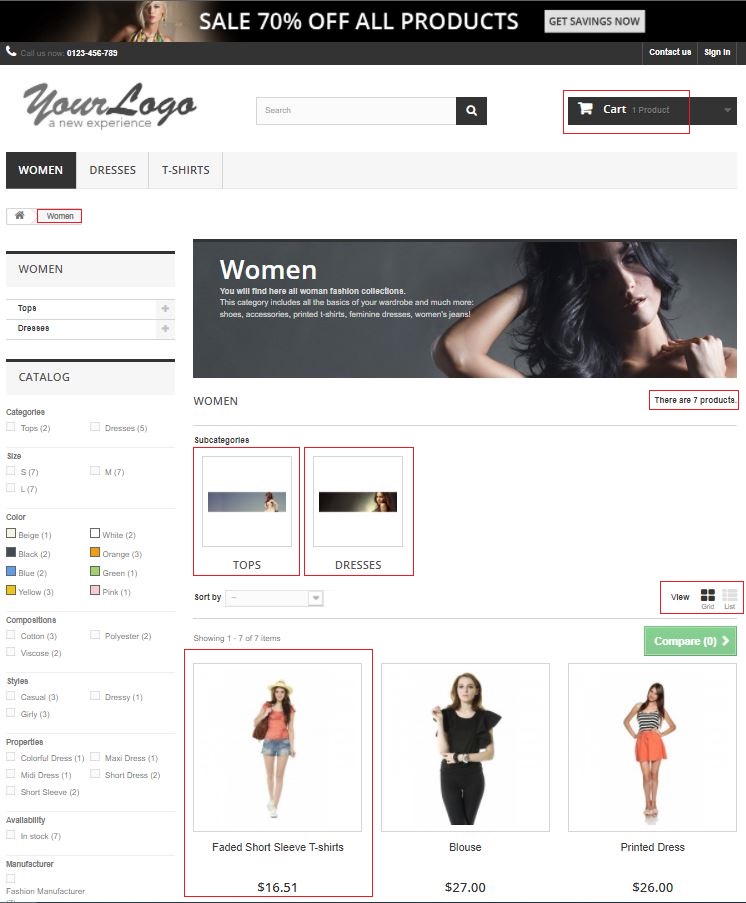
Rysunek 2: Strona pustego koszyka

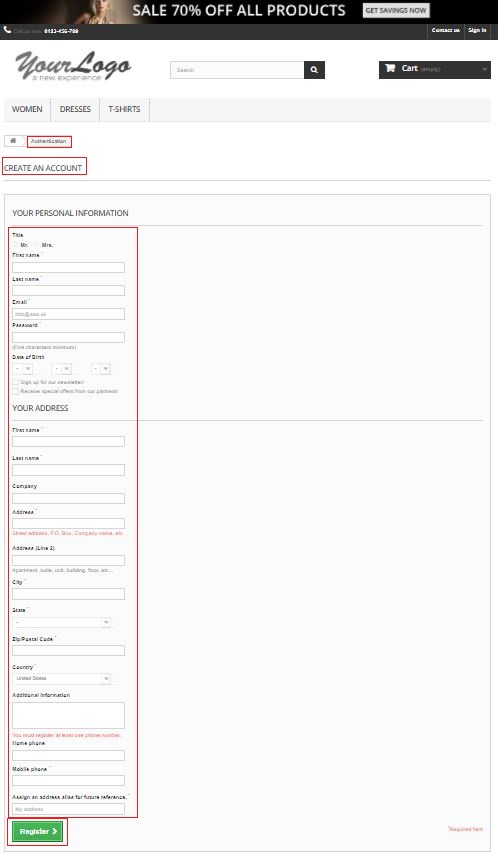
Rysunek 3: Strona z formularzem kontaktowym

Rysunek 4: Strona koszyka zawierający produkt

Rysunek 5: Szybki podgląd produktu

Rysunek 6: Strona logowania i tworzenia konta

Rysunek 7: Strona kategorii Women

Rysunek 8: Strona rejestracji konta

1. **Zastosowane technologie**

Niniejszy rozdział zawiera opis zastosowanych technologii, narzędzi, języków programowania oraz architektury wraz z krótkim opisem środowiska uruchomieniowego.

* 1. **Platforma i środowisko**

Projekt testów automatycznych jest niezależny od systemu operacyjnego. Napisany został w Pythonie z wykorzystaniem bibliotek i narzędzia Selenium Webdriver oraz frameworku Pytest. Jego uruchomienie możliwe jest na przeglądarkach: Chrome, Firefox, Edge pod warunkiem zainstalowania odrębnego webdrivera dla każdej z nich.

* 1. **Selenium WebDriver**

**Wprowadzenie**

Selenium jest zestawem narzędzi służącym do automatyzacji przeglądarek internetowych. Jest wykorzystywany głównie do testowania aplikacji webowych, ale umożliwia również realizację techniki screen scraping oraz nagrywanie i odtwarzanie powtarzających się zadań w oknie przeglądarki. Jest zupełnie darmowe i oparte na licencji open source, m.in. z tego względu cieszy się dużą popularnością oraz posiada bardzo duże wsparcie społeczności developerskiej. Selenium w szczególności dostarcza infrastrukturę dla specyfikacji W3C WebDriver - neutralnego platformowo i językowo interfejsu kodowania, kompatybilnego ze wszystkimi głównymi przeglądarkami internetowymi.

Główne komponenty Selenium to:   
**Selenium IDE** – zintegrowane środowisko developerskie. Instalowane jest jako dodatek do przeglądarek i służy głównie do testowania aplikacji webowych. Jest prostym w obsłudze narzędziem i może stanowić wstęp do automatyzacji testów. Pozwala na nagranie poszczególnych kroków pracy ze stroną w oknie przeglądarki, a następnie odtwarzanie ich w ramach testu. Dzięki temu zwalnia testera ze znajomości języków programowania, a nawet składni HTML i CSS.

**Selenium WebDriver** – zaawansowane narzędzie sterujące przeglądarką w sposób natywny, dokładnie tak, jak zrobiłby to użytkownik, lokalnie lub na zdalnej maszynie za pomocą serwera Selenium. Nazwa Selenium WebDriver odnosi się zarówno wsparcia językowego dla języków programowania jak i samej implementacji kodu sterującego przeglądarkami.

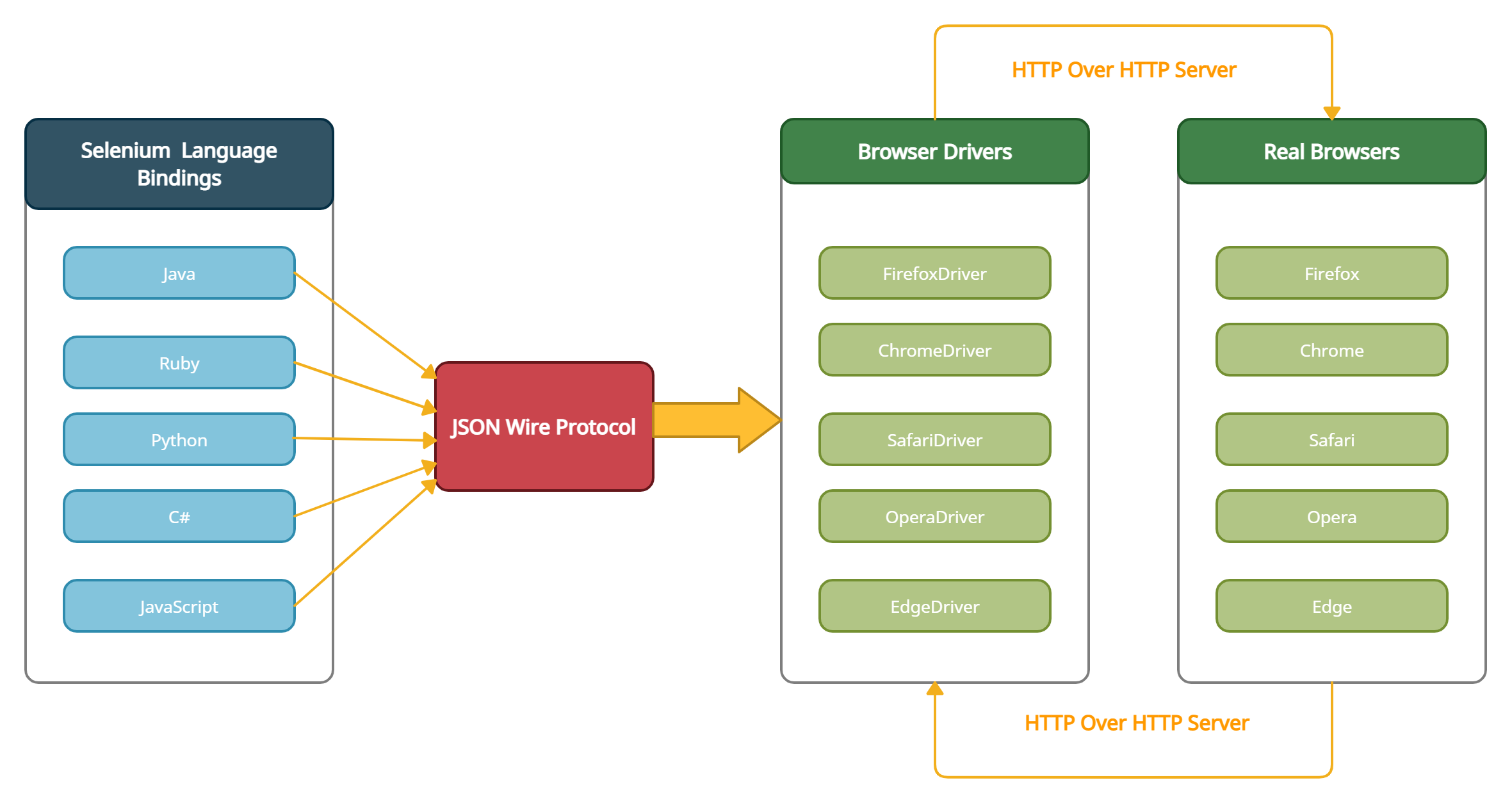
**Selenium Grid** – przenosi obsługę WebDrivera na kolejny poziom. Umożliwia zdalne sterowanie testami uruchamianymi jednocześnie na kilku różnych przeglądarkach.

**Architektura WebDriver**

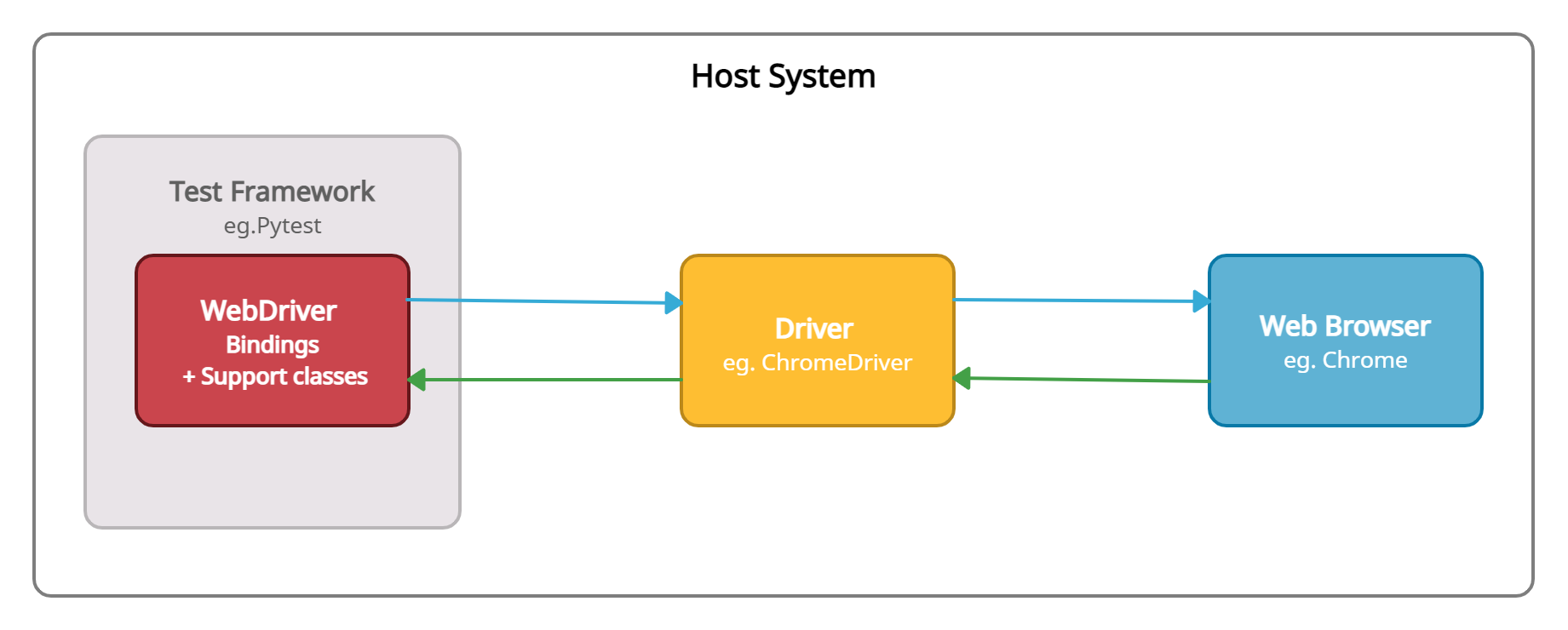
WebDriver jest interfejsem zdalnego sterowania, który umożliwia introspekcję i kontrolę aplikacji klienckich (user agents). Dostarcza neutralny dla platformy i języka protokół JSON Wire do zdalnego instruowania przeglądarek. Zapewnia zestaw interfejsów do odkrywania i manipulowania elementami DOM (Document Object Model) w dokumentach internetowych oraz do kontrolowania zachowania elementów aplikacji klienckich.

Architektura Selenium WebDriver składa się z czterech elementów:

1. Biblioteka Selenium dla języków oprogramowania.
2. Protokół JSON Wire.
3. Driver przeglądarki.
4. Przeglądarka internetowa.

Rysunek 9: Architektura Selenium WebDriver. W oparciu o © SoftwareTestingMaterial.

WebDriver ma tylko jedno zadanie – komunikacja z przeglądarką. Obojętnie czy lokalnie z poziomu tego samego systemu operacyjnego czy zdalnie z pomocą serwera Selenium czy Selenium Grid. Do celów testowych potrzebny jest jeszcze framework, który pozwoli na uruchamianie testów napisanych we współpracującym z Selenium językiem oprogramowania.



Rysunek 10: Komunikacja WebDrivera z przeglądarką. W oparciu o www.selenium.dev

**Wsparcie językowe**

Selenium wspiera wiele języków programowania, w szczególności:

1. Java
2. Ruby
3. Python
4. C#
5. JavaScript

**Protokół JSON Wire**

JSON oznacza JavaScript Object Notation. Protokół JSON Wire jest protokołem REST API używanym do przesyłania danych pomiędzy serwerami HTTP. Każdy driver (FirefoxDriver, ChromeDriver itd.) ma swój własny serwer HTTP.

**Driver**

Każda przeglądarka posiada oddzielny natywny sterownik - driver. Komunikuje się on z daną przeglądarką bez ujawniania jej wewnętrznej logiki funkcjonalności imitując działania zwykłego użytkownika. Kiedy driver otrzyma jakiekolwiek polecenie, zostanie ono wykonane w danej przeglądarce, a odpowiedź wróci w postaci odpowiedzi HTTP.

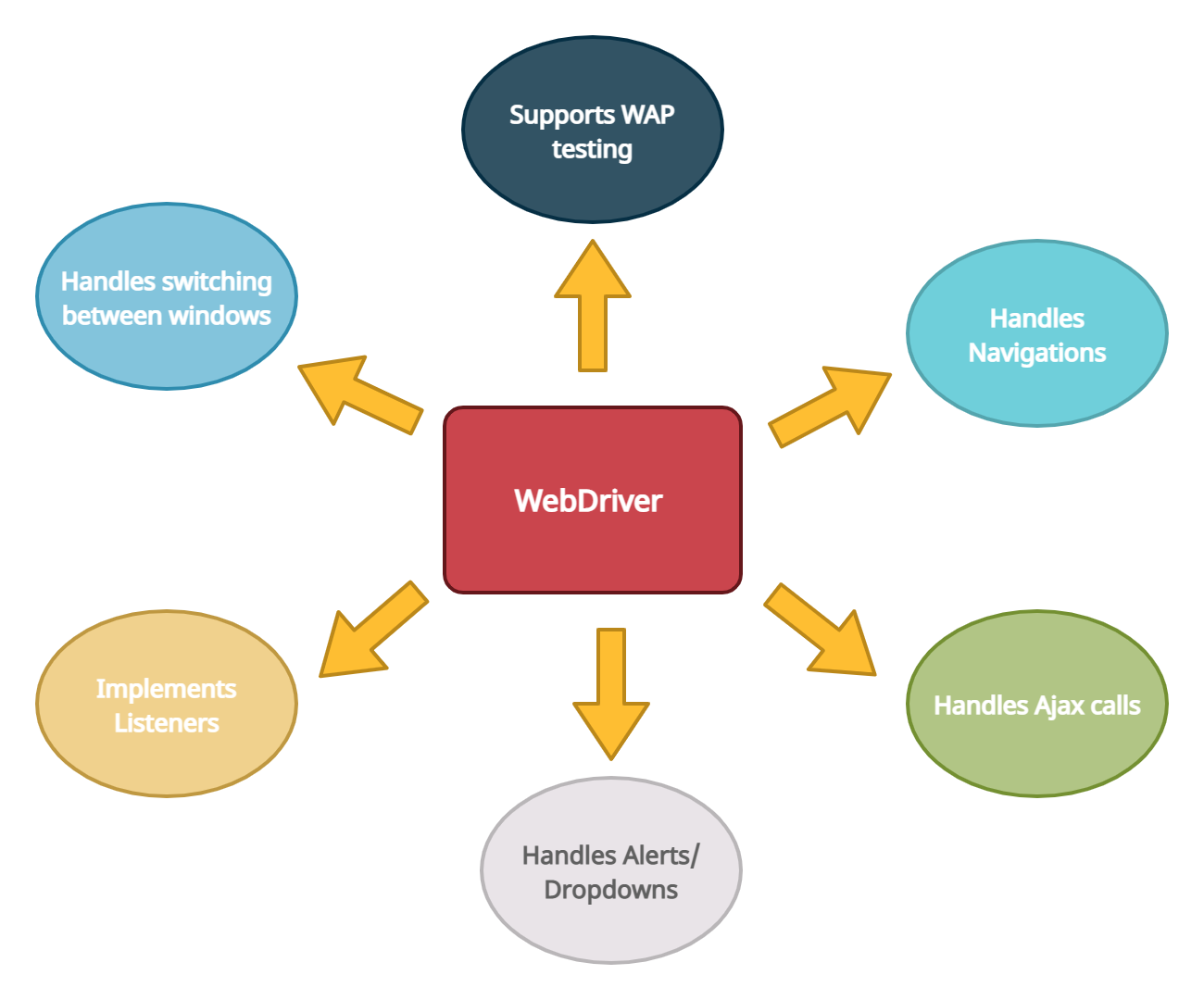
**Zgodność z przeglądarkami**

W 2018 r. WebDriver otrzymał rekomendację W3C - międzynarodowej społeczności pracującej nad opracowywaniem standardów webowych. Oznacza to, że WebDriver wspierany jest przez głównych dostawców przeglądarek (Mozilla, Google, Apple, Microsoft) i jest kompatybilny z takimi przeglądarkami jak:

1. Mozilla Firefox
2. Google Chrome/Chromium
3. Edge
4. Internet Explorer
5. Opera
6. AndroidDriver /IphoneDriver
7. HtmlUnit

**Rozwiązania**

Projektowanie testów automatycznych aplikacji webowych wiąże się z wieloma wyzwaniami. WebDriver oferuje szeroki wachlarz rozwiązań wychodząc naprzeciw potrzebom testerów. Pomaga radzić sobie ze złożonymi typami elementów internetowych jak dropdown’y, checkboxy i alerty za pomocą dynamicznego wyszukiwania. Pozwala na imitowanie ruchów kursorem myszy czy touchpada, przechodzenie między oknami, ramkami (iframe) i ogólną nawigację po stronach internetowych.

Rysunek 11: Metody Selenium WebDriver. W oparciu o © SoftwareTestingMaterial.

* 1. **Języki programowania**

**Python**

Python jest językiem programowania wysokiego poziomu opracowanym przez Guida van Rossuma i opublikowanym po raz pierwszy w 1991 r. Cechuje go łatwość użycia oraz duża skalowalność. Można z jego pomocą tworzyć zarówno niewielkie, szybkie aplikacje jak i bardzo rozbudowane komercyjne projekty. Python jest językiem obiektowym (z ang. *object-oriented programming*), ale w przeciwieństwie do takich języków jak C# czy Java, obiektowość w Pythonie nie jest obligatoryjna, a opcjonalna. Python może zostać zintegrowany z innymi językami, takimi jak C, C++ i Java. Oznacza to, że można wykorzystać silne strony innych języków, np. szybkość jaką mogą zaoferować C czy C++, nie tracąc przy tym łatwości tworzenia kodu, która jest charakterystyczną cechą programowania w Pythonie.

Programy pisane w Pythonie są niezależne od platformy, co sprawia że bez względu na system operacyjny maszyny, na jakiej powstawał kod będzie on działał na innym komputerze, na którym zainstalowano Pythona. Kolejną z zalet tego języka jest też bardzo rozbudowane wsparcie społeczności deweloperskiej oraz fakt, że jest on całkowicie darmowy z otwartym dostępem do kodu źródłowego.

* 1. **Pytest**

Pytest jest nowoczesnym, open sourcowym frameworkiem służącym do uruchamiania testów automatycznych w języku Python. Posiada bardzo dokładną dokumentację zawierającą liczne i szczegółowe przykłady użytkowania. Dzięki wykorzystaniu bibliotek Selenium można go z powodzeniem wykorzystywać do testowania usług webowych. Dopuszcza wiele elastycznych zastosowań dzięki podziałowi testów na zakresy (z ang. *scope*): sesja testowa, moduł testowy, klasa testowa, test. Sesja testowa jest zbiorem modułów zdefiniowanych jako testy, modułem zaś jest pojedynczy plik napisany w Pythonie. Bardzo ciekawą i użyteczną funkcjonalnością Pytestu są tzw. fikstury (z ang. *fixture*). Fikstura ma za zadanie przygotowanie środowiska testowego w określonym przez testera zakresie, a po zakończonym teście wykonuje czynności kończące. Ponadto pozwala na wstrzyknięcie zwracanego przez siebie obiektu do konkretnego testu. Więcej o wykorzystaniu fikstury omawiam w części praktycznej pracy.

Pytest rozpoznaje elementy kodu po słowie kluczowym *test*. Wszystkie pliki, klasy i funkcje, które posiadają w swojej nazwie taki prefiks, zostaną zgromadzone i przygotowane do uruchomienia. Istnieje możliwość zmiany takiego zachowania i wprowadzenie indywidualnych ustawień poprzez stworzenie pliku konfiguracyjnego *pytest.ini*. Jeśli chodzi o konfigurację to Pytest akceptuje kilka formatów plików m.in. wspomniany wyżej *pytest.ini* bądź *setup.cfg*. Ponadto dostarcza jeszcze jeden właściwy dla siebie: *conftest.py*. Plik ten pozwala np. na podanie ścieżki do plików z danymi, wyszczególnienie elementów konfiguracyjnych środowiska testowego (np. opcji przeglądarek), dodawanie nowych flag czy zmianę sposobu zbierania testów.

Rezultaty testów porównywane są z użyciem słowa kluczowego *assert.* Wynik kwalifikowany jest jako PASS lub FAIL. Pytest wprowadza też dwa dodatkowe wyniki: XFAIL oraz XPASS oznaczające spodziewaną porażkę lub sukces (z ang. *expected fail/expected pass*). W przeciwieństwie do innych frameworków (np. Unittestu) pominięcie testu nie oznacza pojawienia się wyjątku.

* 1. **Narzędzia programistyczne**

**PyCharm** to zintegrowane środowisko programistyczne stworzone przez firmę JetBrains dla języka programowania Python. Oprócz Pythona, PyCharm wspiera również JavaScript, TypeScript, SQL, HTML/CSS, AngularJS i inne. Pozwala na szybkie instalowanie dodatkowych modułów m.in. Pytest, Selenium. Jest prosty w obsłudze. Zapewnia łatwą nawigację w projekcie, inteligentne uzupełnianie kodu, sprawdzanie błędów oraz kontrolę jakości zgodną z PEP8 (Python Enhancement Proposal). W swojej pracy wykorzystałam darmową wersję PyCharm Community Edition.

**GIT** to system kontroli wersji pierwotnie stworzony przez Linusa Torvaldsa jako narzędzie pomagające w rozwoju jądra Linux. Wykorzystałam go do śledzenia i kontrolowania zmian wprowadzanych do projektu testów automatycznych. Lokalne repozytorium powstało bezpośrednio na moim komputerze, natomiast jego zdalna wersja na darmowym, hostingowym serwisie GitHub przeznaczonym specjalnie dla projektów programistycznych.

* 1. **Uzasadnienie wyboru technologii**

Wybór opisanych w tej pracy technologii podyktowany był przede wszystkim szeregiem oferowanych przez nie rozwiązań z jednoczesną łatwością ich zastosowania. Python jest obecnie bardzo popularnym językiem programowania. Trendy z kilku ostatnich lat pokazują, że zainteresowanie jakim się cieszy wśród deweloperów stale rośnie. Wg raportu „Developer Nation” biorącego pod uwagę 160 krajów opublikowanego przez SlashData wykorzystywany jest przez 11,3 mln programistów co plasuje go na drugim miejscu listy najpopularniejszych języków. Wydaje się być zatem słusznym pogląd, że zapoznanie się z nim pomoże nie tylko rozwiązać problemy mogące się pojawić podczas pisania kodu dla testów automatycznych, ale również w przyszłości będzie stanowić atut na rynku pracy.

Jednymi z najbardziej interesujących cech pytestu są jego fikstury, możliwość dodania plików konfiguracyjnych, parametryzacja testów oraz możliwość oznaczania testów za pomocą własnych lub wbudowanych dekoratorów. Z kolei Selenium WebDriver stanowił dość oczywisty wybór dla testów aplikacji webowej, ponieważ komunikacja drivera z przeglądarką przebiega w sposób szybki. Jednocześnie nie dzieje się to niejako „pod spodem” tylko odzwierciedla sposób postępowania użytkownika.

1. **Projekt testów**

**4.1 Przygotowanie środowiska**

Najnowszą wersję Pythona można pobrać i zainstalować z oficjalnej strony (https://www.python.org/downloads). Na potrzeby tej pracy była to wersja Python 3.9.6. Ważnym krokiem po udanej instalacji jest dodanie Pythona do ścieżki zmiennych środowiskowych. Dzięki temu komendy Pythona będą rozpoznawane z dowolnego poziomu konsoli. Ponieważ jest możliwe zainstalowanie kilku różnych wersji Pythona na jednym komputerze jednym z bardzo ważnych i przydatnych aspektów pracy z tym językiem jest przygotowanie odpowiedniego wirtualnego środowiska zawierającego moduły i biblioteki niezbędne do konkretnego projektu. Można je przygotować z poziomu zintegrowanego IDE, np. PyCharma albo poprzez terminal konsoli.

Z poziomu katalogu, w którym znajduje się zainstalowany Python wpisujemy komendę:

$ python -m venv %USERPROFILE%\nazwa\_katalogu\nazwa\_srodowiska

Aktywowanie wirtualnego środowiska następuje poprzez wpisanie komendy *activate*

z katalogu Scripts znajdującego się wewnątrz naszego nowo utworzonego środowiska. Po utworzeniu projektu w PyCharmie i przypisaniu mu odpowiedniego interpretera jego aktywacja następuje automatycznie.

Instalowanie modułu Pytest oraz bibliotek Selenium następuje już wewnątrz wirtualnego środowiska. W ten sposób zyskujemy pewność, że projekt będzie zawierał wszystkie niezbędne do poprawnego działania komponenty i nic ponadto.

$ pip install selenium

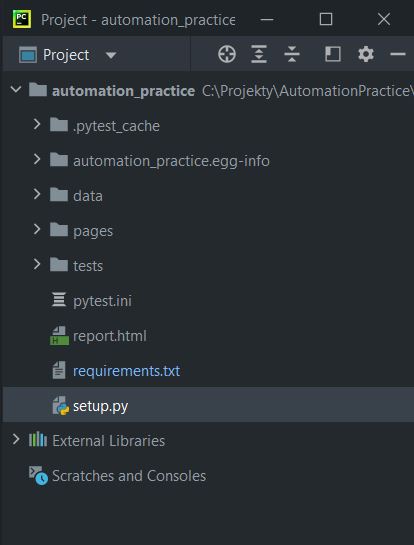
$ pip install pytest

Ciężko sobie wyobrazić proces testowania bez odpowiedniego zbierania raportów z przebiegu testów. Pytest wyświetla komunikaty w oknie terminala konsoli, jednak aby zgromadzić je w bardziej przystępnej formie należy zainstalować moduł, który będzie generował podsumowanie testów w formie czytelnego dokumentu HTML.

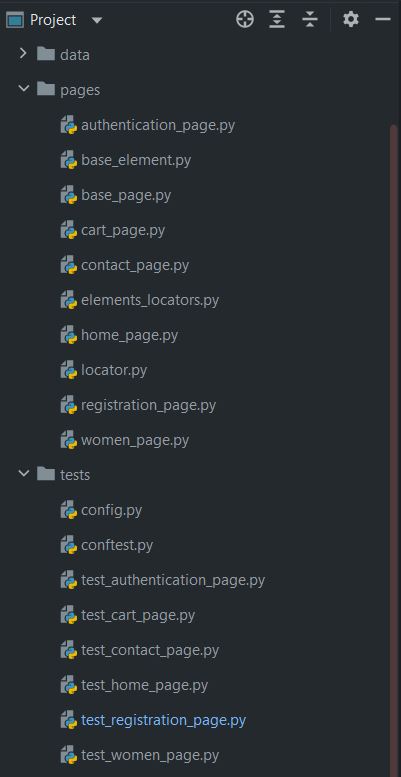
$ pip install pytest-html

**4.2 Struktura projektu**

Projekt testów automatycznych dla aplikacji My Store ma strukturę, którą obrazuje poniższy rysunek:

Rysunek 12: Struktura projektu

* Folder „data” zawiera pliki z danymi testowymi: *test\_login.json* i *test\_registration.json*.
* Folder „pages” zawiera klasę bazowej strony, klasę bazowych elementów oraz wszystkie klasy dla stron internetowych aplikacji My Store, które dziedziczą po stronie bazowej.
* Folder „tests” zawiera wszystkie klasy testów. Ich układ odpowiada klasom z folderu „pages”. Zapewnia to lepszą przejrzystość projektu, ponieważ każda klasa stron aplikacji ma odpowiadającą sobie klasę zawierającą testy.

Rysunek 13: Zawartość folderów "pages" i "tests"

* W folderze „tests” znajdują się również pliki konfiguracyjne dla środowiska uruchomieniowego testów: *config.py* i *conftest.py*.
* Plik pytest.ini jest jednym z plików konfiguracyjnych frameworka Pytest. Określam w nim sposób rozpoznawania i gromadzenia modułów, klas i funkcji testowych oraz jakie dekoratory testów są używane w projekcie. Zawartość pliku *pytest.ini*:

*[pytest]*  
*python\_files* = test\_\*  
*python\_classes* = \*Tests  
*python\_functions* = test\_\*  
  
*markers* =  
 browsertest: Check environment  
 smoke: All critical functions tests  
 homepage: Tests for home page  
 loginpage: Tests fot authentication page  
 womenpage: Tests for women page  
 contactpage: Tests for contact page  
 registrationpage: Tests for account registration page  
 *# scenarios:*  
signin: Tests for sign in functionality  
 signout: Test for sign out functionality  
 cart: Tests for shopping cart functionality  
 purchase\_product: Tests for buying a product  
 remove\_product: Tests for removing product from a cart

* Plik *report.html* zawiera raport z przebiegu testów. Plik ten generuje się automatycznie przy każdym uruchomieniu testów i za każdym razem nadpisuje poprzednią wersję.

**4.3 Page Object Model**

Page Object Model (Page Object Pattern) jest wzorcem projektowym testowania aplikacji internetowych, w którym strony są reprezentowane jako klasy z atrybutami lokalizatora i wszelkimi metodami interakcji. Obiekty klasy zawierają elementy i zachowania związane z daną stroną. Zamiast wywoływać surowe metody WebDrivera testy odwołują się do metod obiektu strony. Dzięki takiemu oddzieleniu testów od danych bazowych łatwiej jest pisać poszczególne przypadki testowe, a także wystarczy zmodyfikować kod w jednym miejscu, by zaktualizować wszystkie testy. Samo modelowanie stron wymaga poświęcenia dodatkowego czasu, ale wysiłek ten zwraca się bardzo szybko, gdy tylko przechodzi się do pisania zestawu metod testowych. Ustrukturyzowanie pracy w ten sposób zwiększa czytelność, ponieważ jasno pokazuje które elementy będą testowane w danym zakresie. Z tego względu POM jest najprawdopodobniej najczęściej używanym podejściem do automatyzacji testów webowych UI (z ang. *User Interface*). Nie istnieje jeden konkretny sposób implementacji tego wzorca, ale większość z nich jest dość podobna. Aby zminimalizować ilość duplikującego się kodu oraz poprawić jego utrzymywanie rozdziela się implementację na przynajmniej trzy sekcje:

* Lokalizatory
* Strony bazujące na testowanej aplikacji webowej
* Skrypty testowe odnoszące się do testowanej strony

**Base page**

Wykorzystałam to podejście w tym projekcie i utworzyłam klasę strony bazowej zawierającą podstawowe atrybuty i metody, które podlegają dziedziczeniu przez konkretne strony odpowiadające stronom aplikacji My Store.

*class* BasePage(object):  
  
 url = *None*  
  
 *def* \_\_init\_\_(*self*, driver):  
 *self*.driver = driver  
  
 *def* go(*self*):  
 *self*.driver.get(*self*.url)  
   
 @property  
 *def* title(*self*):  
 title = *self*.driver.title  
 *return* title  
  
 @property  
 *def* page\_source(*self*):  
 page\_source = *self*.driver.page\_source  
 *return* page\_source

Do atrybutów klasy należy *url* z początkową wartością ustawioną na *None.* Funkcja *def \_\_init\_\_(self)* jest w Pythonie odpowiednikiem konstruktora z podejścia obiektowego w C++. *K*ażda nowo powstała instancja posiada atrybut *self.driver* odnoszący się do WebDrivera. Metoda *def go()* oparta jest na wbudowanej metodzie Selenium *driver.get(url) i* pozwala na otwarcie strony w przeglądarce. Z kolei metody *def title()* i *def page\_source()* opatrzone są dekoratorem *@property*. Dekorator ten odnosi się do wbudowanej pythonowej funkcji *property()*, która używana jest do definiowania właściwości w klasie. Metoda *property()* zapewnia interfejs dla atrybutów instancji. Enkapsuluje je i udostępnia właściwość podobnie jak ma to miejsce w językach Java i C#. Metoda *property()* przyjmuje jako argumenty metody get, set i delete i zwraca obiekt klasy property. Zamiast metody *property()* zalecane jest używanie dekoratora.

W swoim kodzie używam *@property* ponieważ dzięki niemu mogę odnosić się do właściwości jak do atrybutu instancji. Przykład wykorzystania def title() dzięki użyciu @property:

@mark.browsertest  
*def* test\_browser(*self*, get\_home\_page):  
 home = get\_home\_page  
 *assert* home.title == 'My Store'

Jak widać na listingu powyżej instancja obiektu home\_page (dziedziczącego po klasie bazowej) odwołuje się do własności *title* jak do atrybutu, a nie jak do metody.

**Base element**

Oprócz klasy strony bazowej utworzyłam też klasę bazowego elementu będącą modelem podstawowego elementu webowego.

*from* selenium.webdriver.support.wait *import* WebDriverWait  
*from* selenium.webdriver.support *import* expected\_conditions *as* EC  
  
  
*class* BaseElement(object):  
 *def* \_\_init\_\_(*self*, driver, locator):  
 *self*.driver = driver  
 *self*.locator = locator  
  
 *self*.web\_element = *None*  
 *self*.find()  
  
 *def* find(*self*):  
 *""" Finding element on the page using visibility as expected conditions """*  
element = WebDriverWait(*self*.driver, 10).until(  
 EC.visibility\_of\_element\_located(locator=*self*.locator)  
 )  
 *self*.web\_element = element  
 *return None*  
  
 *def* input\_text(*self*, txt):  
 *""" Function to enter text into input fields """*  
 *self*.web\_element.clear()  
 *self*.web\_element.send\_keys(txt)  
 *return None*  
  
 *def* click(*self*):  
 *""" Function to click elements using expected conditions """*  
element = WebDriverWait(*self*.driver, 10).until(  
 EC.element\_to\_be\_clickable(*self*.locator)  
 )  
 element.click()  
 *return None*  
  
 *def* attribute(*self*, attr\_name):  
 *""" Function to get the attribute of an element """*  
attribute = *self*.web\_element.get\_attribute(attr\_name)  
 *return* attribute  
  
 @property  
 *def* text(*self*):  
 *""" Function to retrieve text from an element. Set as @property """*  
text = *self*.web\_element.text  
 *return* text

Na powyższym listingu uwagę może przyciągać metoda *def find()*. Wywoływanie metod w konstruktorze nie należy do częstych praktyk. Zdecydowałam się na to ze względu na powszechny przy automatyzacji testowania stron internetowych problem związany z lokalizowaniem elementów webowych. Omawiam to szerzej w części 4.4 Napotkane problemy.

**Elements Locators i Locator**

Lokatory zamieściłam w pliku *elements\_locators.py* jako atrybuty klas, gdzie każda klasa odpowiada konkretnej stronie aplikacji My Store.

*from* selenium.webdriver.common.by *import* By  
*from* .locator *import* Locator  
*# Locators for all key components used in testing*

*class* ContactPageLocators:  
  
 NAVIGATION\_TAB\_CSS = Locator(by=By.CSS\_SELECTOR, value='.navigation\_page')  
 NAVIGATION\_PAGE\_HEADING\_CSS = Locator(by=By.CSS\_SELECTOR, value='.page-heading.bottom-indent')  
 PAGE\_SUBHEADING\_TEXT = Locator(by=By.CSS\_SELECTOR, value='.page- subheading')  
 *# send message form :*  
SELECT\_SUBJECT\_HEADING\_ID = Locator(by=By.ID, value='id\_contact')  
 INPUT\_EMAIL\_ID = Locator(by=By.ID, value='email')  
 INPUT\_ORDER\_REFERENCE\_CSS = Locator(by=By.CSS\_SELECTOR, value='#id\_order')  
 INPUT\_FILE\_XPATH = Locator(by=By.XPATH, value='//\*[text()="No file selected"]')  
 CHOOSE\_FILE\_BUTTON\_XPATH = Locator(by=By.XPATH, value='//\*[text()="Choose File"]')  
 SEND\_BUTTON\_XPATH = Locator(by=By.XPATH, value='//\*[text()="Send"]')  
 MESSAGE\_TEXTAREA\_ID = Locator(by=By.ID, value='message')  
 *# alert text:*  
SUCCESS\_MESSAGE\_CSS = Locator(by=By.CSS\_SELECTOR, value='.alert.alert-success')  
 FAILURE\_MESSAGE\_CSS = Locator(by=By.CSS\_SELECTOR, value='div[class="alert alert-danger"] ol li')

Przyjmują one formę obiektu Locatora. Locator jest utworzoną przeze mnie krotką nazwaną (z ang. *named tuple*). Jest to lekki typ obiektu zawierający kolekcję klucz-wartość z możliwością indeksowania i iteracji.

Plik locator.py:

*from* collections *import* namedtuple  
  
*# setting namedtuple Locator with 'by' and 'value' used by BaseElement class to find elements on the page*  
Locator = namedtuple('Locator', ['by', 'value'])

Obiekt ten ma stanowić reprezentację lokatora Selenium, czyli sposobu w jaki WebDriver będzie szukał danego WebElementu na stronie. Generalnie możemy to robić przy użyciu:

1. Atrybutu id
2. Atrybutu name
3. XPath
4. CSS Selector
5. Class name
6. Link Text
7. PartialLink Text
8. Tag name

Do dobrych praktyk podczas tworzenia lokatora należy w pierwszej kolejności szukanie atrybutu id, ponieważ z założenia powinno być unikalne, atrybutu name lub class name (jeśli jest unikalna), a następnie XPath czy CSS. Większość użytych przeze mnie lokatorów opiera się na selektorze CSS, ścieżce XPath oraz Id. Dla zwiększenia czytelności można łączyć atrybut class z selektorem CSS, w przypadku użycia tego ostatniego. Przykładem będzie kilka poniższych lokatorów, gdzie kropka oznacza w składni CSS atrybut klasy:

*class* ContactPageLocators:

NAVIGATION\_TAB\_CSS = Locator(by=By.CSS\_SELECTOR, value='.navigation\_page')

NAVIGATION\_PAGE\_HEADING\_CSS = Locator(by=By.CSS\_SELECTOR, value='.page-heading.bottom-indent')  
 PAGE\_SUBHEADING\_TEXT = Locator(by=By.CSS\_SELECTOR, value='.page- subheading')

**4.4 Napotkane problemy**

**Niezależność testów od siebie**

Pierwszym z problemów na jaki natknęłam się podczas tworzenia tej pracy było uzyskanie pełnej niezależności testów od siebie. Przypadki testowe zawarte w scenariuszach zostały ułożone w sposób możliwie od siebie niezależny. Tak, by negatywny wynik jednego z nich nie blokował uruchomienia kolejnego lub nie wywoływał fałszywie negatywnego rezultatu. Niestety tej złotej zasady nie udało mi się zastosować wszędzie i kilka przypadków testowych tworzy pary, gdzie pomyślne zakończenie pierwszego stanowi warunek wstępny drugiego. Mam tu na myśli wylogowywanie się ze strony, czyli funkcjonalność która wymaga uprzedniego zalogowania użytkownika oraz usunięcie bądź zakup produktu z koszyka, który na początku testu musi się tam już znajdować. Dodatkowo zakup produktu możliwy jest tylko dla zalogowanego użytkownika. Najszerszym możliwym do ustawienia zakresem fikstury w pyteście, czyli funkcji przygotowującej środowisko testowe, wykonującej test i czyszczącej środowisko po wszystkim jest *session.* W tym projekcie odpowiada to pojedynczej sesji zautomatyzowanej przeglądarki internetowej. Nieosiągalnym więc stało się np. dodanie produktu do koszyka, a następnie wyczyszczenie przeglądarki, zamknięcie jej i uruchomienie ze stroną, która wciąż by ten produkt w koszyku zawierała. Dlatego te trzy przypadki testowe mogą zakończyć się z wynikiem fałszywie ujemnym. Dla testów wylogowywania jeśli nie powiedzie się logowanie użytkownika, a dla testów usuwania lub kupowania produktu, jeśli nie zostanie on w koszyku odnaleziony.

Miejscem, gdzie udało mi się rozwiązać ten problem była zależność logowania użytkownika od rejestracji. Ponieważ istnienie zarejestrowanego użytkownika nie jest ściśle powiązane z pojedynczą sesją przeglądarki internetowej (a raczej z jakąś bazą danych aplikacji) mogłam sięgnąć po możliwość wstrzyknięcia do testu danych logowania wczytanych z pliku *test\_login.json.* Jedynym warunkiem było stworzenie takiego użytkownika przed pierwszym uruchomieniem testów logowania.

Plik *test\_login.json* przyjmuje formę:

{  
 "valid\_credentials": {  
 "email": "kotytrzy@o2.pl",  
 "password": "5fEb.5aBbmKN3RK",  
 "expected\_result": "Welcome to your account."},  
  
 "invalid\_email": {  
 "email": "aaaa@bbb",  
 "password": "5fEb.5aBbmKN3RK",  
 "expected\_result": "Invalid email address."  
 },  
  
 "invalid\_password": {  
 "email": "kotytrzy@o2.pl",  
 "password": "4536hgbfd",  
 "expected\_result": "Authentication failed."  
 },  
  
 "restore\_password": {  
 "email": "kotytrzy@o2.pl",  
 "confirmation": "A confirmation email has been sent to your address: "  
 }  
}

Poza testem logowania z użyciem poprawnych danych istniejącego użytkownika (część „valid credentials”) wykorzystywany jest jeszcze w trzech innych przypadkach testowych składających się na scenariusz sprawdzający poprawność funkcjonalności logowania.

Kod testu logowania w pliku *test\_authentication\_page.py* wygląda następująco:

@mark.smoke  
@mark.signin  
*def* test\_sign\_in\_with\_valid\_credentials(*self*, get\_authentication\_page, user\_test\_data):  
 authentication\_page = get\_authentication\_page  
 test\_data = user\_test\_data  
 authentication\_page.login(email=test\_data['valid\_credentials']['email'],  
 password=test\_data['valid\_credentials']['password'])  
 expected\_result = test\_data['valid\_credentials']['expected\_result']  
 result = re.search(expected\_result, authentication\_page.page\_source)  
 *assert* result *is not None*

Natomiast wczytanie danych testowych następuje z pomocą fikstury zapisanej w pliku *conftest.py:*

*# data for login tests*  
data\_path2 = 'data/test\_login.json'  
  
  
*def* load\_test\_data(path):  
 *""" Function to load test data from json file to dict. """*  
 *with* open(path) *as* test\_data:  
 data = json.load(test\_data)  
 *yield* data

@fixture(params=load\_test\_data(data\_path2), scope='function')  
*def* user\_test\_data(request):  
 data = request.param  
 *yield* data

**Lokalizowanie WebElementów**

Automatyzacji czynności w przeglądarkach internetowych opiera się na właściwym lokalizowaniu elementów strony przez WebDrivera. W idealnej sytuacji każdy element znajdujący się na stronie jest dobrze opisany i posiada unikalny atrybut: id lub name. W rzeczywistych projektach często można natknąć się na trudne do jednoznacznego zlokalizowania elementy dlatego twórcy aplikacji My Store postarali się to odzwierciedlić. Dzięki temu mogłam wykorzystać i przećwiczyć różne formy tworzenia lokatorów. Do najłatwiejszych z nich należały oczywiście lokatory oparte na id, selektorze css, ścieżce Xpath i tekście linku np:

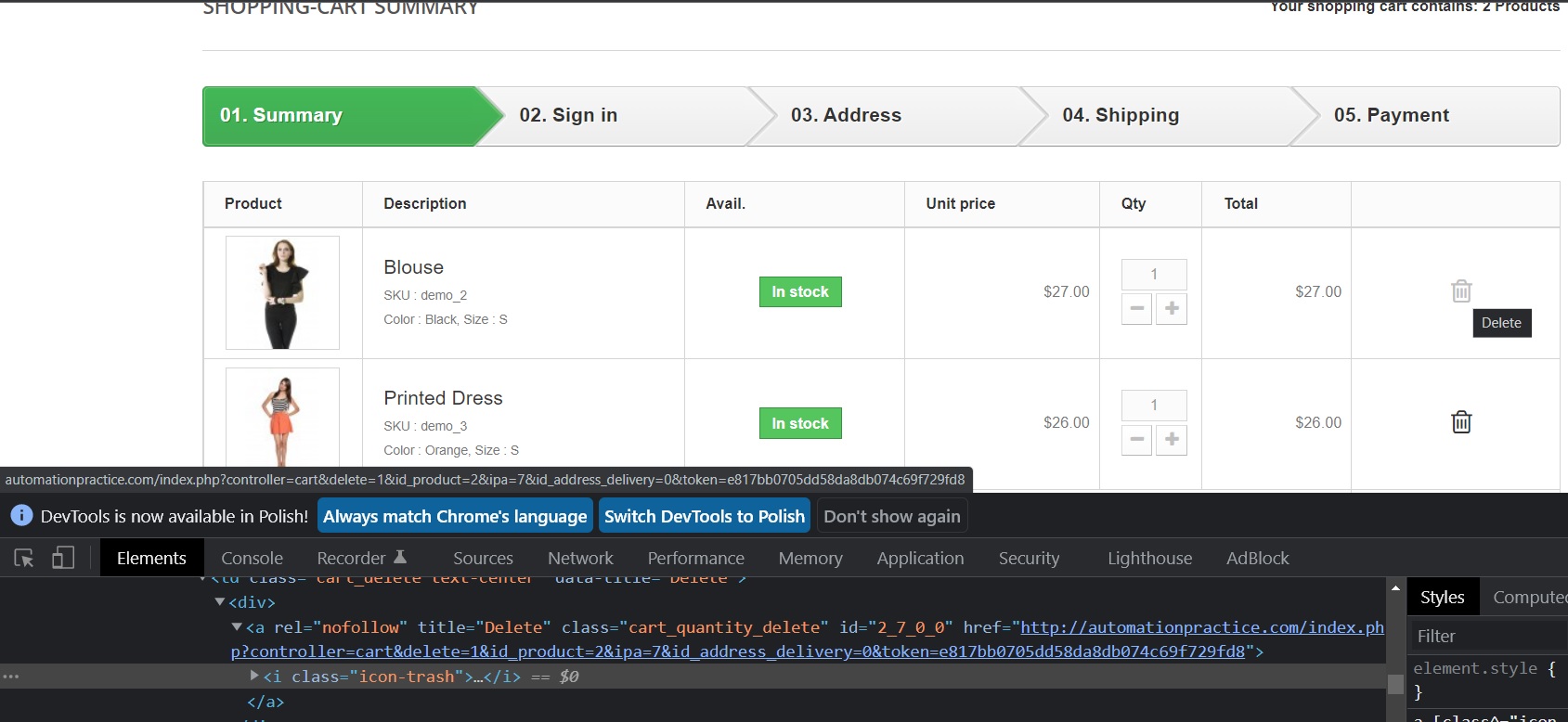
SELECT\_SUBJECT\_HEADING\_ID = Locator(by=By.ID, value='id\_contact')

MENU\_WOMEN\_TAB\_CSS = Locator(by=By.CSS\_SELECTOR, value='a[title="Women"]')

RETRIEVE\_PASSWORD\_BUTTON = Locator(by=By.XPATH, value='//span[normalize-space()="Retrieve Password"]')

SIGN\_IN\_BUTTON\_CSS = Locator(by=By.CSS\_SELECTOR, value='button[id="SubmitLogin"] span')

Wyzwaniem okazało się stworzenie lokatora dla ikony kosza (przycisk „delete”) produktu znajdującego się w koszyku., ponieważ nie posiada ona atrybutu name, class name dzieli z innymi elementami, a jej id składa się z cyfr, z których tylko pierwsza jest unikalna i odpowiada id produktu.

Rysunek 14: Kod HTML ikony kosza

Rozwiązanie kryło wśród bogactwa składni CSS: „[id^="' + product\_id + '"] gdzie znak ^ oznacza w tym wypadku id zaczynające się od product\_id.

*def* remove\_product\_from\_cart(*self*, product\_id):  
 *""" Function to remove a product from the cart by product id. """*  
delete\_button = *self*.driver.find\_element(by=By.CSS\_SELECTOR,  
 value='a[id^="' + product\_id + '"]')  
 delete\_button.click()  
 *return None*

**Metody czekające (Implicit i Explicit Wait)**

Podczas korzystania z Selenium można często natknąć się na błąd *NoSuchElementExeption.* Pojawia się on kiedy WebDriver nie może odnaleźć na stronie szukanego elementu. Najczęściej jest to w sposób bezpośredni powiązane z opóźnionym ładowaniem się elementu. Przyciski, linki, obrazki itp. mogą ładować się z różną prędkością. Dlatego sytuacje, kiedy WebDriver nie może odnaleźć czegoś, z czym miał wejść w interakcję są bardzo częste. Selenium rozwiązuje ten problem na dwa sposoby. Pierwszy z nich to metoda Implicit Wait, a drugi Explicit Wait.

Implicit Wait jest prostą w użyciu metodą, której działanie opiera się na określeniu czasu, na jaki ma zostać wstrzymana egzekucja polecenia bądź kroku testu. Przykładowo gdy Implicit Wait zostanie powiązany z metodą find\_element(by, value) i WebDriver nie znajdzie szukanego elementu od razu, będzie odpytywał DOM (z ang. Document Object Model) przez zadany czas. Prostota tej metody czekania posiada swoje wady ponieważ narzucony z góry czas może okazać się niewystarczający albo wręcz przeciwnie, nagminne używanie może doprowadzić do niepotrzebnego wydłużenia testów.

Sama sięgnęłam po to rozwiązanie w kilku miejscach, m.in. w pliku *home\_page.py* i metodzie nawigującej do strony kontaktowej:

*# redirect to contact us page*  
 *def* click\_contact\_button(*self*):  
 *""" Function for clicking the "Contact Us" button to redirect to the contact page."""*  
 *self*.contact\_button.click()  
 contact\_page = ContactPage(*self*.driver)  
 *self*.driver.implicitly\_wait(10)  
 *return* contact\_page

Explicit Wait jest bardziej rozbudowany i jego implementacja wygląda nieco inaczej. Aby go użyć musimy utworzyć obiekt klasy *WebDriverWait*, przekazać mu WebDrivera i timeout, a następnie wywołać dla niego metodę *until()*, w której zdefiniujemy interesujące nas warunki. Tutaj z pomocą przyjdzie nam klasa *ExpectedConditions,* która zawiera szereg interesujących warunków m.in.: *elementToBeClickable(),* textToBePresentInElement()*,* alertIsPresent()*.* Taka konstrukcja pozwala uściślić na co dokładnie i przez jaką wartości timeoutu WebDriver ma czekać.

Taka konstrukcja występuje w metodzie *find()* dla bazowego elementu w klasie *BaseElement*:

*from* selenium.webdriver.support.wait *import* WebDriverWait  
*from* selenium.webdriver.support *import* expected\_conditions *as* EC

*def* find(*self*):  
 *""" Finding element on the page using visibility as expected conditions """*  
element = WebDriverWait(*self*.driver, 10).until(  
 EC.visibility\_of\_element\_located(locator=*self*.locator)  
 )  
 *self*.web\_element = element  
 *return None*

Dzięki temu każde użycie WebDrivera do zlokalizowania elementu zachodzi z użyciem warunku *visibility\_of\_element\_located().* Metoda ta nie tylko sprawdza czy dany element znajduje się w DOM, ale również czy jest widoczny.

Z kolei metoda *click()* w klasie *BaseElement* opiera się na warunku sprawdzającym czy dany element jest klikalny i można z nim wejść w interakcję.

*def* click(*self*):  
 *""" Function to click elements using expected conditions """*  
element = WebDriverWait(*self*.driver, 10).until(  
 EC.element\_to\_be\_clickable(*self*.locator)  
 )  
 element.click()  
 *return None*

* 1. **Wybrane zagadnienia programistyczne**

**Pliki konfiguracyjne**

Użycie plików konfiguracyjnych znacznie poprawia czytelność, utrzymywalność i reużywalność kodu. Dzięki nim przygotowanie środowiska do testów staje się znacznie prostsze i przyjemniejsze niż próba wstrzykiwania tego typu informacji do skryptów testowych.. W tej pracy zostały użyte trzy pliki konfiguracyjne: *pytest.ini*, o którym była mowa w rozdziale 4.2 „Struktura programu”., *conftest.py* będący specyficznym plikiem frameworka pytest oraz *config.py*.

Ponieważ niniejszy projekt zakłada możliwość wykonania testów na trzech różnych przeglądarkach internetowych kwestią priorytetową stało się określenie, na której z nich zostaną one uruchomione. Zatem plik *config.py* zawiera klasę Config, której obiekt posiada atrybut drivera dla wybranej przeglądarki. Jeśli którykolwiek z driverów potrzebowałby dodatkowo specyficznych opcji uruchomieniowych klasa ta byłaby odpowiednim miejscem, by je wymienić. Zawartość *config.py* została przedstawiona poniżej:

*from* selenium *import* webdriver  
  
  
*class* Config:  
 *def* \_\_init\_\_(*self*, env):  
 *self*.driver = {  
 'chrome': webdriver.Chrome,  
 'firefox': webdriver.Firefox,  
 'edge': webdriver.Edge  
 }[env]  
 *if* env == 'chrome':  
 *self*.driver = *self*.driver(

options=

webdriver.ChromeOptions().add\_experimental\_option(

'excludeSwitches',  
 ['enable-logging']))  
 *else*:  
 *self*.driver = *self*.driver()

Pytest pozwala na dodanie własnych opcji wczytywanych z linii komend podczas zbierania plików do testów. Aby to osiągnąć potrzebne jest utworzenie funkcji *pytest\_addption(parser)* oraz odpowiedniej fikstury. W *conftest.py* wygląda to następująco:

*from* pytest *import* fixture  
*from* config *import* Config

*def* pytest\_addoption(parser):  
 *""" Function to add option from command line. """*  
parser.addoption(  
 "--env",  
 action="store",  
 help="Browser to run tests"  
 )  
  
  
@fixture(scope='session')  
*def* env(request):  
 *return* request.config.getoption('env')

Komenda dla uruchomienia przykładowego testu strony głównej na przeglądarce Firefox z linii komend będzie w takim przypadku wyglądało następująco:

$ python -m pytest -m homepage --env firefox

Jak widać na powyższym przykładzie poza markerem interesującego nas zestawu testów (-m homepage) należy podać wartość –env.

Samo przygotowanie i uruchomienie zautomatyzowanej przeglądarki realizowane jest dzięki fiksturze:

@fixture(scope='session')  
*def* browser(request, env):  
 *""" Function to initialize the browser."""*  
cfg = Config(env)  
 driver = cfg.driver  
 driver.maximize\_window()  
 *yield* driver  
 driver.quit()

W tym miejscu zachodzi utworzenie obiektu klasy *Config*, który dzięki podanemu parametrowi *env* przyjmuje postać odpowiedniego drivera. Na uwagę zasługuje fakt, że wszystko, co znajduje się po poleceniu *yield* wykona się dopiero po skończonym teście. Dzięki temu na początku testu zostanie uruchomiona przeglądarka, potem odbędzie się wykonywanie poleceń znajdujących się w teście, a na końcu driver zamknie przeglądarkę. Pozostając na przykładzie strony głównej kod potrzebny do załadowania strony i sprawdzenia jej tytułu wygląda w przedstawiony niżej sposób:  
Fikstura w *conftest.py:*

@fixture(scope='class')  
*def* get\_home\_page(request, browser):  
 home\_page = HomePage(driver=browser)  
 home\_page.go()  
 *yield* home\_page

Metoda testowa w *test\_home\_page.py* wywołująca powyższą fiksturę:

@mark.browsertest  
*def* test\_browser(*self*, get\_home\_page):  
 home = get\_home\_page  
 *assert* home.title == 'My Store'

**ActionChains**

Selenium dysponuje zestawem metod klasy ActionChains wykorzystywanych do automatyzacji interakcji z witryną. Odzwierciedlają one ruchy myszy, akcje przyciśnięć myszy, interakcje menu kontekstowego (prawego przycisku myszy) oraz naciśnięć klawiszy. Bez nich nie byłoby możliwe wykonywanie takich działań jak drag and drop czy hover over. W celu ich użycia należy zainicjalizować obiekt klasy ActionChains z przekazaną jako parametr instancją WebDrivera. Wywoływane metody są przechowywane w postaci kolejki. Dopiero w momencie wywołania *perform()* poszczególne akcje są wykonywane w kolejności, w jakiej zostały dodane do kolejki.

Przykład wykorzystania ActionChains znajduje się w pliku *women\_page.py* i metodzie *get\_quick\_view():*

*def* get\_quick\_view(*self*, selected\_product, product\_id):  
 *""" Function to get quick view of the selected product. """*  
hover = ActionChains(*self*.driver)  
 products = *self*.driver.find\_elements(by=By.XPATH, value='//div[@class="product- container"]')  
 *for* product *in* products:  
 *if* selected\_product *in* product.text:  
 hover.move\_to\_element(product).perform()  
 quick\_view = *self*.driver.find\_element(  
 by=By.XPATH,  
 value='//a[@class="quick-view"][@href="http://automationpractice.com/i ndex.php?id\_product='  
 + product\_id + '&controller=product"]/span')  
 hover.move\_to\_element(quick\_view).click().perform()  
 *break*  
 *self*.driver.implicitly\_wait(10)

Poszczególne akcje mogą być zapisane w postaci jednego łańcucha jak w linijce:

hover.move\_to\_element(quick\_view).click().perform()

Ewentualnie mogą przyjąć formę pojedynczych wywołań:

hover.move\_to\_element(quick\_view)

hover.click()

hover.perform()

Oba sposoby zapisu zostaną wykonane dokładnie w ten sam sposób.

* 1. **Uruchamianie testów**

Uruchamianie zestawu testów w pyteście realizowane jest za pomocą prostej komendy *pytest* wpisanej w konsoli. Aby została ona poprawnie wykonana należy znajdować się w katalogu zawierającym skrypty testowe. Pytest docelowo przeszuka cały katalog oraz wszystkie znajdujące się w nim podkatalogi tworząc kolekcję testów do uruchomienia. Testy zostaną zgromadzone według klucza (najczęściej prefiksu) ustalonego w pliku *pytest.ini.* Kolekcję testów bez potrzeby uruchomiania można wyświetlić za pomocą opcji –*collect-only.*

Sytuacja wygląda nieco inaczej jeśli struktura projektu rozdziela folder zawierający testy od folderu z klasami stron. W takim wypadku podanie komendy *pytest* zwróci błąd.

(automationpractice) C:\Projekty\AutomationPractice\automation\_practice>pytest

ImportError while loading conftest 'C:\Projekty\AutomationPractice\automation\_practice\tests\conftest.py'.

tests\conftest.py:4: in <module>

from pages.home\_page import HomePage

E ModuleNotFoundError: No module named 'pages'

Wynika to z faktu, że foldery *test* i *pages* są sobie równorzędne. Rozwiązań tego problemu jest kilka. Najprostsze z nich, nie wymagające instalowania żadnych dodatkowych elementów to wywołanie pytestu jako modułu pythona. Z poziomu głównego katalogu projektu należy w konsoli wpisać komendę:

$ python -m pytest

Możliwość podawania własnych opcji za pomocą komend zostało już przedstawione. Pozostaje jeszcze omówić kwestię wyboru konkretnego testu bądź zestawu testów do uruchomienia. Możliwe jest podanie nazwy wybranego pliku np.

$ python -m pytest tests/test\_home\_page.py

Pytest pozwala też na użycie dekoratora z markerem wybranych testów. Odpowiada za to przełącznik *-m <nazwa\_markera>* np.:

$ python -m pytest -m homepage

Marker przed klasą odnosi się do wszystkich testów zawartych w klasie. Poniżej przedstawiono odpowiadający za to fragment kodu klasy HomePageTests:

*from* pytest *import* mark  
  
@mark.homepage  
*class* HomePageTests:

* 1. **Raportowanie**

Samo wykonanie zestawu testów nie niosłoby ze sobą żadnych korzyści informacyjnych, gdyby po zakończeniu nie został wyświetlony raport z ich przebiegu.

Do tego celu służy flaga *-r* przyjmująca argumenty z listy:

* f – zakończone porażką
* E – zakończone błędem
* s – pominięte
* x – zakończone spodziewaną porażką
* X – zakończone spodziewanym sukcesem
* p – zakończone sukcesem
* P – zakończone sukcesem z outputem

Ponadto możliwe jest oznaczenie grup testów:

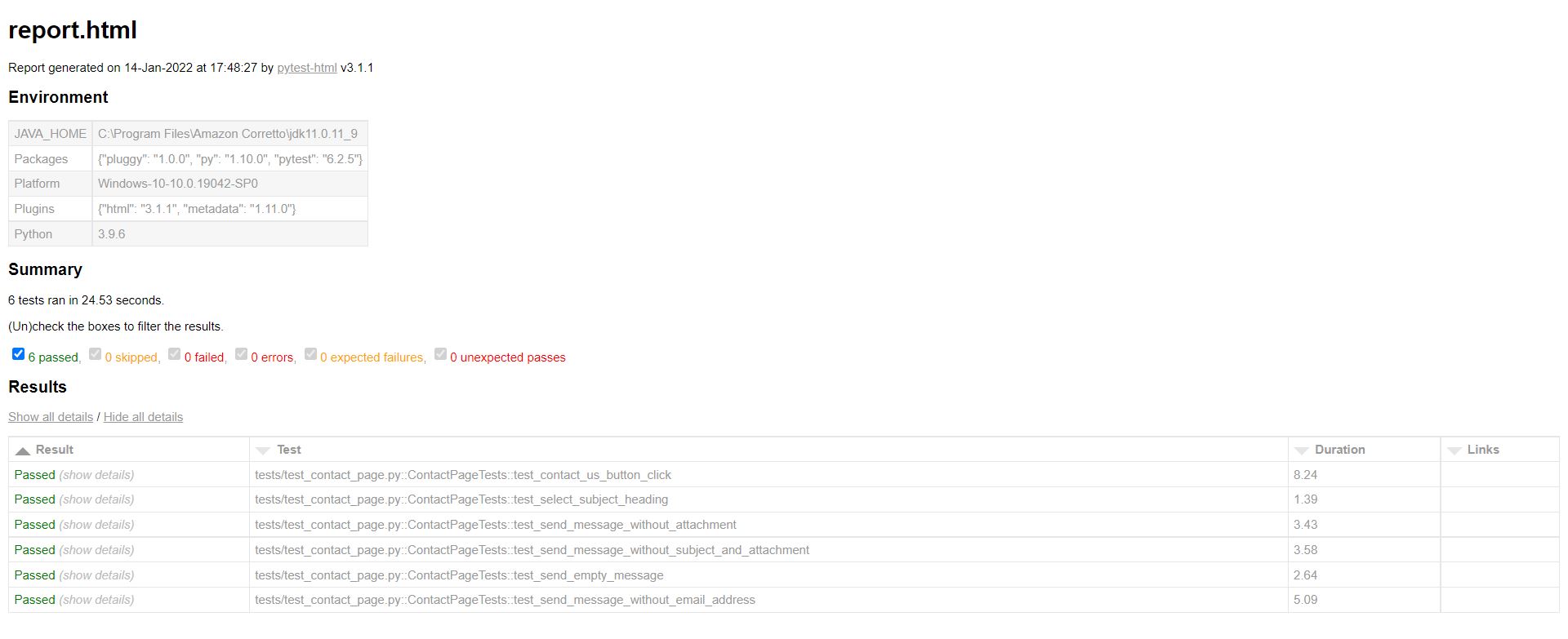
* a - wszystkie testy z wyjątkiem pP
* A – wszystkie
* N – żadne

Poniżej znajduje się przykład uruchomienia testów z dekoratorem *@mark.contactpage* w przeglądarce chrome z flagą *-rA*:

Rysunek 15: Przykład raportu z testów wyświetlony w konsoli

Raportowanie przebiegu testów w oknie konsoli sprawdza się tylko jeśli tester monitoruje sytuację na bieżąco. Inną możliwością jest wykorzystanie wbudowanego modułu i komendy *pytest –junitxml report.xml*, która utworzy plik xml domyślnie w głównym katalogu projektu (lub we wskazanej ścieżce). Pytest pozwala również na doinstalowanie dodatkowych pluginów np. *pytest-exel* i *pytest-html*, których zadaniem jest generowanie pliku zawierającego raport odpowiednio w formacie xls lub html. W tym projekcie został wykorzystany *pytest-html* z uwagi na czytelność dokumentu internetowego.

Dla porównania report.html dla tych samych testów @mark.contactpage wygląda następująco:

Rysunek 16: Przykładowy plik report.html

1. **Zakończenie**

**5.1 Rozwój testów**

W przyszłości prace nad rozwojem testów dla aplikacji My Store mogłyby być związane z pokryciem większej ilość przypadków użycia. Obecne testy najbardziej podstawowych funkcji stanowią dobry punkt wyjścia. Projekt został stworzony w oparciu o Page Object Model, więc dokładanie kolejnych stron i testów, a następnie integracja z już istniejącymi nie powinna stanowić żadnego problemu. Można by też pomyśleć o metodach uruchamiania testów w równoległych procesach z pomocą pluginu *pytest-xdist.* Ewentualnie zdalnie na kilku przeglądarkach jednocześnie. W tym celu należałoby wzbogacić środowisko testowe o Selenium Grid bądź Server Selenium. Pozwoliłoby to znacznie przyspieszyć realizację procesu testowego. Kolejną możliwością rozwoju projektu jest rozszerzenie go o współpracę z przeglądarkami na urządzeniach mobilnych. Ten ostatni kierunek, z uwagi na popularność i dostępność mobilnych urządzeń wydaje się szczególnie interesujący.

**5.2 Wnioski**

Stworzenie projektu testów automatycznych z użyciem Selenium WebDriver, języka Python oraz frameworku PyTest okazało się bardzo satysfakcjonującym zadaniem. Te trzy elementy bardzo dobrze ze sobą współpracują pozwalając na stworzenie prostego i czytelnego kodu. Mnogość dostępnych bibliotek i modułów pozwala uporać się z trudnościami, z jakimi wiąże się automatyzacja przeglądarki oraz obsługa aplikacji webowej. Poruszone w pracy zagadnienia zaledwie dotykają powierzchni niezwykle interesującego tematu jakim jest tworzenie testów automatycznych. Zarówno jeśli chodzi o kwestie czysto teoretyczne, opierające się na wytycznych ISTQB®, jak i kwestie programistyczne związane z konkretnym językiem czy narzędziem. Zachęcają za to do sięgnięcia głębiej, uruchomienia kolejnych zapasów dociekliwości i sprawdzenia, jakie jeszcze możliwości niesie ze sobą Selenium WebDriver. Czas spędzony nad tym projektem z pewnością przyniesie wymierne korzyści w postaci zdobytej wiedzy oraz doświadczenia niezbędnego na rynku pracy. Kolejnym etapem będzie zatem ugruntowanie tych umiejętności.

1. **Bibliografia**
   1. Analiza rynku E-commerce w Polsce
   2. „Developer Nation” SlashData - https://slashdata-website-cms.s3.amazonaws.com/sample\_reports/\_TPqMJKJpsfPe7ph.pdf
   3. Dokumentacja Pytest - https://docs.pytest.org/en/latest/contents.html
   4. Dokumentacja Selenium – www.selenium.dev/documentation
   5. „E-commerce w Polsce 2020. Gemius dla e-commerce w Polsce” https://www.gemius.pl/wszystkie-artykuly-aktualnosci/e-commerce-w-polsce-2020.html
   6. PwC „Strategie, które wygrywają. Liderzy e-commerce o rozwoju handlu cyfrowego”
   7. „Pytest – pierwsze kroki” – www.blog.qalabs.pl
   8. „Python dla każdego. Podstawy oprogramowania” Michael Dawson wyd.III
   9. Selenium-Python: https://selenium-python.readthedocs.io/
   10. Słownik terminów testowych ISTQB® wersja 3.4 (2020) International Software Testing Qualifications Board® oraz Stowarzyszenie Jakości Systemów Informatycznych
2. **Dodatkowe elementy pracy**

**Spis tabel**

Tabela 1. Przypadki testowe dla scenariusza nr 1….…..….…..……….……………….8

Tabela 2. Przypadki testowe dla scenariusza nr 2….…..….…..……….……………….9

Tabela 3. Przypadki testowe dla scenariusza nr 3….…..….…..……….……………...10

Tabela 4. Przypadki testowe dla scenariusza nr 4….…..….…..……….……………...12

Tabela 5. Przypadki testowe dla scenariusza nr 5….…..….…..……….……………...13

Tabela 6. Przypadki testowe dla scenariusza nr 6….…..….…..……….……………...14

Tabela 7. Przypadki testowe dla scenariusza nr 7….…..….…..……….……………...15

Tabela 8. Przypadki testowe dla scenariusza nr 8….…..….…..……….……………...16

**Spis rysunków**

Rysunek 1. Strona główna…………………………...….…..….…..……….…………...18

Rysunek 2. Strona pustego koszyka………………..….…..….…..……….…………...18

Rysunek 3. Strona z formularzem kontaktowym…..….…..….…..……….…………...19

Rysunek 4. Strona koszyka zawierającego produkt….…..….…..……….……………20

Rysunek 5. Szybki podgląd produktu….…..….…..……….…………………………….21

Rysunek 6. Strona logowania i tworzenia konta.....….…..….…..……….…………….22

Rysunek 7. Strona kategorii Women….…..….…..……………………….……………..23

Rysunek 8. Strona rejestracji konta….…..….…..……….………………………………24

Rysunek 9. Architektura Selenium Webdriver….…..….…..……….…………………..27

Rysunek 10. Komunikacja Webdrivera z przeglądarką..….…..……….………………27

Rysunek 11. Metody Selenium WebDriver….…..….…..……….………………………29

Rysunek 12. Struktura projektu….…..….…..……….……………….…………………..34

Rysunek 13. Zawartość folderów „pages” i „tests”….….…..….…..……….…………..35

Rysunek 14. Kod HTML ikony kosza….…..….…..……….……………….…………….46

Rysunek 15. Przykład raportu z testów wyświetlony w konsoli….…..….…..………...55

Rysunek 16. Przykładowy plik report.html….…..….……….……………………………56

**Załączniki**

Płyta CD z kodem źródłowym projektu.