O'REILLY®

Selenium WebDriver w Javie

Praktyczne wprowadzenie do tworzenia testów systemowych

Helion

Boni García

Tytuł oryginału: Hands-On Selenium WebDriver with Java:
A Deep Dive into the Development of End-to-End Tests

Tłumaczenie: Katarzyna Bogusławska

ISBN: 978-83-283-9982-2

Authorized Polish translation of the English edition of *Hands-On Selenium WebDriver with Java*
ISBN 9781098110000 © 2022 Boni García

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/sewebd
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/sewebd.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 3.

Podstawy WebDriver

Ten rozdział przedstawia podstawowe aspekty interfejsu API Selenium WebDriver. Zaczyna się od omówienia różnych sposobów tworzenia instancji obiektów hierarchii `WebDriver` (np. `ChromeDriver`, `EdgeDriver`, `FirefoxDriver` etc.). Przedstawiam także główne metody udostępniane przez te obiekty. Spośród nich kluczową jest znajdowanie elementów na stronie. W związku z tym poznasz różne lokalizatory, tj. strategie do znajdowania elementów w ramach strony (nazywanych w interfejsie API Selenium WebDriver `WebElement`), m.in. nazwy znaczników, teksty odnośników, atrybuty HTML (identyfikatory, nazwy, klasy), selektory CSS czy wyrażenia XPath. Innym istotnym elementem API WebDriver omawianym w tym rozdziale jest naśladowanie działań użytkownika (tj. automatyzacja interakcji ze stroną internetową za pomocą klawiatury czy myszki). W ostatniej części rozdziału prezentuję możliwości czekania na elementy strony. Ta funkcjonalność ma olbrzymie znaczenie ze względu na dynamiczną i asynchroniczną naturę aplikacji webowych.

Podstawy zastosowania WebDriver

W tym podrozdziale znajdziesz informacje na temat trzech zasadniczych aspektów związanych z obiektami WebDriver. Po pierwsze, różne sposoby ich tworzenia. Po drugie, ich podstawowe działania. Wreszcie, różne opcje ich usuwania (zwykle na koniec testu w celu zamknięcia przeglądarki).

Tworzenie obiektów WebDriver

Jak wspomniałem w rozdziale 2., do kontrolowania przeglądarek z Selenium WebDriver w Javie w pierwszej kolejności potrzebne są instancje `WebDriver`. Co za tym idzie, chcąc posługiwać się przeglądarką Chrome, musisz stworzyć obiekt `ChromeDriver`, `EdgeDriver` dla Edge czy `FirefoxDriver` dla Firefoksa. Podstawowy sposób tworzenia instancji tych typów obejmuje wykorzystanie operatora `new` w Javie. Przykładowo obiekt `ChromeDriver` można stworzyć w następujący sposób:

```
WebDriver driver = new ChromeDriver();
```

Użycie operatora `new` do tworzenia nowych instancji `WebDriver` jest w pełni poprawne i możesz używać tej metody w swoich testach. Niemniej jednak warto mieć świadomość innych możliwości i ich potencjalnych korzyści w zależności od konkretnego przypadku użycia. Te alternatywy obejmują mechanizmy budowania WebDriver i WebDriverManager.

Mechanizm budowania WebDriver

Interfejs API Selenium WebDriver udostępnia do tworzenia instancji `WebDriver` wbudowane metody zgodne ze wzorcem projektowym Budowniczy. Ta funkcjonalność jest też dostępna poprzez statyczną metodę `builder()` w klasie `RemoteWebDriver` i oferuje płynny interfejs API do instancji obiektów `WebDriver`. Tabela 3.1 przedstawia metody dostępne dla tego mechanizmu budowania. Przykład 3.1 prezentuje szkielet testu z jego zastosowaniem.

Tabela 3.1. Metody mechanizmu budowania WebDriver

Metoda	Opis
`oneOf(Capabilities opcje)`	Opcje dopasowane do przeglądarki
`addAlternative(Capabilities opcje)`	Alternatywne opcje zależne od przeglądarki (patrz rozdział 5.)
`addMetadata(String klucz, Object wartość)`	Dodawanie specjalnych metadanych, zwykle wykorzystywanych do żądania dodatkowych funkcjonalności od dostawców chmury (patrz rozdział 6.)
`setCapability(String nazwaOpcji, Object wartość)`	Pojedyncze opcje zależne od przeglądarki (patrz rozdział 5.)
`address(String adres)`	Ustawianie adresu zdalnego serwera (patrz rozdział 6.)
`address(URL adres)`	
`address(URI adres)`	
`config(ClientConfig konfig)`	Specjalistyczna konfiguracja dotycząca korzystania z serwera zdalnego (np. maksymalny czas oczekiwania na połączenie, ustawienia proxy)
`withDriverService(DriverService usługa)`	Specjalistyczne ustawienia lokalnego sterownika (np. chromedriver), takie jak lokalizacja plików, używany port, czas oczekiwania czy argumenty)
`build()`	Ostatnia metoda we wzorcu Budowniczy, poświęcona stworzeniu instancji WebDriver

W rozdziale 5. wyjaśniam szczegóły związane z *opcjami zależnymi od przeglądarki* (np. `ChromeOptions`). Na tym etapie korzystam z tych klas wyłącznie do wybrania typu przeglądarki (tj. `ChromeOptions` dla Chrome'a, `EdgeOptions` dla Edge czy `Firefox`
`↪Options` dla Firefoksa).

Przykład 3.1. Szkielet testu z wykorzystaniem mechanizmu budowania WebDriver

```
class WebDriverBuilderJupiterTest {

    WebDriver driver;

    @BeforeAll
    static void setupClass() {
        WebDriverManager.chromedriver().setup(); ❶
    }

    @BeforeEach
    void setup() {
        driver = RemoteWebDriver.builder().oneOf(new ChromeOptions()).build(); ❷
    }
```

```
    @AfterEach
        void teardown() {
            driver.quit();
    }

    @Test
    void test() {
        // TODO: wykorzystać zmienną „driver" do wywołania interfejsu API Selenium WebDriver
    }
}
```

❶ Jak zwykle przed stworzeniem instancji `WebDriver` rozwiązuję wymagany sterownik (w tym przypadku `chromedriver`) za pomocą WebDriverManager.

❷ Tworzę instancję `WebDriver` z wykorzystaniem mechanizmu budowania WebDriver. Ponieważ w teście chcę używać przeglądarki Chrome, posługuję się obiektem `ChromeOptions` w ramach argumentu opcji (korzystając z metody `oneOf()`).

Z funkcjonalnego punktu widzenia ten przykład działa dokładnie tak samo jak zwykły test „witaj, świecie" przedstawiony w rozdziale 2. Niemniej jednak interfejs API mechanizmu budowania WebDriver pozwala na proste wskazywanie różnych przeglądarek. Spójrz na poniższy przykład. W tym kodzie zmienia się metoda konfiguracyjna i tworzy instancję `SafariDriver`. Zastanówmy się, co się stanie, jeśli inicjalizacja nie jest możliwa (zwykle dzieje się tak, gdy test nie jest wykonywany na systemie macOS, a więc przeglądarka Safari nie jest dostępna w systemie). W takim przypadku korzystam z Chrome'a jako alternatywnej przeglądarki.

```
@BeforeEach
void setup() {
    driver = RemoteWebDriver.builder().oneOf(new SafariOptions())
            .addAlternative(new ChromeOptions()).build();
}
```

Mechanizm budowania WebDriverManager

Inną możliwością jest tworzenie obiektu `WebDriver` przez WebDriverManager. Poza rozwiązywaniem sterowników od wersji 5. WebDriverManager udostępnia mechanizm budowania obiektu `WebDriver`. Przykład 3.2 przedstawia szkielet testu z wykorzystaniem tego mechanizmu.

Przykład 3.2. Szkielet testu z wykorzystaniem mechanizmu budowania WebDriverManager

```
class WdmBuilderJupiterTest {

    WebDriver driver;

    @BeforeEach
    void setup() {
        driver = WebDriverManager.chromedriver().create(); ❶
    }

    @AfterEach
    void teardown() {
        driver.quit();
    }
```

```
    @Test
    void test() {
        // TODO: wykorzystać zmienną „driver" do wywołania interfejsu API Selenium WebDriver
    }
}
```

❶ WebDriverManager rozwiązuje wymagany sterownik (`chromedriver` w tym przypadku) i tworzy instancję właściwego typu `WebDriver` (w tym przypadku `ChromeDriver`) w jednej linii.

To podejście przynosi różne korzyści. Po pierwsze, pozwala na bardziej zwięzłe testy, ponieważ rozwiązywanie i tworzenie instancji `WebDriver` jest jednoczesne. Po drugie, umożliwia wskazywanie typu przeglądarki (np. Chrome, Firefox itd.) za pomocą samego wyboru menedżera (np. `chromedriver()`, `firefoxdriver()` itd.). Ponadto z łatwością można parametryzować dobór menedżera na potrzeby testów na wielu przeglądarkach (patrz rozdział 8.). Wreszcie, WebDriverManager pozwala konfigurować typowe dla przeglądarki opcje (patrz rozdział 5.) i bez wysiłku posługiwać się przeglądarkami w kontenerach Dockera (patrz rozdział 6.).

WebDriverManager zachowuje wskaźniki do obiektów `WebDriver`, które tworzy w tym podejściu. Co więcej, uruchamia mechanizm hook, który nadzoruje poprawną likwidację instancji `WebDriver`. Jeśli sesje WebDriver są aktywne, wirtualna maszyna Javy kończy je, a WebDriverManager zamyka przeglądarki. Możesz poćwiczyć te funkcjonalności poprzez usunięcie metody `teardown` z przykładu powyżej.

Chociaż WebDriverManager pozbywa się obiektów `WebDriver` automatycznie, zalecam robienie tego jawnie w każdym teście. Jeśli tego nie zrobisz, w przypadku zwyczajnego uruchomienia zestawu testów wszystkie przeglądarki pozostaną otwarte aż do końca wykonywania całego zestawu.

Metody WebDriver

Interfejs `WebDriver` udostępnia grupę metod, które są podstawą interfejsu API Selenium WebDriver. Tabela 3.2 przedstawia podsumowanie tych metod. Z kolei przykład 3.3 ilustruje podstawowy test z wykorzystaniem części z nich.

Tabela 3.2. Metody WebDriver

Metoda	Typ zwracany	Opis
`get(String adres)`	`void`	Ładuje stronę internetową w bieżącej przeglądarce.
`getCurrentUrl()`	`String`	Pobiera URL bieżącej przeglądarki.
`getTitle()`	`String`	Pobiera tytuł (znacznik HTML `<title>`) bieżącej strony.
`findElement(By lokalizator)`	`WebElement`	Znajduje pierwszy `WebElement` na stronie odpowiadający wskazanemu lokalizatorowi. Innymi słowy, jeśli lokalizator wskazuje kilka dopasowań, pierwsze (w obiektowym modelu dokumentu DOM) jest zwracane (patrz podrozdział „Znajdowanie elementów WebElement" w dalszej części rozdziału).

Tabela 3.2. Metody WebDriver (ciąg dalszy)

Metoda	Typ zwracany	Opis
`findElements(By lokalizator)`	`List<WebElement>`	Znajduje na stronie wszystkie elementy `WebElement` odpowiadające wskazanemu lokalizatorowi (patrz podrozdział „Znajdowanie elementów WebElement").
`getPageSource()`	`String`	Pobiera kod źródłowy HTML bieżącej strony.
`navigate()`	`Navigation`	Uzyskuje dostęp do historii przeglądarki i przechodzi pod wskazany adres URL (patrz rozdział 4.).
`getWindowHandle()`	`String`	Pobiera **uchwyt okna** (ang. *window handle*), czyli niepowtarzalny identyfikator otwartego okna przeglądarki (patrz rozdział 4.).
`getWindowHandles()`	`Set<String>`	Pobiera zbiór uchwytów okien otwartych w bieżącej przeglądarce (patrz rozdział 4.).
`switchTo()`	`TargetLocator`	Wybiera ramkę lub okno w bieżącej przeglądarce (patrz rozdział 4.).
`manage()`	`Options`	Ogólna funkcjonalność zarządzania różnymi aspektami przeglądarki (np. rozmiarem okna, pozycją, ciasteczkami, maksymalnym czasem oczekiwania czy logami).
`close()`	`void`	Zamyka bieżące okno i całą przeglądarkę, jeśli nie ma więcej otwartych okien.
`quit()`	`void`	Zamyka wszystkie okna i przeglądarkę.

Od tej chwili będę przedstawiał przykłady zawierające tylko logikę testową. Testy te posługują się obiektem `WebDriver` stworzonym wcześniej (w metodzie konfigurującej) i zamykają go na koniec (w ramach sprzątania po teście). Jako konwencję przyjąłem przykłady w JUnit 5 (ale wszystkie wersje: JUnit 4, Selenium-Jupiter i TestNG znajdziesz w repozytorium przykładów).

Przykład 3.3. Test wykorzystujący kilka podstawowych metod interfejsu Selenium WebDriver API

```
@Test
void testBasicMethods() {
    String sutUrl = "https://bonigarcia.dev/selenium-webdriver-java/";
    driver.get(sutUrl); ❶

    assertThat(driver.getTitle())
        .isEqualTo("Hands-On Selenium WebDriver with Java"); ❷
    assertThat(driver.getCurrentUrl()).isEqualTo(sutUrl); ❸
    assertThat(driver.getPageSource()).containsIgnoringCase("</html>"); ❹
}
```

❶ Otwieram stronę z ćwiczeniami.

❷ Weryfikuję, że jej tytuł jest zgodny z oczekiwanym.

❸ Potwierdzam, że adres URL pozostał bez zmian.

❹ Sprawdzam, że źródłowy kod HTML strony zawiera oczekiwany znacznik.

Konwencje nazw testów i klas w repozytorium przykładów

Dla ułatwienia znajdowania testów w repozytorium przykładów przestrzegam pewnej konwencji nazewniczej. Nazwa każdego zawartego tam testu zaczyna się od słowa test, po którym następuje opisowa etykieta. Następnie posługuję się tą etykietą w nazwie klasy w Javie zawierającej ten test. Przykładowo poprzedni test (o nazwie `testBasicMethods`) znajdziesz w klasach `BasicMethods ↪JUnit4Test` (używającej JUnit 4), `BasicMethodsJupiterTest` (używającej JUnit 5), `Basic ↪MethodsSelJupTest` (używającej JUnit 5 i Selenium-Jupiter) oraz `BasicMethodsNGTest` (używającej TestNG).

Identyfikator sesji

Za każdym razem, gdy tworzy się instancja `WebDriver`, obsługujący ją sterownik (np. `chromedriver`, `geckodriver` itd.) tworzy niepowtarzalne identyfikatory nazywane *sessionId* do śledzenia sesji przeglądarki. Możesz korzystać z tej wartości w teście, by ponad wszelką wątpliwość zidentyfikować sesję. W tym celu musisz wywołać metodę `getSessionId()` w obiekcie sterownika. Zwróć uwagę, że nie wymieniłem tej metody w tabeli 3.2, ponieważ należy ona do klasy `RemoteWebDriver`. W praktyce typy, których używasz do kontrolowania przeglądarki (np. `ChromeDriver`, `FirefoxDriver` itd.), dziedziczą po tej klasie. Wynika z tego, że musisz jedynie rzutować obiekt `WebDriver` na `RemoteWeb ↪Driver`, by wywołać metodę `getSessionId()`. Test zawierający takie działanie przedstawiłem w przykładzie 3.4.

Przykład 3.4. Test odczytujący wartość sessionId

```
@Test
void testSessionId() {
    driver.get("https://bonigarcia.dev/selenium-webdriver-java/");

    SessionId sessionId = ((RemoteWebDriver) driver).getSessionId(); ❶
    assertThat(sessionId).isNotNull(); ❷
    log.debug("The sessionId is {}", sessionId.toString()); ❸
}
```

❶ Rzutuję obiekt sterownika na typ `RemoteWebDriver` i odczytuję wartość `sessionId`.

❷ Weryfikuję, że `sessionId` ma pewną wartość.

❸ Zapisuję wartość `sessionId` na standardowe wyjście.

Pozbywanie się obiektów WebDriver

Jak widzisz w tabeli 3.2, istnieją dwie metody kończenia działania obiektów `WebDriver`: `close()` i `quit()`. Co do zasady posługuję się metodą `quit()` w przykładach, ponieważ zamyka ona przeglądarkę i wszystkie związane z nią okna. Z kolei metoda `close()` zamyka tylko bieżące okno. W związku z tym używam metody `close()` tylko wtedy, gdy obsługuję różne okna (lub karty) w tej samej przeglądarce i chcę zamknąć tylko niektóre z nich, a resztę nadal wykorzystywać.

Znajdowanie elementów WebElement

Jedną z najważniejszych cech interfejsu API Selenium WebDriver jest możliwość wchodzenia w interakcje z różnymi elementami strony internetowej. Elementy te są obsługiwane przez Selenium WebDriver dzięki interfejsowi `WebElement` — abstrakcji dotyczącej elementów HTML. Jak zaznaczyłem w tabeli 3.2, do wyboru są dwie metody do znajdowania elementów `WebElement` na stronie. Pierwsza z nich — `findElement()` — zwraca pierwsze wystąpienie (jeśli takie jest) danego węzła w obiektowym modelu dokumentu DOM. Druga — `findElements()` — zwraca listę elementów. Każda z nich przyjmuje parametr `By`, który wskazuje strategię lokalizowania.

Obiektowy model dokumentu DOM

Obiektowy model dokumentu DOM to niezależny od platformy interfejs pozwalający na przedstawianie dokumentów w typie XML (np. stron internetowych opartych na HTML) w strukturze drzewiastej. Przykład 3.5 prezentuje małą stronę internetową. Związana z nią struktura DOM w pamięci jest przedstawiona na rysunku 3.1. Jak widzisz, każdy znacznik HTML (np. `<html>`, `<head>`, `<body>`, `<a>` itd.) tworzy węzeł (element) drzewa. Następnie każdy standardowy atrybut HTML (`charset`, `href` itd.) tworzy odpowiednią *właściwość* DOM. Ponadto także wartość tekstowa znaczników HTML jest dostępna w drzewie. Języki takie jak JavaScript posługują się metodami DOM do udostępniania i zmiany struktury drzewa. Dzięki temu strony internetowe są dynamiczne i mogą zmieniać swój układ oraz treść w odpowiedzi na działania użytkownika.

Przykład 3.5. Podstawowa strona internetowa

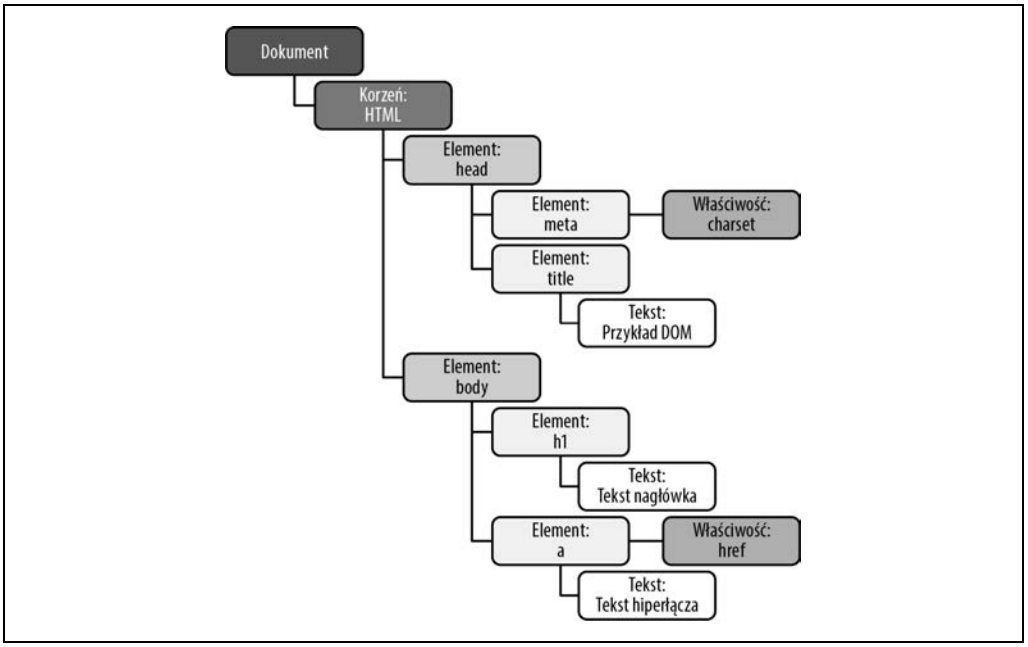
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Przykład DOM</title>
</head>
<body>
    <h1>Tekst nagłówka</h1>
    <a href="#">Tekst hiperłącza</a>
</body>
</html>
```

Metody WebElement

Tabela 3.3 zawiera podsumowanie metod dostępnych w klasie `WebElement`. Przykłady użycia każdej z nich znajdziesz w dalszej części tego punktu.

Strategie lokalizacji

Selenium WebDriver udostępnia osiem podstawowych strategii lokalizowania elementów, które podsumowuję w tabeli 3.4 . Ponadto, jak wyjaśniam w kolejnych podpunktach, istnieją inne zaawansowane strategie lokalizacji, tj. lokalizatory złożone i względne.



Rysunek 3.1 Struktura DOM wygenerowana dla przykładu 3.5

Tabela 3.3. Metody WebElement

Metoda	Typ zwracany	Opis
`click()`	`void`	Wykonuje kliknięcie myszą (tj. kliknięcie lewym przyciskiem) na bieżącym elemencie.
`submit()`	`void`	Wysyła formularz (gdy bieżący element jest formularzem).
`sendKeys(CharSequence... klucze)`	`void`	Symuluje pisanie na klawiaturze (np. w polach tekstowych).
`clear()`	`void`	Resetuje wartość pola tekstowego.
`getTagName()`	`String`	Pobiera nazwę znacznika.
`getDomProperty(String nazwa)`	`String`	Pobiera wartość wskazanej właściwości DOM.
`getDomAttribute(String nazwa)`	`String`	Pobiera wartość atrybutu elementu zadeklarowaną w jego kodzie HTML.
`getAttribute(String nazwa)`	`String`	Pobiera wartość podanego atrybutu HTML (np. `class`) jako łańcuch znaków. Dokładniej: metoda ta próbuje uzyskać sensowną wartość właściwości DOM o wskazanej nazwie, jeśli taka właściwość istnieje. Przykładowo dla atrybutów logicznych (np. `readonly`) zwraca `true`, gdy właściwość istnieje, i `null`, gdy nie istnieje.
`getAriaRole()`	`String`	Pobiera rolę elementu według specyfikacji W3C WAI-ARIA (*https://www.w3.org/TR/wai-aria*).
`getAccessibleName()`	`String`	Pobiera dostępną nazwę zgodną ze specyfikacją WAI-ARIA.

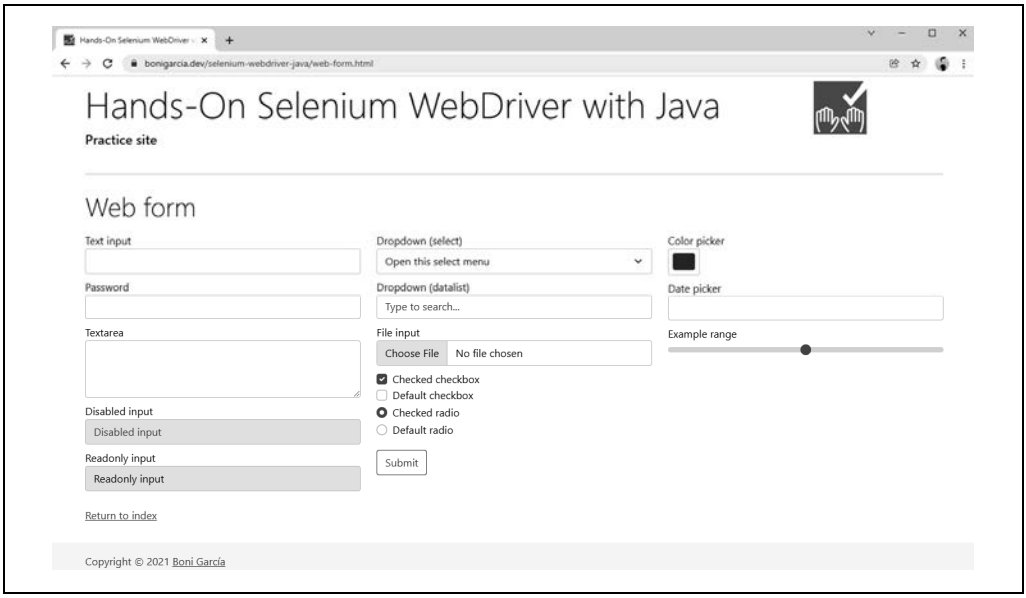
Tabela 3.3. Metody WebElement (ciąg dalszy)

Metoda	Typ zwracany	Opis
`isSelected()`	`boolean`	Określa, czy pole wyboru, opcja lub przycisk radio są zaznaczone.
`isEnabled()`	`boolean`	Określa, czy element jest aktywny, czy nie (np. pole formularza).
`isDisplayed()`	`boolean`	Określa, czy element jest widoczny, czy nie.
`getText()`	`String`	Pobiera widoczny tekst elementu, łącznie z jego elementami dziedzicznymi.
`getLocation()`	`Point`	Pobiera pozycje (koordynaty x i y) lewego górnego rogu wyrenderowanego elementu.
`getSize()`	`Dimension`	Pobiera szerokość i wysokość wyrenderowanego elementu.
`getRect()`	`Rectangle`	Pobiera pozycję i rozmiar wyrenderowanego elementu.
`getCssValue (String nazwaWłaściwości)`	`String`	Pobiera wartość właściwości CSS elementu.
`getShadowRoot()`	`SearchContext`	Pobiera element Shadow Root, by przeszukiwać płytkie drzewo elementu (patrz podrozdział „Shadow DOM" w rozdziale 4.).
`findElements(By lokalizator)`	`List< WebElement>`	Znajduje wszystkie podelementy, które pasują do lokalizatora w drzewie poniżej bieżącego elementu.
`findElement(By lokalizator)`	`WebElement`	Znajduje pierwszy podelement, który pasuje do lokalizatora w drzewie poniżej bieżącego elementu.

Tabela 3.4. Podsumowanie strategii lokalizacji w Selenium WebDriver

Lokalizator	Znajduje elementy na podstawie
Nazwa znacznika	Nazwy znacznika HTML (np. `a`, `p`, `div`, `img` itd.).
Tekst odnośnika	Dokładnej wartości tekstowej wyświetlanej na odnośniku (tj. w znaczniku HTML).
Cząstkowy tekst odnośnika	Tekstu zawartego na odnośniku (tj. w znaczniku HTML).
Nazwa	Wartości atrybutu `name`.
Identyfikator	Wartości atrybutu `id`.
Nazwa klasy	Wartości atrybutu `class`.
Selektor CSS	Wzorca zgodnego z rekomendacją W3C Selectors (*https://www.w3.org/TR/selectors*). Pierwotnym celem wzorców CSS jest wybieranie elementów ze strony, by opatrzyć je stylami CSS. Selenium WebDriver pozwala na wykorzystanie tych selektorów do znajdowania elementów i wchodzenia w interakcje z nimi.
XPath	Zapytań o składni języka XPath (*https://www.w3.org/TR/xpath*) (XML Path Language). XPath to standardowy język zapytań do wybierania węzłów w dokumentach typu XML (np. stronach internetowych).

Konkretny lokalizator określa się poprzez użycie klasy `By` w interfejsie API Selenium WebDriver. Poniższe podpunkty przedstawiają przykłady użycia każdej z nich. W tym celu posługuję się stroną z ćwiczeniami (*https://bonigarcia.dev/selenium-webdriver-java/web-form.html*). Rysunek 3.2 przedstawia zrzut strony z zamieszczonym na tej stronie formularzem.



Rysunek 3.2. Formularz do ćwiczeń wykorzystywany w przykładach związanych z lokalizatorami

Znajdowanie na podstawie nazwy znacznika HTML

Jedną z najbardziej podstawowych strategii znajdowania elementów jest ta oparta na nazwie znacznika. Przykład 3.6 przedstawia test wykorzystujący tę strategię. Test ten znajduje dostępne na stronie z ćwiczeniami pole tekstowe, którego kod HTML wygląda następująco:

```
<textarea class="form-control" id="my-textarea" rows="3"></textarea>
```

Przykład 3.6. Test oparty na strategii szukania na podstawie nazwy znacznika

```
@Test
void testByTagName() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    WebElement textarea = driver.findElement(By.tagName("textarea")); ❶
    assertThat(textarea.getDomAttribute("rows")).isEqualTo("3"); ❷
}
```

❶ Korzystam z lokalizatora `By.tagName("textarea")`, by znaleźć element. W tym przypadku, ponieważ na stronie zadeklarowane jest tylko jedno pole tekstowe, mogę być pewien, że metoda `findElement()` znajdzie go.

❷ Upewniam się, że wartość atrybutu `rows` jest taka sama jak zadeklarowana w kodzie HTML.

Znajdowanie na podstawie atrybutu HTML (nazwy, id, klasy)

Inną prostą strategią znajdowania elementów jest szukanie na podstawie atrybutów HTML, m.in. nazwy, identyfikatora id lub klasy. Przyjrzyj się polu tekstowemu na stronie z ćwiczeniami. Zauważ, że zawiera ono standardowe atrybuty `class`, `name`, `id` oraz niestandardowy atrybut `myprop` (dodany,

by pokazać różnicę między kilkoma metodami `WebDriver`). Przykład 3.7 przedstawia test oparty na tej strategii.

```
<input type="text" class="form-control" name="my-text" id="my-text-id" myprop="myvalue">
```

Przykład 3.7. Test wykorzystujący lokalizatory oparte na atrybutach HTML (name, id i class)

```
@Test
void testByHtmlAttributes() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    // Po nazwie
    WebElement textByName = driver.findElement(By.name("my-text")); ❶
    assertThat(textByName.isEnabled()).isTrue(); ❷

    // Po identyfikatorze
    WebElement textById = driver.findElement(By.id("my-text-id")); ❸
    assertThat(textById.getAttribute("type")).isEqualTo("text"); ❹
    assertThat(textById.getDomAttribute("type")).isEqualTo("text");
    assertThat(textById.getDomProperty("type")).isEqualTo("text");

    assertThat(textById.getAttribute("myprop")).isEqualTo("myvalue"); ❺
    assertThat(textById.getDomAttribute("myprop")).isEqualTo("myvalue");
    assertThat(textById.getDomProperty("myprop")).isNull();

    // Po nazwie klasy
    List<WebElement> byClassName = driver.findElements(By.className("form-control")); ❻
    assertThat(byClassName.size()).isPositive(); ❼
    assertThat(byClassName.get(0).getAttribute("name")).isEqualTo("my-text"); ❽
}
```

❶ Znajduję pole tekstowe na podstawie nazwy.

❷ Dokonuję asercji, że pole jest aktywne (tj. użytkownik może w nim pisać).

❸ Znajduję ten sam element na podstawie jego identyfikatora id.

❹ Ta asercja (jak i dwie kolejne) zwraca tę samą wartość, ponieważ atrybut `type` jest standardowy i, jak wcześniej wyjaśniałem, staje się *właściwością* DOM.

❺ Ta asercja (jak i dwie kolejne) zwraca różne wartości, ponieważ atrybut `myprop` nie jest standardowy i w związku z tym nie staje się właściwością DOM.

❻ Znajduję listę elementów na podstawie klasy.

❼ Weryfikuję, że lista ma więcej niż jeden element.

❽ Sprawdzam, że pierwszy element znaleziony na podstawie klasy jest taki sam jak pole tekstowe znalezione wcześniej.

Znajdowanie na podstawie tekstu odnośnika

Ostatnim podstawowym lokalizatorem jest ten oparty na tekście odnośnika. Ta strategia ma dwie wersje: szukanie dokładnego dopasowania tekstu i szukanie częściowego dopasowania. W przykładowym formularzu posługuję się odnośnikiem o następującym kodzie HTML. Dalej, w przykładzie 3.8 przedstawiam test używający tych lokalizatorów.

```
<a href="./index.html">Return to index</a>
```

Przykład 3.8. Test wykorzystujący lokalizatory oparte na tekście odnośnika

```
@Test
void testByLinkText() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    WebElement linkByText = driver.findElement(By.linkText("Return to index")); ❶
    assertThat(linkByText.getTagName()).isEqualTo("a"); ❷
    assertThat(linkByText.getCssValue("cursor")).isEqualTo("pointer"); ❸

    WebElement linkByPartialText = driver.findElement(By.partialLinkText("index")); ❹
    assertThat(linkByPartialText.getLocation()).isEqualTo(linkByText.getLocation()); ❺
    assertThat(linkByPartialText.getRect()).isEqualTo(linkByText.getRect());
}
```

❶ Znajduję element na podstawie pełnego tekstu odnośnika.

❷ Sprawdzam, że nazwa jego znacznika to `a`.

❸ Sprawdzam, że jego właściwość CSS `cursor` ma wartość `pointer` (tj. styl typowo wykorzystywany dla elementów, które można kliknąć).

❹ Znajduję element na podstawie częściowego dopasowania tekstu odnośnika. Jest to ten sam element co w kroku 1.

❺ Weryfikuję, że oba elementy mają tę samą pozycję i rozmiar.

Znajdowanie na podstawie selektorów CSS

Strategie, które poznałeś do tej pory, są łatwe do zastosowania, ale mają pewne ograniczenia. Po pierwsze, znajdowanie elementów na podstawie nazwy znacznika może być nieskuteczne, ponieważ jest bardzo prawdopodobne, że na stronie będzie wiele elementów o tym samym znaczniku. Z kolei znajdowanie elementów na podstawie atrybutów HTML (nazwy, identyfikatora id czy klasy) ma ograniczone zastosowanie, ponieważ te atrybuty nie zawsze są dostępne. Ponadto identyfikatory id mogą być autogenerowane i zmienne między sesjami. Wreszcie szukanie na podstawie tekstu odnośnika może być stosowane tylko do odnośników. Aby pokonać te trudności, Selenium WebDriver proponuje dwie strategie o szerokim zastosowaniu: selektory CSS i XPath.

Istnieje wiele możliwości tworzenia selektorów CSS. Tabela 3.5 przedstawia podsumowanie podstawowych selektorów.

Tabela 3.5. Podstawowe selektory CSS

Kategoria	Składnia	Opis	Przykład	Wyjaśnienie przykładu
Uniwersalny	*	Wybiera wszystkie elementy.	*	Odpowiadają mu wszystkie elementy.
Typ	*nazwaElementu*	Wybiera wszystkie elementy z danym znacznikiem.	`input`	Odpowiadają mu wszystkie elementy `<input>`.
Klasa	*.nazwaKlasy*	Wybiera wszystkie elementy o danej wartości atrybutu `class`	`.form-control`	Odpowiadają mu wszystkie elementy o klasie `form-control`.

Tabela 3.5. Podstawowe selektory CSS (ciąg dalszy)

Kategoria	Składnia	Opis	Przykład	Wyjaśnienie przykładu
Id	`#identyfikator`	Wybiera wszystkie elementy o danej wartości atrybutu id.	`#my-text-id`	Odpowiadają mu wszystkie elementy z identyfikatorem `my-text-id`.
Atrybut	[*atrybut*]	Wybiera wszystkie elementy z danym atrybutem.	`[target]`	Odpowiadają mu wszystkie elementy zawierające atrybut `target`.
	[*atrybut=wartość*]	Wybiera wszystkie elementy zawierające dany atrybut o konkretnej wartości.	`[target=_blank]`	Odpowiadają mu wszystkie elementy zawierające atrybut `target="blank"`.
	[*atrybut~=wartość*]	Wybiera wszystkie elementy zawierające dany atrybut i pewną wartość tekstową.	`[title~=hands]`	Odpowiadają mu wszystkie elementy `title` zawierające słowo `hands`.
	[*atrybut|=wartość*]	Wybiera wszystkie elementy, w których wartość danego atrybutu jest równa lub zaczyna się od pewnych danych.	`[lang|=en]`	Odpowiadają mu wszystkie elementy o wartości równej lub zaczynającej się od `en`.
	[*atrybut^=wartość*]	Wybiera wszystkie elementy, których wartość danego atrybutu zaczyna się od pewnych danych.	`a[href^="https"]`	Odpowiadają mu wszystkie odnośniki, których atrybut `href` zaczyna się od `https`.
	[*atrybut$=wartość*]	Wybiera wszystkie elementy, których wartość danego atrybutu kończy się pewnymi danymi.	`a[href$=".pdf"]`	Odpowiadają mu wszystkie odnośniki, których atrybut `href` kończy się na `.pdf`.
	[*atrybut*=wartość*]	Wybiera wszystkie elementy, których wartość danego atrybutu zawiera pewien łańcuch znaków.	`a[href*="github"]`	Odpowiadają mu wszystkie odnośniki, których atrybut `href` zawiera słowo `github`.

Poniższy fragment kodu HTML pokazuje ukryte pole tekstowe znajdujące się w formularzu ćwiczeniowym. Następnie przykład 3.9 przedstawia możliwe sposoby znalezienia tego elementu na podstawie selektorów CSS. Jedną z zalet tego lokalizatora jest to, że selektor będzie dalej działać, nawet gdy atrybut `name` w HTML zmieni swoją wartość.

```
<input type="hidden" name="my-hidden">
```

Przykład 3.9. Test wykorzystujący lokalizatory oparte na selektorach CSS

```
@Test
void testByCssSelectorBasic() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    WebElement hidden = driver.findElement(By.cssSelector("input[type=hidden]")); ❶
    assertThat(hidden.isDisplayed()).isFalse(); ❷
}
```

❶ Wykorzystuję selektor CSS do znalezienia ukrytego pola.

❷ Sprawdzam, że pole nie jest widoczne.

Istnieje mnóstwo możliwości tworzenia zaawansowanych selektorów CSS. Tabela 3.6 pokazuje niektóre z nich. Pełny opis selektorów CSS jest dostępny w oficjalnej rekomendacji W3C (*https://www.w3.org/TR/selectors*).

Tabela 3.6. Zaawansowane selektory CSS

Kategoria	Składnia	Opis	Przykład	Wyjaśnienie przykładu
Grupa	`,`	Grupa dwóch (lub więcej) selektorów.	`div, span`	Odpowiadają mu wszystkie elementy `<span>` i `<div>`.
Połączenia	(*spacja*)	Wybiera elementy, które następują po sobie.	`div span`	Odpowiadają mu wszystkie elementy `<span>` wewnątrz elementów `<div>`.
	A `>` *B*	Wybiera elementy, które są bezpośrednimi następnikami innego elementu.	`ul > li`	Odpowiadają mu wszystkie elementy `<li>` zagnieżdżone bezpośrednio w elemencie `<ul>`.
	A `~` *B*	Wybiera elementy posiadające ten sam poprzednik (tj. *rodzeństwo*), a drugi element następuje popierwszym (niekoniecznie bezpośrednio).	`p ~ span`	Odpowiadają mu wszystkie elementy `<span>` następujące po `<p>` (bezpośrednio lub nie).
	A `+` *B*	Elementy będące rodzeństwem, a drugi z nich występuje bezpośrednio obok pierwszego.	`h2 + p`	Odpowiadają mu wszystkie elementy `<p>` następujące bezpośrednio po `<h2>`.
Pseudo	`:`	Wybiera *pseudoklasy* CSS (tj. specjalny stan wybranego elementu).	`a:visited`	Odpowiadają mu wszystkie kliknięte odnośniki.
	`:nth-child(`*n*`)`	Wybiera elementy na podstawie ich pozycji w grupie, zaczynając od początku.	`p:nth-child(2)`	Odpowiada mu co drugi następnik elementu `<p>`.
	`:not(`*selektor*`)`	Wybiera elementy nieodpowiadające danemu selektorowi.	`:not(p)`	Odpowiada mu każdy element różny od `<p>`.
	`:nth-lastchild(`*n*`)`	Wybiera elementy na podstawie ich pozycji w grupie, zaczynając od końca.	`p:nth-lastchild(2)`	Odpowiada mu co drugi następnik elementu `<p>` (licząc od ostatniego następnika).
	`::`	Wybiera *pseudoelement* CSS (tj. konkretną część wybranego elementu).	`p::firstline`	Odpowiada mu pierwsza linia każdego elementu `<p>`.

Przyjrzyj się poniższemu kodowi HTML (jak zwykle zawartemu w formularzu ćwiczeniowym). Jak widzisz, jest w nim kilka pól wyboru: jedno z nich jest zaznaczone, a drugie nie. Możesz

określić, które z nich jest zaznaczone dzięki interfejsowi API Selenium WebDriver i selektorom CSS. W tym celu przykład 3.10 posługuje się pseudoklasą.

```
<input class="form-check-input" type="checkbox" name="my-check" id="my-check-1" checked>
<input class="form-check-input" type="checkbox" name="my-check" id="my-check-2">
```

Przykład 3.10. Test wykorzystujący zaawansowane lokatory oparte na selektorach CSS

```
@Test
void testByCssSelectorAdvanced() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    WebElement checkbox1 = driver
        .findElement(By.cssSelector("[type=checkbox]:checked")); ❶
    assertThat(checkbox1.getAttribute("id")).isEqualTo("my-checkbox-1"); ❷
    assertThat(checkbox1.isSelected()).isTrue(); ❸

    WebElement checkbox2 = driver
        .findElement(By.cssSelector("[type=checkbox]:not(:checked)")); ❹
    assertThat(checkbox2.getAttribute("id")).isEqualTo("my-checkbox-2"); ❺
    assertThat(checkbox2.isSelected()).isFalse(); ❻
}
```

❶ Korzystam z pseudoklasy `checked` do znalezienia kliknięte pola wyboru.

❷ Sprawdzam, że identyfikator `id` elementu odpowiada oczekiwaniom.

❸ Potwierdzam, że element jest zaznaczony.

❹ Korzystam z pseudoklasy `checked` i operatora `not` do znalezienia domyślnego pola wyboru.

❺ Sprawdzam, że identyfikator `id` elementu odpowiada oczekiwaniom.

❻ Potwierdzam, że element nie jest zaznaczony.

Znajdowanie na podstawie XPath

XPath (ang. *XML Path Language*, język ścieżek XML) to potężny sposób nawigowania po drzewie DOM dokumentów w typie XML, m.in. stron HTML. Obejmuje on przeszło dwieście wbudowanych funkcji do tworzenia złożonych zapytań szukających węzłów. Istnieją dwa typy zapytań XPath. Pierwsze, *absolutne*, wykorzystują symbol ukośnika (/) do przechodzenia po drzewie DOM od korzenia. Przykładowo, wracając do podstawowej strony HTML z przykładu 3.5, aby wybrać odnośnik znajdujący się na tej stronie, należałoby posłużyć się następującym zapytaniem:

```
/html/body/a
```

Absolutne ścieżki XPath łatwo zbudować, ale mają jedną poważną niedogodność — najdrobniejsza nawet zmiana w układzie strony spowoduje, że zbudowany w ten sposób lokalizator zawiedzie. Z tego powodu użycie ścieżek absolutnych nie jest zalecane. Zamiast nich bardziej przydatne są ścieżki *względne*.

Ogólna składnia względnych wyrażeń XPath wygląda następująco:

```
//nazwaZnacznika[@atrybut='wartość']
```

Przykład 3.11 przedstawia test używający lokalizatorów XPath do wybrania ukrytego na stronie z ćwiczeniami pola.

Przykład 3.11. Test wykorzystujący podstawowy lokalizator XPath

```
@Test
void testByXPathBasic() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    WebElement hidden = driver
        .findElement(By.xpath("//input[@type='hidden']")); ❶
    assertThat(hidden.isDisplayed()).isFalse(); ❷
}
```

❶ Znajduję ukryty element na stronie z ćwiczeniami.

❷ Weryfikuję, że element nie jest widoczny dla użytkownika.

Prawdziwa moc wyrażeń XPath pochodzi z wbudowanych funkcji. Tabela 3.7 podsumowuje najważniejsze z nich. Możesz znaleźć pełny opis XPath w rekomendacji W3C XPath Recommendations (*https://www.w3.org/TR/xpath*).

Tabela 3.7. Podsumowanie najważniejszych wbudowanych funkcji XPath

Kategoria	Składnia	Opis	Przykład	Wyjaśnienie przykładu
Atrybuty	`contains(@`*`atrybut`*`, '`*`łańcuch znaków`*`')`	Sprawdza, czy atrybut zawiera łańcuch znaków.	`//a[contains (@href, 'github')]`	Odpowiada mu odnośnik, którego atrybut `href` zawiera wartość `github`.
	`startswith(@`*`atrybut`*`, '`*`łańcuch znaków`*`')`	Sprawdza, czy atrybut zaczyna się łańcuchem znaków.	`//a[startswith (@href, 'https')]`	Odpowiada mu odnośnik do strony HTTPS.
	`endswith(@`*`atrybut`*`, '`*`łańcuch znaków`*`')`	Sprawdza, czy atrybut kończy się łańcuchem znaków.	`//a[endswith (@href, "https")]`	Odpowiada mu odnośnik do pliku PDF.
Tekst	`text()='`*`łańcuch znaków`*`'`	Znajduje elementy na podstawie zawartości tekstowej.	`//*[text()="kliknij"]`	Odpowiadają mu wszystkie elementy z napisem `kliknij`
Następnik	`[`*`indeks`*`]`	Znajduje następniki (dzieci).	`//div/*[0]`	Odpowiada mu pierwsze dziecko elementu `<div>`.
Wartości logiczne	`or`	Logiczny operator `or`.	`//@type='submit' or @type='reset']`	Odpowiadają mu przyciski do wysyłania lub czyszczenia formularzy
	`and`	Logiczny operator `and`.	`//@type='submit' and @id=' my-button']`	Odpowiadają mu przyciski o określonym id.
	`not()`	Logiczny operator `not`.	`//@type='submit' and not(@id=' my-button')]`	Odpowiadają mu przyciski o id różnym od wskazanego.

Tabela 3.7. Podsumowanie najważniejszych wbudowanych funkcji XPath (ciąg dalszy)

Kategoria	Składnia	Opis	Przykład	Wyjaśnienie przykładu
Osie (używane do znajdowania względnych węzłów)	`following::element`	Węzły następujące bezpośrednio po bieżącym.	`//*[@type='text']//following::input`	Odpowiadają mu pola tekstowe występujące po pierwszym polu.
	`descendant::element`	Wybiera następniki bieżącego węzła (dzieci etc.) .	`//*[@id='my-id']//descendant::a`	Odpowiadają mu wszystkie odnośniki pochodne od wskazanego rodzica.
	`ancestor::element`	Wybiera poprzedniki bieżącego węzła (rodzic etc.) .	`//input[@id='myid']//ancestor::label`	Odpowiadają mu wszystkie etykiety poprzedzające dane pole tekstowe.
	`child::element`	Wybiera dzieci bieżącego węzła.	`//*[@id='my-id']//child::li`	Odpowiadają mu wszystkie listy pod danym węzłem.
	`preceding::element`	Wybiera wszystkie węzły, które występują przed bieżącym elementem.	`//*[@id='my-id']//preceding::input`	Odpowiadają mu wszystkie pola tekstowe przed danym węzłem.
	`following-sibling::element`	Wybiera węzły, które poprzedzają bieżący węzeł, ale są na tym samym poziomie drzewa.	`//*[@id='my-id']//following--sibling::input`	Odpowiada mu pole tekstowe obok (przed) danym węzłem.
	`parent::element`	Wybiera rodzica bieżącego elementu.	`//*[@id='my-id']//parent::div`	Odpowiada mu element `div` będący rodzicem danego węzła.

Przykład 3.12 pokazuje, jak używać lokalizatorów XPath do przycisków radio dostępnych na stronie z ćwiczeniami. Kod HTML tych przycisków wygląda następująco:

```
<input class="form-check-input" type="radio" name="my-radio" id="my-radio-1" checked>
<input class="form-check-input" type="radio" name="my-radio" id="my-radio-2">
```

Przykład 3.12. Test wykorzystujący złożone lokalizatory XPath

```
@Test
void testByXPathAdvanced() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    WebElement radio1 = driver
        .findElement(By.xpath("//*[@type='radio' and @checked]")); ❶
    assertThat(radio1.getAttribute("id")).isEqualTo("my-radio-1"); ❷
    assertThat(radio1.isSelected()).isTrue(); ❸

    WebElement radio2 = driver
        .findElement(By.xpath("//*[@type='radio' and not(@checked)]")); ❹
    assertThat(radio2.getAttribute("id")).isEqualTo("my-radio-2"); ❺
    assertThat(radio2.isSelected()).isFalse(); ❻
}
```

❶ Korzystam ze ścieżki XPath do znalezienia kliknięte go pola radio.

❷ Sprawdzam, że identyfikator `id` elementu odpowiada oczekiwaniom.

❸ Potwierdzam, że element jest zaznaczony.

❹ Korzystam ze ścieżki XPath do znalezienia niekliknięte go pola radio.

❺ Sprawdzam, że identyfikator `id` elementu odpowiada oczekiwaniom.

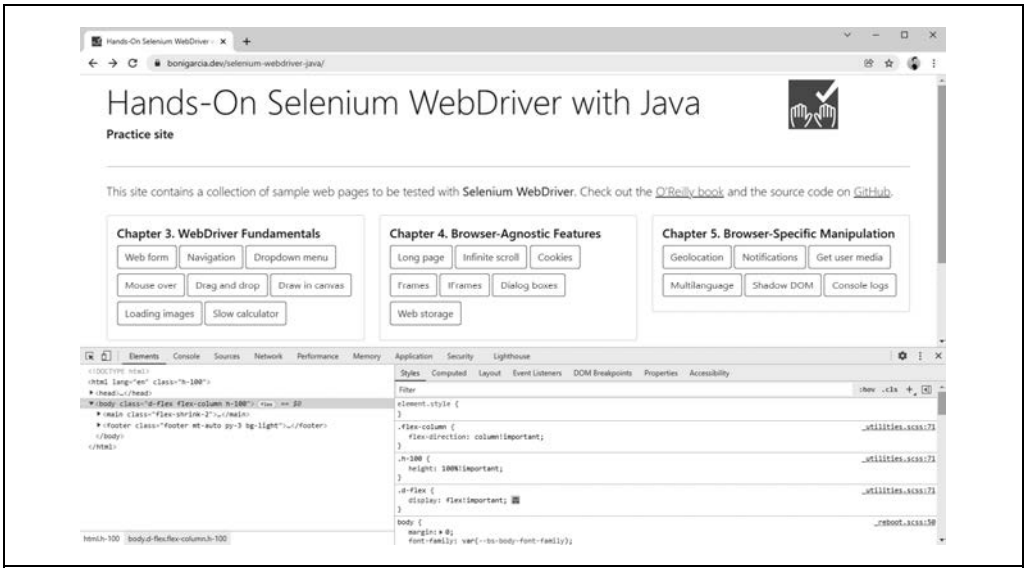
❻ Potwierdzam, że element nie jest zaznaczony.

Punkt „Której strategii używać?" w dalszej części rozdziału zawiera porównanie selektorów CSS i XPath oraz oferuje kilka rad dotyczących używania jednej i drugiej strategii.

Znajdowanie lokalizatorów na stronie internetowej

Jak wskazywałem w tabeli 1.4 w rozdziale 1., do wyboru jest szereg narzędzi, z których można skorzystać, by ułatwić sobie generowanie lokalizatorów na potrzeby testów WebDriver. Ten punkt pokazuje, jak używać głównych funkcjonalności wbudowanych we wszystkie główne przeglądarki, tj. Chrome DevTools (*https://developer.chrome.com/docs/devtools*) dla przeglądarek opartych na Chromium (np. Chrome czy Edge) i Firefox Developer Tools (*https://developer.mozilla.org/en-US/docs/Tools*) (dla Firefoksa).

Każde z tych narzędzi możesz uruchomić poprzez kliknięcie prawym przyciskiem myszy na tej części interfejsu UI strony internetowej, który chcesz przetestować, i wybranie opcji *Zbadaj*. Rysunek 3.3 przedstawia zrzut ekranu z konsoli Chrome DevTools uruchomionej u dołu okna przeglądarki (to położenie możesz zmienić, jeśli chcesz).

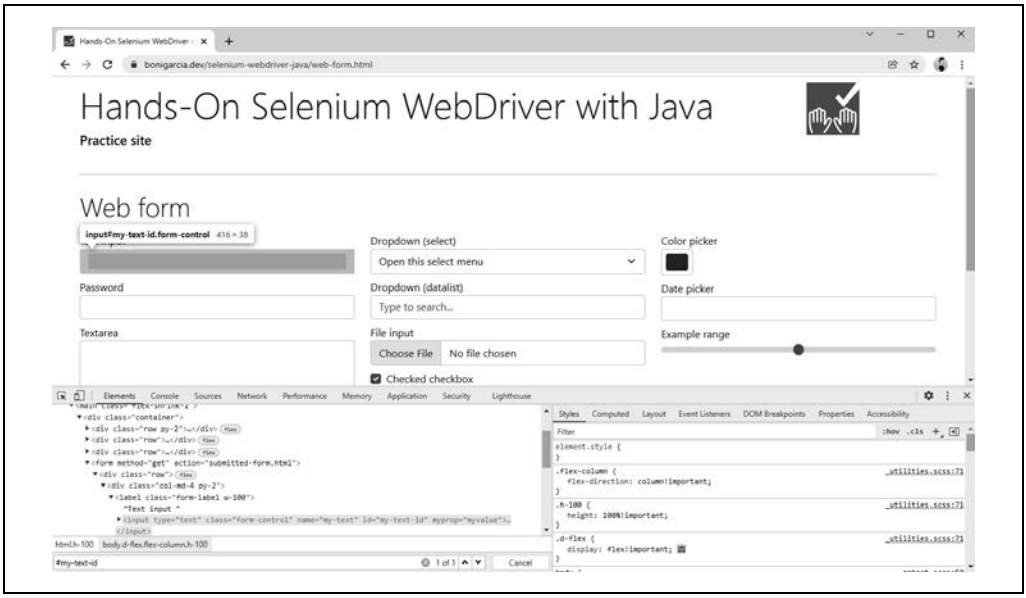


Rysunek 3.3. Wykorzystanie Chrome DevTools podczas nawigowania po stronie z ćwiczeniami

Narzędzia programistyczne oferują różne sposoby znajdowania elementów na stronie. Po pierwsze, korzysta się z selektora elementu poprzez kliknięcie ikonki (strzałki na tle prostokąta) w lewym górnym rogu okna narzędzi programistycznych. Następnie należy poruszać się myszą po ekranie, by zaznaczyć element i zbadać jego kod oraz atrybuty w wyświetlanym panelu.

W tym samym widoku można skorzystać z narzędzia do kopiowania selektorów CSS i XPath poprzez kliknięcie elementu prawym przyciskiem myszy i wybranie opcji *Skopiuj*. Ten mechanizm pozwala na uzyskanie pełnego selektora CSS lub XPath. To może stanowić pierwszy, szybki krok do wygenerowania lokalizatora, chociaż nie zalecam polegania na nim, ponieważ wygenerowane w ten sposób wskaźniki są zwykle mało stabilne (tj. silnie związane z bieżącym układem strony) i mało czytelne.

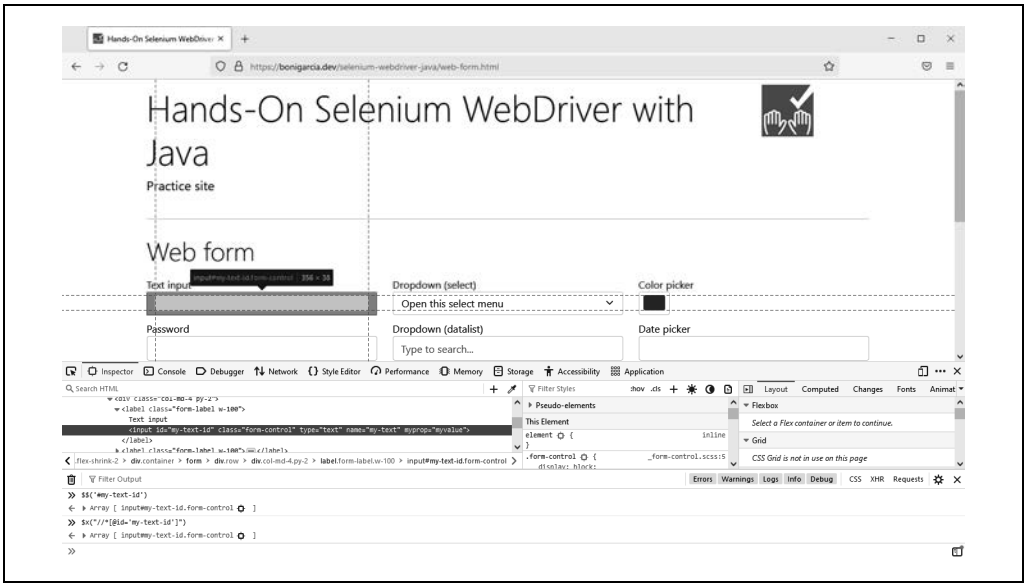
Aby stworzyć godne zaufania lokalizatory CSS czy XPath, trzeba zastanowić się nad cechami charakterystycznymi strony, nad którą się pracuje, i zbudować specjalne selektory na podstawie tej wiedzy. Również w tym zadaniu wesprą Cię narzędzia programistyczne przeglądarki. Możesz przycisnąć kombinację klawiszy *Ctrl*+F i szukać łańcucha znaków, selektora CSS czy XPath w Chrome DevTools. Rysunek 3.4 przedstawia przykład tej funkcjonalności w działaniu.



Rysunek 3.4. Poszukiwanie selektora CSS w Chrome DevTools

Zwróć uwagę, że korzystam ze strony z ćwiczeniami i wpisuję łańcuch znaków `#my-text-id`, odpowiadający elementowi z określonym identyfikatorem, na który wskazuje selektor CSS. Narzędzie DevTools znalazło ten element na stronie i podkreśliło go.

Podobne podejście można zastosować w przeglądarce Firefox. W tym celu należy skorzystać z panelu konsoli i wpisać `$$("css-selector")`, by znaleźć selektor CSS lub `$x("xpath-query")` w przypadku zapytań XPath. Rysunek 3.5 pokazuje, jak znaleźć pierwsze pole tekstowe na stronie z ćwiczeniami, wykorzystując selektor CSS i zapytanie XPath.



Rysunek 3.5. Poszukiwanie selektorów CSS i XPath w Firefox Developer Tools

Lokalizatory złożone

Interfejs API Selenium WebDriver posiada wiele klas pomocniczych umożliwiających łączenie różnych typów lokalizatorów, które już poznałeś. Do tych klas należy:

`ByIdOrName(String` *`identyfikatorLubNazwa`*`)`
: Szuka po identyfikatorze, a jeśli nie przynosi to rezultatu — po nazwie.

`ByChained(By...` *`lokalizatory`*`)`
: Szuka elementów w sekwencji (tj. drugi powinien znajdować się wewnątrz pierwszego itd.).

`ByAll(By...` *`lokalizatory`*`)`
: Szuka elementów, które odpowiadają kilku strategiom lokalizacyjnym (stosując do tych lokalizatorów logiczny operator `or`).

Przykład 3.13 przedstawia test wykorzystujący `ByIdOrName`. Test ten szuka pola wyboru pliku widocznego na stronie z ćwiczeniami. Zwróć uwagę, że pole to zawiera atrybut `name` (ale nie identyfikator `id`).

```
<input class="form-control" type="file" name="my-file">
```

Przykład 3.13. Test wykorzystujący złożony lokalizator oparty na atrybutach `name` oraz `id`

```
@Test
void testByIdOrName() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    WebElement fileElement = driver.findElement(new ByIdOrName("my-file")); ❶
    assertThat(fileElement.getAttribute("id")).isBlank(); ❷
    assertThat(fileElement.getAttribute("name")).isNotBlank(); ❸
}
```

❶ Korzystam z lokalizatora opartego na atrybucie `id` lub `name`.

❷ Sprawdzam, że element nie posiada atrybutu `id`.

❸ Weryfikuję, że ten sam element posiada atrybut `name`.

Przykład 3.14 przedstawia z kolei dwa testy, które ilustrują różnicę między `ByChained` oraz `ByAll`. Oba lokalizatory są ponownie używane na stronie z ćwiczeniami. Jeśli przyjrzysz się jej kodowi źródłowemu, zauważysz, że znajdują się tam cztery elementy `<div class="row">`, w tym jeden wewnątrz elementu `<form>`.

Przykład 3.14. Test wykorzystujący złożone lokalizatory ByChained i ByAll

```
@Test
void testByChained() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    List<WebElement> rowsInForm = driver.findElements(
        new ByChained(By.tagName("form"), By.className("row"))); ❶
    assertThat(rowsInForm.size()).isEqualTo(1); ❷
}

@Test
void testByAll() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    List<WebElement> rowsInForm = driver.findElements(
        new ByAll(By.tagName("form"), By.className("row"))); ❸
    assertThat(rowsInForm.size()).isEqualTo(5); ❹
}
```

❶ Korzystam z lokalizatora `ByChained`.

❷ Znajduję tylko jeden element `row`, ponieważ tylko jeden z nich znajduje się w formularzu.

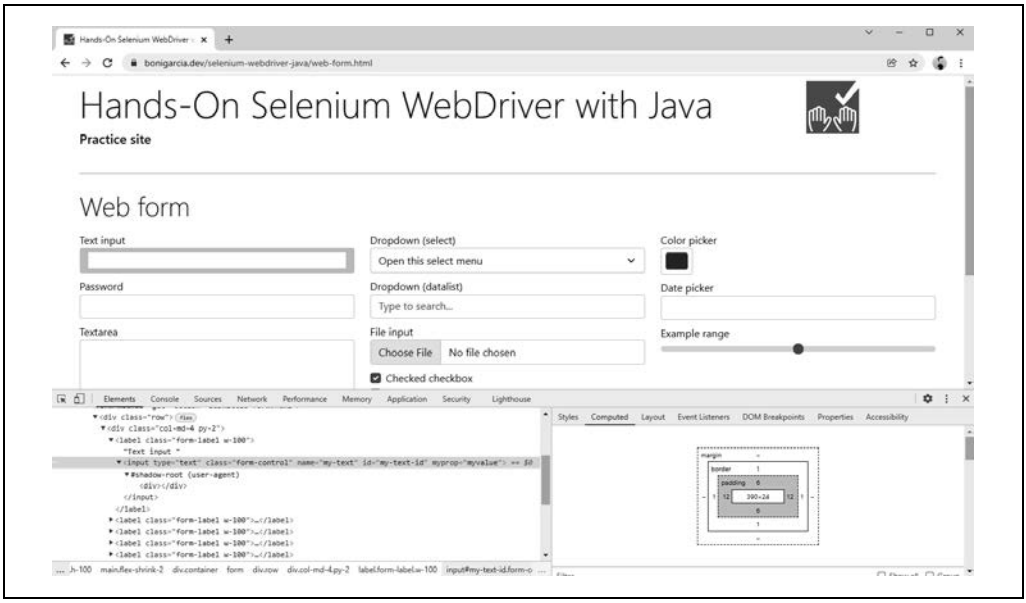
❸ Korzystam z lokalizatora `ByAll`.

❹ Znajduję pięć elementów, ponieważ temu lokalizatorowi odpowiada element `<form>` oraz cztery elementy `<div class="row">` znajdujące się na stronie.

Lokalizatory względne

Selenium WebDriver w wersji 4. udostępnia nowy sposób znajdowania elementów na stronach internetowych: **lokalizatory względne**. Te nowe lokalizatory mają za zadanie znajdować elementy względem pewnego znanego już węzła. Ta funkcjonalność opiera się na modelu pudełkowym CSS. Model ten zakłada, że każdy element dokumentu jest renderowany w prostokątnym polu („pudełku”). Rysunek 3.6 przedstawia przykład takiego pola dla konkretnego elementu w formularzu ćwiczeniowym.

Posługując się tym modelem, lokalizatory względne w interfejsie API Selenium WebDriver pozwalają na znajdowanie elementów w odniesieniu do pozycji innego elementu. W tym celu należy najpierw znaleźć pewien element przy użyciu jednej z tradycyjnych strategii (np. na podstawie identyfikatora id, nazwy, atrybutu itd.). Następnie trzeba określić typ lokalizatora uzyskany w przybliżeniu



Rysunek 3.6. Formularz ćwiczeniowy przedstawiający model pudełkowy elementu webowego

do oryginalnego elementu dzięki statycznej metodzie `with` z klasy `RelativeLocator`. W ten sposób uzyskuje się obiekt `RelativeBy`, który dziedziczy po abstrakcyjnej klasie `By`, wykorzystywanej w standardowych strategiach lokalizacyjnych. Obiekt `RelativeBy` udostępnia następujące metody do wyszukiwania względnego:

`above()`
: Znajduje element(y) powyżej oryginalnego elementu.

`below()`
: Znajduje element(y) poniżej oryginalnego elementu.

`near()`
: Znajduje element(y) w pobliżu oryginalnego elementu. Domyślną odległością, jaka traktowana jest jako bliska, jest sto pikseli. Ten lokalizator można przeciążać, by wskazać inną odległość.

`toLeftOf()`
: Znajduje element(y) na lewo od oryginalnego elementu.

`toRightOf()`
: Znajduje element(y) na prawo od oryginalnego elementu.

Przykład 3.15 przedstawia prosty test wykorzystujący lokalizatory względne. Podobnie jak wcześniej, posługuję się stroną z ćwiczeniami do zobrazowania tej funkcjonalności.

Przykład 3.15. Test używający lokalizatorów względnych

```
@Test
void testRelativeLocators() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");
```

```
        WebElement link = driver.findElement(By.linkText("Return to index")); ❶
        RelativeBy relativeBy = RelativeLocator.with(By.tagName("input")); ❷
        WebElement readOnly = driver.findElement(relativeBy.above(link)); ❸
        assertThat(readOnly.getAttribute("name")).isEqualTo("my-readonly"); ❹
    }
```

❶ Znajduję odnośnik o tekście `Return to index`.

❷ Wskazuję typ lokalizatora względnego, którym będzie nazwa znacznika — `input`.

❸ Korzystam z lokalizatora względnego do znalezienia elementu (którym powinno być pole `input`) powyżej oryginalnego elementu (tj. odnośnika).

❹ Weryfikuję, że element powyżej referencyjnego odnośnika jest polem do odczytu (patrz rysunek 3.2, by się upewnić).

Lokalizatory względne mogą być pomocne w znajdowaniu elementów w odniesieniu do pozycji innych węzłów. Z drugiej strony ta strategia może być bardzo narażona na zmiany w układzie strony. Przykładowo musisz uważać, używając lokalizatorów względnych na stronach responsywnych, gdzie układ będzie zależał od obszaru renderowania treści viewport.

Wyzwanie

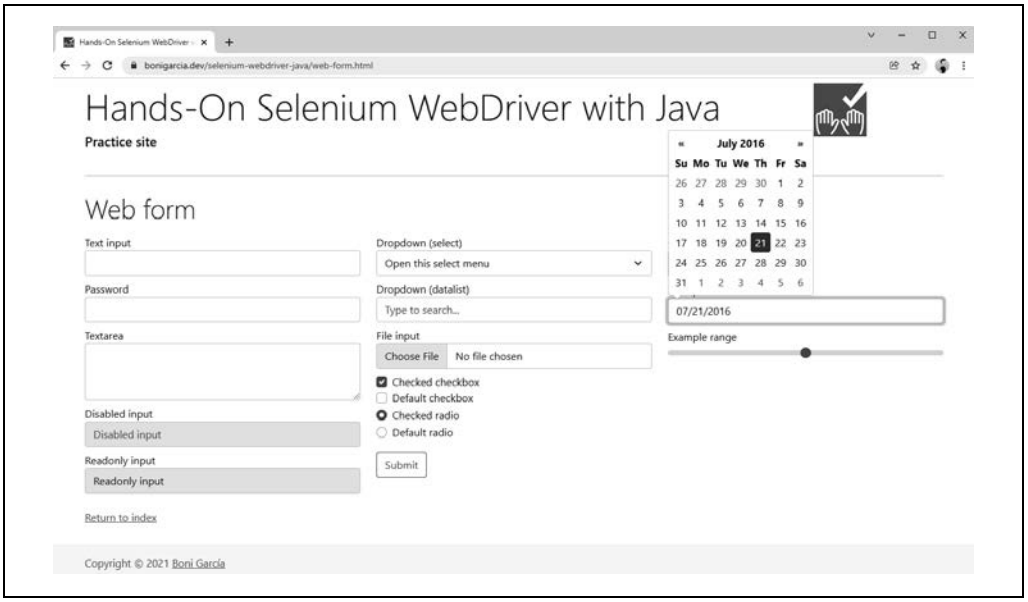
Przykłady prezentowane do tej pory były względnie nieskomplikowane. Spójrzmy więc na bardziej złożony przypadek użycia. Niedomyślnym elementem na stronie z ćwiczeniami jest *kalendarz*. Jak wskazuje nazwa, element ten pozwala na prosty wybór daty za pośrednictwem graficznego interfejsu użytkownika. Ponieważ biblioteką, jakiej używam na stronie z ćwiczeniami, jest Bootstrap (*https://getbootstrap.com*), zaimplementowałem wybór daty z wykorzystaniem *bootstrap-datepicker* (*https://github.com/uxsolutions/bootstrap-datepicker*). Jest on powiązany z polem typu `input`. Gdy użytkownik je kliknie, na stronie pojawia się kalendarz (patrz rysunek 3.7). Użytkownik może wtedy wybrać datę, poruszając się między dniami, miesiącami i latami.

Mam za zadanie przygotować automatyczny test w Selenium WebDriver, który wybiera ten sam dzień i miesiąc co dziś, ale rok temu za pomocą interfejsu GUI kalendarza. Przykład 3.16 przedstawia efekt mojej pracy.

Gdy będziesz czytał ten przykład, polecam wejść na stronę z ćwiczeniami (adres URL w kodzie) i za pomocą narzędzi programistycznych zbadać elementy wewnątrz kalendarza. Zwracaj uwagę na wykorzystanie różnych strategii lokalizacyjnych.

Przykład 3.16. Test interakcji z kalendarzem

```
@Test
void testDatePicker() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");
```



Rysunek 3.7. Kalendarz w formularzu na stronie z ćwiczeniami

```
// Pobierz bieżącą datę z zegara systemowego ❶
LocalDate today = LocalDate.now();
int currentYear = today.getYear();
int currentDay = today.getDayOfMonth();

// Kliknij pole, by otworzyć kalendarz ❷
WebElement datePicker = driver.findElement(By.name("my-date"));
datePicker.click();

// Kliknij bieżący miesiąc znaleziony na podstawie tekstu ❸
WebElement monthElement = driver.findElement(By.xpath(
    String.format("//th[contains(text(),'%d')]", currentYear)));
monthElement.click();

// Kliknij strzałkę w lewo znalezioną dzięki lokalizatorowi względnemu ❹
WebElement arrowLeft = driver.findElement(
    RelativeLocator.with(By.tagName("th")).toRightOf(monthElement));
arrowLeft.click();

// Kliknij ten sam miesiąc w zeszłym roku ❺
WebElement monthPastYear = driver.findElement(RelativeLocator
    .with(By.cssSelector("span[class$=focused]")).below(arrowLeft));
monthPastYear.click();

// Kliknij dzisiejszy dzień tego miesiąca ❻
WebElement dayElement = driver.findElement(By.xpath(String.format(
    "//td[@class='day' and contains(text(),'%d')]", currentDay)));
dayElement.click();
```

```
        // Pobierz ostateczną datę z pola input ❼
        String oneYearBack = datePicker.getAttribute("value");
        log.debug("Final date in date picker: {}", oneYearBack);

        // Sprawdź, że oczekiwana data jest równa tej wybranej w kalendarzu ❽
        LocalDate previousYear = today.minusYears(1);
        DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("MM/dd/yyyy");
        String expectedDate = previousYear.format(dateFormat);
        log.debug("Expected date: {}", expectedDate);

        assertThat(oneYearBack).isEqualTo(expectedDate);
    }
```

❶ Pobierz bieżącą datę z zegara systemowego. W tym celu korzystam ze standardowego interfejsu API Javy.

❷ Kliknij pole, by otworzyć kalendarz. Korzystam tu z lokalizatora (`By.name("my-date")`).

❸ Kliknij bieżący miesiąc znaleziony na podstawie tekstu. W tym celu posługuję się ścieżką XPath. Po wykonaniu tego kroku w interfejsie graficznym kalendarza pojawiają się pozostałe miesiące roku.

❹ Kliknij strzałkę w lewo znalezioną dzięki lokalizatorowi względnemu (tj. na lewo od elementu miesiąca). Po wykonaniu tego kroku kalendarz przesuwa się do poprzedniego roku.

❺ Kliknij ten sam miesiąc w zeszłym roku. W tym miejscu używam selektora CSS.

❻ Kliknij dzisiejszy dzień tego miesiąca. Na tym etapie posługuję się ścieżką XPath. Po wykonaniu tego kroku data jest wybrana i wartość pojawia się w polu tekstowym.

❼ Pobierz ostateczną datę z pola `input`. Korzystam z podstawowego lokalizatora opartego na atrybucie.

❽ Sprawdź, czy oczekiwana data jest równa tej wybranej w kalendarzu. Obliczam oczekiwaną datę dzięki standardowej bibliotece Javy i, jak zwykle, używam AssertJ do asercji.

Której strategii używać?

W tym punkcie przyglądam się różnym opcjom, jakie interfejs API Selenium WebDriver udostępnia w celu znajdowania elementów na stronach internetowych. Temat ten jest jedną z najbardziej podstawowych praktyk w automatyzacji pracy przeglądarki za pomocą Selenium WebDriver. Być może zadajesz sobie pytanie: „Której strategii używać?”. Jak powiedziałby dr Alfred Lanning (bohater powieści i filmu *Ja, robot*): „To jest właściwe pytanie, detektywie”. Moim zdaniem jest także trudne i nie ma na nie prostej odpowiedzi. Czyli mógłbym powiedzieć: „To zależy”. W tym punkcie przedstawiam kilka rad dotyczących wyboru najlepszej strategii lokalizacyjnej w powszechnych przypadkach użycia. Zaczynam od tabeli 3.8, która porównuje różne strategie.

Jak widzisz, selektory CSS i XPath mają wspólne wady, zalety i przypadki użycia. Czy znaczy to, że te strategie są takie same? Nie! Obie dają duże możliwości i pozwalają na tworzenie skomplikowanych lokalizatorów. Niemniej jednak istnieją między nimi poważne różnice. Podsumowuje je tabela 3.9.

Tabela 3.8. Zalety, wady i zwyczajowe zastosowania różnych strategii lokalizacyjnych

Lokalizator	Zalety	Wady	Zwyczajowe zastosowania
Atrybut (identyfikator, nazwa, klasa)	Proste w użyciu	Atrybuty nie zawsze są dostępne	Elementy, w których atrybuty te są niezmienne (tj. nie zmieniają się dynamicznie)
Tekst odnośnika (pełny lub cząstkowy)	Proste w użyciu	Ma zastosowanie tylko do odnośników	Tekst odnośnika
Nazwa znacznika	Proste w użyciu	Trudno wybrać konkretny element, gdy znacznik pojawia się na stronie wielokrotnie	Unikalna nazwa znacznika lub stała pozycja węzła w drzewie DOM
Selektor CSS lub XPath	Dają duże możliwości	Pisanie solidnych selektorów nie jest proste	Skomplikowane lokalizatory
Lokalizatory złożone	Proste sposoby łączenia istniejących lokalizatorów	Ograniczony do konkretnych sytuacji	Poszukiwanie identyfikatora lub nazwy (`ByIdOrName`), elementy zagnieżdżone (`ByChained`), łączenie wielu strategii jednocześnie (`ByAll`)
Lokalizatory względne	Podejście przypominające język naturalny	Trzeba je łączyć z innymi lokalizatorami	Elementy położone względem innych, znanych elementów (np. powyżej, poniżej, obok)

Tabela 3.9. Wybrane różnice między selektorami CSS i XPath

XPath	CSS
Ścieżki XPath pozwalają na lokalizowanie w obu kierunkach, tj. przechodzenie od rodzica do dziecka i od dziecka do rodzica.	Selektory CSS pozwalają na lokalizowanie w jednym kierunku, tj. od rodzica do dziecka.
Ścieżki XPath są wolniejsze pod względem wydajności.	Selektory CSS są szybsze niż XPath.
Ścieżki XPath pozwalają na wskazywanie widocznego na ekranie tekstu dzięki funkcji `text()`.	Selektory CSS nie umożliwiają znajdowania elementów na podstawie tekstu.

Aby lepiej ukazać różnice między selektorami XPath i CSS, tabela 3.10 porównuje konkretne lokalizatory z obu strategii.

Tabela 3.10. Przykłady porównań selektorów XPath i CSS

Lokalizator	XPath	CSS
Każdy element	`//*`	`*`
Każdy element `<div>`	`//div`	`div`
Element o identyfikatorze	`//*[@id='my-id']`	`#my-id`
Element o klasie	`//*[contains(@class='my-class')]`	`.my-class`
Element o atrybucie	`//*[@atrybut]`	`*[atrybut]`
Tekst w elemencie `<div>`	`//div[text()='search-string']`	Niedostępne
Pierwsze dziecko elementu `<div>`	`//div/*[1]`	`div>*:first-child`
Wszystkie elementy `<div>` z dzieckiem w postaci odnośnika	`//div[a]`	Niedostępne
Element następujący po `<div>`	`//div/following-sibling::*[1]`	`div + *`
Poprzednik elementu `<div>`	`//div/preceding-sibling::*[1]`	Niedostępne

Jak widać, ścieżki XPath stanowią najbardziej ogólną strategię. Niemniej jednak w niektórych przypadkach selektory CSS mają bardziej przyjazną składnię (np. znajdowanie na podstawie klasy czy id) i lepszą wydajność.

Działania z klawiaturą

Jak zaznaczałem w tabeli 3.3, dwie główne metody obiektów `WebDriver` pozwalają na symulowanie działań użytkownika na klawiaturze. Są to `sendKeys()` i `clear()`. Przykład 3.17 przedstawia test wykorzystujący te metody.

Przykład 3.17. Test naśladujący zdarzenia klawiaturowe

```
@Test
void testSendKeys() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    WebElement inputText = driver.findElement(By.name("my-text")); ❶
    String textValue = "Hello World";
    inputText.sendKeys(textValue); ❷
    assertThat(inputText.getAttribute("value")).isEqualTo(textValue); ❸

    inputText.clear(); ❹
    assertThat(inputText.getAttribute("value")).isEmpty(); ❺
}
```

❶ Korzystam ze strony z ćwiczeniami, gdzie znajduję element o nazwie `my-text`.

❷ Naśladuję pisanie na klawiaturze poprzez użycie metody `sendKeys()`.

❸ Oceniam, czy wartość pola jest zgodna z oczekiwaniami.

❹ Resetuję wartość poprzez metodę `clear()`.

❺ Oceniam, czy wartość pola jest pusta.

Nieautomatyczne oczekiwanie w metodach kończących test

Jeśli przyjrzysz się pełnemu kodowi niektórych testów w repozytorium z przykładami, zauważysz, że zawierają one pauzę `Thread.sleep()` w metodach kończących test. Jak wyjaśnię w podrozdziale „Strategie czekania”, taki rodzaj oczekiwania jest uważany za *złą praktykę* (cechę kodu źródłowego, która może prowadzić do niepożądanych efektów). Niemniej jednak pozwoliłem sobie korzystać z niego w niektórych przypadkach w celach szkoleniowych, tj. by ułatwić ręczne przejrzenie przeglądarki przy uruchamianiu testów lokalnie. Zalecam usunięcie tych linii, jeśli zamierzasz wykorzystać te testy jako podstawę prawdziwego zestawu.

Wysyłanie plików

Istnieje kilka przypadków zastosowania, w których będziesz musiał naśladować działania z klawiaturą podczas pracy nad automatyzacją w Selenium WebDriver. Pierwszy z nich to wysyłanie plików.

Standardowy mechanizm wysyłania plików w aplikacjach webowych obejmuje wykorzystanie elementów `<input>` o typie `"file"`. Formularz ćwiczeniowy zawiera taki przykładowy element:

```
<input class="form-control" type="file" name="my-file">
```

Interfejs API Selenium WebDriver nie udostępnia możliwości obsługi plików jako danych wejściowych. Zamiast tego należy traktować elementy do wysyłania plików jak zwykłe pola tekstowe, więc potrzebne jest naśladowanie wprowadzania w nie znaków. Chodzi przede wszystkim o wpisanie absolutnej ścieżki do wysyłanego pliku. Przykład 3.18 przedstawia taki test.

Przykład 3.18. Test wysyłający plik

```
@Test
void testUploadFile() throws IOException {
    String initUrl = "https://bonigarcia.dev/selenium-webdriver-java/web-form.html";
    driver.get(initUrl);

    WebElement inputFile = driver.findElement(By.name("my-file")); ❶

    Path tempFile = Files.createTempFile("tempfiles", ".tmp"); ❷
    String filename = tempFile.toAbsolutePath().toString();
    log.debug("Using temporal file {} in file uploading", filename);
    inputFile.sendKeys(filename); ❸

    driver.findElement(By.tagName("form")).submit(); ❹
    assertThat(driver.getCurrentUrl()).isNotEqualTo(initUrl); ❺
}
```

❶ Znajduję pole dzięki strategii `By.name`.

❷ Tworzę plik tymczasowy, posługując się standardową biblioteką Javy.

❸ Wpisuję ścieżkę absolutną do niego w polu.

❹ Wysyłam formularz.

❺ Weryfikuję, że URL strony z rezultatem (zdefiniowana jako atrybut `action` formularza) jest różna od początkowego adresu strony.

Ścieżka pliku, jaką wpisuje się w polu, powinna odpowiadać zasobowi istniejącemu na maszynie uruchamiającej test. Jeśli tak nie będzie, test zwróci błąd `InvalidArgument` ↪`Exception`. Zajrzyj do podrozdziału „Wyjątki WebDriver” w rozdziale 5. po więcej informacji na temat wyjątków.

Podczas wysyłania pliku do zdalnej przeglądarki (jak wyjaśniam w rozdziale 6.) należy załadować plik jawnie z lokalnego systemu plików. Poniższa linia pokazuje, jak wskazać mechanizm wykrywania pliku:

```
((RemoteWebDriver) driver).setFileDetector(new LocalFileDetector());
```

Suwaki

Podobna sytuacja ma miejsce, gdy zajmujesz się polami formularza `<input type="range">`. Te elementy pozwalają użytkownikowi wybrać liczbę z zakresu za pomocą suwaka graficznego. Przykład możesz znaleźć na stronie z ćwiczeniami:

```
<input type="range" class="form-range" name="my-range" min="0" max="10" step="1" value="5">
```

Podobnie jak w przypadku plików, interfejs API Selenium WebDriver nie udostępnia specjalnych metod do obsługi tych pól. Można jednak wejść z nimi w interakcje poprzez naśladowania działań z klawiatury w Selenium WebDriver. Przykład 3.19 przedstawia test obejmujący takie działanie.

Przykład 3.19. Test wybierający liczbę za pomocą suwaka

```
@Test
void testSlider() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    WebElement slider = driver.findElement(By.name("my-range"));
    String initValue = slider.getAttribute("value");
    log.debug("The initial value of the slider is {}", initValue);

    for (int i = 0; i < 5; i++) {
        slider.sendKeys(Keys.ARROW_RIGHT); ❶
    }

    String endValue = slider.getAttribute("value");
    log.debug("The final value of the slider is {}", endValue);
    assertThat(initValue).isNotEqualTo(endValue); ❷
}
```

❶ Do elementu suwaka wysyłam zdarzenie kliknięcia klawisza. Posługuję się klasą `Keys` dostępną w interfejsie API Selenium WebDriver, która ma za zadanie obsługiwać specjalne znaki klawiaturowe. Konkretnie wysyłam do suwaka zdarzenie kliknięcia strzałki w prawo, a w konsekwencji suwak przesuwa się w tym samym kierunku, czyli wartość rośnie.

❷ Dokonuję asercji, że wybrana wartość jest inna niż oryginalna pozycja suwaka.

Działania myszką

Oprócz klawiatury urządzeniem wejścia używanym do interakcji z aplikacjami webowymi bywa myszka. Po pierwsze, pojedyncze kliknięcie (kliknięcie lewym przyciskiem myszy czy po prostu kliknięcie) jest naśladowane przez API Selenium WebDriver przez metodę `click()`, będącą jedną z metod obiektów `WebElement`. W tym podrozdziale przedstawiam dwa typowe przypadki użycia tej funkcjonalności: nawigowanie i interakcje z polami wyboru czy przyciskami radio w formularzach.

Do innych popularnych działań należy: kliknięcie prawym przyciskiem myszki (otwieranie menu kontekstowego), podwójne kliknięcie, ruch kursora, przeciąganie elementów czy skierowanie myszki na element. Selenium WebDriver pozwala na naśladowanie tych działań dzięki pomocniczej klasie `Actions`. W kolejnym punkcie poświęcę jej więcej miejsca. Wreszcie przewijanie strony jest możliwe dzięki wykonywaniu skryptów JavaScript. Omawiam tę funkcjonalność w podrozdziale „Wykonywanie skryptów JavaScript” w rozdziale 4.

Nawigacja

Przykład 3.20 przedstawia test implementujący automatyczną nawigację po stronie z wykorzystaniem Selenium WebDriver. Test znajduje odnośniki na podstawie selektorów XPath i klika je poprzez wywołanie metody `click()`. Na koniec test czyta tekstową zawartość elementu `body` i weryfikuje, że zawiera ona oczekiwany łańcuch znaków.

Przykład 3.20. Test nawigacji poprzez klikanie odnośników

```
@Test
void testNavigation() {
    driver.get("https://bonigarcia.dev/selenium-webdriver-java/");

    driver.findElement(By.xpath("//a[text()='Navigation']")).click();
    driver.findElement(By.xpath("//a[text()='Next']")).click();
    driver.findElement(By.xpath("//a[text()='3']")).click();
    driver.findElement(By.xpath("//a[text()='2']")).click();
    driver.findElement(By.xpath("//a[text()='Previous']")).click();

    String bodyText = driver.findElement(By.tagName("body")).getText();
    assertThat(bodyText).contains("Lorem ipsum");
}
```

Pola wyboru i przyciski radio

Przykład 3.21 przedstawia kolejne podstawowe zastosowanie metody `click()` do pracy z polami wyboru i przyciskami radio. Aby zweryfikować oczekiwany stan elementów po kliknięciu, korzystam z asercji opartej na rezultacie metody `isSelected()`.

Przykład 3.21. Test działań na polach wyboru i przyciskach radio

```
@Test
void testNavigation() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");

    WebElement checkbox2 = driver.findElement(By.id("my-checkbox-2"));
    checkbox2.click();
    assertThat(checkbox2.isSelected()).isTrue();

    WebElement radio2 = driver.findElement(By.id("my-radio-2"));
    radio2.click();
    assertThat(radio2.isSelected()).isTrue();
}
```

Gesty użytkownika

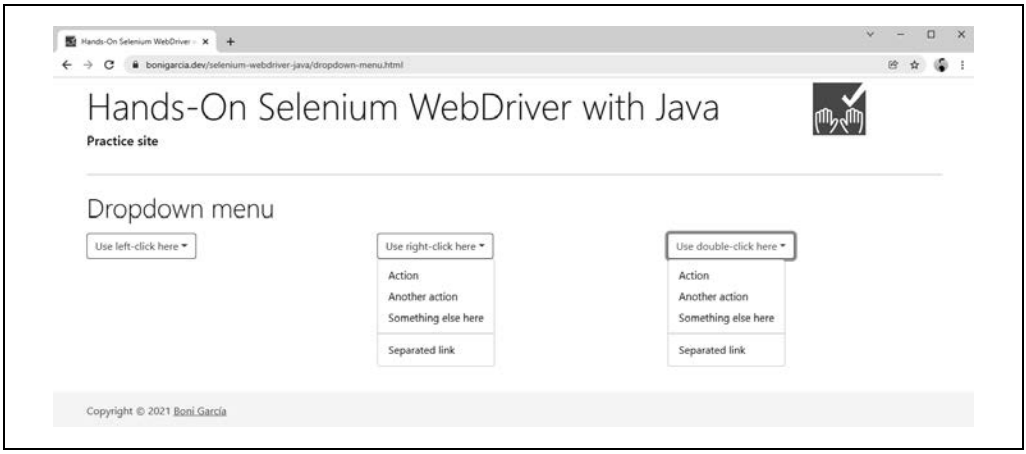
Selenium WebDriver udostępnia klasę `Actions` — potężne narzędzie do automatyzacji różnych działań użytkownika, zarówno z klawiaturą, jak i z myszką. Klasa ta opiera się na wzorcu Budowniczy. Dzięki temu możesz łączyć wiele metod w łańcuch (tj. wiele działań) i wykonywać je, na końcu wywołując metodę `build()`. Tabela 3.11 podsumowuje publiczne metody dostępne w tej klasie. Omawiam je szczegółowo w przykładach w kolejnych punktach.

Tabela 3.11. Metody klasy Actions

Metoda	Opis
`keyDown(CharSequence klawisz)` `keyDown(WebElement cel, CharSequence klawisz)`	Wysyła pojedynczy klawisz (może to być specjalny znak z klasy `Keys`) z danej pozycji (lub elementu). Klawisz pozostaje wciśnięty do momentu wywołania `keyUp()`.
`keyUp(CharSequence klawisz)`	Zwalnia klawisz wciśnięty uprzednio za pomocą `keyDown()`.
`keyUp(WebElement cel, CharSequence klawisz)`	
`sendKeys(CharSequence... klawisze)` `sendKeys(WebElement cel, CharSequence... klawisze)`	Wysyła sekwencję klawiszy z danej pozycji (lub elementu). Ta metoda różni się od `WebElement#sendKeys(CharSequence...)` na dwa sposoby: 1) klawisze modyfikujące (np. `Keys.CONTROL`, `Keys.SHIFT`) nie są jawnie zwalniane. 2) Fokus nie wraca automatycznie do elementu, więc klikanie `Keys.TAB` powinno działać.
`clickAndHold()`	Klika bez zwalniania bieżącej pozycji (lub środka elementu).
`clickAndHold(WebElement cel)`	
`release()`	Zwalnia lewy przycisk myszy kliknięty wcześniej poprzez `clickAndHold()`.
`release(WebElement cel)`	
`click()`	Klika bieżącą pozycję (lub element).
`click(WebElement cel)`	
`doubleClick()`	Klika podwójnie bieżącą pozycję (lub element).
`doubleClick(WebElement cel)`	
`contextClick()`	Klika prawym przyciskiem myszy bieżącą pozycję (lub element).
`contextClick(WebElement cel)`	
`moveToElement(WebElement cel)`	Przesuwa kursor myszy do środka elementu (lub przesuwa o dany odstęp).
`moveToElement(WebElement cel, int xOdstęp, int yOdstęp)`	
`moveByOffset(int xOdstęp, int yOdstęp)`	Przesuwa kursor myszy z danej pozycji (domyślnie `0, 0`) o wskazany odstęp.
`dragAndDrop(WebElement źródło, WebElement cel)` `dragAndDropBy(WebElement źródło, int xOdstęp, int yOdstęp)`	Działanie to obejmuje trzy kroki. 1. Kliknięcie i przytrzymanie w środku (lub z przesunięciem o wskazany odstęp) lokalizacji elementu źródłowego. 2. Przesunięcie myszy na lokalizację docelową. 3. Zwolnienie przycisku myszy.
`pause(long pauza)` `pause(Duration okres)`	Wykonuje przerwę w łańcuchu działań (jej długość jest określona w milisekundach lub jako obiekt Javy `Duration`).
`build()`	Generuje złożoną akcję zawierającą wszystkie poprzednie instrukcje.
`perform()`	Wykonuje złożoną akcję.

Kliknięcie kontekstowe i podwójne kliknięcie

Na stronie z ćwiczeniami znajdziesz widok zawierający trzy menu rozwijane (patrz rysunek 3.8). Pierwsze z nich rozwija się, gdy kliknie się przycisk, drugie — gdy kliknie się je prawym przyciskiem, a trzeci — gdy kliknie się podwójnie. Przykład 3.22 przedstawia test wykorzystujący tę stronę w celu naśladowania ruchów użytkownika za pomocą klasy WebDriver `Actions`.



Rysunek 3.8. Strona ćwiczeniowa z menu rozwijanymi

Przykład 3.22. Test implementujący kliknięcie kontekstowe i podwójne

```
@Test
void testContextAndDoubleClick() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/dropdown-menu.html");
    Actions actions = new Actions(driver);

    WebElement dropdown2 = driver.findElement(By.id("my-dropdown-2"));
    actions.contextClick(dropdown2).build().perform(); ❶
    WebElement contextMenu2 = driver.findElement(By.id("context-menu-2"));
    assertThat(contextMenu2.isDisplayed()).isTrue(); ❷

    WebElement dropdown3 = driver.findElement(By.id("my-dropdown-3"));
    actions.doubleClick(dropdown3).build().perform(); ❸
    WebElement contextMenu3 = driver.findElement(By.id("context-menu-3"));
    assertThat(contextMenu3.isDisplayed()).isTrue(); ❹
}
```

❶ Korzystam z metody `contextClick()` w odniesieniu do środkowego menu rozwijanego.

❷ Weryfikuję, czy menu wyświetla się prawidłowo.

❸ Korzystam z metody `doubleClick()` w odniesieniu do menu rozwijanego po prawej stronie.

❹ Weryfikuję, czy menu wyświetla się prawidłowo.

Przesunięcie myszki

Drugi przykład obsługi `Actions` obejmuje implementację przesunięcia myszki nad element na stronie z ćwiczeniami. Strona ta zawiera cztery obrazki. Każdy z nich wyświetla etykietę tekstową, gdy kursor myszy znajdzie się nad obszarem grafiki. Przykład 3.23 przedstawia test dotyczący tej strony. Rysunek 3.9 prezentuje z kolei stronę, gdy kursor myszy znajduje się nad pierwszym obrazkiem.

Przykład 3.23. Test implementujący ruchy myszką

```
@Test
void testMouseOver() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/mouse-over.html");
    Actions actions = new Actions(driver);

    List<String> imageList = Arrays.asList("compass", "calendar", "award", "landscape");
    for (String imageName : imageList) { ❶
        String xpath = String.format("//img[@src='img/%s.png']", imageName);
        WebElement image = driver.findElement(By.xpath(xpath)); ❷
        actions.moveToElement(image).build().perform(); ❸

        WebElement caption = driver.findElement(
            RelativeLocator.with(By.tagName("div")).near(image)); ❹

        assertThat(caption.getText()).containsIgnoringCase(imageName); ❺
    }
}
```

❶ W pętli przechodzę przez listę łańcuchów znaków, by znaleźć na stronie cztery obrazki.

❷ Korzystam z selektora XPath, by znaleźć każdy element `<img>`.

❸ Używam metody `moveToElement()`, by przesunąć kursor myszy na środek obrazka.

❹ Posługuję się lokalizatorem względnym, by znaleźć pojawiającą się etykietę.

❺ Dokonuję asercji, by potwierdzić, że tekst jest zgodny z oczekiwaniami.

Rysunek 3.9. Strona z ćwiczeniami zawierająca grafiki, których etykiety pojawiają się przy odpowiedniej pozycji kursora

Przeciąganie elementów

Przykład 3.24 przedstawia użycie mechanizmu przeciągania elementów. Test ten korzysta ze strony z ćwiczeniami widocznej na rysunku 3.10.

Przykład 3.24. Test wykorzystujący mechanizm przeciągania elementów

```
@Test
void testDragAndDrop() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/drag-and-drop.html");
    Actions actions = new Actions(driver);

    WebElement draggable = driver.findElement(By.id("draggable")); ❶
    int offset = 100;
    Point initLocation = draggable.getLocation();
    actions.dragAndDropBy(draggable, offset, 0)
        .dragAndDropBy(draggable, 0, offset)
        .dragAndDropBy(draggable, -offset, 0)
        .dragAndDropBy(draggable, 0, -offset).build().perform(); ❷
    assertThat(initLocation).isEqualTo(draggable.getLocation()); ❸

    WebElement target = driver.findElement(By.id("target")); ❹
    actions.dragAndDrop(draggable, target).build().perform(); ❺
    assertThat(target.getLocation()).isEqualTo(draggable.getLocation()); ❻
}
```

❶ Znajduję element o identyfikatorze `draggable`.

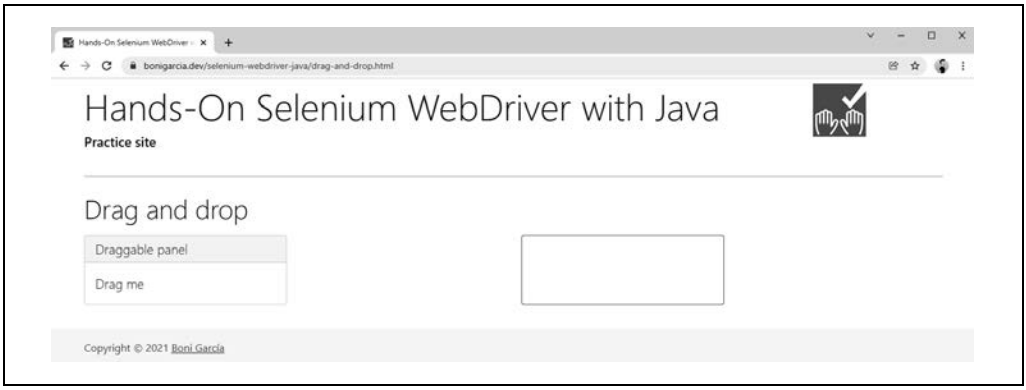
❷ Korzystam z metody `dragAndDropBy()`, by przesunąć element o określoną liczbę pikseli (100) cztery razy (na prawo, na dół, na lewo i w górę).

❸ Sprawdzam, czy pozycja elementu jest taka sama jak na początku.

❹ Znajduję drugi element (którego nie da się ruszyć na stronie).

❺ Korzystam z metody `dragAndDrop()`, by przesunąć pierwszy element na drugi.

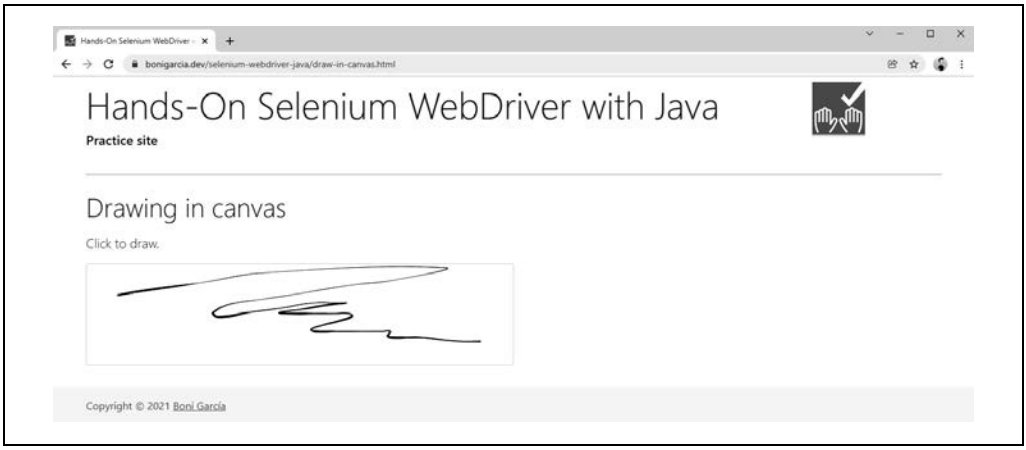
❻ Sprawdzam, czy pozycja obu elementów jest taka sama.



Rysunek 3.10. Strona z ćwiczeniami zawierająca element do przeciągania

Kliknięcie i przytrzymanie

Poniższy przykład przedstawia złożone działania użytkownika obejmujące kliknięcie i przytrzymanie. W tym celu testuję stronę widoczną na rysunku 3.11.



Rysunek 3.11. Strona z ćwiczeniami zawierająca pole do rysowania

Strona ta korzysta z wolnoźródłowej biblioteki JavaScript Signature Pad (*https://github.com/szimek/signature_pad*) do umieszczania odręcznych podpisów na płótnach HTML za pomocą myszki. Przykład 3.25 przedstawia test wykorzystujący tę funkcjonalność.

Przykład 3.25. Test rysujący okrąg na płótnie

```
@Test
void testClickAndHold() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/draw-in-canvas.html");
    Actions actions = new Actions(driver);

    WebElement canvas = driver.findElement(By.tagName("canvas")); ❶
    actions.moveToElement(canvas).clickAndHold(); ❷

    int numPoints = 10;
    int radius = 30;
    for (int i = 0; i <= numPoints; i++) { ❸
        double angle = Math.toRadians(360 * i / numPoints);
        double x = Math.sin(angle) * radius;
        double y = Math.cos(angle) * radius;
        actions.moveByOffset((int) x, (int) y); ❹
    }

    actions.release(canvas).build().perform(); ❺
}
```

❶ Znajduję element płótna dzięki nazwie znacznika.

❷ Przesuwam kursor do niego za pomocą metody `moveToElement()`, a następnie rozpoczynam działanie kliknięcia i przytrzymania `clickAndHold()` (w celu rysowania na płótnie) w ciągu akcji.

❸ Przechodzę przez ustaloną liczbę punktów, posługując się równaniem do znalezienia punktów na obwodzie.

❹ Wykorzystuję punkty na obwodzie (`x`, `y`) do przesuwania myszki o odstęp (`moveByOffset()`). Ponieważ już od poprzedniego kroku element jest kliknięty, powstała w ten sposób złożona akcja będzie przesuwać kursor w czasie, gdy przycisk jest wciśnięty.

❺ Zwalniam kliknięcie, buduję ciąg działań i wywołuję jego wykonanie. W konsekwencji na płótnie powinien pojawić się okrąg.

Mechanizm kopiuj-wklej

Ostatni przykład naśladowania ruchów użytkownika automatyzuje jedną z najbardziej popularnych czynności z użyciem klawiatury, tj. kopiowanie i wklejanie zawartości. W tym celu wykorzystuję formularz dostępny na stronie z ćwiczeniami. Przykład 3.26 przedstawia test naśladujący wykonanie tej powszechnej operacji.

Przykład 3.26. Test naśladujący mechanizm kopiowania i wklejania zawartości

```
@Test
void testCopyAndPaste() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/web-form.html");
    Actions actions = new Actions(driver);

    WebElement inputText = driver.findElement(By.name("my-text")); ❶
    WebElement textarea = driver.findElement(By.name("my-textarea"));

    Keys modifier = SystemUtils.IS_OS_MAC ? Keys.COMMAND : Keys.CONTROL; ❷
    actions.sendKeys(inputText, "hello world").keyDown(modifier)
        .sendKeys(inputText, "a").sendKeys(inputText, "c")
        .sendKeys(textarea, "v").build().perform(); ❸

    assertThat(inputText.getAttribute("value"))
        .isEqualTo(textarea.getAttribute("value")); ❹
}
```

❶ Znajduję dwa elementy: pole do wprowadzania tekstu i pole z tekstem.

❷ Posługuję się różnymi klawiszami modyfikującymi do wysłania kombinacji *Ctrl+C* (w przypadku systemu operacyjnego Windows i Linux) oraz *Cmd+C* (w przypadku macOS) do kopiowania. Aby określić system operacyjny, korzystam z klasy `SystemUtils`, dostępnej w wolnoźródłowej bibliotece Apache Commons IO (*https://commons.apache.org/proper/commons-io*) (ta zależność jest używana jako przechodnia w projektach Maven/Gradle).

❸ Implementuję łańcuch działań złożony z następujących kroków:

1. Wpisuję łańcuch znaków w polu do tekstu.
2. Przyciskam klawisz modyfikujący (*Ctrl* lub *Cmd*, w zależności od systemu operacyjnego). Pamiętaj, że klawisz pozostanie wciśnięty, dopóki jawnie go nie zwolnię.
3. Wysyłam klawisz *a* do pola do tekstu. Ponieważ modyfikator pozostanie wciśnięty, powstaje w ten sposób kombinacja *Ctrl+A* (lub *Cmd+A*). Dzięki temu cały tekst zostaje zaznaczony.

4. Wysyłam klawisz *c* do pola do tekstu. Znów, ponieważ modyfikator pozostanie wciśnięty, powstaje w ten sposób kombinacja *Ctrl+C* (lub *Cmd+C*), a zawartość tekstowa zostaje skopiowana do schowka.
5. Wysyłam klawisz *v* do pola tekstowego. Znów, ponieważ modyfikator pozostanie wciśnięty, powstaje w ten sposób kombinacja *Ctrl+V* (lub *Cmd+V*), a tekst ze schowka zostaje wklejony w polu tekstowym.

❹ Sprawdzam, czy zawartość obu elementów jest taka sama na końcu testu.

Strategie oczekiwania

Aplikacje webowe to rozproszone usługi w modelu klient-serwer, w których klientami są przeglądarki, a serwerami sieciowymi zwykle jakieś zdalne maszyny. Powstałe między nimi opóźnienie sieciowe może wpływać na stabilność testów WebDriver. Przykładowo w przypadku sieci z dużym opóźnieniem lub przeciążonych serwerów powolna odpowiedź może negatywnie wpłynąć na oczekiwane warunki w testach WebDriver. Ponadto współczesne aplikacje często są dynamiczne i asynchroniczne. Obecnie JavaScript pozwala na wykonywanie nieblokujących (asynchronicznych) operacji z wykorzystaniem różnych mechanizmów, takich jak wywołania zwrotne, obietnice czy mechanizm async/await. Co więcej, dane z innych serwerów również można pozyskiwać asynchronicznie, np. używając usług AJAX (Asynchronous JavaScript and XML) lub REST (REpresentational State Transfer).

Podsumowując, kluczowa jest możliwość implementacji pauzy i czekania na spełnienie pewnych warunków w testach WebDriver. W tym celu interfejs WebDriver API udostępnia szereg zasobów związanych z czekaniem. Trzy główne obejmują mechanizmy *implicit*, *explicit* i *fluent wait* (*oczekiwanie bezwzględne*, *względne* i *płynne*). W poniższych punktach omawiam je i prezentuję przykłady.

Jeśli chodzi o czekanie, może przyjść Ci do głowy polecenie `Thread.sleep()`. Z jednej strony to proste rozwiązanie, ale z drugiej jest uważane za *złą praktykę* i podatność kodu mogącą prowadzić do niestabilnych testów (ponieważ warunki oczekiwania mogą się zmieniać). Co do zasady nalegam na niestosowanie tej opcji. Zamiast niej polecam wspomniane już strategie.

Implicit wait (oczekiwanie bezwzględne)

Pierwsza ze strategii udostępnianych przez Selenium WebDriver jest *bezwzględna*. Mechanizm ten pozwala na określenie czasu, po upływie którego zgłoszony zostanie wyjątek, jeśli element nie zostanie znaleziony. Domyślnie ta wartość wynosi zero (tj. WebDriver nie czeka wcale). Niemniej jednak, gdy tę wartość się zdefiniuje, Selenium WebDriver będzie sprawdzał dokument DOM przez ten czas, poszukując wskazanego elementu. Częstotliwość odpytywania o element jest zależna od implementacji sterownika, ale często wynosi mniej niż pięćset milisekund. Jeśli przed upływem wyznaczonego czasu element uda się znaleźć, wykonanie skryptu postępuje. Jeśli nie — zgłaszany jest wyjątek.

Przykład 3.27 przedstawia tę strategię. Widoczny tam test korzysta ze strony z ćwiczeniami (patrz rysunek 3.12) i dynamicznie ładuje kilka obrazków do drzewa DOM. Ponieważ grafika ta nie jest dostępna, dopóki strona się nie załaduje, trzeba poczekać, aż obrazki pojawią się na stronie.

Rysunek 3.12. Strona z ćwiczeniami ładująca grafikę

Przykład 3.27. Test wykorzystujący oczekiwanie bezwzględne

```
@Test
void testImplicitWait() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/loading-images.html");
    driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10)); ❶

    WebElement landscape = driver.findElement(By.id("landscape")); ❷
    assertThat(landscape.getAttribute("src"))
        .containsIgnoringCase("landscape");
}
```

❶ Przed wchodzeniem w interakcje z elementami określam strategię bezwzględnego czekania. W tym przypadku czekanie będzie trwało 10 sekund.

❷ W kolejnych wywołaniach posługuję się interfejsem API Selenium WebDriver tak jak do tej pory.

Możesz poeksperymentować z tą funkcjonalnością poprzez usunięcie mechanizmu czekania z testu (krok 1.). Jeśli tak zrobisz, zauważysz, że test zwraca negatywny wynik w kroku 2. z powodu wyjątku `NoSuchElementException`.

Choć interfejs API Selenium WebDriver wspiera ten mechanizm, bezwzględne oczekiwanie wiąże się z pewnymi niedogodnościami, o których powinieneś wiedzieć. Po pierwsze, ten rodzaj oczekiwania działa wyłącznie w odniesieniu do znajdowania elementów. Po drugie, nie można dopasować tego zachowania do swoich potrzeb, ponieważ jego implementacja jest zależna

od sterownika. Wreszcie, oczekiwanie bezwzględne stosowane jest globalnie, a oczekiwanie na nieobecność elementów zwykle wydłuża czas wykonania całego skryptu. Z tego względu strategię tę również uważa się za złą praktykę w większości przypadków i zaleca się oczekiwanie względne lub płynne.

Explicit wait (oczekiwanie względne)

Druga strategia oczekiwania, nazywana *względną*, pozwala na zatrzymanie wykonania testu na pewien maksymalny okres, dopóki spełniony nie zostanie określony warunek. Aby z niej skorzystać, należy stworzyć instancję klasy `WebDriverWait`, korzystając z obiektu `WebDriver` jako pierwszego argumentu konstruktora i instancji klasy `Duration` jako drugiego (by określić maksymalny czas oczekiwania).

Selenium WebDriver zapewnia szeroki wybór spodziewanych warunków za pośrednictwem klasy `ExpectedConditions`. Są one bardzo czytelne i ich zastosowanie nie wymaga dalszych wyjaśnień. Polecam skorzystanie z opcji autouzupełniania w swoim ulubionym środowisku IDE, by odkryć wszystkie możliwości. Rysunek 3.13 przedstawia taką listę w IDE Eclipse.

Rysunek 3.13. Autouzupełnianie dla klasy ExpectedConditions w środowisku programistycznym Eclipse

Przykład 3.28 przedstawia test używający strategii oczekiwania względnego. Korzystam tam z warunku `presenceOfElementLocated`, by poczekać, aż jeden z obrazków będzie dostępny na stronie.

Przykład 3.28. Test wykorzystujący oczekiwanie względne na stronie ładującej grafikę

```
@Test
void testExplicitWait() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/loading-images.html");
    WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10)); ❶
```

```
    WebElement landscape = wait.until(ExpectedConditions
        .presenceOfElementLocated(By.id("landscape"))); ❷
    assertThat(landscape.getAttribute("src"))
        .containsIgnoringCase("landscape");
}
```

❶ Tworzę instancję `wait`. W tym przypadku wybrany okres oczekiwania wynosi 10 sekund.

❷ Jawnie czekam na spełnienie pewnego warunku (w tym przypadku obecności konkretnego elementu) poprzez wywołanie metody `until()` na obiekcie `WebDriverWait`. Dla czytelności możesz też statycznie zaimportować oczekiwane warunki (`presenceOfElementLocated`). W tej książce zdecydowałem się jednak zachować nazwę klasy (`ExpectedConditions`), by ułatwić korzystanie z możliwości autouzupełniania, o której wspominałem wcześniej.

Przykład 3.29 przedstawia inny test używający oczekiwania względnego. Korzystam w nim z innej strony z ćwiczeniami (nazwanej „powolnym kalkulatorem"), która zawiera interfejs graficzny prymitywnego kalkulatora, zmanipulowany w taki sposób, by oczekiwał podlegający konfiguracji czas przed zwróceniem rezultatu podstawowej operacji arytmetycznej (domyślnie 5 sekund). Rysunek 3.14 przedstawia zrzut ekranu obrazujący tę stronę.

Przykład 3.29. Test wykorzystujący oczekiwanie względne na stronie z „powolnym kalkulatorem"

```
@Test
void testSlowCalculator() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/slow-calculator.html");
    // 1 + 3
    driver.findElement(By.xpath("//span[text()='1']")).click(); ❶
    driver.findElement(By.xpath("//span[text()='+']")).click();
    driver.findElement(By.xpath("//span[text()='3']")).click();
    driver.findElement(By.xpath("//span[text()='=']")).click();

    // ...powinno dać 4, poczekaj na to
    WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
    wait.until(ExpectedConditions.textToBe(By.className("screen"), "4")); ❷
}
```

❶ Posługuję się selektorem XPath do kliknięcia przycisków odpowiadających dodawaniu 1 + 3.

❷ Ponieważ test powinien poczekać, aż wynik będzie gotowy, sięgam po jawne oczekiwanie. W tym przypadku warunek polega na tym, że wartość elementu tekstowego o nazwie klasy `screen` ma być równa 4.

Fluent wait (płynne czekanie)

Ostatnią strategią jest płynne czekanie. Ten mechanizm jest uogólnieniem czekania względnego. Innymi słowy, korzysta się z niego w celu zawieszenia testu do czasu spełnienia pewnego warunku, ale w odróżnieniu od oczekiwania względnego strategia ta daje wiele możliwości precyzyjnej konfiguracji. Tabela 3.12 podsumowuje metody dostępne w klasie `FluentWait`. Jak wskazuje nazwa, klasa oferuje płynny interfejs, pozwalający na łączenie wielu wywołań w tej samej linii. Przykład 3.30 przedstawia test używający płynnego oczekiwania.

Rysunek 3.14. Strona z ćwiczeniami zawierająca „powolny kalkulator”

Tabela 3.12. Metody płynnego oczekiwania

Metoda	Opis
`withTimeout(Duration` *`czasOczekiwania`*`)`	Maksymalny czas oczekiwania określony za pomocą klasy Javy `Duration`
`pollingEvery(Duration` *`interwał`*`)`	Częstotliwość odpytywania o spełnienie warunku (domyślnie 500 milisekund)
`withMessage(String` *`wiadomość`*`)`	Spersonalizowane komunikaty błędów
`withMessage(Supplier<String>` *`dostawcaWiadomości`*`)`	
`ignoring(Class<? extends Throwable>` *`typWyjątku`*`)`	Ignorowanie określonych wyjątków podczas czekania na spełnienie warunku
`ignoring(Class<? extends Throwable>` *`pierwszyTyp`*`, Class<? extends Throwable>` *`drugiTyp`*`)`	
`ignoreAll(Collection<Class<? extends Throwable>>` *`typy`*`)`	
`until(Function<? super T, V>` *`jestPrawdą`*`)`	Oczekiwany warunek

Przykład 3.30. Test wykorzystujący płynne oczekiwanie

```
@Test
void testFluentWait() {
    driver.get(
        "https://bonigarcia.dev/selenium-webdriver-java/loading-images.html");
    Wait<WebDriver> wait = new FluentWait<>(driver)
        .withTimeout(Duration.ofSeconds(10))
        .pollingEvery(Duration.ofSeconds(1))
        .ignoring(NoSuchElementException.class); ❶

    WebElement landscape = wait.until(ExpectedConditions
        .presenceOfElementLocated(By.id("landscape")));
    assertThat(landscape.getAttribute("src"))
        .containsIgnoringCase("landscape");
}
```

❶ Jak widzisz, ten test jest bardzo podobny do przykładu 3.28, choć opiera się na instancji `FluentWait`, dzięki której mogę dodać szereg opcji konfiguracyjnych. W tym przypadku zmieniam częstotliwość odpytywania na jedną sekundę.

Klasa `WebDriverWait` (prezentowana w poprzednim punkcie) dziedziczy po bardziej ogólnej klasie `FluentWait`. Tak więc możesz też używać wszystkich metod przedstawionych w tabeli 3.12 z mechanizmem oczekiwania względnego.

Powiązane strategie w Selenium WebDriver

Poza omówionymi już strategiami oczekiwania powinieneś mieć świadomość istnienia kilku innych dodatkowych funkcjonalności Selenium WebDriver. Należą do nich:

Strategie ładowania

Selenium WebDriver pozwala na wskazywanie różnych strategii ładowania stron. Ta funkcjonalność jest dostępna poprzez funkcjonalności zależne od typu przeglądarki (tj. za pomocą `ChromeOptions`, `FirefoxOptions` itd.). Z tego powodu omawiam ją szerzej w punkcie „Strategie ładowania strony" w rozdziale 5.

Czas oczekiwania

Selenium WebDriver pozwala także na wskazywanie maksymalnego czasu oczekiwania na ładowanie strony i skryptu. Więcej o tej funkcjonalności mówię w podrozdziale „Maksymalny czas oczekiwania" w rozdziale 4.

Podsumowanie

Ten rozdział zawierał podstawy interfejsu API Selenium WebDriver. Po pierwsze, nauczyłeś się, jak tworzyć i likwidować instancje `WebDriver`. Obiekty te reprezentują przeglądarkę kontrolowaną przez Selenium WebDriver. Dzięki temu możesz korzystać z `ChromeDriver` dla przeglądarki Chrome, `FirefoxDriver` dla Firefox itd. Po drugie, zapoznałeś się z cechami klasy `WebElement` — klasy przedstawiającej różne elementy strony (np. odnośniki, obrazki, pola formularzy itd.). Masz do wyboru kilka opcji znajdowania tych elementów na stronie: na podstawie atrybutów HTML (id, nazwy czy klasy), nazwy znacznika, tekstu odnośnika (pełnego lub cząstkowego), selektorów CSS lub XPath. Zaprezentowałem także zupełnie nowe strategie pochodzące z Selenium WebDriver 4 — lokalizatory względne. Następnie pokazałem Ci sposoby naśladowania działań użytkownika z wykorzystaniem klawiatury i myszki. Możesz korzystać z nich do wykonywania prostych operacji (np. kliknięcia odnośnika, wypełnienia pola tekstowego) lub skomplikowanych ruchów (np. przeciągania elementów, klikania i przesuwania kursora). Na koniec dowiedziałeś się, jakie możliwości w zakresie czekania daje Selenium WebDriver. Ta funkcjonalność jest kluczowa ze względu na obecną rozproszoną, dynamiczną i asynchroniczną naturę aplikacji webowych. Selenium WebDriver wyróżnia trzy podstawowe strategie oczekiwania: bezwzględna (wskazywanie ogólnego czasu oczekiwania na elementy), względna (wstrzymanie testu do czasu spełnienia pewnego warunku) i płynna (rozszerzenie oczekiwania względnego o zaawansowane opcje konfiguracyjne).

Rozdział 4. zgłębia dalsze szczegóły interfejsu API Selenium WebDriver. W szczególności przygląda się możliwościom wspólnym dla różnych przeglądarek (np. Chrome, Edge, Firefox itd.). Dowiesz się dzięki temu, jak wykonywać skrypty JavaScript, wskazywać mechanizmy nasłuchiwania na elementy, konfigurować maksymalny czas oczekiwania na ładowanie strony i skryptów, zarządzać historią przeglądarki, wykonywać zrzuty ekranu, obsługiwać ciasteczka, manipulować listami rozwijanymi (w elementach typu select i list), pracować z typami okien (zakładkami, ramkami i ramkami typu iframe) oraz oknami dialogowymi (alertami, modalami, potwierdzeniami, wskazówkami), wykorzystywać schowek web storage i radzić sobie z wyjątkami WebDriver.

Selenium: testowanie, które daje niezawodność i bezpieczeństwo!

Selenium pozwala na automatyzację pracy przeglądarek internetowych. Kluczowym komponentem tego projektu open source jest Selenium WebDriver — biblioteka do programistycznego kontrolowania przeglądarek. Podstawowym zastosowaniem Selenium jest implementacja testów systemowych w celu weryfikacji funkcjonowania aplikacji. Narzędzie to zdobyło ogromną popularność — stanowi ono jedno z wiodących rozwiązań w zakresie testów systemowych. Jest chętnie używane zarówno przez duże organizacje, jak i samodzielnych programistów.

Ten praktyczny przewodnik po Selenium WebDriver w wersji 4 z uwzględnieniem implementacji w Javie jest przeznaczony dla programistów Javy, inżynierów jakości i testerów. Przedstawiono w nim główne aspekty zautomatyzowanej nawigacji po stronie, manipulacji w przeglądarce, interakcji z elementami, naśladowania działań użytkownika i automatycznego zarządzania sterownikami. Opisano koncepcję wzorca projektowego POM, który pozwala na modelowanie stron internetowych w klasach zorientowanych obiektowo. Zaprezentowano różne sposoby przeprowadzania testów i wprowadzania odpowiedniej ich kolejności, omówiono też zasady analizy błędnych wykonań w celu określenia przyczyn niepowodzenia. Pokazano także możliwości wzbogacania testów o inne technologie, służące na przykład do raportowania wyników, generowania danych czy implementacji szczególnych przypadków użycia.

W książce między innymi:

- przygotowanie środowiska do testów end-to-end z Selenium WebDriver
- automatyczne interakcje z aplikacjami internetowymi
- strategie testowania na wielu przeglądarkach
- testowanie działania formularzy, komunikatów w oknach i skryptów JavaScript
- posługiwanie się złożoną infrastrukturą w testach Selenium WebDriver
- wykorzystanie programowania zorientowanego obiektowo w testowanych aplikacjach

Dr Boni García jest wykładowcą wizytującym na Uniwersytecie Karola III w Madrycie, a także autorem ponad 45 publikacji naukowych. Prowadzi kilka projektów open source, między innymi WebDriver Manager i Selenium-Jupiter. Jego zainteresowania naukowe koncentrują się wokół inżynierii oprogramowania, a zwłaszcza testów automatycznych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-9982-2
9 788328 399822
Cena: 89,00 zł