Technologia i rozwiązania

Selenium i testowanie aplikacji

Receptury

Wydanie II

Unmesh Gundecha

Helion

[PACKT] PUBLISHING open source community experience distilled

Tytuł oryginału: Selenium Testing Tools Cookbook, Second Edition

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-3411-3

First published in the English language under the title
'Selenium Testing Tools Cookbook - Second Edition - (9781784392512)'

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 4

Korzystanie z Selenium API

Oto zagadnienia, jakie zostaną omówione w tym rozdziale.

- Sprawdzanie istnienia elementu.
- Sprawdzanie stanu elementu.
- Wykorzystanie API zaawansowanych interakcji użytkownika (ang. *Advanced User Interactions*) do obsługi zdarzeń myszy i klawiatury.
- Wykonywanie podwójnych kliknięć elementu.
- Wykonywanie operacji „przeciągnij i upuść".
- Obsługa menu kontekstowych.
- Wykonywanie kodu JavaScript.
- Przechwytywanie zrzutów ekranu za pomocą Selenium WebDriver.
- Maksymalizowanie okna przeglądarki.
- Obsługa plików cookie sesji.
- Obsługa mechanizmów nawigacji przeglądarki.
- Korzystanie ze zdarzeń klasy WebDriver.

Wprowadzenie

Selenium WebDriver implementuje kompleksowe API do pracy z elementami webowymi, które umożliwia wykonywanie zaawansowanych interakcji z użytkownikiem, takich jak złożone zdarzenia związane z myszą i klawiaturą, wykonywanie kodu JavaScript, przechwytywanie zrzutów ekranu i tak dalej.

W tym rozdziale omówię sposób wykorzystania tych własności do budowania kroków testów o różnym poziomie złożoności. Lektura rozdziału pomoże również w przezwyciężeniu niektórych typowych problemów, które mogą powstawać podczas tworzenia testów bazujących na frameworku Selenium WebDriver.

Sprawdzanie istnienia elementu

Selenium WebDriver nie implementuje metody Selenium RC `isElementPresent()`, która sprawdza, czy element istnieje na stronie. Metoda jest przydatna do budowania niezawodnego testu, który mógłby sprawdzić istnienie elementu przed wykonaniem na nim jakiegokolwiek działania.

W tej recepturze napiszę metodę podobną do metody `isElementPresent()`.

Jak to zrobić?

Aby zaimplementować metodę `isElementPresent()`, wykonaj następujące czynności.

1. Utwórz metodę `isElementPresent()` w module dostępnym do testów i umieść w niej następujący kod.

```
{{Private boolean isElementPresent(By by)
    try {
        driver.findElement(by);
            return true;
        } catch {} (NoSuchElementException e)
            return false;
    }
}
```

2. Teraz utwórz kod testu, który wywołuje metodę `isElementPresent()`. Metoda będzie sprawdzać, czy żądany element istnieje na stronie. Jeśli będzie znaleziony, zostanie wykonane kliknięcie elementu, w przeciwnym razie test nie przejdzie. Można to zrobić w następujący sposób.

```
@Test
Public void testIsElementPresent() {}
    // Sprawdzenie, czy element spełniający kryteria lokalizatora istnieje na stronie
    if (isElementPresent(By.name("airbags"))) {
        // Utwórz reprezentację pola wyboru i zaznacz je
        WebElement airbag = driver.findElement(By.name("airbags"));
        if (!airbag.isSelected()) {
            airbag.click();
        }
    } else {
        fail("Pole wyboru poduszki powietrznej nie istnieje!!");
    }
}
```

Jak to działa?

Metoda `isElementPresent()` pobiera argument lokalizatora z wykorzystaniem egzemplarza klasy `By`. Następnie wywołuje metodę `findElement()`. Jeśli element nie zostanie znaleziony, będzie zgłoszony wyjątek `NoSuchElementException`. Dzięki użyciu bloku `try` i `catch` metoda `isElementPresent()` zwróci `true`, gdy element zostanie znaleziony. W tym przypadku nie zostanie zgłoszony wyjątek. W przeciwnym razie zwróci `false`, a metoda `findElement()` zgłosi wyjątek `NoSuchElementException`.

Zobacz też

- Receptura „Sprawdzanie stanu elementu”.

Sprawdzanie stanu elementu

Wielokrotnie zdarza się, że próba kliknięcia elementu lub wprowadzenia tekstu w polu zawodzi ze względu na to, że element jest wyłączony lub istnieje w modelu DOM, ale jest ukryty na stronie. To powoduje zgłoszenie błędu i niepowodzenie testu. Zbudowanie wiarygodnych testów, które można uruchomić w trybie nienadzorowanym, wymaga utworzenia w przepływie testów solidnego mechanizmu obsługi błędów i wyjątków.

Powstające problemy można obsłużyć, sprawdzając stan elementów. Interfejs `WebElement` udostępnia następujące metody sprawdzania stanu elementu.

Metoda	Przeznaczenie
`isEnabled()`	Metoda sprawdza, czy element jest aktywny. Zwraca `true`, jeśli jest aktywny, i `false` w przypadku przeciwnym.
`isSelected()`	Metoda sprawdza, czy element (przełącznik, pole wyboru i tak dalej) jest zaznaczony. Zwraca `true`, jeśli jest zaznaczony, i `false` w przypadku przeciwnym.
`isDisplayed()`	Metoda sprawdza, czy element jest wyświetlany.

W tej recepturze zastosuję niektóre z wymienionych metod w celu sprawdzenia stanu elementów i obsługi błędów, jeśli to możliwe.

Jak to zrobić?

Utworzymy test, w ramach którego sprawdzimy, czy na stronie zostało zaznaczone pole wyboru włączające reflektor LED. To pole wyboru będzie włączone lub wyłączone, w zależności od wcześniej wybranych opcji. Przed zaznaczeniem tego pola wyboru upewnimy się, czy jest ono aktywne. Można to zrobić w następujący sposób.

```
@Test
Public void testElementIsEnabled() {}
    // Utworzenie reprezentacji pola wyboru w postaci obiektu WebElement z wykorzystaniem atrybutu name
    WebElement ledheadlamp = driver.findElement(By.name("ledheadlamp"));
    // Sprawdzenie, czy element jest aktywny przed jego zaznaczeniem
    if (ledheadlamp.isEnabled()) {
        // Sprawdzenie, czy pole wyboru już jest zaznaczone. W przeciwnym razie zaznaczenie go
         if (!ledheadlamp.isSelected()) {
            ledheadlamp.click();
        }
    } else {
         fail("Pole wyboru reflektora LED jest nieaktywne!!");
    }
}
```

Jak to działa?

Zaznaczamy pole wyboru, sprawdzając dwa stany elementu; po pierwsze, czy pole to jest aktywne, i po drugie, czy już nie jest zaznaczone. Do tego celu możemy użyć funkcji `isEnabled()` interfejsu `WebElement`, która zwraca `true`, jeśli element jest aktywny, lub `false`, gdy jest nieaktywny. Jeśli pole wyboru jest nieaktywne, test zakończy się niepowodzeniem. Kiedy nie sprawdzimy tego warunku, test prawdopodobnie zgłosi wyjątek informujący o tym, że obiekt jest nieaktywny, tak jak pokazano poniżej.

```
// Sprawdzenie, czy element jest aktywny przed jego zaznaczeniem
if (ledheadlamp.isEnabled()) {
    // Sprawdzenie, czy pole wyboru jest już zaznaczone. W przeciwnym razie zaznaczenie pola wyboru
    if (!ledheadlamp.isSelected()) {
        ledheadlamp.click();
    }
} else {
    fail("Pole wyboru reflektora LED jest nieaktywne!!");
}
```

Wykorzystanie API Advanced User Interactions do obsługi zdarzeń myszy i klawiatury

API *Advanced User Interactions* frameworka Selenium WebDriver pozwala wykonywać zarówno proste, jak i złożone operacje obsługi zdarzeń klawiatury i myszy. Klasa `Actions` udostępnia tak złożone operacje jak „przeciągnij i upuść”, czy przytrzymanie klawisza, a następnie wykonanie operacji myszą. API umożliwia także budowanie złożonych łańcuchów zdarzeń imitujących działania ręcznie wykonywane przez użytkownika.

W celu utworzenia złożonych działań obejmujących grupę innych działań klasa `Actions` implementuje wzorzec „budowniczy”.

W tej recepturze użyję klasy `Actions` do utworzenia łańcucha zdarzeń wymaganych do zaznaczenia wierszy w tabeli.

Jak to zrobić?

Utwórzmy test, który zaznacza wiele wierszy z różnych pozycji w tabeli za pomocą klawisza *Ctrl* (w komputerach Mac *Command*). Możemy wybrać wiele wierszy: zaznaczamy pierwszy wiersz, przytrzymujemy klawisz *Ctrl* (*Command* na komputerach Mac), a następnie wybieramy inny wiersz i zwalniamy klawisz *Ctrl* (*Command*). Takie działanie spowoduje zaznaczenie żądanych wierszy z tabeli, tak jak pokazano w poniższym fragmencie kodu.

```
@Test
public void testRowSelectionUsingControlKey() {

    List<WebElement> tableRows = driver.findElements
        (By.xpath("//table[@class='iceDatTbl']/tbody/tr"));

    // Zaznacz drugi i czwarty wiersz w tabeli za pomocą klawisza Control.
    //Wiersze rozpoczynają się od indeksu 0
    Actions builder = new Actions(driver);
    builder.click(tableRows.get(1))
           .keyDown(Keys.CONTROL)
           .click(tableRows.get(3))
           .keyUp(Keys.CONTROL)
           .build().perform();

// Sprawdzenie, czy w tabeli są zaznaczone dwa wiersze
List<WebElement> rows = driver.findElements
    (By.xpath("//div[@class='icePnlGrp
    exampleBox']/table[@class='iceDatTbl']/tbody/tr"));
assertEquals(2,rows.size());
}
```

W komputerach z systemem Mac OS X należy użyć składni klawisza `Command` zamiast składni klawisza `Control`. Oto przykład.

```
Actions builder = new Actions(driver);
builder.click(tableRows.get(1)).keyDown(Keys.COMMAND)
    click(tableRows.get(3)).keyUp(Keys.COMMAND).perform();}
```

Jak to działa?

Należy utworzyć egzemplarz klasy `Actions`, przekazując do konstruktora egzemplarz klasy `driver` w następujący sposób.

```
Actions builder = new Actions(driver);
```

Zbudujemy łańcuch zdarzeń, które należy wykonać, aby zaznaczyć wiersze. Będzie to wymagało wykonania operacji `click()` na pierwszym wierszu, następnie przytrzymania klawisza *Ctrl* (*Command* w systemie Mac) przy użyciu operacji `keyDown()`, kliknięcia wiersza końcowego, a następnie zwolnienia klawisza *Ctrl* (*Command*) przez wywołanie `keyUp()`. Klasa `Actions` oferuje różne metody do wykonywania operacji z klawiaturą i myszą.

```
Actions builder = new Actions(driver);
builder.click(tableRows.get(1)).keyDown(Keys.CONTROL)
    .click(tableRows.get(3)).keyUp(Keys.CONTROL)
    .build().perform();
```

Możemy utworzyć złożone działania, które wykonamy przy użyciu wywołania metody `perform()` klasy `Actions`.

Klasa `Keys` reprezentuje wszystkie nietekstowe klawisze na klawiaturze, na przykład klawisz *Ctrl*, klawisz *Shift*, klawisze funkcyjne i tak dalej. W poprzednim przykładzie użyliśmy wywołania `keyDown(Keys.CONTROL)` w celu naciśnięcia i przytrzymania klawisza *Ctrl* (*Command*) aż do zakończenia następnej operacji.

> Działania mogą nie uruchamiać się poprawnie w przypadku elementów, które nie są widoczne lub są nieaktywne. Przed użyciem opisanych zdarzeń należy zadbać o to, aby elementy były widoczne i aktywne.

Zobacz też

- Receptura „Wykonywanie dwukrotnych kliknięć elementów”.

Wykonywanie dwukrotnych kliknięć elementów

W aplikacjach webowych są elementy, które do wykonania niektórych czynności wymagają dwukrotnych kliknięć. Przykładowo dwukrotne kliknięcie wiersza tabeli powoduje wyświetlenie nowego okna. API Advanced User Interaction dostarcza metodę umożliwiającą wykonywanie dwukrotnych kliknięć.

W tej recepturze do wykonywania operacji dwukrotnego kliknięcia użyję klasy `Actions`.

Jak to zrobić?

Utwórzmy test lokalizujący element, dla którego zaimplementowano zdarzenie dwukrotnego kliknięcia. Dwukrotne kliknięcie tego elementu spowoduje zmianę jego koloru.

```
package com.secookbook.examples.chapter04;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.By;
import org.openqa.selenium.interactions.Actions;
import static org.junit.Assert.*;
import org.junit.Test;

public class DoubleClickTest {
    @Test
    public void testDoubleClick() throws Exception {
        WebDriver driver = new ChromeDriver();
        driver.get("http://cookbook.seleniumacademy.com/DoubleClickDemo.html");

        try {
            WebElement message = driver.findElement(By.id("message"));

            // Sprawdzenie, czy element jest niebieskiego koloru
            assertEquals("rgba(0, 0, 255, 1)", message.
            ↪getCssValue("background-color"));

            Actions builder = new Actions(driver);
            builder.doubleClick(message).perform();

            // Sprawdzenie, czy element jest żółtego koloru
            assertEquals("rgba(255, 255, 0, 1)", message.
            ↪getCssValue("background-color"));
        } finally {
            driver.quit();
        }
    }
}
```

Jak to działa?

Aby wykonać dwukrotne kliknięcie elementu, należy wywołać metodę `doubleClick()` klasy `Actions`. W celu wywołania tej metody trzeba utworzyć egzemplarz klasy `Actions`, tak jak pokazano poniżej.

```
Actions builder = new Actions(driver);
```

Metoda `doubleClick()` wymaga elementu, na którym zostanie zainicjowane zdarzenie dwukrotnego kliknięcia. Metodę `doubleClick()` można wywołać, przekazując element w następujący sposób.

```
builder.doubleClick(message).perform();
```

Zobacz też

- Receptura „Wykorzystanie API Advanced User Interactions do obsługi zdarzeń myszy i klawiatury”,
- Receptura „Wykonywanie operacji «przeciągnij i upuść»”.

Wykonywanie operacji „przeciągnij i upuść”

Selenium WebDriver, za pośrednictwem klasy `Actions`, implementuje polecenie Selenium RC `dragAndDrop`. Klasa `Actions`, jak pokazywałem we wcześniejszych recepturach, obsługuje zaawansowane interakcje użytkownika, takie jak wyzwalanie zdarzeń myszy i klawiatury. Za pośrednictwem tej klasy można budować proste lub złożone łańcuchy zdarzeń.

W tej recepturze wykorzystam klasę `Actions` do wykonywania operacji „przeciągnij i upuść”.

Jak to zrobić?

Zaimplementujemy test, który za pomocą klasy `Actions` wykona na stronie operację „przeciągnij i upuść”.

```
@Test
Public void testDragDrop() {}
    Driver.get ("http://cookbook.seleniumacademy.com/DragDropDemo.html");

    WebElement source = driver.findElement(By.id("draggable"));
    WebElement target = driver.findElement(By.id("droppable"));

    Actions builder = new Actions(driver);
    builder.dragAndDrop(source, target) .perform();
    assertEquals("Upuszczono!", target.getText());
}
```

Jak to działa?

Aby przeciągnąć element na inny element i go na nim upuścić, trzeba zlokalizować oba elementy i przekazać je do metody `dragAndDrop()` klasy `Actions`. Żeby wywołać tę metodę, należy utworzyć egzemplarz klasy `Actions` w następujący sposób.

```
Actions builder = new Actions(driver);
```

Metoda `dragAndDrop()` wymaga elementu źródłowego oraz elementu docelowego, na który element źródłowy zostanie przeciągnięty i upuszczony. Metodę `dragAndDrop()` można wywołać w następujący sposób.

```
builder.dragAndDrop(source, target).perform();
```

W czasie powstawania tej książki framework Selenium WebDriver nie obsługiwał operacji „przeciągnij i upuść" dla HTML5. Więcej informacji na ten temat można znaleźć pod adresem *https://code.google.com/p/selenium/issues/detail?id=3604*. Istnieje obejście umożliwiające symulację własności „przeciągnij i upuść" w aplikacjach webowych bazujących na HTML5. Więcej szczegółów na ten temat można znaleźć pod adresem *https://gist.github.com/rcorreia/2362544*.

Zobacz też

- Receptura „Wykorzystanie API Advanced User Interactions do obsługi zdarzeń myszy i klawiatury",
- Receptura „Wykonywanie dwukrotnych kliknięć elementów".

Obsługa menu kontekstowych

Menu kontekstowe (znane także jako menu skrótów, wyskakujące menu lub menu podręczne) to lista poleceń wyświetlanych na stronie WWW w odpowiedzi na kliknięcie elementu prawym przyciskiem myszy. Przykładowo pod adresem *http://bit.ly/1CAV05I* zamieszczono kod korzystający z wtyczki jQuery `contextMenu`, który wyświetla menu edycji, gdy użytkownik kliknie przycisk prawym przyciskiem myszy.

Klasa `Actions` frameworka Selenium WebDriver dostarcza metodę `contextClick()`, która pozwala wykonać operację kliknięcia prawym przyciskiem myszy. W tej recepturze pokażę, w jaki sposób zautomatyzować interakcje z menu kontekstowymi.

Jak to zrobić?

Spróbujmy zaimplementować test, który otworzy menu kontekstowe i, korzystając z klasy `Actions`, wybierze jedną z opcji.

```
@Test
Public void testContextMenu() {}
    WebElement clickMeElement =
        driver.findElement (By.cssSelector ("div.context-menuone.Box.menu-1 "));
    WebElement editMenuItem =
        driver.findElement (By.cssSelector ("li.context-menuitem.icon-edit"));

    Actions builder = new Actions(driver);
    builder.contextClick(clickMeElement)
        .moveToElement(editMenuItem)
        .Click()
        .perform();

    WebDriverWait wait = new WebDriverWait(driver, 10);
```

```
        Alert alert = wait.until(ExpectedConditions.alertIsPresent());
        assertEquals("clicked: edit", alert.getText());
        alert.dismiss();
    }
```

Jak to działa?

Do otwarcia menu kontekstowego w odpowiedzi na kliknięcie prawym przyciskiem myszy służy metoda `contextClick()` klasy `Actions`. Poniższy kod wykonuje kliknięcie prawym przyciskiem myszy podanego elementu.

```
WebElement clickMeElement =
        driver.findElement(By.cssSelector("div.context-menuone.
box.menu-1"));
    WebElement editMenuItem =
        driver.findElement(By.cssSelector("li.context-menuitem.
icon-edit"));

    Actions builder = new Actions(driver);
    builder.contextClick(clickMeElement)
        .moveToElement(editMenuItem)
        .click()

        .perform();
```

Następnie można przejść do żądanego elementu — w tym przypadku jest nim element `<li>`, który reprezentuje jedną z opcji menu. Aby wykonać akcję menu, wywoływana jest metoda `click()`. W tym przykładzie wywołanie metody `click()` powoduje wyświetlenie alertu, a test sprawdza komunikat w oknie alertu.

Co dalej?

W poprzednim przykładzie test wyszukuje pozycję menu, a następnie wykonuje operację kliknięcia. Czasami w menu są dostępne klawisze skrótów. Za pośrednictwem klasy `Actions` można wykorzystać kombinację zdarzeń myszy i naciśnięć klawiszy w celu otwarcia żądanej opcji menu. Przykładowo klawiszem skrótu dla pozycji menu *Edit* jest „e”. Możemy otworzyć menu kontekstowe, a następnie wysłać kombinację klawiszy *Alt+E* w następujący sposób.

```
@Test
public void testContextMenuWithKeys() {
    WebElement clickMeElement =
        driver.findElement(By.cssSelector("div.context-menu-one.box.menu-1"));

    Actions builder = new Actions(driver);
    builder.contextClick(clickMeElement)
    .sendKeys(Keys.chord(Keys.ALT, "e"))
    .perform();
```

```
    WebDriverWait wait = new WebDriverWait(driver, 10);

    Alert alert = wait.until(ExpectedConditions.alertIsPresent());
    assertEquals("clicked: edit", alert.getText());
    alert.dismiss();
}
```

Zobacz też

- Receptura „Wykorzystanie API Advanced User Interactions do obsługi zdarzeń myszy i klawiatury".

Wykonywanie kodu JavaScript

API Selenium WebDriver zapewnia możliwość wykonywania w oknie przeglądarki kodu JavaScript. Jest to bardzo przydatna funkcja, w przypadku gdy testy muszą wykonywać operacje na stronie przy użyciu JavaScript. Dzięki temu API frameworka Selenium WebDriver może być wykorzystane również do testowania kodu JavaScript po stronie klienta. Selenium WebDriver oferuje interfejs `JavascriptExecutor`, który może służyć do uruchamiania dowolnego kodu JavaScript w kontekście przeglądarki.

W tej recepturze pokażę, w jaki sposób używać interfejsu `JavascriptExecutor` do uruchamiania kodu JavaScript. W tej książce zamieszczono różne receptury, w których wykorzystano interfejs `JavascriptExecutor` do wykonywania niektórych zaawansowanych operacji, które nie są jeszcze obsługiwane przez Selenium WebDriver.

Jak to zrobić?

Utwórzmy test, który wywołuje kod JavaScript w celu zwrócenia tytułu i liczby linków (tzn. liczby tagów `Anchor`). Aby zwrócić tytuł strony, można również wykorzystać metodę `driver.getTitle()`. Poniżej przedstawiono przykładowy kod takiego działania.

```
@Test
public void testJavaScriptCalls() throws Exception {
    WebDriver driver = new ChromeDriver();
    driver.get("http://www.google.com");
    try {
        JavascriptExecutor js = (JavascriptExecutor) driver;

        String title = (String) js.executeScript("return document.title");
        assertEquals("Google", title);

        long links = (Long) js
```

```
                .executeScript("var links = document.
                ↪getElementsByTagName('A'); return links.length");
            assertEquals(42, links);
        } finally {
            driver.quit();
        }
    }
```

Jak to działa?

Zrzutowanie egzemplarza WebDriver na interfejs `JavascriptExecutor` pozwala wykonać w Selenium WebDriver kod JavaScript.

```
JavascriptExecutor js = (JavascriptExecutor) driver;
```

W poniższym przykładzie uruchomiono pojedynczy wiersz kodu JavaScript, który zwraca tytuł strony wyświetlanej przez sterownik. Interfejs `JavascriptExecutor` dostarcza metodę `executeScript()`, do której należy przekazać kod JavaScript.

```
String title = (String) js.executeScript("return document.title");
```

> Podczas zwracania wartości z kodu JavaScript trzeba użyć słowa kluczowego `return`. Wynik metody `executeScript()` trzeba zrzutować na podstawie typu zwróconej wartości. Dla wartości dziesiętnych można użyć typu `Double`, niedziesiętne wartości liczbowe wymagają użycia typu `Long`, natomiast dla wartości logicznych należy stosować typ `Boolean`. Jeśli kod JavaScript zwraca element HTML, można wykorzystać typ `WebElement`. Dla wartości tekstowych należy skorzystać z typu `String`. Jeśli jest zwracana lista obiektów, można zastosować dowolny z wymienionych typów, w zależności od typu obiektów na liście. W przeciwnym razie zostanie zwrócona wartość `null`.

W poniższym przykładzie w celu pobrania liczby linków na stronie uruchomiono wielowierszowy kod JavaScript.

```
long links = (Long) js.executeScript("var links =
    document.getElementsByTagName('A'); return links.length");
```

Co dalej?

Do kodu JavaScript uruchamianego za pomocą metody `executeScript()` mogą być przekazywane argumenty. W poniższym przykładzie chcemy ustawić wartość elementu. Wewnątrz kodu JavaScript zostanie wykorzystana specjalna tablica `arguments`, tak jak pokazano w poniższym kodzie.

```
js.executeScript("document.getElementByID('name').value =
    arguments[0]","Jan");
```

Przechwytywanie zrzutów ekranu za pomocą Selenium WebDriver

Selenium WebDriver dostarcza interfejs `TakesScreenshot`, który umożliwia przechwytywanie zrzutów ekranu stron internetowych. Pomaga to podczas uruchamiania testu, ponieważ pokazuje, co dokładnie się stało, gdy został zgłoszony wyjątek lub wystąpił błąd podczas uruchamiania. Można również przechwytywać zrzuty ekranu podczas weryfikacji stanu elementu, wartości wyświetlanych wewnątrz elementów lub stanu po zakończeniu działania.

Przechwytywanie zrzutów ekranu pomaga także w weryfikacji takich obszarów jak układ strony czy sposób wyrównania pól, kiedy porównywane są zrzuty ekranu wykonane podczas działania testów z ilustracjami bazowymi.

W tej recepturze użyję interfejsu `TakesScreenshot` do wykonania zrzutu ekranu testowanej strony WWW.

Jak to zrobić?

Utwórzmy test, który otworzy aplikację testową i wykona zrzut ekranu strony w formacie **PNG** (ang. *Portable Network Graphics*), tak jak pokazano w poniższym przykładowym kodzie.

```
@Test
public void testTakesScreenshot() throws Exception {
    File scrFile = ((TakesScreenshot) driver)
        .getScreenshotAs(OutputType.FILE);
    FileUtils.copyFile(scrFile, new File("target/main_page.png"));
}
```

Jak to działa?

Interfejs `TakesScreenshot` dostarcza metodę `getScreenshotAs()`, która pozwala przechwycić zrzut ekranu strony wyświetlanej wewnątrz egzemplarza obiektu `driver`. W poniższym przykładzie użyliśmy wartości `OutputType.FILE` jako argumentu metody `getScreenshotAs()`. Dzięki temu przechwycony zrzut ekranu zostanie zwrócony w pliku.

```
File scrFile =
    ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
```

Obiekt `file` zwrócony przez metodę `getScreenshotAs()` możemy zapisać za pomocą metody `copyFile()` klasy `FileUtils` należącej do klasy `org.apache.commons.io.FileUtils`.

> Interfejs `TakesScreenshot` wykonuje zrzuty ekranu z wykorzystaniem API przeglądarki. Sterownik `HtmlUnit` nie obsługuje interfejsu `TakesScreenshot`.

Co dalej?

Klasa `OutputType` oferuje wiele sposobów prezentowania wyników wykonania zrzutów ekranu za pomocą metody `getScreenshotAs()`. W poprzednim przykładzie widzieliśmy zrzut ekranu zapisany w pliku. Zrzuty ekranu mogą być również przechwytywane w tekstowym formacie Base64 lub w formacie surowych bajtów. W poniższym przykładzie wykonano zrzut ekranu w formacie ciągu Base64.

```
String base64 =
    ((TakesScreenshot)driver).getScreenshotAs(OutputType.BASE64);
```

Przechwytywanie zrzutów ekranu w środowisku RemoteWebDriver (Selenium Grid)

Gdy testy są uruchamiane z wykorzystaniem interfejsu `RemoteWebDriver` lub Selenium Grid, nie można wykonywać zrzutów ekranu, ponieważ interfejs `TakesScreenshot` nie jest zaimplementowany w klasie `RemoteWebDriver`.

Możemy jednak użyć klasy `Augmenter`, która dodaje interfejs `TakesScreenshot` do egzemplarza `RemoteWebDriver`, tak jak pokazano w poniższym przykładzie kodu.

```
driver = new Augmenter().augment(driver);
File scrFile =
    ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
FileUtils.copyFile(scrFile, new File("c:\\tmp\\screenshot.png"));
```

Klasa `Augmenter` ulepsza klasę `RemoteWebDriver` poprzez dodanie do niej różnych interfejsów, w tym interfejsu `TakesScreenshot`.

```
driver = new Augmenter().augment(driver);
```

Po wykonaniu takiego rozszerzenia możemy użyć interfejsu `TakesScreenshot` z klasy `RemoteWebDriver` do przechwytywania ekranu.

Zobacz też

- Receptura „Przechwytywanie zrzutów ekranu elementów w Selenium WebDriver” w rozdziale 9., „Rozszerzanie Selenium”.
- Receptura „Porównywanie obrazów w Selenium” w rozdziale 9., „Rozszerzanie Selenium”.

Maksymalizowanie okna przeglądarki

W Selenium WebDriver brakowało polecenia `windowMaximize()` z API Selenium RC. Jednakże od wersji 2.21 Selenium WebDriver obsługuje maksymalizowanie okna przeglądarki.

W tej krótkiej recepturze pokażę, jak zmaksymalizować okno przeglądarki.

Przygotuj się

Utwórz nowy test, który zbuduje egzemplarz `WebDriver`, przejdzie do witryny i wykona kilka podstawowych działań i weryfikacji.

Jak to zrobić?

Aby zmaksymalizować okno przeglądarki, trzeba wywołać metodę `maximize()` interfejsu `Window` klasy `driver`. Dodaj drugi wiersz kodu z poniższego fragmentu, ten za instrukcją definiującą egzemplarz klasy `FirefoxDriver`.

```
driver = new FirefoxDriver();
driver.manage().window().maximize();
```

Obsługa plików cookie sesji

W witrynach internetowych używane są pliki cookie do przechowywania preferencji użytkownika, informacji o logowaniu oraz różnych innych szczegółów dotyczących klienta. Selenium WebDriver API zapewnia różne metody zarządzania plikami cookie podczas testowania. Za pomocą tych metod możemy odczytywać wartości plików cookie, dodawać pliki cookie oraz usuwać takie pliki podczas testowania. Może to służyć do testowania reakcji aplikacji podczas wykonywania działań z plikami cookie. Interfejs `WebDriver.Options` udostępnia następujące metody zarządzania plikami cookie.

Metoda	Opis
`addCookie(Cookie cookie)`	Dodaje plik cookie.
`getCookieNamed(String name)`	Zwraca plik cookie o podanej nazwie.
`getCookies()`	Zwraca wszystkie pliki cookie dla bieżącej domeny.
`deleteCookieNamed(String name)`	Usuwa plik cookie o podanej nazwie.
`deleteCookie(Cookie cookie)`	Usuwa plik cookie.
`deleteAllCookies()`	Usuwa wszystkie pliki cookie dla bieżącej domeny.

W tej recepturze pokażę, jak odczytywać pliki cookie i sprawdzać ich wartość.

Przygotuj się

Utwórz nowy test, który zbuduje egzemplarz `WebDriver`, przejdzie do witryny i wykona kilka podstawowych działań i weryfikacji.

Jak to zrobić?

Utwórzmy test, który odczytuje plik cookie i sprawdza jego wartość, tak jak pokazano w poniższym przykładzie kodu.

```
@Test
Public void testCookies() {}
    Driver.get ("http://demo.magentocommerce.com/");

    // Utworzenie reprezentacji rozwijanego menu wyboru języka w postaci egzemplarza klasy Select
    Select language = new Select(driver.findElement(By
        .id("select-language")));

    // Sprawdzenie, czy domyślnie zaznaczona opcja to język polski
    assertEquals("Polski", language.getFirstSelectedOption().getText());

    // Zmienna storeCookie powinna mieć wartość null
    Cookie storeCookie = driver.manage().getCookieNamed("store");
    assertEquals(null, storeCookie);

    // Zaznacz opcję przy użyciu metody SelectByVisibleText
    language.selectByVisibleText("English");

    // Zmienna storeCookie powinna zawierać informację o wybranym języku
    storeCookie = driver.manage().getCookieNamed("store");
    assertEquals("english", storeCookie.getValue());
}
```

Jak to działa?

Interfejs `WebDriver.Options` dostarcza różne metody do dodawania, czytania, modyfikowania i usuwania plików cookie. W tym przykładzie, gdy zmieniamy język sklepu, do przechowywania preferencji językowych używany jest plik cookie. Ten plik cookie oraz jego wartość możemy odczytać w następujący sposób.

```
Cookie storeCookie = driver.manage().getCookieNamed("store");
assertEquals("english", storeCookie.getValue());
```

Wywołaliśmy metodę `getCookieNamed()`, przekazując nazwę pliku cookie. Metoda zwraca egzemplarz obiektu `Cookie`. Obiekt `Cookie` zawiera różne metody czytania wartości, domeny i tak dalej.

Korzystanie z mechanizmów nawigacji przeglądarki

Przeglądarki zapewniają wiele metod nawigacji umożliwiających dostęp do stron WWW z historii przeglądarki. Pozwalają również na odświeżanie bieżącej strony za pomocą przycisków *Wstecz*, *Dalej* oraz *Odśwież* na pasku narzędzi okna przeglądarki. API Selenium WebDriver zapewnia dostęp do tych przycisków za pomocą różnych metod interfejsu `WebDriver.Navigation`. Przy użyciu tych metod możemy testować zachowanie aplikacji.

Metoda	Opis
`back()`	Metoda powoduje przejście do poprzedniej strony w historii przeglądarki.
`forward()`	Metoda powoduje przejście do następnej strony w historii przeglądarki.
`refresh()`	Metoda powoduje ponowne załadowanie bieżącej strony.
`to(String url)` `to(java.net.URL url)`	Metoda ładuje podany adres URL w bieżącym oknie przeglądarki.

W tej recepturze zaprezentuję metody nawigacji w przeglądarce.

Przygotuj się

Utwórz nowy test, który zbuduje egzemplarz klasy `WebDriver`, przejdzie do witryny i wykona kilka podstawowych działań i weryfikacji.

Jak to zrobić?

Utwórzmy test, który wywołuje różne metody nawigacji i sprawdza zachowanie aplikacji. Jego kod zamieszczono poniżej.

```
@Test
public void testNavigation() {
    driver.get("http://www.google.com");

    // Utworzenie reprezentacji pola tekstowego wyszukiwania
    WebElement searchField = driver.findElement(By.name("q"));
    searchField.clear();

    // Wprowadzenie słowa kluczowego do wyszukiwania i przesłanie żądania
    searchField.sendKeys("selenium webdriver");
    searchField.submit();
    WebElement resultLink = driver.findElement(By.linkText("Selenium WebDriver"));
    resultLink.click();
```

```
        new WebDriverWait(driver, 10).until(ExpectedConditions
            .titleIs("Selenium WebDriver"));

        assertEquals("Selenium WebDriver", driver.getTitle());

        driver.navigate().back();

        new WebDriverWait(driver, 10).until(ExpectedConditions
            .titleIs("selenium webdriver - Szukaj w Google"));

        assertEquals("selenium webdriver - Szukaj w Google", driver.getTitle());

        driver.navigate().forward();

        new WebDriverWait(driver, 10).until(ExpectedConditions
            .titleIs("Selenium WebDriver"));

        assertEquals("Selenium WebDriver", driver.getTitle());

        driver.navigate().refresh();

        new WebDriverWait(driver, 10).until(ExpectedConditions
            .titleIs("Selenium WebDriver"));

        assertEquals("Selenium WebDriver", driver.getTitle());
    }
```

Jak to działa?

Interfejs `WebDriver.Navigation` dostarcza metody `back()` i `forward()` do ładowania stron z historii przeglądarki. Metody te reprezentują przyciski *Wstecz* i *Dalej* dostępne w dowolnej przeglądarce WWW. Możemy również odświeżyć lub ponownie załadować stronę poprzez wywołanie metody `refresh()`.

Korzystanie ze zdarzeń frameworka WebDriver

Selenium WebDriver zawiera klasę `EventFiringWebDriver`, która nasłuchuje różnych zdarzeń zachodzących podczas wykonywania testu. Przykładowo zdarzenia są wyzwalane przy wejściu na stronę, gdy zostanie wykonane kliknięcie elementu albo kiedy zostanie zmieniona wartość. W poniższej tabeli zestawiono listę wszystkich zdarzeń, które możemy śledzić podczas wykonywania testu.

Zdarzenie	Opis
`BeforeNavigateTo`	Metoda jest wywoływana przed wywołaniem metod `get(String url)` lub `navigate().to(String url)`.
`afterNavigateTo`	Metoda jest wywoływana po wywołaniu metod `get(String url)` lub `navigate().to(String url)`.
`beforeNavigateBack`	Metoda jest wywoływana przed wywołaniem metody `navigate().back()`.
`afterNavigateBack`	Metoda jest wywoływana po wywołaniu metody `navigate().back()`.
`beforeNavigateFor`	Metoda jest wywoływana przed wywołaniem metody `navigate().forward()`.
`afterNavigateForward`	Metoda jest wywoływana po wywołaniu metody `navigate().forward()`.
`beforeFindBy`	Metoda jest wywoływana przed wywołaniem następujących metod: `WebDriver.findElement(...)`, `WebDriver.findElements(...)`, `WebElement.findElement(...)`, `WebElement.findElements(...)`.
`afterFindBy`	Metoda jest wywoływana po wywołaniu następujących metod: `WebDriver.findElement(...)`, `WebDriver.findElements(...)`, `WebElement.findElement(...)`, `WebElement.findElements(...)`.
`beforeChangeValueOf`	Metoda jest wywoływana przed wywołaniem metod `WebElement.clear()` lub `WebElement.sendKeys(...)`.
`afterChangeValueOf`	Metoda jest wywoływana po wywołaniu metod `WebElement.clear()` lub `WebElement.sendKeys(...)`.
`beforeClickOn`	Metoda jest wywoływana przed wywołaniem metody `WebElement.click()`.
`afterClickOn`	Metoda jest wywoływana po wywołaniu metody `WebElement.click()`.
`beforeScript`	Metoda jest wywoływana przed metodą `RemoteWebDriver.executeScript(java.lang.String, java.lang.Object[])`.
`afterScript`	Metoda jest wywoływana po metodzie `RemoteWebDriver.executeScript(java.lang.String, java.lang.Object[])`.
`onException`	Metoda ta jest wywoływana zawsze wtedy, gdy ma być zgłoszony wyjątek.

Wymienionych procedur obsługi zdarzeń można użyć do wykonania dodatkowych poleceń. Przykładowo przed wprowadzeniem wartości w polu tekstowym możemy wyczyścić istniejącą wartość lub przechwycić zrzut ekranu nawet wtedy, gdy `WebDriver` zgłosi wyjątek. Można to zrobić, wykonując następujące czynności.

- Utwórz własną klasę nasłuchiwania zdarzeń. Do tej klasy możesz dodać kod, który zostanie uruchomiony po zgłoszeniu specyficznych zdarzeń.

- Za pomocą egzemplarza `WebDriver` utwórz egzemplarz klasy `EventFiringWebDriver`.
- Zarejestruj klasę słuchacza zdarzeń w egzemplarzu klasy `EventFiringWebDriver`.

Klasę słuchacza zdarzeń można utworzyć na dwa sposoby:

- przez zaimplementowanie interfejsu `WebDriverEventListener`,
- przez rozszerzenie klasy `AbstractWebDriverEventListener`.

W tej recepturze pokażę, w jaki sposób używać klasy `EventFiringWebDriver` do nasłuchiwania zdarzeń `WebDriver`.

Przygotuj się

Utwórz nowy test, który zbuduje egzemplarz `WebDriver`, przejdzie do witryny i wykona kilka podstawowych działań oraz weryfikacji.

Jak to zrobić?

Na początek zdefiniujemy klasę nasłuchiwania zdarzeń poprzez zaimplementowanie interfejsu `WebDriverEventListener` w następujący sposób.

```
package com.secookbook.examples.chapter04;

import org.openqa.selenium.*;
import org.openqa.selenium.support.events.WebDriverEventListener;

public class MyListener implements WebDriverEventListener {
    public void beforeChangeValueOf(WebElement element, WebDriver
driver) {
        element.clear();
    }
}
```

Następnie utworzymy test, który używa klasy `EventFiringWebDriver`.

```
package com.secookbook.examples.chapter04;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.events.EventFiringWebDriver;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class EventFiringTest {
    private WebDriver driver;
```

```
    @Before
    public void setUp() throws Exception {
        driver = new FirefoxDriver();
    }

    @Test
    public void testEventFiringWebDriver() throws Exception {

        EventFiringWebDriver eventDriver =
        new EventFiringWebDriver(driver);
        MyListener myListener = new MyListener();
        eventDriver.register(myListener);
        eventDriver.get("http://bit.ly/1DbdhsW");
        eventDriver.findElement(By.id("q"))
            .sendKeys("Selenium i testowanie aplikacji. Receptury");
    }

    @After
    public void tearDown() throws Exception {
        driver.quit();
    }
}
```

Co dalej?

Dodajemy do klasy odbiornika zdarzeń jedną dodatkową procedurę obsługi zdarzenia, która przechwytuje zrzut ekranu w momencie zgłaszania wyjątku.

```
public void onException(Throwable exception, WebDriver driver) {
    try {
    if
(driver.getClass().getName().equals("org.openqa.selenium.remote.Re
moteWebDriver")) {
    driver = new Augmenter().augment(driver);
    }
    File scrFile = ((TakesScreenshot)
driver).getScreenshotAs(OutputType.FILE);
    FileUtils.copyFile(scrFile, new
File("target/screenshots/error.png"));
    } catch (Exception e) {
      e.printStackTrace();
    }
}
```

Skorowidz

E

F

G

H

I

J

K

L

M

N

O

P

W

X

Z

Ż

Selenium i testowanie aplikacji

Receptury. Wydanie II

Projektanci nowoczesnych aplikacji internetowych muszą obecnie sprostać wielu wyzwaniom. Oprogramowanie, które tworzą, musi działać wydajnie i bezbłędnie w różnych przeglądarkach, na różnych platformach, na różnych urządzeniach. Wobec rosnących wymagań testowanie aplikacji staje się bardzo ważnym etapem jej rozwoju. Jako że zaprojektowanie i przeprowadzenie takich testów jest dość złożonym zadaniem, warto zapewnić sobie sprawdzone narzędzie umożliwiające automatyzację testów aplikacji. Takim narzędziem z pewnością jest Selenium Web Driver — framework służący do automatyzacji przeglądarek internetowych.

W tej książce przedstawiono zaawansowane techniki testowania aplikacji internetowych za pomocą Selenium Web Driver i narzędzi pokrewnych. Zaprezentowano skuteczne i efektywne metodyki testowania aplikacji przeznaczonych dla komputerów stacjonarnych, przeglądarek mobilnych i działających w środowisku rozproszonym. Opisano tu również wzorce projektowe, takie jak testy bazujące na danych, obiekty stron i mapy obiektów. Ponadto omówiono techniki rozszerzania frameworka Selenium i dostosowywania go do szczególnych działań. W książce znalazło się ponad 80 receptur, które z pewnością okażą się przydatne podczas automatyzacji testów aplikacji.

Nowoczesne testowanie: Selenium Web Driver!

W książce znajdziesz:

- konfigurację Selenium i narzędzi współpracujących
- techniki i strategie lokalizacji
- stosowanie Selenium do tworzenia i synchronizacji testów
- techniki Behavior-Driven Development (BDD)
- testowanie aplikacji w wielu różnych przeglądarkach, w tym w środowiskach mobilnych

Unmesh Gundecha od kilkunastu lat specjalizuje się w rozwijaniu i testowaniu oprogramowania. Brał udział w licznych projektach automatyzacji testów funkcjonalnych przy użyciu standardowych i niestandardowych frameworków automatyzacji testów w połączeniu z wiodącymi narzędziami. Obecnie jest architektem testów w międzynarodowej firmie w Pune w Indiach.

[PACKT] open source PUBLISHING community experience distilled

Helion

księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-3411-3

cena: 59,00 zł