Testy automatyczne: JavaScript & Protractor

Wprowadzenie

Istnieje wiele języków programowania, z których możemy korzystać w pracy jako testerzy. Na przełomie lat 2015/2016 język *JavaScript* stał się bardzo popularny w branży IT. W tym czasie specyfikacja tego języka została ustandaryzowana zgodnie z **ECMAScript** [14]**.** ECMA to ustandaryzowana specyfikacja skryptowego języka programowania, gdzie najbardziej znane implementacje to JavaScript, JScript i ActionScript. ECMA-262 definiuje semantykę języka oraz niektóre podstawowe typy danych (String, Boolean, Number, Object itp.) i obiekty (np. Math, Array). Standard jest cały czas uaktualniany. Od roku 2015 powstaje nowa wersja, którą zajmuje się komisja TC39, w skład której wchodzą przedstawiciele wszystkich głównych przeglądarek internetowych. Wspomniana standaryzacja była bardzo ważnym krokiem oraz stanowi przełom dla języka JavaScript.

Język JavaScript dotarł również do testów automatycznych. Istnieje również framework o nazwie *Protractor*, który został utworzony w celu umożliwienia automatyzacji testów aplikacjach webowych pisanych w technologii *Angular*. Jest oficjalnym frameworkiem służącym do testowania aplikacji webowych napisanych w technologii angular. Istnieją również inne frameworki, które umożliwiają automatyzację aplikacji webowych pisanych w angularze np. *Cypress.io, Nightwatch.js.* Zalety, jakie posiada protractor można znaleźć czytając artkuł [15].

Protractor współpracuje również z *TypeScript*, który stosuje się jako nakładkę na JavaScript. Typescript pozwala na statyczne typowanie zmiennych, tak jak w językach *C#, Java*. W artykule jednak skupimy się na języku JavaScript.

Korzystanie z języka JavaScript i Protractora wymaga uprzedniej znajomości trzech podstawowych pojęć, takich jak:

a) node.js,
b) NPM (Node Package Module),
c) asynchroniczność,

które zostaną omówione w kolejnych rozdziałach.

Celem artykułu jest przedstawienie czytelnikowi, jak można pisać testy automatyczne wykorzystując JavaScript i Protractor oraz jakie korzyści daje to podejście. Pisząc artykuł zakładamy, że czytelnik zna podstawy języka JavaScript oraz miał styczność z tym językiem.

1. Node.js i NPM

1.1 Czym jest Node.js?

Node to środowisko przeznaczone głównie do tworzenia aplikacji webowych w JavaScript. Sam Node.js został stworzony przez *Ryana Dahla* na początku 2009 roku, jego rozwój sponsorowany był przez firmę *Joyent*, w której pracował [3]. Node.js uruchamia silnik JavaScript V8 (utworzony przez Google) poza przeglądarką, dzięki czemu może być bardzo wydajny, również w testach automatycznych.

Aplikacja Node.js jest uruchamiana na ogół w jednym procesie, bez tworzenia nowego wątku dla każdego żądania. Aczkolwiek są rozwiązania pozwalające na uruchomienie wielu procesów. Node.js zawiera zestaw podstawowych operacji asynchronicznych We/Wy w swojej standardowej bibliotece, które zapobiegają blokowaniu kodu JavaScript. Biblioteki w Node.js są pisane przy użyciu paradygmatów nieblokujących, co powoduje, że zachowanie blokujące wykonanie pewnych poleceń jest raczej wyjątkiem niż normą. Jeżeli Node.js musi wykonać operację We/Wy, np. odczyt z sieci, dostęp do bazy danych to zamiast blokowania wątku i marnowania czekających cykli procesora,

Node.js wznowi operacje po powrocie odpowiedzi [4]. Instalacja Node.js na stacji roboczej jest niezbędna aby korzystać z Protractora.

1.2 Czym jest NPM?

NPM *(Node Package Manager*) to domyślny manager pakietów (bibliotek) dla środowiska Node.js, który może być używany do zarządzania warstwą front-end aplikacji webowych pisanych w języku JavaScript. Może też być używany do pisania aplikacji backendowych w dzisiejszych czasach. Słowo *paczka* oznacza w JavaScript to samo co *biblioteka* w innych językach programowania. Składa się z wiersza poleceń (zwanego także npm), internetowej bazy danych publicznych oraz płatnych pakietów prywatnych. Wspomniana baza często jest nazywana *rejestrem npm.* Rejestr ten jest dostępny za pośrednictwem klienta, natomiast dostępne z nim pakiety można przeglądać i wyszukiwać za pośrednictwem strony internetowej npm.

Aby rozpocząć korzystanie z npm, należy utworzyć plik *package.json*, który zawiera informację o zastosowanych paczkach [13]. Służy do tego instrukcja

npm init

wywołana w odpowiednim katalogu w konsoli windowsa. Niektóre programy np. *InteliJ* mają własną konsolę, w której również można wpisywać tego rodzaju polecenia. Polecenie *npm init* utworzy w projekcie plik *package.json*, który zawiera informacje na temat projektu. Nie jest to jednak obowiązkowe. W przypadku testów automatycznych, w których korzystamy z różnych paczek jest to bardzo pomocne. Osoba, która ściąga kod na swój komputer lokalny, musi już tylko wywołać polecenie

npm install

a to z kolei spowoduje zainstalowanie paczek, które są umieszczone w pliku package.json. Polecenie *npm install* musi być wywołane z tego katalogu (na ogół w głównym katalogu projekt), w którym jest umieszczony ten plik w projekcie. Przykładowy plik może mieć postać:

```
{
  "name": "testProject",
  "version": "1.0.0",
  "description": "",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "doc": "jsdoc -c jsdoc.json"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/jasmine": "3.3.9",
    "@types/jasminewd2": "2.0.6",
    "@types/node": "11.11.0",
    "protractor": "^5.4.2"
  },
  "dependencies": {
    "better-docs": "^1.4.7",
    "chromedriver": "^74.0.0",
    "fs-extra": "^8.1.0",
    "jasmine": "^3.4.0",
    "jasmine-bail-fast": "0.0.1",
    "jasmine-reporters": "^2.0.0",
    "jsdoc": "^3.6.3",
    "module-alias": "^2.2.2",
    "moment": "^2.24.0",
    "protractor-beautiful-reporter": "^1.2.7",
    "testlink-api-client": "0.0.8"
```

```
    }
}
```

Wymagane w projekcie paczki znajdują się w sekcji *dependencies.* Podczas instalowania paczek warto również znać polecenie *-g,* którego użycie spowoduje zainstalowanie paczek globalnie [1]. W takim przypadku paczka będzie dostępna wszędzie tam, gdzie zainstalowany jest Node.js. Aby skorzystać z paczki podczas definiowania metod dla potrzeb testów należy daną paczkę zaimportować w pliku **.js* za pomocą instrukcji:

```
const moment = require('moment');
```

Wywołanie wyżej przedstawionej instrukcji spowoduje zaimportowanie paczki *moment,* która ułatwia pisanie metod operujących na dacie i czasie. Instalowanie nowej paczki wykonuje się za pomocą polecenia:

npm install <package name>

lub skrótowo,

npm i <package name>

UWAGA!!! Udostępniając projekt innym osobom nie powinniśmy przekazywać katalogu *node_modules* (katalogu będący częścią projektu) jako katalogu, który jest wersjonowany, ponieważ mogą wystąpić konflikty uniemożliwiające instalację paczek przez inną osobę. Podczas gdy npm aktualizuje pakiety, generowany jest plik *package-lock.json*, który zawiera listę rzeczywistych wersji pakietu npm używanych w aplikacji, w tym wszystkich pakietów zagnieżdżonych.

2. Asynchroniczność a JavaScript

Pojęcie *asynchroniczności* jest bardzo ważne w programowaniu w JavaScripcie. Nie są to skomplikowane rzeczy, ale chodzi o to aby wiedzieć jakie mogą być tutaj następstwa. JavaScript do tej pory był językiem jednowątkowym, ale można w pewnych przypadkach zasymulować wielowątkowość. Wątki zostały wprowadzone do Node.js, ale póki co nie są jeszcze powszechnie używane. [5] Taka symulacja wymaga przemyślenia i podstawowej wiedzy związanej z asynchronicznością i obiektami typu *Promise*.

Problem, który występuje w jednowątkowych językach programowania jest taki, że w przypadku długich operacji np. wczytywanie zawartości pliku wstrzymane będą pozostałe instrukcje, co spowalnia działanie programu. Jeżeli jednak wczytywanie pliku będzie asynchroniczne, to nie spowolni to wykonania pozostałych instrukcji programu, ponieważ program nie będzie czekał na wczytanie zawartości pliku. Ta operacja będzie zainicjowana, a użytkownik poczeka na jej rezultat „w międzyczasie".

2.1 Callbacki, obiekty typu Promise

W języku JavaScript istnieje pewien mechanizm, który należy znać, jeżeli chcemy pisać testy w protraktorze. Są to tzw. **funkcje zwrotne** nazywane również **callbackami** [16]. W języku JavaScript funkcje są obiektami, dlatego:

- mogą być przekazywane jako parametry do innych funkcji,
- mogę być zwracane przez inne funkcje.

Ważną rzeczą, którą należy wziąć pod uwagę pisząc testy w protraktorze jest to, że **funkcje asynchroniczne nie zwracają wartości.** Zwracają „obietnicę" wykonania pewnej metody lub ciągu instrukcji (funkcje anonimowe również mogą być asynchroniczne). Wspomniana obietnica to obiekt typu *Promise* (zwany również czasami *Future*) [5]. API dostarczone przez twórców protractora działa w sposób asynchroniczny. Metody protractora, które umożliwiają wykonywanie czynności na stronie webowej na ogół będą zwracały promise. Każda funkcja lub ciąg instrukcji użyty przez promise nazywany jest *funkcją wywołania zwrotnego.* Sam callback jest również funkcją zwrotną.

Obiekty typu Promise "obiecują" wykonanie pewnej czynności i następnie zwrócenie rezultatu lub wyrzucenie błędu. Obiekt ten zawsze występuje w jednym z trzech stanów :

- *Pending* - wywołanie zostało zainicjowane ale jeszcze nie ukończone,
- *Fullfilled* - wywołanie zakończone sukcesem,
- *Rejected* - wywołanie zakończone porażką.

Promise, w zależności od wyniku operacji, trafia do jednego z dwóch stanów ponieważ nie może istnieć w obu naraz. Funkcja lub zestaw instrukcji wew. Metody asynchronicznej jest nazywana *callback'iem.* W asynchronicznych metodach obsługujących zakończone czynności możemy zwrócić kolejnego Promise'a, którego będziemy oczekiwali w następnej metodzie itd.

Przykład 1.

```
async getAllElement() {
    await detailsElements.getElement().click().getText();
}
```

Metoda pobiera adres pewnego elementu na formularzu aplikacji a następnie wykonuję akcję kliknięcia. Protractor jest zaprojektowany tak, że jego API (instrukcje służące do wykonania akcji na testowanym formularzu) zawiera metody asynchroniczne. Funkcja *click()* jest również asynchroniczna. Jeśli dojdzie do sytuacji, że element nie pojawi się na stronie, to możemy dostać komunikat typu, że ***click() is not a function.*** Ale konstrukcja Promise'ów pozwala rozwiązać ten problem w taki sposób, jak pokazuje kolejny przykład:

Przykład 2.

```
async getAllElements() {
    const tmp = await detailsElements.getElements()
    await tmp.click();
    await tmp.getText ();
}
```

Różnica pomiędzy przykładami 1 oraz 2 jest taka, że w drugim przykładzie metoda poczeka, aż będzie możliwe wykonanie akcji *click()* (co jest zalecane w przypadku protractora). Taki sposób programowania nazywany pozwala na wygodną obsługę wyniku zwróconego przez Promise'a [6].

Na koniec tego rozdziału musimy jeszcze wspomnieć o dwóch kluczowych słowach: *async i await.* Wspomniane słowa są najczęściej stosowane razem oraz są używane do obsługi promise'ów, przy czym jednocześnie definiują callbacki. Słowo *async* stosuje się tylko do funkcji, co oznacza, że funkcja zawsze zwraca promise'a - nawet jeśli zwracana wartość nie jest promisem (zostanie zwrócony obiekt typu promise, nawet jeżeli funkcja dodatkowo zwróci np. liczbę). Słowo await stosujemy tylko wewnątrz funkcji asynchronicznej. Await z kolei spowoduje, że program poczeka z przejściem do następnej linii kodu, aż nie zakończy się poprzednia instrukcja, co dokładnie pokazuje przykład.

2.2 Promise chaining a async/await

W języku JavaScript metody takie jak *then(), catch(), finally()* zwracają obiekt Promise. Tego typu metody można wywoływać jedną po drugiej. Takie zjawisko jest nazwane jako *promise chaining.*
Kod przedstawiony niżej:

Przykład 3.

```
const result = new Promise((resolve) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
});

result.then((result) => {
    console.log(result); // 10
```

```
    return result * 2;
}).then((result) => {
    console.log(result); // 20
    return result * 3;
}).then((result) => {
    console.log(result); // 60
    return result * 4;
});
```

to przykład programowania nazywanego promise chaining. Taki sposób programowania jest poprawny, ale nie jest najprostszy w zastosowaniu oraz zrozumieniu w przypadku testów automatycznych. Kod napisany w Java Script w postaci promise chaining może mieć np. taką postać w przypadku testów automatycznych:

Przykład 4.

```
verifyOption(expectedText) {
    optionPage.getOptions().then(() => {
        browser.actions()
         .mouseMove(optionPage.getOptions()
         .get(6))}).getText().then((text) => {
        expect(text).toBe(expectedText);
    });
};
```

Taki sposób pisania kodu, jak wyżej, nie jest jednak zbyt czytelny. Idąc po kolei:

i) pobieramy adres lokatora,
ii) przenosimy na niego mysz (takie instrukcje niekiedy są potrzebne, jeśli lokator nie jest wyświetlony na stronie, bo się nie mieści w oknie aplikacji),
iii) pobieramy element listy o indeksie 6,
iv) pobieramy tekst elementu,
v) sprawdzamy, czy tekst jest poprawny.

Jeżeli napiszemy wyżej funkcję z przykładu 4, używając oznaczeń *async* oraz *await,* to jej kod będzie mógł mieć postać:

```
async verifyOptionOne(expectedText) {
    const options = await optionPage.getOptions();
    await browser.actions().mouseMove(options[6]);
    const text = await options.getText();
    return expect(text).toBe(expectedText);
}
```

lub

```
async verifyOptionTwo(index, expectedText) {
    const options = await optionPage.getOptions();
    await browser.actions().mouseMove(options[index]);
    const text = await options.getText();
    return expect(text).toBe(expectedText);
}
```

lub

```
async verifyOptionThree(index, expectedText) {
    const options = optionPage.getOptions();
    const chosenOption = await options.get(index)
    await browser.actions().mouseMove(chosenOption);
    const text = await options.getText();
```

```
    return expect(text).toBe(expectedText);
}
```

Funkcja *verifyOptionOne* na szytwno przyjmuje indeks czytanego elementu. Funkcje *verifyOptionTwo* oraz *verifyOptionThree* różnia się drugą linijką:
- *verifyOptionTwo* ma await przed instrukcją `optionPage.getOptions();`
- *verifyOptionThree* nie ma awaita przed instrukcją `optionPage.getOptions();`

Dodanie awaita przed pobieraniem listy spowoduje, że zwrócona zostanie indeksowana tablica. Brak awaita spowoduje zwrócenie obiektu. Jeśli nastąpi zwrócenie tylko samego obiektu bez indeksów, to należy wskazać ten indeks w kolejnym kroku, jeśli chcemy odczytać atrybut konkretnego pola czy literału.

Teoria mówi, że nie musimy rozbijać metody na pojedyncze atomowe instrukcje, tak jak w wyżej zaprezentowanych przykładach. Słowo await powinno spowodować wywołanie procesu promise chaining. Wobec tego, teoretycznie kod metody *verifyOptionTwo* mógłby być krótszy np.:

```
async verifyOptionTwo(index, expectedText) {
    const options = await optionPage.getOptions().get(index);
    await browser.actions().mouseMove(options);
    return expect(await options.getText()).toBe(expectedText);
}
```

Jednak z praktyki wiemy, że testy automatyczne oraz przeglądarki mają swoje „humory" i czasami dobrą praktyką są atomowe instrukcje, a nie takie, które są pisanie ciągiem, jedna po drugiej.

Przykład 5.

Ciekawym przykładem procesu promise chaining są metody:

```
async checkDescription(text) {
    let desc = null;
    try {
        browser.ignoreSynchronization = true;
        await browser.getWindowHandle();
        const EC = protractor.ExpectedConditions;
        desc = await browser.wait(EC.textToBePresentInElement(await elements.getDesc(), text, 2000));
    } finally {
        browser.ignoreSynchronization = false;
    }
    if (desc !== null) return Promise.resolve(true);
    return Promise.resolve(false);
}
```

lub

```
async describeClickedElement(element) {
  return element.click().then(async () => {
    console.elementClickedLog(element);
    return element;
  }).catch((error) => {
    console.log(error);
    throw error;
  });
}
```

Metoda *checkDescription* sprawdza poprawność tekstu na wyskakującym popupie. W tym celu wymagane jest chwilowe wyłączenie synchronizacji z angularem `browser.ignoreSynchronization = true;` (nie jest to reguła, ale zdarzają się takie sytuacje). Pierwszy wykonywany jest blok instrukcji zawartych w *try*. Jeśli dowolna instrukcja z tego bloku nie zadziała z jakiegoś powodu, to synchronizacja z angularem i tak zostanie włączona, ponieważ wykonany będzie blok *finally* `browser.ignoreSynchronization = false;` .

Z kolei metoda *describeClickedElement* próbuje kliknąć element na stronie webowej i wypisuje do logu w co klika. Jeśli element nie jest klikalny lub z jakiegoś powodu nie pojawi się na testowanej stronie, zwrócony zostanie komunikat błędu za pomocą bloku *catch* oraz instrukcji `throw error;`.

3. Jasmine, Proctartor, Webdriver

Każdy projekt zawierający testy automatyczne składa się z następujących elementów, które muszą współistnieć oraz współpracować ze sobą:

a) dane testowe wejściowe,
b) dane testowe wyjściowe (oczekiwane rezultaty testów),
c) język programowania oraz kod testów,
d) framework zawierający metody pozwalające wykonywać na elementach aplikacji, takie czynności jak: kliknięcie, wpisanie znaku w pole, odczyt tekstu z pola, ciąg akcji (rozwijalne zakładki), identyfikowanie elementów strony za pomocą określonego atrybutu,
e) zestaw atrybutów albo funkcji (zależnie od języka oraz frameworka testowego), które sterują uruchomieniem, wykonaniem oraz zakończeniem testów.

Wyżej wymienione cechy są wspólne dla każdego projektu związanego z automatyzacją. Różnice pojawiają się tylko wtedy, gdy zmienia się technologia lub frameworki przeznaczone do automatyzacji.

3.1 Sterowanie testami za pomocą Jasmine

Jasmine to oddzielny framework, który został wykorzystany jako biblioteka przez twórców Protractora. Jest to zbiór metod odpowiadających w protractorze za sterowanie wykonaniem testów oraz współpracujących z technologią Node.js. Jasmine może być również wykorzystywany do tworzenia testów automatycznych w oparciu o podejście BDD.

Każdy projekt związany z testami automatycznymi musi posiadać tego rodzaju funkcje lub atrybuty. W przypadku Jasmine są to funkcje. Wśród podstawowych takich funkcji możemy wyróżnić [7]:

- *describe(description, specDefinitions*) – tworzy grupę testów, zwaną często „test suite". Pierwszy parametr to opis funkcji. Jako drugi parametr podaje się definicję scenariusza testowego, np. w postaci funkcji anonimowej (funkcja, która nie posiada swojego aliasu),

- *it(description, testFunction, timeoutopt)* – definiuje jeden test case. Pierwszy parametr to opis tekstowy, drugi to callback, który musi zawierać przynajmniej jedną instrukcję. Jeżeli wszystkie wyrażenia zawarte wew. funkcji *it* zostaną wykonane (z oczekiwanym rezultatem, jeśli taki jest zdefiniowany), to test case zakończy się sukcesem. Jeśli natomiast dowolna instrukcja nie zostanie wykonana, to test zakończy się niepowodzeniem,

- *beforeEach(function, timeoutopt)* – uruchamia zestaw instrukcji, które są wykonywane przed każdym testem umieszczonym wewnątrz funkcji describe; beforEach również należy umieścić wewnątrz describe,

- *afterEach(functionopt, timeoutopt)* – uruchamia zestaw instrukcji, które są wykonywane po każdym jednym teście umieszczonym wewnątrz funkcji describe. Tę funkcję również umieszcza się wewnątrz describe,

- *beforeAll(functionopt, timeoutopt)* – służy do tego, aby wykonać zestaw instrukcji jeden raz, przed wszystkimi testami. Tę funkcję również umieszcza się wewnątrz describe,
- *expect(actual)* – jest wykorzystywana do tworzenia asercji,
- *xdescribe, xit* - te funkcje pomijają odpowiednio zestaw testów albo pojedynczy test,
- *fdescribe* – zastosowanie tej funkcji spowoduje wykonanie tylko jednego zestawu testów (oznaczonego przez fdescribe), natomiast spowoduje pominięte tych zestawów, które są oznaczone funkcją decsribe,
- *fit* – spowoduje wykonanie tylko jednego testu z całego zestawu, natomiast pozostałe testy (it-y) zostaną pominięte.

Funkcji xit, xdescribe, fit oraz fdescribe można używać gdy zachodzi potrzeba uruchomienia tylko część testów, np. wykonanie testów dymnych (smoke tests). Przykładowa struktura zestawu testów może mieć postać:

```
const userData = require('@data/usersData.td');
const testData = require('@data/operations.td');

const helper = require('helper');

describe(First_test, () => {
    const testCase = testCaseData.test_1;
    const user = userData.user_1;
    const credentials = {
        username: user.username,
        password: user.password,
        sms: user.sms
    };

    beforeAll(async () => {
        await browser.get(browser.baseUrl);
        await helper.login(credentials);
    });

    beforeEach(async () => {
        await helper.navigateToDashboard ();
    });

    afterEach(async () => {
        await browser.refresh();
    });

    afterALL(async () => {
        await browser.quit();

    });

    it(start test 1, async () => {
        global.caseId = testCase.testId;
        expect(await helper.validMailScreen(testData.operations))
            .toBe(true, Main screen is not correct);
    });

    it(start test 2, async () => {
        global.caseId = testCase.testId;
        expect(await helper.validPrints(testData.prints))
            .toBe(true, Main screen is not correct);
    });
});
```

Wyżej przedstawiony przykład zawiera dwa testy oraz odpowiednie funkcje sterujące, które ustawiają warunki początkowe. Przed definicją testów odbywa się import klas, które zawierają metody wykorzystywane w testach oraz w warunkach początkowych. Służy do tego funkcja *require.* Każdy zaimportowany plik jest zmienną stałą, poprzedzoną atrybutem *const*. Każda funkcja sterująca zawiera wewnątrz funkcję anonimową (która jest callbackiem). Słowa kluczowe *async* oraz *await* są używane, ponieważ API dostarczone przez protractor zawiera funkcje asynchroniczne.

Uruchomienie testów na serwerze można zdefiniować za pomocą plików konfiguracyjnych, które zawierają odwołania do testów. Pliki konfiguracyjne to zwykłe pliki **.js:*

```
const config = base.config;

config.suites = {
        dashboard: '../spec/dashboard',
        prints: '../spec/prints,
};

exports.config = config;
```

Pliki z zestawem testów na ogół umieszcza się w katalogu *spec,* a ich nazwy zwykle zawierają słowo spec np. *testSuite-dashboard.spec.js.* Wywołanie takiego pliku z konsoli Windowsa spowoduje uruchomienie testów.

3.2 Webdriver && Protractor API

3.2.1 Webdriver manager

Serwer selenium (na którym jest oparty protractor) oraz driver manager są zawarte w repozytorium protractora. Aby rozpocząć pracę z testami automatycznymi, należy wywołać komendę:

npm install -g webdriver-manager

w konsoli systemu operacyjnego lub w konsoli środowiska (np. InteliJ), w którym piszemy testy automatyczne. To polecenie spowoduje zainstalowanie biblioteki, która umożliwi współpracę metod API protractora z przeglądarką. Metody protractora to instrukcje pozwalające wykonywać akcje na stronie internetowej, analogicznie jak API samego selenium. Po wywołaniu tego polecenia, w projekcie pojawi się plik *webdriver.js*, a również zostanie zainstalowany globalnie [8]. Zainstalowana zostanie też biblioteka *Selenium Standalone*, która domyślnie zawiera takie drivery jak:

- *ChromeDriver*
- *FirefoxDriver (GeckoDriver)*
- *IEDriver*
- *Edge WebDriver*

Ścieżka do plików z driverami na lokalnej stacji roboczej zazwyczaj ma postać:
C:\...*node_modules**protractor**node_modules**webdriver-manager**selenium*

Javascript oraz przeglądarka ciągle są rozwijane, w związku z czym co jakiś czas należy wykonywać aktualizację webdrivera. Służy do tego polecenie:

webdriver-manager update

Domyślnie serwer selenium będzie uruchamiany pod adresem *http://localhost:4444/wd/hub*. W celu uruchomienia serwera wykonuje się polecenie:

webdriver-manager start.

3.2.2 Uruchomienie testów z protractorze

Uruchomienie testów w protractorze wymaga instalacji takich komponentów jak:

a) *npm* (analogicznie jak w rozdziale 1.1),
b) *Java Development Kit (JDK)* – wymagana do uruchomienia standalone Selenium Server,
c) Instalacja protractora za pomocą pakietu npm (zostaje również zainstalowany webdriver-manager).

Instalację protractora można wykonać w konsoli systemu za pomocą polecenia:

i) *npm install -g protractor*

Kolejne polecenia jakie należy wykonać to:

ii) *webdriver-manager update* – to polecenie zaktualizuje webdrivera, jeżeli na danej stacji roboczej lub na danym serwerze nie mamy aktualnej wersji.

oraz

iii) *webdriver-manager start* – to polecenie spowoduje uruchomienie serwera Selenium. Jeśli serwer uruchomi się poprawnie, to w konsoli systemu operacyjnego bądź w konsoli narzędzie do programowania, otrzymamy komunikaty, o podobnej jak niżej treści:

```
C:\TEST\auto>webdriver-manager start
webdriver-manager: using local installed version xxxx
……..
 C:\TEST\auto\node_modules\webdriver-manager\selenium\selenium-server-standalone-3.141.59.jar -port 4444
[19:39:34] I/start - seleniumProcess.pid: 25940
19:39:34.899 INFO [GridLauncherV3.parse] - Selenium server version: 3.141.59, revision: e82be7d358
19:39:34.994 INFO [GridLauncherV3.lambda$buildLaunchers$3] - Launching a standalone Selenium Server on port 4444
19:39:35.066:INFO::main: Logging initialized @547ms to org.seleniumhq.jetty9.util.log.StdErrLog
19:39:35.326 INFO [WebDriverServlet.<init>] - Initialising WebDriverServlet
19:39:35.664 INFO [SeleniumServer.boot] - Selenium Server is up and running on port 4444
```

Ostatnia czynność jaka została, to wskazanie pliku *.spec.js, który zawiera test lub zestaw testów oraz uruchomienie tego pliku. Przykładowy plik config.js może mieć postać:

Przykład 6.

```
// config.js
exports.config = {
    framework: 'jasmine',
    seleniumAddress: 'http://localhost:4444/wd/hub',
    specs: ['spec.js']
}
```

Jeżeli plik spec.js zawiera testy (poprzedzone funkcją *describe*), to po wywołaniu w konsoli polecenia:

protractor conf.js

zostaną uruchomione testy zawarte w pliku *spec.js.* plik config.js może mieć inny schemat niż ten z przykładu 5. Do pliku konfiguracyjnego można przekazać ustawienia przeglądarki, środowiska itd. Zależy to od sposobu ułożenia struktury projektu z testami. O tym wspomnimy w rozdziale 3.3.

3.2.3 ElementFinder && ElementArrayFinder

W każdym frameworku przeznaczonym do testów automatycznych GUI istnieją metody, które umożliwiają pobieranie adresów elementów danego formularza. Protractor zawiera dwie klasy, które służą do tego celu: *ElementFinder* oraz *ElementArrayFinder.*

ElementArrayFinder służy do operacji na tablicy elementów (w przeciwieństwie do pojedynczego elementu), natomiast ElementFinder reprezentuje pojedynczy element ElementArrayFinder. W rezultacie wszystko, co można zrobić za pomocą ElementFinder, można również wykonać za pomocą ElementArrayFinder. Dokumentacja dotycząca lokatorów w protractorze jest w bardzo czytelny sposób dostępna pod linkami [9], [10]. Protractor umożliwia identyfikowanie lokatorów poprzez łączenie metod z zestawów ElementFinder oraz ElementArrayFinder:

Przykład 7.

```
getActions() {
    return $("[class *= 'summary-bar-actions']").$$("button > span");
}
```

lub

```
getActions() {
    return $('.summary-bar-actions').$$("button > span");
}
```

Pierwszy selektor zadziała tak, że najpierw znajdujemy element o klasie *summary-bar-actions* a następnie wyszukuje wewnątrz klasy zawierającej tekst *summary-bar-action* elemety, *span* poprzedzone atrybutem *button*. Drugi selektor zadziała analogicznie ale z tą różnicą, że znajdzie element o tekście **równym** *summary-bar-actions*.

Łączyć ze sobą można również listy:

Przykład 8.

```
wrappers() {
    return $$("[class *= 'wrappers]");
}
```

```
async wrappersOptions(index) {
    const wrappers = await this.wrappers();
    const options = await wrappers[index].$$("[class *= 'options']");
    return options;
}
```

Pierwszy selektor zwróci listę o klasie *wrappers.* Drugi z kolei zwróci listę wrapper o ustalonym indeksie oraz zawarte w nim opcje o klasie zawierającej tekst *options*. Metoda wrappersOptions jest asynchroniczna, ponieważ zawiera instrukcje:

```
const wrappers = await this.wrappers();
```

Użycie słowa await spowoduje zwrócenie tablicy elementów, przez co odwołujemy się już do konkretnego elementu tablicy po indeksie. Brak awaita spowodowałby z kolei zwrócenie obiektu. Nie jest konieczne używanie awaita. W takim przypadku kod selektora miałby taką postać:

```
wrappersOptions(index) {
    const wrapper = this.wrappers().get(index);
    const options = wrapper.$$("[class *= 'options']");
    return options;
}
```

Słowo *this* oznacza w tym przypadku, że bierzemy metodę z tej samej klasy w której jesteśmy. Znaczenie tego słowa w języku JavaScript jest dość obszerne. Więcej można poczytać na ten temat w pozycji [17]. Instrukcja:

```
const wrappers = await this.wrappers().get(index);
```

spowoduje wybranie konkretnego elementu z obiektu zwróconego za pomocą metody wrappers(). Słowo await utworzy promise, a przekazany selektor będzie callbackiem. Wynik takiej instrukcji będzie rozwiązaniem w postaci zwróconej tablicy.

Klasa ElementArrayFinder zawiera m. in. takie metody jak:

a) *first() – zwraca pierwszy znaleziony element listy :*

```
return $$("[class *= 'comboOptions']").first();
```

b) *last() – zwraca ostatni napotkany element listy:*

```
return $$("[class *= 'comboOptions']").last();
```

c) *get(index) – zwraca wskazany element dla wczytanej listy, jest uogólnieniem metod first() oraz last():*

```
return $$("[class *= 'comboOptions']").get(3);
```

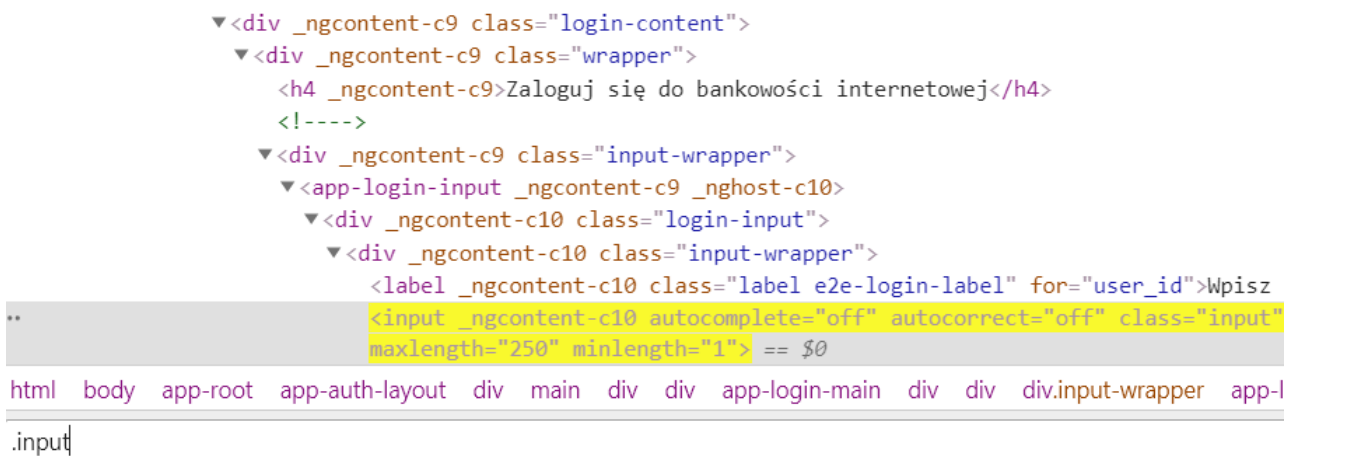
d) *map() – przypisuje elementom listy konkretny atrybut, który istnieje dla tej listy:*

```
const tab = await listElements.getList();
const tabText = await tabs.map(async element => await element.getText());
const textTab = [];
for (let i = 0; i < 4; i++) {
    await actual.push(tabsLiterals[i]);
}
```

e) *count() – zwraca liczbę elementów listy:*

```
return $$("[class *= 'comboOptions']").count();
```

element() – jest odpowiednikiem funkcji $(). W protractorze możemy też wyszukać klasę input za pomocą funkcji $('.input')




```
return element(by.name("login"));
```

f) element.all() – jest odpowiednikiem funkcji $$() (szukanie listy elementów):

```
return element.all(by.css('menu-item'));
```

Protractor umożliwia również tworzenie bardzo zawiłych selektorów, złożonych z kilku funkcji np.:

```
selectList: element.all(by.css("[class *= 'content']")).last()

getList: function () {
return this.selectList;
}

getCss: function () {
  return this.getList().element(by.css("['css']"));
}

getCheckbox: function (text) {
  return this.getList().element(by.cssContainingText('checkbox', text));
}
```

Takie podejście nie jest jednak zalecane. Powinno się je stosować tylko wtedy, gdy nie istnieje inny sposób. Długie selektory zwłaszcza te, które są identyfikowane poprzez funkcję *cssContainingText()* działają wolniej niż krótkie i proste selektory.

4. Wybrane funkcje użyciem protractora

4.1. Testowanie pobierania plików

W testowaniu trafiają się przypadki testowe dotyczące pobierania plików. Załóżmy, że nie musimy weryfikować zawartości pliku, natomiast sprawdzić tylko jego nazwę i rozszerzenie. W tym celu niezbędne będą takie metody jak:

1. *pobranie nazwy przeglądarki,*
2. *weryfikacja, czy przeglądarka wspiera testowanie pobierania pliku,*

3. *Pobranie ścieżki do której trafi pobrany plik,*
4. *czekanie aż plik się ściągnie,*
5. *sprawdzenie nazwy pobranego pliku.*

W tym celu wykorzystane zostaną biblioteki z zestawu npm, takie jak: *rimraf, fs, path*. Biblioteki *fs* i *path* wchodzą w skład node'a, natomiast bibliotekę *rimraf* musimy zainstalować z zewnątrz. Importu tych paczek należy dokonać za pomocą instrukcji:

```
const rimraf = require('rimraf');
const path = require('path');
const fs = require('fs');
```

Ad. 1 Metoda, która pobiera nazwę przeglądarki przykładowo może mieć postać:

```
async getBrowserName() {
    const capabilities = await browser.getCapabilities();
    const browserName = capabilities.get('browserName');
    return browserName;
}
```

Metoda getCapabilities() zwraca promise, którego rozwiązaniem są własności przeglądarki. Z kolei metoda get('browserName') zwraca nazwę przeglądarki.

Ad. 2 Przykładowa metoda, która weryfikuje możliwość pobierania pliku może mieć postać:

```
async isDownloadSupported() {
    const browserName = await this.getBrowserName();
    return browserName === 'chrome' || 'firefox';
}
```

Dlaczego taka metoda ma sens? Dlatego że nie każdy framework współpracuje z przeglądarką IE lub Edge. Nie zawsze będziemy mogli skonfigurować parametry przeglądarki tak, aby na każdej z nich było możliwe pobieranie pliku. Metoda isDownloadSupported() zwróci wartość true, jeśli nazwa przeglądarki jest równa chrome lub firefox. W przeciwnym przypadku zwróci wartość false.

Ad. 3 Metoda, która pobiera ścieżką docelową pobranego pliku może zostać zaimplementowana w ten sposób:

```
async getDownloadDirectory() {
    const browserName = await this.getBrowserName();
    const dir = await browser.getProcessedConfig();

    switch (browserName) {
    case 'chrome':
        return dir.env.chromeCapabilities.chromeOptions.prefs.download.default_directory;
    case 'firefox':
        return dir.env.firefoxCapabilities['moz:firefoxOptions'].prefs['browser.download.dir'];
    default:
        return null;
    }
}
```

Funckja getProcessedConfig() zwraca własności aktualnie uruchomionej przeglądarki. Wśród nich można odczytać docelowy katalog, do którego trafi pobrany plik.

Ad. 4 Metoda, która pozwoli poczekać na plik do momentu jego pobrania, może mieć postać:

```
async waitForFile(absolutePath, time) {
    return browser.driver.wait(() => fs.existsSync(absolutePath), time);
}
```

Funkcja existsSync(absolutePath) sprawdza, czy istnieje ścieżka, do której powinien trafić plik. Nie zwraca obiektu typu promise. Time to czas, który mówi o tym, ile czekamy na plik.

Ad. 5 Metoda, która pobierze plik, może być zaimplementowana tak:

```
async isFileDownloaded(expectedFileExtention) {
    const isSupported = await this.isDownloadSupported();
    if (!isSupported) {
        return false;
    }
    const action = await elements.getDownloadActions()
    await action.click();
    const downloadDir = await this.getDownloadDirectory();
    const filePath = path.join(downloadDir, path.extname(expectedFileExtention));
    const isDownloaded = await waitHelper.waitForFile(filePath);
    return isDownloaded;
}
```

Instrukcja:

```
const filePath = path.join(downloadDir, path.extname(expectedFileExtention));
```

utworzy stałą (obiekt), która będzie zawierała takie własności, jak ścieżka docelowego pliku oraz oczekiwane rozszerzenie pobranego pliku (path.extname(expectedFileExtention)). Jeśli rozszerzenie oraz katalog będą zgodne z oczekiwaniami, to metoda zwróci wartość true, w przeciwnym przypadku wartość false. Jeśli chcemy weryfikować nazwę pliku, to wystarczy zmienić wyżej zaprezentowaną metodę w taki sposób:

```
async isFileDownloaded(expectedFileName) {
    const isSupported = await this.isDownloadSupported();
    if (!isSupported) {
        return false;
    }
    const action = await elements.getDownloadActions();
    await action.click();
    const downloadDir = await this.getDownloadDirectory();
    const filePath = path.join(downloadDir, expectedFileName);
    const isDownloaded = await waitHelper.waitForFile(filePath);
    return isDownloaded;
}
```

Druga wersja tej metody jest dokładniejsza, ponieważ sprawdzi zarówno nazwę pobranego pliku, jak i jego rozszerzenie. Pisząc testy automatyczne, które sprawdzają pobieranie pliku, możemy mieć potrzebę usuwania plików już pobranych. Wobec tego należy napisać metodę, która to zrobi za nas. W tym celu można wykorzystać bibliotekę *rimraf.* [11] Kod takiej metody może mieć postać:

```
async cleanDownloadDirectory() {
    const downloadDirectory = await this.getDownloadDirectory();
    if (downloadDirectory) {
        return new Promise((resolve, reject) => {
            rimraf(path.join(downloadDirectory, '*.*'), (error) => {
                if (!error) {
                    resolve(downloadDirectory);
                } else {
                    reject(error);
                }
            });
        });
    }
    return Promise.resolve(downloadDirectory);
}
```

Metoda cleanDownloadDirectory() działa w ten sposób, że na początku sprawdza czy istnieje katalog, w którym mogą znajdować się pobrane pliki. Jeśli taki istnieje, to zwracamy obiekt typu promise, który czyści wskazany katalog. Jeśli czyszczenie nie zadziała (nie będzie pobrany katalog), to utworzony zostanie promise, który zwróci wynik kodu:

```
await this.getDownloadDirectory();
```

Instrukcja :

```
rimraf(path.join(downloadDirectory, '*.*')
```

wyczyści wszystkie pliki z katalogu downloadDirectory.

4.2. Pobieranie własności ukrytych elementów

W aplikacjach webowym może zaistnieć taka sytuacja, że wartości pól edytowalnych lub nieedytowalnych, dowolne przyciski czy, linki nie mieszczą się na ekranie komputera, na którym uruchamiane są testy automatyczne. Jeżeli chcemy wykonać akcję na dowolnym elemencie formularza, to najpierw należy ten element wczytać za pomocą odpowiedniego selektora, a później już wykonać akcję.

W praktyce, może zaistnieć taka sytuacja, że element zostanie znaleziony za pomocą selektora, natomiast nie będzie możliwa na nim do wykonania pewna akcja, na której nam zależy. Taki problem można często rozwiązać za pomocą bardzo prostej instrukcji:

```
await browser.actions().mouseMove(webElement).perform();
```

Metoda *actions()* pozwala na wykonanie sekwencji zdarzeń, takich jak:

- *mouseDown(),*
- *mouseMove(webElement),*
- *mouseUp(),*
- *doubleClick(),*
- *sendKeys(text),*
- *click(),*
- *dragAndDrop(webElement).*

Ciąg akcji należy zawsze zakończyć instrukcją *perform()*. Instrukcja *actions()* nie zwraca promise'a, natomiast metody protractora już tak. Wobec tego, w większości przypadków użycia, polecenie *browser.actions()…* będzie poprzedzone słowem *await'em.* A to z kolei pociąga za sobą słowo *async* przed nazwą metody, która będzie zawierała sekwencję:

```
await browser.actions()................perform();
```

4.3 Biblioteka *moment*

W testach automatycznych mamy do czynienia bardzo często z datami i różnego rodzaju formatami dat**.** W tego rodzaju przypadkach bardzo dobrze sprawdza się biblioteka *moment*. Zawiera ona metody, które pozwalają wykonywać operacje na datach. Więcej o tej bibliotece można przeczytać w pozycji [11]. Bibliotekę należy zaimportować analogicznie, jak inne biblioteki:

```
const moment = require('moment');
```

Załóżmy, że chcemy mieć funkcję, która dodaje do bieżącej daty (bądź odejmuje od niej) rok, miesiąc albo dzień. Przykładowa taka metoda może mieć postać :

```
addPeriodToDate(days, months, years) {
    const date = moment();
    date.add(days, 'days');
    date.add(months, 'month');
    date.add(years, 'year');
    return date.format('DD.MM.YYYY');
}
```

Funkcja *addPeriodToDate()* działa w ten sposób, że wpierw tworzy stałą *date* która zawiera rok, miesiąc, dzień godzinę, minutę z momentu wywołania metody *moment().* Polecenie *add(numer, period)* dodaje bądź odejmuje pewien okres czasu od stałej *date*, gdzie

- *numer* – liczba całkowita (może być ujemna),
- *period* - okres który dodajemy/odejmujemy.

Funkcja format() tworzy taki format daty, jaki jest potrzebny w teście lub w metodzie wykorzystującej datę. Formatowanie możemy definiować na kilka sposobów, np.:

```
moment().format('DDMMYYYYhhmm');
moment().format('DDMMYYYY');
moment().format('DD.MM.YYYY');
moment().format('YYYY.MM.DD');
moment().format(' DDMMYYYY'');
```

5. Podsumowanie

Celem napisania artykułu było przede wszystkim zebranie najważniejszych wiadomości, jakie należy mieć, aby rozpocząć pracę jako tester automatyczny, który korzysta z protractora. W tym przypadku okazuje się, że nie wystarczy tylko wiedzieć, że są metody które coś robią na stronie. Dochodzi tutaj asynchroniczność, callbacki, promise'y, które należy zrozumieć, aby móc poprawnie używać JS oraz protractora. Artykuł z pewnością pomoże rozpocząć pracę z protractorem każdej osobie, która zacznie z niego korzystać po raz pierwszy.

Bibliografia:

[1] https://blog.gutek.pl/2014/05/26/co-to-jest-npm/

[2] https://www.protractortest.org/#/

[3] https://pl.wikipedia.org/wiki/Node.js

[4] https://nodejs.dev/

[5] https://fsgeek.pl/post/asynchronicznosc-w-javascript/

[6] https://webroad.pl/javascript/746-synchroniczna-asynchronicznosc

[7] https://jasmine.github.io/api/3.5/global

[8] https://www.npmjs.com/package/webdriver-manager

[9] https://www.protractortest.org/#/api?view=ElementArrayFinder

[10] https://www.protractortest.org/#/api?view=ElementFinder

[11] https://www.npmjs.com/package/rimraf

[12] https://momentjs.com/docs/

[13] https://docs.npmjs.com/

[14] https://pl.wikipedia.org/wiki/ECMAScript

[15] https://hackernoon.com/cypress-io-vs-protractor-e2e-testing-battle-d124ece91dc7

[16] https://www.nafrontendzie.pl/javascript-metody-obiektu-jako-callback

[17] https://typeofweb.com/this-js-kontekst-wywolania-funkcji/

Recenzent: Paweł Kołodziej, Krzysztof Kołodziejczyk

Marek Żukowicz jest absolwentem matematyki na Uniwersytecie Rzeszowskim. Jest testerem oprogramowania w firmie **Ailleron**. Jego zainteresowania skupiają się wokół testowania, matematyki, AI, zastosowania modeli matematycznych w procesie testowania, automatyzacji testów.