Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Financial Services

Protractor Whitepaper

Angular JS - Protractor E-2-E Testing

Prepared By: Subrahmanyam Baratam

ING - Hyderabad

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Table of Contents

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Financial Services

Executive Summary

This white paper discusses about Protractor & how can Angular JS application be end to end tested.

Setting up the environment which includes Protractor, Selenium Server, and Browser set up in order to test the application.

Protractor scripts to different scenarios in order to cover the functionality of the application and comparing expected and actual results.

Protractor API, listing classed and methods exposed by Protractor API.

Why to use Protractor:

1. It is an end-to-end test framework for Angular JS applications. It runs tests against the application running in a real browser, interacting with it as a user would☺.

2. **Testing like real user:**
   It is built on top of Web Driver JS, which uses native events and also browser specific drivers to interact with the application.

3. **Less setup effort:** Supports Angular-specific strategies, which allows testing Angular-specific elements without any much setup effort.

4. **Automatic Waiting:** No longer need to add explicit waits and explicit sleeps to the test. Protractor can automatically execute the next step in the test the moment the webpage finishes pending tasks, so no need to worry about waiting for the test and webpage to synchronise.

Prerequisites for setup

Protractor is a Node js program, so in order to run Node js should be installed.
Download the protractor package using npm which comes with Node js.
Check the version of Node JS (node –version) and check the compatibility of the protractor README to make sure both are compatible with each other.

Protractor setup

Use npm to install Protractor globally, ignore –g if you would prefer not to install globally.

Command:

npm install -g protractor

Verify:

Run the command protractor –version to find out the installed protractor is working fine or not.

Set up of Selenium Server

When working with protractor, specifying how to connect to browser drivers which will start up and control the browsers in which the application is going to be tested.

Selenium server acts as proxy between the test script and the browser driver.
The server forwards the commands from script to driver and returns responses from driver to script. The server can even handle multiple scripts and multiple browsers.

Test Scripts < --------- > Selenium Server < --------- > Browser Drivers

Run Selenium Server

To run the selenium server on local machine below installation are required.

JDK

To run the Selenium server on local machine JDK(Java development Kit) is mandatory, check if JDK is already installed on the local machine, run the below command in the command line to know the java version.

Command: Java –version

Installing and Start of Selenium Server:
To install & starting the selenium server use the webdriver-manager command line tool which comes from protractor.

Install: To install server & chrome driver
Webdriver-manager update

Start: To start the server
Webdriver-manager start

Note: While running the scripts the server should be running.

Set up of Browser

Protractor works with Selenium Web Driver, a browser automation framework. Selenium Web Driver supports several browser implementations or drivers which are discussed below.

Browser Support:
Protractor supports the latest major versions of Chrome, Firefox, Safari, and IE.

Configuring Browsers:
In the protractor config file all the browser related setup is done by capabilities object, this object is directly passed to the web driver builder.

Using Browsers:
To use a browser other than Chrome, simply set a different browser name in the capabilities object.

```
capabilities: {
  'browserName': 'firefox'
}
```

Multiple Browsers:

In order to test the scripts against multiple browsers multiCapabilities object needs to be used.
Protractor will run tests in parallel against each set of capabilities.
If multiCapabilities are defined, the capabilities configuration object is ignored.

```
multiCapabilities: [
{
  'browserName': 'firefox'
},

 {
  'browserName': 'chrome'
}
]
```

Writing a Script

Protractor needs two files to run, a **spec file** and a **configuration file**.
Let's write a simple script which checks the browser title.

```
//spec.js (spec file)

describe('Simple App', function() {
  it('should have a title', function() {
    browser.get('http://localhost/.../protractor-demo/');

    expect(browser.getTitle()).toEqual('Simple App Title');
  });
});
```

By default Jasmine is the default test framework.

Describe, it syntax is from the Jasmine framework.
Browser is a global created by Protractor, which is used for browser-level commands such as navigation with **browser. get**.

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

The above configuration file tells Protractor where the test files (specs) are, and where to talk to the Selenium Server (selenium Address).
Also specifies using Jasmine for the test framework.

To run our spec in which the script is written, run the below command,

Command: protractor conf.js

After executing, you should see a browser window open up and navigate to the application, then close itself (this should be very fast!).
The test output should be 1 test, 1 assertion, 0 failures.

Interacting with Elements

Let's try by modifying the elements on the page, let's modify the spec.js,

```
//spec.js (spec file)
describe('Simple App to check result', function() {
  it('should have a title', function() {
    browser.get('http://localhost/.../protractor-demo/');

//set value 1 to text field
element(by.model('first')).sendKeys(1);
//set value 1 to text field
    element(by.model('second')).sendKeys(2);
//select and submit the button
    element(by.id('submitbutton')).click();
//check the result in the field latest
    expect(element(by.binding('latest')).getText()).
       toEqual('5'); // This is wrong!
  });
});
```

This script uses the global element, by which are also created by Protractor. The element function is used for finding HTML elements on your webpage.
We used sendKeys to type into <input>, click to click a button, and getText to return the content of an element.

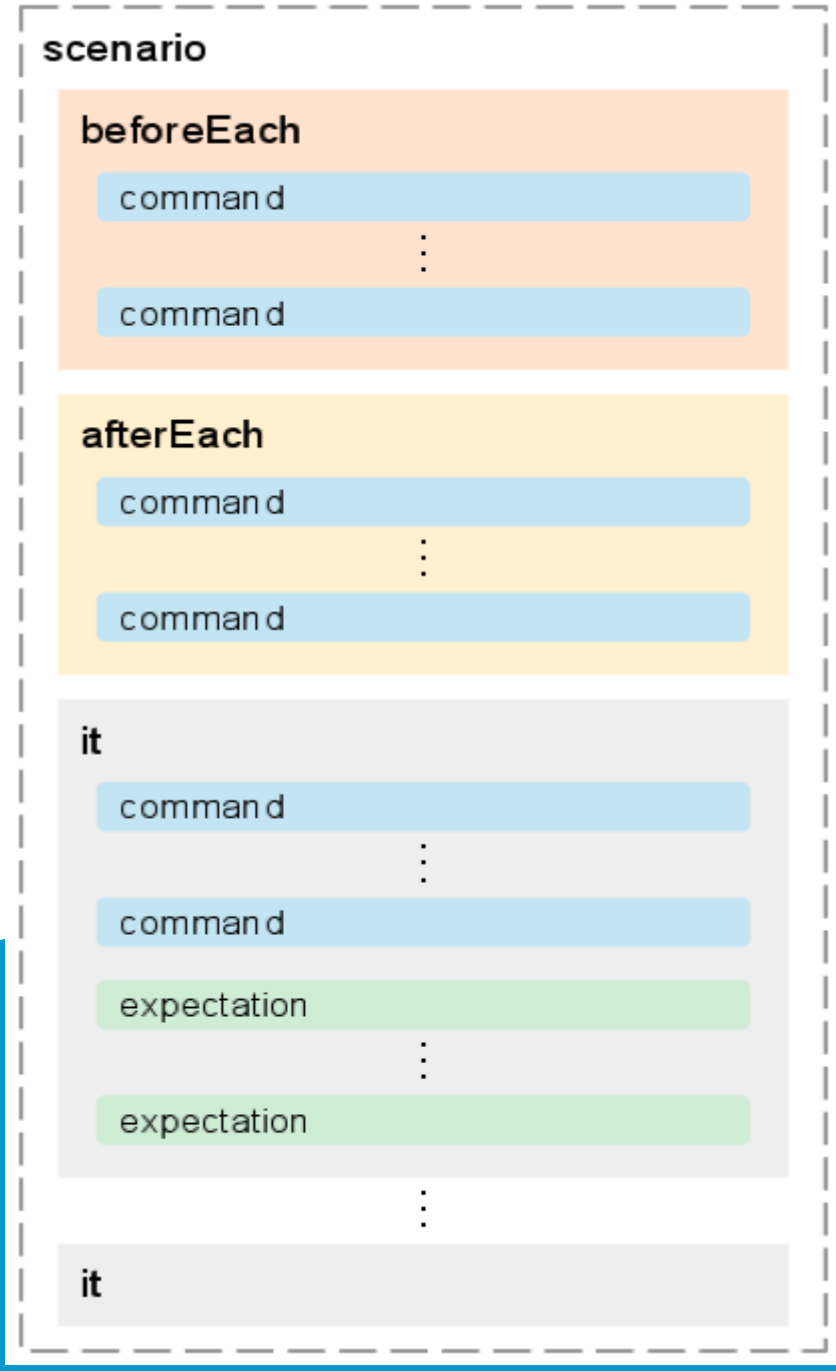
After executing the above script, the page enters two numbers and wait for the result to be displayed. Because the result is 3, not 5, our test fails. Fix the test and try running it again.

Multiple Scenarios

We have written two scripts & now let's put these two tests together and clean them up a bit.

```
// spec.js
describe('Simple App Combined', function() {
  var firstNumber = element(by.model('first'));
  var secondNumber = element(by.model('second'));
  var goButton = element(by.id('gobutton'));
  var latestResult = element(by.binding('latest'));

  beforeEach(function() {
    browser.get('http://localhost/.../protractor-demo/');
  });

  it('should have a title', function() {
    expect(browser.getTitle()).toEqual('Simple App Title');
  });

  it('should add one and two', function() {
    firstNumber.sendKeys(1);
    secondNumber.sendKeys(2);
    goButton.click();                    //click the button
    expect(latestResult.getText()).toEqual('3');
  });

});
```

Please observe, here, we've pulled the navigation out into a **before Each** function which is run before every" **it"** block.
We have stored the element finders for the first and second input into variables that can be reused.


scenario
beforeEach
command
command
afterEach
command
command
it
command
command
expectation
expectation
it

Protractor API

In the above examples we have seen selecting some text fields and set some data to field, now let's see how can we select a option in drop down box.

```
element(by.css('#customerAccountNumber')).element(by.cssContaining Text('option', '156456456')).click();

var numberText = "123456789";
element(by.model('form.selectRknm')).element(by.xpath("option[text() ='"+numberText+"']")).click();
```

Selecting a button by partial name:

```
<button>Register User</button>
element(by.partialButtonText("Register")).click();
```

Selecting a hyper link on partial url:

```
<a href=" https://en.wikipedia.org/wiki/History_of_India ">Doge meme</a>
var url = browser.findElement(by.partialLinkText('India'));
url.click();
```

Verify the first element:

```
<ul class="items">
  <li>India</li>
  <li>Russia</li>
  <li>USA</li>
</ul>

let first = element.all(by.css('.items li')).first();
expect(first.getText()).toBe('India');
```

In the similar way we have **last**, **each**, **count**.

Verify the Tag:

```
<span>{{country.name}}</span>

expect(element(by.binding('person.name')).getTagName()).toBe('span');
```

Verify the Attribute:

```
<div id="myDiv" class="country"></div>

var elmVar = element(by.id('myDiv'));
expect(elmVar.getAttribute(class)).toEqual('country');
```

Checkbox:

```
<input id="country" type="checkbox">

var myCountry = element(by.id('country'));
expect(myCountry.isSelected()).toBe(false);
myCountry.click();
expect(myCountry.isSelected()).toBe(true);
```

Clear:

```
<input id="country" value="India">

var myElement = element(by.id('country'));
expect(myElement.getAttribute('value')).toEqual('India');
myElement.clear();
expect(myElement.getAttribute('value')).toEqual('');
```

Element by Class name:

```
<ul class="countriesCss">
  <li class="statesCss">Punjab</li>
</ul>
// Returns the web element for dog
var stateElement = browser.findElement(by.className('statesCss'));
expect(stateElement.getText()).toBe('Punjab');
```

Deep Css:

Find an element by css selector within the Shadow DOM.

```
<div>
 <span id="outerspan">
 <"shadow tree">
  <span id="span1"></span>
  <"shadow tree">
   <span id="span2"></span>
  </>
 </>
</div>

var spans = element.all(by.deepCss('span'));
expect (spans.count()).toEqual(3);
```

Main functions are discussed here and some more important API functions can be seen here … http://www.protractortest.org/#/api

Conclusion

The main objective of this white paper is to bring the capabilities of Protractor to fore front, this helps in a full-fledged end to end testing of an Angular JS application with very minimal effort and good performance.

We have discussed protractor with Jasmine framework which is provided default, there are other frameworks as well like Mocha, Cucumber.

Interested can use Cucumber, Mocha based on the respective requirement, protractor support using Mocha, Cucumber and provides a very easy configuration in order to switch from jasmine.

Also customised frameworks can be configured to protractor.
Please refer to http://www.protractortest.org/.