**Protractor Testing Tool for End-to-end Testing of AngularJS Applications**

Posted In | [Automation Testing](http://www.softwaretestinghelp.com/category/automation-testing/), [Software Testing Tools](http://www.softwaretestinghelp.com/category/software-testing-tools/), [Web Testing](http://www.softwaretestinghelp.com/category/web-testing/) | Last Updated: "April 17, 2017"

**What is Protractor?**

The Protractor is an automation testing tool for web applications testing; combining powerful technologies such as Jasmine, Selenium Webdriver, Node.js etc.

The Protractor testing tool is an end to end behavior-driven testing framework designed keeping Angular JS applications in mind. Even though that might sound like Protractor won’t work with non-angular JS applications, it does.

It works with both Angular and non-Angular JS applications equally well.

* [Protractor vs Selenium WebDriver](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#Protractor_vs_Selenium_WebDriver)
* [Features](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#Features)
* [How can Protractor help me?](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#How_can_Protractor_help_me)
* [Which Framework to use?](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#Which_Framework_to_use)
* [How to Download and Setup Protractor](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#How_to_Download_and_Setup_Protractor)
* [Ready to create your first test case?](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#Ready_to_create_your_first_test_case)
* [How to run your test cases?](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#How_to_run_your_test_cases)
* [Some more cool features of Protractor](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#Some_more_cool_features_of_Protractor)
  + [#1) Assertions & annotations](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#1_Assertions_annotations)
  + [#2) Handling multiple browsers/windows/tabs](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#2_Handling_multiple_browserswindowstabs)
  + [#3) Use Page Objects to make your framework even better](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#3_Use_Page_Objects_to_make_your_framework_even_better)
  + [#4) Reporting](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#4_Reporting)
  + [#5) Integrate with other powerful tools such as Git/Jenkins/Browserstack/Grunt](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#5_Integrate_with_other_powerful_tools_such_as_GitJenkinsBrowserstackGrunt)
* [Conclusion](http://www.softwaretestinghelp.com/protractor-testing-tutorial/#Conclusion)

**Protractor vs Selenium WebDriver**

**What makes Protractor different from traditional** [**Selenium WebDriver**](http://www.softwaretestinghelp.com/selenium-webdriver-selenium-tutorial-8/)**?**

Take a minute to answer these questions:

* Is it a struggle to determine when exactly the web page is finally loaded (All of the asynchronous elements are ready and processed)?
* Are you tired of adding waits and sleeps to your code?
* Do you want to get rid of the cumbersome effort to locate the angular elements?
* Frustrated with locating elements with changing Ids?
* Do you want to create your own locator?
* Are you writing lengthy code even for the simplest assertions?
* Are you a JavaScript enthusiast/fan?

*If you answered Yes to these questions,* [*Protractor*](https://www.protractortest.org/#/) *can help.*

It’s a wrapper built on top of Selenium Webdriver and thus, provides all the capabilities of Selenium along with many useful additions. It offers:

**Features**

It offers:

**1) WaitForAngular**

From documentation:

“*Instruct WebDriver to wait until Angular has finished rendering and has no outstanding $http or $timeout calls before continuing. Note that Protractor automatically applies this command before every WebDriver action.*”

What this means is that there is no need to manually add waits to your script and Protractor will automatically wait for the web elements to load and only then executes the next steps.

**2)** It has the ability to export a global function *element*, which takes a locator and will return an ElementFinder. This ElementFinder has a set of action methods, such as click(), getText(), sendKeys() etc. This is the core of how to interact with the element and get information from it.

This global function helps to reduce the element locating syntax. Take a look at the following statement to locate the element in both Selenium WebDriver and Protractor:

**Selenium Webdriver**:

|  |  |
| --- | --- |
| 1 | driver.findElement(By.cssSelector("css selector")); |

**Protractor**:

|  |  |
| --- | --- |
| 1 | element(by.css('some-css')); |

The syntax looks compact, Isn’t it?

**3)** Some new locator strategies and functions provided to help locate the Angular elements are: *By.binding, By.repeater, By.textarea, By.model, WebElement.all, WebElement.evaluate*, etc.

**How can Protractor help me?**

Take a look at the following salient features:

* Simple syntax to write test cases
* The ability to run multiple browsers at once using Selenium Grid
* Angular-specific locators
* Support for Behavior-driven development such as Jasmine/Mocha
* No need to add sleeps/waits
* Supported integration with Jenkins/Browser Stack/Grunt etc.
* Get rid of dealing with synchronization issue in Angular JS websites
* Multiple browser support (Firefox, Chrome, Safari, Internet explorer)
* Ability to run the same scripts in mobile browsers also without the need to change the code

**Which Framework to use?**

It supports two Behavior-driven development (BDD) test frameworks right out of the box:

* **Jasmine:** [Jasmine](https://jasmine.github.io/) is the default test framework when Protractor is installed. We will use Jasmine in this article.
* **Mocha:** [Mocha](https://mochajs.org/) is a JavaScript test framework which runs on Node.js. If you wish to use Mocha as your test framework, you will have to set it up with your Protractor and also will need to use Behavior Driven Development interface and Chai Assertions with Chai as Promised (refer [this link](http://chaijs.com/plugins/chai-as-promised/) for more details).

**How to Download and Setup Protractor**

As it is a node.js program, you need to install node.js to get it running. Node.js can be downloaded from this URL by selecting the operating system: [Node.js download](https://nodejs.org/en/download/)

With node.js, you also get *Protractor npm* package, which can now be used to install Protractor.

Now that node.js is installed to your machine, open the command prompt(cmd) and hit the below command to install it globally:

|  |  |
| --- | --- |
| 1 | npm install -g protractor |

*‘-g’* is used to install this tool globally. Remove it if you do not wish to install it globally. This command also installs the Protractor API along with a default selenium server, which means that you don’t necessarily have to start a standalone server.

Now, we need to install the selenium server and ChromeDriver. Use the following command (also comes with Protractor) in cmd:

|  |  |
| --- | --- |
| 1 | webdriver-manager update |

This is all we need to start with our first test case writing using this framework. If you wish, you can install any IDE/editor to write your code. Eclipse IDE is the popular one, but there are also some more cool editors to consider. Personally, I prefer ‘Atom’ editor for my Protractor code writing.

**Ready to create your first test case?**

It needs 2 files to run:

* Configuration file
* Spec file.

The *Configuration file* is the one that tells Protractor where to find the test files (specs)/which browser to pick/which framework to use (Jasmine/Mocha)/where to talk to your Selenium browser and other configurations. If any configuration is not defined in configuration files, it will use defaults.

Spec file is the one where we write our actual test code. All of our test functional flow/assertions will be in this specific file. There might be several spec files based on the number of test cases but just 1 spec file will be able to run the entire test suite of several specs.

**Example TEST CASE:**

Now, we will write a simple test case where we will navigate to an URL and check for the page title.

**Here are the steps:**

* Create a new folder for your test suite.
* Create a new file with name as ‘*js*’.(any name will do) All of the spec/configuration files will be of ‘.js’ extension.
* Create a new file with the name specified as ‘*js*’.

|  |  |
| --- | --- |
| 1 | //CheckTitleSpec.js |
| 2 | describe('Protractor Demo', function() { | |

|  |  |  |
| --- | --- | --- |
| 3 | it('to check the page title', function() { | |
| 4 | browser.ignoreSynchronization = true; |

|  |  |
| --- | --- |
| 5 | browser.get('http://www.softwaretestinghelp.com/'); |
| 6 | browser.driver.getTitle().then(function(pageTitle) { | |

|  |  |  |
| --- | --- | --- |
| 7 | expect(pageTitle).toEqual('Software Testing Help - A Must Visit Software Testing Portal'); | |
| 8 | }); |

|  |  |  |
| --- | --- | --- |
| 9 | }); | |
| 10 | | }); | |

Now, we are all set to start with our coding. See the below code of ‘testCaseNameSpec.js’ file.

**Here is what the configuration file looks like:**

|  |  |
| --- | --- |
| 1 | // conf.js |
| 2 | exports.config = { | |

|  |  |  |
| --- | --- | --- |
| 3 | framework: 'jasmine', | |
| 4 | capabilities: { |

|  |  |  |
| --- | --- | --- |
| 5 | browserName: 'chrome', | |
| 6 | }, |

|  |  |  |
| --- | --- | --- |
| 7 | specs: ['simpleTestSpec.js'] | |
| 8 | }; |

**Now, let break these 2 files and see how it works.**

**#1) *Spec.js***

* All of the browser level commands will be handled by ‘*browser*’, a global created by Protractor.
* As we are following Jasmine framework, ‘*describe’* and ‘*it*’ are the syntaxes of Jasmine. Describe will contain the entire end to end the flow of your test case, whereas ‘it’ might contain some of test steps/scenarios etc. You can have multiple ‘*it*’ blocks in your program if you wish so.
* *browser.get* is a simple Selenium syntax which tells Protractor to hit a specific URL in the browser.
* As the website we are trying to hit is a non-angular website, we set the *ignoreSynchronization* tag to ‘*true*’ as displayed at line# 4. If you do not make this tag true, your test will fail with error “Angular could not be found on the page”. The reason behind this is that Protractor expects to work with angular websites by default, and if we are using Protractor to validate the non-angular website, we need to tell this to Protractor explicitly. However, if you are working on angular websites, there is no need to use this statement as Protractor will by default consider the web page to be angular.
* ‘expect’ is nothing but the assertion where we are comparing the web page title to equal some predefined data. We will discuss more assertions in detail.

**#2) *conf.js***

* As discussed earlier, the configuration file is the one which tells Protractor the core details. As displayed in the code, the framework is ‘Jasmine’.
* Inside the capabilities section, browser configurations are set. You can define the browser name such as Firefox/chrome etc. You can also set the maximum instances of the browsers so that at one time, you can run multiple test cases on different available browser windows.
* In the ‘*specs*’ section, we give the path of the spec file, i.e. exactly where the spec file is located with respect to the configuration file.
* There are also many other cool features you can attach to your configuration file such as reporting/onPrepare function/threshold timeout etc. We will cover few of these in this tutorial.

**How to run your test cases?**

We have written the code and now all we need is a little push to get our code running. Hit the following command in cmd to run your program:

|  |  |
| --- | --- |
| 1 | protractor conf.js |

This command will start running the selenium server followed by running your test script. You can view the logs in cmd itself or if you wish, logs can be captured in a .txt file as well (just *add >> textFileName.txt* after the above command and the logs will be saved in the text file located in the same directory as of configuration file).

A chrome window will open, where ***SoftwareTestingHelp.com*** website should be opened. The output would be ‘1 spec, 0 failure’ when you run the code. What this means is that we had 1 ‘it’ blocks, which executed with 0 failures.

Now, consider the below spec file where we are performing some actions on a web page build on AngularJS so that you can see protractor’s take on Selenium when it comes to testing the Angular website:

|  |  |
| --- | --- |
| 1 | // spec.js |
| 2 | describe('Code to interact with Angular JS elements', function() { | |

|  |  |
| --- | --- |
| 3 | it('should multiply two integers', function() { |
| 4 | browser.get('http://juliemr.github.io/protractor-demo/'); | |

|  |  |
| --- | --- |
| 5 | element(by.model('first')).sendKeys(5); |
| 6 | element(by.model('second')).sendKeys(5); | |

|  |  |
| --- | --- |
| 7 | element(by.model('operator')).click(); |
| 8 | element(by.xpath(".//option[@value= 'MULTIPLICATION']")).click(); | |

|  |  |  |
| --- | --- | --- |
| 9 | element(by.id('gobutton')).click(); | |
| 10 | | expect(element(by.binding('latest')) .getText()).toEqual('10'); //Incorrect expectation | |

|  |  |  |
| --- | --- | --- |
| 11 | expect(element(by.binding('latest')) .getText()).toEqual('25'); //Correct expectation | |
| 12 | }); |

|  |  |
| --- | --- |
| 13 | }); |

You can use the same conf.js for running this spec too. Just make sure to update the specification file name.

Now, in this spec file, we are playing with an angular JS website so that you can see what exactly Protractor is capable of.

If you run this file just like we did earlier, a web page will open with 2 text boxes, a drop down, a button and few other web elements. As you might have already guessed, it’s a calculator page. We are giving 2 integers as input and performing a multiplication operation.

------------

As we already discussed, one of the biggest advantages of using this tool is its unique techniques to locate the Angular elements. *‘by.model’* is one such way to locate elements. *‘.sendKeys()’* is usual Selenium syntax to send values in text boxes and *‘.click()’* is used to click on buttons.

As discussed earlier, *‘expect’* is an assertion, which is asking Protractor to capture the result of the multiplication of numbers and compare it with ‘10’ and ‘25’ sequentially. Simple Mathematics tells us that the output should be ‘25’ and hence, the first assertion fails and second passes.

As a result, you will get ‘*1 Spec, 1 Failure*’ in the logs when you run the code, which is expected.

**Some more cool features of Protractor**

**#1) Assertions & annotations**

Assertions are an important part of the automation scripts. Annotations are also very useful in effectively tagging certain methods in a class to have special meaning.

It provides a variety of assertions & annotations and in addition to that, also provides the ability to create your own assertions.

**Consider the below example:**

|  |  |  |
| --- | --- | --- |
| 1 | describe('Code to understand assertions/annotations', function() { | |
| 2 | beforeEach(function() { |

|  |  |  |
| --- | --- | --- |
| 3 | browser.get('http://juliemr.github.io/protractor-demo/'); | |
| 4 | }); |

|  |  |
| --- | --- |
| 5 | afterEach(function() { |
| 6 | browser.get('https://www.madewithangular.com/#/'); | |

|  |  |
| --- | --- |
| 7 | }); |
| 8 | var multiplyNumbers = function(a, b) { | |

|  |  |  |
| --- | --- | --- |
| 9 | element(by.model('first')).sendKeys(a); | |
| 10 | | element(by.model('second')).sendKeys(b); | |

|  |  |  |
| --- | --- | --- |
| 11 | element(by.model('operator')).click(); | |
| 12 | element(by.id('gobutton')).click(); |

|  |  |
| --- | --- |
| 13 | }; |
| 14 | it('should multiply two integers', function() { | |

|  |  |
| --- | --- |
| 15 | multiplyNumbers(2, 2); |
| 16 | expect(element.all(by.repeater('result in memory')).count()).toEqual(2); | |

|  |  |
| --- | --- |
| 17 | multiplyNumbers(3, 3); |
| 18 | expect(element.all(by.repeater('result in memory')).count()).toEqual(2); | |

|  |  |
| --- | --- |
| 19 | }); |
| 20 | }); |

In the above example, we are using 2 annotations, *‘beforeEach’* and *‘afterEach’*. These annotations are also available in TestNG (traditional Selenium). These annotations make sure that a particular piece of code will be executed before/after respectively the execution of remaining code.

**So, here is how the execution of the code will take place,**

1. Protractor will reach inside the ‘*beforeEach*’ block first and it will hit ‘*http://juliemr.github.io/protractor-demo/*’ URL in the browser.
2. Now, the flow will move to the ‘it’ block and function ‘*multiplyNumbers’* will be called which will, in turn, perform the specified actions in it sending the control back to where the function was called.
3. At last, the assertion will do its job. Now, if we want to tackle multiple elements at the same time, you can use ‘element.all’ a feature of this tool. It will identify all of the available elements with the specified locator (by.repeater in this case). It’s up to you what you wish to do with the identified elements. In this case, we are comparing the calculation history with a given number.
4. Since, in the first assertion, we are comparing the calculation history count with ‘2’ even though we performed calculation just once, the assertion will fail. The second assertion, however, will pass as after the second calculation, history count would be ‘2’.

**There are many more available types of assertions. Some of which are given below:**

**a) Get text from a web element and compare it with a certain value:**

|  |  |  |
| --- | --- | --- |
| 1 | element(by.locator('someLocator')).getText(text) .then(function() { | |
| 2 | expect(text).toEqual('someData'); |

|  |  |
| --- | --- |
| 3 | expect(text).not.toEqual('someData'); |
| 4 | expect(text).toContain('someOtherData'); | |

|  |  |
| --- | --- |
| 5 | }); |

**b) Verify if a web element is displayed on the page or not:**

|  |  |
| --- | --- |
| 1 | expect(browser.driver.findElement(by.locator(someLocator)) .isDisplayed()).toBe(true); |

**#2) Handling multiple browsers/windows/tabs**

There can be multiple cases when it comes to handling the browser. Some of these cases are highlighted below:

**a) A new tab opens up by clicking on any link**

Sometimes, when you click on any link, a new tab opens up and rest of the actions need to take place in the newly opened window. In this case, when you write the code to the point a new tab is opened, you need to implement Window Handler using the following way:

|  |  |  |
| --- | --- | --- |
| 1 | //Get all of the opened windows and store the count in handles | |
| 2 | browser.getAllWindowHandles().then(function(handles) { |

|  |  |
| --- | --- |
| 3 | //Now switch the control to the newly opened window |
| 4 | browser.switchTo().window(handles[1]).then(function() { | |

|  |  |  |
| --- | --- | --- |
| 5 | //Write the code which needs to be executed in the new tab | |
| 6 | }); |

|  |  |
| --- | --- |
| 7 | }); |

First get a count of all the available windows and then use indexing to switch control between the windows. The original window which initiated the new window will have index 0, whereas the subsequent windows will have increasing indexes.

**b) Opening an entirely new browser with new session**

When you need to perform some actions on a browser and need to do further actions on a different session of the browser, we need to use the ***forkNewDriverInstance***. In this case, we create a new browser instance with a fresh browser name in the following way:

|  |  |  |
| --- | --- | --- |
| 1 | describe('Code to understand assertions/annotations', function() { | |
| 2 | //Create a new browser instance |

|  |  |
| --- | --- |
| 3 | var newBrowser = browser.forkNewDriverInstance(); |
| 4 | it('should should open multiple browsers instances', function() { | |

|  |  |
| --- | --- |
| 5 | //Opens a URL in the 1st browser instance |
| 6 | browser.get('http://juliemr.github.io/protractor-demo/'); | |

|  |  |
| --- | --- |
| 7 | //Opens a URL in the 2nd browser instance |
| 8 | newBrowser.get('https://www.madewithangular.com/#/'); | |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | newBrowser.driver.quit(); | | |
| 10 | | }); |

|  |  |
| --- | --- |
| 11 | }); |

**c) Running your test case in multiple browsers:**

Running your test case in 2 browsers at a time is something configuration file can do for you. Just add the below code in your configuration file:

As soon as you run this configuration file, you will see tests running in both Firefox and Chrome simultaneously and run logs will be displayed in the command prompt separately.

|  |  |
| --- | --- |
| 1 | // conf.js |
| 2 | exports.config = { | |

|  |  |  |
| --- | --- | --- |
| 3 | framework: 'jasmine', | |
| 4 | specs: ['SomeSpec.js'], |

|  |  |
| --- | --- |
| 5 | multiCapabilities: [{ |
| 6 | browserName: 'firefox' | |

|  |  |
| --- | --- |
| 7 | }, { |
| 8 | browserName: 'chrome' | |

|  |  |  |
| --- | --- | --- |
| 9 | }] | |
| 10 | | } |

**#3) Use Page Objects to make your framework even better**

This tool is good on its own but it becomes invincible when combined with Page Object Model(POM). Most of its shortcomings (if any) are overcome with the page object model. Moreover, POM also helps maintain your project in a more structured way.

If you do not know what POM is, no worries. POM is a way to segregate your test case on the basis of the pages.

**Take this example:**

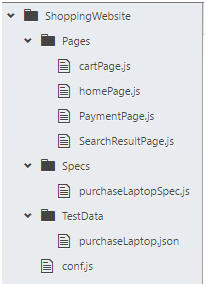
There is a shopping website. Your test case is to select a product, add it to cart and then purchase it.

Now, there are two ways to manage your test script code for this:

1. Write a plain test case with all of the locators on the same page your logic is written in,
2. Write all of your flow of test case, your logic in your spec file and segregate your locators and test data in different files. Each web page will have an equivalent .js page file. In this way, your code will be structured and if there is any other test case which requires the same locator, you do not need to write these locators again, just import this locator file and use it as per your need.

Maintaining your test cases can be a real pain. If you use POM, your code will be in a much more structured manner.

**Here is an example of using the page object model:**

[](http://cdn2.softwaretestinghelp.com/wp-content/qa/uploads/2017/03/page-object-model.jpg)

**This is the flow is in the above snapshot:**

1. There is a test case which purchases Laptop. The code for the flow and logic will be in the purchaseLaptopSpec.js.
2. All the pages which are encountered to purchase the Laptop will have 1 ‘.js’ file with a proper title. All of the elements required to be used for purchasing the Laptop, their locators will be inside the respective page file.
3. The data required for this test case can be saved in the TestData folder either in a ‘.json’ format or in the excel format.
4. Once done with the pages and locators, just import these files in your spec file to use the locator/test data and you are all set with your test case.

**#4) Reporting**

NPM (Node Package Manager) provides various reporting packages, with which screenshot of each test step can be captured and also, once the test run completes, it will generate an HTML report for you. All you have to do is to install those packages by opening a command prompt and hitting the below commands:

|  |  |
| --- | --- |
| 1 | npm install -g protractor-jasmine2-html-reporter |
| 1 | npm install protractor-jasmine2-screenshot-reporter --save-dev | |

Once these packages are installed, every time you run your configuration file, all the screenshots of your test cases will be saved and also there will be an HTML report generated displaying the test case pass/fail result.

**#5) Integrate with other powerful tools such as Git/Jenkins/Browserstack/Grunt**

There are multiple tools available in the market to help make your test cases even better. Git/Jenkins/BrowserStack/Grunt are some such tools which add significant value to your normal Protractor test scripts. And the best part is that you don’t have to touch your spec file to integrate your Protractor with any of these tools. It is your configuration file, which will take all of these things for you.

**Git** is a very powerful version control tool. It is always a best practice to keep your code in Git if there are multiple developers involved.

**Jenkins** is a continuous integration tool with which, you can schedule your test cases and run it as per your need. Protractor scripts can also be configured with Jenkins. Best use of running your test cases on Jenkins is that it is very fast and also you can run multiple test cases at a time.

**BrowserStack** is a cross-browser testing tool which can also be used to test your applications across different browsers. It can also be integrated with Protractor by adding the browserStack credentials in your config file.

**Grunt** is a JavaScript task runner. It provides you with the ability to perform several tasks for you. Its awesomeness is that there are more than 4000 tasks and you can create even more tasks as per your requirement. Following are few of the important daily use tasks we can use Grunt for:

1. List down all of the coding best practices and inform immediately whenever you violate any of these.
2. To create multiple spec files at run time. **For example**, if there is any test case that you wish to run several times (ranging from 1 to any number). This might seem unnecessary at this point of time but think of running any shopping website checkout flow to be run on every single available country. It would be tedious to create multiple specs manually. So, let Grunt do this for you.
3. The watch feature. You write a test case and every time as soon as save your code after making any change in it, you want your test case to run, Grunt has got it.
4. Concatenating multiple files.

Just give it a try and you will love it.

**Conclusion**

You can make Selenium work like protractor but why to re-invent the wheel? If Google has put in so much effort to make Protractor work like a charm, let use it to the fullest. At least for the AngularJS websites, it will make your life much easier.

Also, this article is not just about Protractor. There is a big world of Protractor out there and there are hundreds of packages available in the market offered by NPM to add more features to your test in addition to simple test scenarios. Feel free to use these packages and make it even better.