Table of Contents

Tutorial

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.

Prerequisites

Protractor is a Node.js program. To run, you will need to have Node.js installed. You will download Protractor package using npm, which comes with Node.js. Check the version of Node.js you have by running `node --version`.

Protractor 5 is compatible with nodejs v6 and newer.

Protractor works with AngularJS versions greater than 1.0.6/1.1.4, and is compatible with Angular applications. Note that for Angular apps, the `binding` and `model` locators are not supported. We recommend using `by.css`.

By default, Protractor uses the Jasmine test framework for its testing interface. This tutorial assumes some familiarity with Jasmine, and we will use version 2.4.

You will need to have the Java Development Kit (JDK) installed to run the standalone Selenium Server. Check this by running `java -version` from the command line.

Why Do We Need Protractor Framework?

JavaScript is used in almost all web applications. As the applications grow, JavaScript also increases in size and complexity. In such case, it becomes a difficult task for Testers to test the web application for various scenarios.

Sometimes it is difficult to capture the web elements in AngularJS applications using JUnit or Selenium WebDriver.

Protractor is a NodeJS program which is written in JavaScript and runs with Node to identify the web elements in AngularJS applications, and it also uses WebDriver to control the browser with user actions.

Ok, fine now let's discuss what exactly is an AngularJS application?

AngularJS applications are Web Applications which uses extended HTML's syntax to express web application components. It is mainly used for dynamic web applications. These applications use less and flexible code compared with normal Web Applications.

Why can't we find Angular JS web elements using Normal Selenium Web driver?

Angular JS applications have some extra HTML attributes like ng-repeater, ng-controller, ng-model.., etc. which are not included in Selenium locators. Selenium is not able to identify those web elements using Selenium code. So, Protractor on the top of Selenium can handle and controls those attributes in Web Applications.

The protractor is an end to end testing framework for Angular JS based applications. While most frameworks focus on conducting unit tests for Angular JS applications, Protractor focuses on testing the actual functionality of an application.

Setup

Use npm to install Protractor globally with:

```
npm install -g protractor
```

This will install two command line tools, `protractor` and `webdriver-manager`. Try running `protractor --version` to make sure it's working.

The `webdriver-manager` is a helper tool to easily get an instance of a Selenium Server running. Use it to download the necessary binaries with:

```
webdriver-manager update
```

Now start up a server with:

```
webdriver-manager start
```

This will start up a Selenium Server and will output a bunch of info logs. Your Protractor test will send requests to this server to control a local browser. Leave this server running throughout the tutorial. You can see information about the status of the server at `http://localhost:4444/wd/hub`.

Step 0 - write a test

Open a new command line or terminal window and create a clean folder for testing.

Protractor needs two files to run, a **spec file** and a **configuration file**.

Example to test demo website :

http://juliemr.github.io/protractor-demo/

Copy the following into spec.js:

```
// spec.js
describe('Protractor Demo App', function() {
  it('should have a title', function() {
    browser.get('http://juliemr.github.io/protractor-demo/');

    expect(browser.getTitle()).toEqual('Super Calculator');
  });
});
```

The `describe` and `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`.

Now create the configuration file. Copy the following into conf.js:

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

This configuration tells Protractor where your test files (`specs`) are, and where to talk to your Selenium Server (`seleniumAddress`). It specifies that we will be using Jasmine for the test framework. It will use the defaults for all other configuration. Chrome is the default browser.

Now run the test with

```
protractor conf.js
```

You should see a Chrome browser window open up and navigate to the Calculator, then close itself (this should be very fast!). The test output should be `1 tests, 1 assertion, 0 failures`. Congratulations, you've run your first Protractor test!

Step 1 - interacting with elements

Now let's modify the test to interact with elements on the page. Change spec.js to the following:

```
// spec.js
describe('Protractor Demo App', function() {
  it('should add one and two', function() {
    browser.get('http://juliemr.github.io/protractor-demo/');
    element(by.model('first')).sendKeys(1);
    element(by.model('second')).sendKeys(2);

    element(by.id('gobutton')).click();
    expect(element(by.binding('latest')).getText()).
        toEqual('5'); // This is wrong!
  });
});
```

This uses the globals `element` and `by`, which are also created by Protractor. The `element` function is used for finding HTML elements on your webpage. It returns an ElementFinder object, which can be used to interact with the element or get information from it. In this test, we use `sendKeys` to type into `<input>`s, `click` to click a button, and `getText` to return the content of an element.

`element` takes one parameter, a Locator, which describes how to find the element. The `by` object creates Locators. Here, we're using three types of Locators:

- `by.model('first')` to find the element with `ng-model="first"`. If you inspect the Calculator page source, you will see this is `<input type="text" ng-model="first">`.
- `by.id('gobutton')` to find the element with the given id. This finds `<button id="gobutton">`.
- `by.binding('latest')` to find the element bound to the variable `latest`. This finds the span containing `{{latest}}`

Using Locators

The heart of end-to-end tests for webpages is finding DOM elements, interacting with them, and getting information about the current state of your application. This doc is an overview of how to locate and perform actions on DOM elements using Protractor.

Overview

Protractor exports a global function `element`, which takes a *Locator* and will return an *ElementFinder*. This function finds a single element - if you need to manipulate multiple elements, use the `element.all` function.

The *ElementFinder* has a set of *action methods*, such as `click()`, `getText()`, and `sendKeys`. These are the core way to interact with an element and get information back from it.

When you find elements in Protractor all actions are asynchronous. Behind the scenes, all actions are sent to the browser being controlled using the JSON Webdriver Wire Protocol. The browser then performs the action as a user natively would.

Locators

A locator tells Protractor how to find a certain DOM element. Protractor exports locator factories on the global `by` object. The most common locators are:

```
// Find an element using a css selector.
by.css('.myclass')

// Find an element with the given id.
by.id('myid')

// Find an element using an input name selector.
by.name('field_name')

// Find an element with a certain ng-model.
// Note that at the moment, this is only supported for AngularJS apps.
by.model('name')

// Find an element bound to the given variable.
// Note that at the moment, this is only supported for AngularJS apps.
by.binding('bindingname')
```

The locators are passed to the `element` function, as below:

```
element(by.css('some-css'));
element(by.model('item.name'));
element(by.binding('item.name'));
```

When using CSS Selectors as a locator, you can use the shortcut $() notation:

```
$('my-css');

// Is the same as:

element(by.css('my-css'));
```

Actions

The `element()` function returns an ElementFinder object. The ElementFinder knows how to locate the DOM element using the locator you passed in as a parameter, but it has not actually done so yet. It will not contact the browser until an *action* method has been called.

The most common action methods are:

```
var el = element(locator);


// Click on the element.
el.click();


// Send keys to the element (usually an input).
el.sendKeys('my text');


// Clear the text in an element (usually an input).
el.clear();


// Get the value of an attribute, for example, get the value of an inp
ut.
el.getAttribute('value');
```

Since all actions are asynchronous, all action methods return a promise. So, to log the text of an element, you would do something like:

```
var el = element(locator);
el.getText().then(function(text) {
  console.log(text);
});
```

Finding Multiple Elements

To deal with multiple DOM elements, use the `element.all` function. This also takes a locator as its only parameter.

```
element.all(by.css('.selector')).then(function(elements) {
  // elements is an array of ElementFinders.
});
```

`element.all()` has several helper functions:

```
// Number of elements.
element.all(locator).count();

// Get by index (starting at 0).
element.all(locator).get(index);

// First and last.
element.all(locator).first();
element.all(locator).last();
```

When using CSS Selectors as a locator, you can use the shortcut $$() notation:

```
$$('.selector');

// Is the same as:

element.all(by.css('.selector'));
```

Finding Sub-Elements

To find sub-elements, simply chain element and element.all functions together as shown below.

Using a single locator to find:

- an element

  ```
  element(by.css('some-css'));
  ```

- a list of elements:

  ```
  element.all(by.css('some-css'));
  ```

Using chained locators to find:

- a sub-element:

```
element(by.css('some-css')).element(by.tagName('tag-within-css'));
```

- to find a list of sub-elements:

```
element(by.css('some-css')).all(by.tagName('tag-within-css'));
```

You can chain with get/first/last as well like so:

```
element.all(by.css('some-css')).first().element(by.tagName('tag-within-css'));

element.all(by.css('some-css')).get(index).element(by.tagName('tag-within-css'));

element.all(by.css('some-css')).first().all(by.tagName('tag-within-css'));
```

Behind the Scenes: ElementFinders versus WebElements

If you're familiar with WebDriver and WebElements, or you're just curious about the WebElements mentioned above, you may be wondering how they relate to ElementFinders.

When you call `driver.findElement(locator)`, WebDriver immediately sends a command over to the browser asking it to locate the element. This isn't great for creating page objects, because we want to be able to do things in setup (before a page may have been loaded) like

```
var myButton = ??;
```

and re-use the variable `myButton` throughout your test. ElementFinders get around this by simply storing the locator information until an action is called.

```
var myButton = element(locator);

// No command has been sent to the browser yet.
```

The browser will not receive any commands until you call an action.

```
myButton.click();

// Now two commands are sent to the browser - find the element, and then click it.
```

ElementFinders also enable chaining to find subelements, such as `element(locator1).element(locator2)`.

All WebElement actions are wrapped in this way and available on the ElementFinder, in addition to a couple helper methods like `isPresent`.

You can always access the underlying WebElement using `element(locator).getWebElement()`, but you should not need to.

Step 2 - writing multiple scenarios

Let's put these two tests together and clean them up a bit. Change spec.js to the following:

```
// spec.js
describe('Protractor Demo App', 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://juliemr.github.io/protractor-demo/');
  });

  it('should have a title', function() {
    expect(browser.getTitle()).toEqual('Super Calculator');
  });

  it('should add one and two', function() {
    firstNumber.sendKeys(1);
    secondNumber.sendKeys(2);

    goButton.click();

    expect(latestResult.getText()).toEqual('3');
  });
  it('should add four and six', function() {
    // Fill this in.
    expect(latestResult.getText()).toEqual('10');
  });

  it('should read the value from an input', function() {
    firstNumber.sendKeys(1);
    expect(firstNumber.getAttribute('value')).toEqual('1');
  });
});
```

Here, we've pulled the navigation out into a `beforeEach` function which is run before every `it` block. We've also stored the ElementFinders for the first and second input in nice variables that can be reused. Fill out the second test using those variables, and run the tests again to ensure they pass.

In the last assertion we read the value from the input field with `firstNumber.getAttribute('value')` and compare it with the value we have set before.

Step 3 - changing the configuration

Now that we've written some basic tests, let's take a look at the configuration file. The configuration file lets you change things like which browsers are used and how to connect to the Selenium Server. Let's change the browser. Change conf.js to the following:

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

Try running the tests again. You should see the tests running on Firefox instead of Chrome. The `capabilities` object describes the browser to be tested against.

You can also run tests on more than one browser at once. Change conf.js to:

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

Try running once again. You should see the tests running on Chrome and Firefox simultaneously, and the results reported separately on the command line.

Step 4 - lists of elements

Let's go back to the test files. Feel free to change the configuration back to using only one browser.

Sometimes, you will want to deal with a list of multiple elements. You can do this with `element.all`, which returns an ElementArrayFinder. In our calculator application, every operation is logged in the history, which is implemented on the site as a table with `ng-repeat`. Let's do a couple of operations, then test that they're in the history. Change spec.js to:

```
// spec.js
describe('Protractor Demo App', 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'));
  var history = element.all(by.repeater('result in memory'));

  function add(a, b) {
    firstNumber.sendKeys(a);
    secondNumber.sendKeys(b);
    goButton.click();
  }

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

  it('should have a history', function() {
    add(1, 2);
    add(3, 4);

    expect(history.count()).toEqual(2);

    add(5, 6);

    expect(history.count()).toEqual(0); // This is wrong!
  });
});
```

We've done a couple things here - first, we created a helper function, `add`. We've added the variable `history`. We use `element.all` with the `by.repeater` Locator to get an ElementArrayFinder. In our spec, we assert that the history has the expected length using the `count` method. Fix the test so that the second expectation passes.

`ElementArrayFinder` has many methods in addition to `count`. Let's use `last` to get an ElementFinder that matches the last element found by the Locator. Change the test to:

```
  it('should have a history', function() {
    add(1, 2);
    add(3, 4);

    expect(history.last().getText()).toContain('1 + 2');
    expect(history.first().getText()).toContain('foo'); // This is wro
ng!
```

```
  });
```

Since the Calculator reports the oldest result at the bottom, the oldest addition (1 + 2) be the last history entry. We're using the `toContain` Jasmine matcher to assert that the element text contains "1 + 2". The full element text will also contain the timestamp and the result.

Fix the test so that it correctly expects the first history entry to contain the text "3 + 4".