
Table of Contents

Introduction	1.1
Install Protractor	1.2
Run Tasks	1.3
Protractor Conf	1.4
Your First Test	1.5
Web Elements	1.6
Forms	1.7
Input Text	1.7.1
Check Boxes	1.7.2
Select Options	1.7.3
Tricks	1.8
Hidden Elements	1.8.1
Async Await	1.8.2

Protractor: Learn Testing Angular

This book serves as a start to learning Protractor which is a testing framework specifically for end to end tests on AngularJS.



Install Protractor

```
npm install protractor --save-dev
mkdir protractor
cd protractor
touch protractor.conf.js
touch spec.js
```

Run Tasks

Open up your *package.json* file and add couple of run tasks to the scripts section.

```
"scripts": {  
  "update": "node_modules/.bin/webdriver-manager update",  
  "ptr": "node_modules/.bin/protractor test/protractor.conf.js"  
}
```

This will allow you to have a clean way to run your suite soon.

Updating Selenium and ChromeDriver

```
$ npm run update
```

To run Protractor test suite.

```
$ npm run ptr
```

Protractor Conf

Getting started all you must have in the config to get started is...

```
//protractor.conf.js
exports.config = {
  specs: ['spec.js']
};
```

Although this is enough to get started lets add a few more things to the config. We can touch on the specifics of them more in depth later.

```
//protractor.conf.js
exports.config = {
  specs: ['spec.js'],

  directConnect: true, //chrome only

  capabilities: {
    'browserName': 'chrome'
  },

  jasmineNodeOpts: {},

  onPrepare: function() {}
};
```

Your First Test

We are going to start writing your first test. To begin lets say we have a scenario where we want to make sure that there is an image with the certain class present on the screen.

```
describe('Your first test', function() {  
  
  it('should have angular image', function(){  
    browser.get('http://www.angularjs.org');  
    expect($('.AngularJS-large').isPresent()).toBeTruthy();  
  });  
  
});
```

By default Protractor uses Jasmine as the test runner of choice. You can select other frameworks but for this lesson we will stick with the default.

Jasmine uses test descriptive functions like "describe" and "it", to describe your scenarios.

The "it" block is the specific test that you want to validate.

In our test scenario the first thing we need to accomplish is to go to the AngularJS website. To make this happen we use *browser.get()*.

Once we get to the Angular website then we write an expectation that says element with class "AngularJS-large" is to be present.

To run your test...

```
$ npm run ptr
```

Web Elements

A key attribute to testing web applications is accessing a specific element within DOM (Document Object Model) and performing an action. The DOM is basically a programming interface that creates a tree representation of the HTML. Lets imagine we have this simple Angular web page below to test.

```
<!DOCTYPE html>
<html ng-app="myapp">
<head>
  <script src="./angular.min.js"></script>
  <title>My App</title>
</head>
<body>
  <a href="www.angularjs.org">Link to Angular</a>
</body>
</html>
```

Test Scenario

As a user of this website, I expect to see a link to www.angularjs.org.

```
describe('As a user of website', function() {
  it('should have link to www.angularjs.org', function() {
    browser.get('http://thisWebsite/');
    var el = element(by.linkText('Link to Angular'));
    expect(el.getAttribute('href')).toBe('www.angularjs.org');
  });
});
```

Elements

To get to a specific element on the DOM you will use the *element* function with 1 argument that is a *by* function. The *by* function has several possible options that can help locate a specific element.

- `by.css`
- `by.linkText`
- `by.binding`
- `by.model`

- `by.repeater`
- find more www.protractortest.org

Actions

Once you have an element then you need to perform an action to that element.

- `click()`
- `sendKeys()`
- `getAttribute('attrID')`

Bonus syntax

If you are familiar with JQuery, you may used to a syntax that looks like.

```
$('.class')
```

This works in Protractor as well. It is the same as.

```
element(by.css('.class'));
```


Forms

Forms are probably one of the most used reasons for automation within a web application. We will look at a variety of form elements and show how to automate an action and also how to test the value of an element.

A basic form that we are going to deal with is below.

```
<html ng-app="myform">
<head>
  <script src="./angular.min.js"></script>
  <title>Form</title>
</head>
<body>
  <form method="POST" action="someaction">
    Username: <input type="text" class="frm1" id="username"/> <br>
    Password: <input type="password" class="frm1" id="password"/> <br>
    Remember me: <input type="checkbox" id="remember"/>
    <input type="submit" />
  </form>
</body>
</html>
```

Input Text

Input text can be done by finding a specific element and calling the `sendKeys` function.

Test Scenario

As a user I want to type in my username and password to login to the web page.

```
describe('As a user of website', function() {
  it('should allow me to enter username', function() {
    browser.get('http://thisWebsite/');

    //using element by.id syntax
    element(by.id('username')).sendKeys('sholmes');
    expect(element(by.id('username')).getAttribute('value'))
      .toBe('sholmes');
  });

  it('should allow me to enter password', function() {
    browser.get('http://thisWebsite/');

    //using JQuery style by css
    $('[id="password"]').sendKeys('pmoriarty');
    expect($('[id="password"]').getAttribute('value'))
      .toBe('pmoriarty');
  });
});
```

In this basic form we have 2 input fields that we need to run some operations on. The first is entering a username and the second is a password. Functionally both these tests do the same thing to different fields. Notice the difference in the JQuery style to find an element by id on password versus how it was done on the username. Its rather short. With the JQuery like syntax you can find any element by a specific attribute withing the HTML tag. for example if you had a custom attribute like.

```
<input type="text" custom_attr="custom" />
```

You could access it by.

```
$('[custom_attr="custom"]')
```


Check boxes

Check boxes can seem a little out of place as far as inputs are concerned. When you are getting text from a text input you are generally just looking for the value. However on a check box the DOM doesn't return a value but it does add an attribute *checked*.

Testing Scenario

As a user of the website I want to be able to click the Remember me check box and make sure it is selected.

```
describe('As a user of website', function() {
  it('should allow me to click Remember me checkbox', function() {
    browser.get('http://thisWebsite/');

    // Check the checkbox
    $(' [id="remember"]').click();

    // Expect that the checkbox is checked.
    expect($(' [id="remember"]').getAttribute('checked')).toBeTruthy();

    // Alternate way to check if checkbox is checked
    expect($(' [id="remember"]').isSelected()).toBeTruthy();
  });
});
```

Select Options

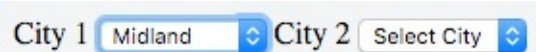
Let's say you have a form on your Angular app that has some select boxes like the code below.



```
<form method="somemethod" action="someaction">
  <label>City 1</label>
  <select name="city1">
    <option disabled selected>Select City</option>
    <option>Atlanta</option>
    <option>Midland</option>
  </select>
  <label>City 2</label>
  <select name="city2">
    <option disabled selected>Select City</option>
    <option>Atlanta</option>
    <option>Midland</option>
  </select>
</form>
```

If you had just one select box on the page you could do a simple `by.cssContainingText`.

```
element(by.cssContainingText('option', 'Midland')).click();
```



Running this you would get the result.

That just does not give you much control over your application at all. Selenium will search and return the first available element in the DOM. For this task we need to use nested elements by finding the select box first, and then the option.

```
$('#[name="city2"]').element(by.cssContainingText('option', 'Midland')).click();
```



ES6 Cookbook

Hidden Elements

```
<span style="display: none" name="hiddenSpan">Hidden Text</span>
```

```
$( '[name="hiddenSpan"]' ).getText();  
// returns ""  
$( '[name="hiddenSpan"]' ).getAttribute( 'textContent' );  
// returns "Hidden Text"
```

Async Await

When you are searching for an element with Protractor you are returning a promise that should be fulfilled before moving on to the next step. Lets say you are wanting to debug your tests using console.log statements.

```
console.log($('.someclass').getText());
```

This would return a promise object for that elements text. But that is not very readable when you need it. Alternatively you could wrap that into a then() but that involve a lot more text as well. With ES6 there is the idea of async functions. In order to make this work from out test we have to make the function nested in the "it" block async. After that we can use await to allow the promise to complete and then we can continue processing the text.

```
it('should give us text', async function() {  
  console.log(await $('.someclass').getText());  
});
```

Now if you are trying this at home you will also need some extra libraries added.

Install Babel and Plugins.

```
$ npm install babel-core babel-plugin-transform-async-to-generator
```

Add require to **protractor-conf.js**

```
require('babel-core/register');
```

Create a **.babelrc** file

```
{  
  "plugins": ["transform-async-to-generator"]  
}
```