Client Side: Protractor End to End Tests

Angular’s integration with [Protractor](https://angular.github.io/protractor) to provide end to end tests can be extremely powerful. Using this, in addition to the unit tests of your server and client side code, provides a set of tests that truly validate your source code’s functionality.

If you’re not familiar with Protractor, it provides some additional tooling around [Selenium’s WebDriver](http://docs.seleniumhq.org/projects/webdriver/), which allows you to automate an actual interaction with the browser (typing the keys to fill out a form, clicking buttons, etc). See the [Protractor API](https://angular.github.io/protractor/#/api) for more information on what’s available.

protractor-conf.js

Protractor has a configuration file which you use to state where your test files are located, what browsers will be used during testing, the test framework, etc. Similar to the unit tests, I generally place my end to end tests within the test/client/e2e directory. Below is an example protractor-conf.js file:

"use strict";

exports.config = {

allScriptsTimeout: 11000,

specs: [

"test/client/e2e/\*.js"

],

multiCapabilities: [{

browserName: "firefox"

}, {

browserName: "chrome"

}],

baseUrl: "http://localhost:3000/",

beforeLaunch: function() {

console.log("Starting setup...");

// Return a promise when dealing with asynchronous

// functions here (like creating users in the database)

},

afterLaunch: function() {

console.log("Starting cleanup...");

// Return a promise when dealing with asynchronous

// functions here (like removing users from the database)

},

framework: "jasmine",

jasmineNodeOpts: {

defaultTimeoutInterval: 30000

}

};

This file takes advantage of the multiCapabilities configuration attribute to state that the tests should run in both Firefox and Chrome. There is also beforeLaunch and afterLaunch functions defined which provide us a hook for any test setup or tear down (for example, create a test user to use during login).

Another key configuration attribute is the baseUrl which states where Protractor will begin it’s tests. Note that we’ve supplied http://localhost:3000, which means we’ll need to start our web server prior to running the Protractor tests.

Example Test

The [Protractor tutorial](https://angular.github.io/protractor/#/tutorial) provides a great overview on how to write tests which I recommend looking over. Below is an example test used in the same [meanjs-template](https://github.com/dylants/meanjs-template):

"use strict";

describe("meanjs-template App", function() {

beforeEach(function() {

// get the root of the project

browser.get("/");

});

it("should display heading", function(done) {

browser.getTitle().then(function(title) {

expect(title).toEqual("meanjs-template");

done();

});

});

});

This test navigates to the root of the project, then verifies the page title is correct. Protractor provides many additional functions to get elements on the page using Angular specific concepts, such as [by binding](https://angular.github.io/protractor/#/api?view=ProtractorBy.prototype.binding) or [by model](https://angular.github.io/protractor/#/api?view=ProtractorBy.prototype.model).

Using Grunt to Wire it All Together

Now that we have all three of the test groups setup, the difficulty comes in running them all together. A build system like [Grunt](http://gruntjs.com/) helps solve this problem by enabling us to run these groups of tests, one after another, resulting in full validation of our application. (In addition to the tests above, we can also add in [jshint](https://www.npmjs.com/package/grunt-contrib-jshint) along with[csslint](https://www.npmjs.com/package/grunt-contrib-csslint) to validate our source code semantics.)

View the [getting started guide for Grunt](http://gruntjs.com/getting-started) to get your project initially setup. The three Grunt plugins I chose to use to run the tests are [grunt-jasmine-node](https://www.npmjs.com/package/grunt-jasmine-node), [grunt-karma](https://www.npmjs.com/package/grunt-karma), and [grunt-protractor-runner](https://www.npmjs.com/package/grunt-protractor-runner). With all of that installed, the Gruntfile.js for our tests becomes:

"use strict";

module.exports = function(grunt) {

grunt.initConfig({

jasmine\_node: {

options: {

forceExit: true,

matchall: true,

showColors: true,

includeStackTrace: true

},

all: ["test/server"]

},

karma: {

all: {

configFile: "karma.conf.js",

singleRun: true

}

},

protractor: {

all: {

configFile: "protractor-conf.js",

keepAlive: true

}

}

});

require("load-grunt-tasks")(grunt);

grunt.registerTask("server", "Start the server", function() {

require("./server.js");

});

grunt.registerTask("test", ["jasmine\_node", "karma", "server", "protractor"]);

};

The above configuration file specifies the options for Jasmine, Karma, and Protractor (mostly default options), loads all the necessary Grunt tasks, and defines two of our own tasks: one to start the server, and one which will run our tests. We’ll need to start the server prior to running the Protractor tests, since Protractor is a true end to end test harness.

With this Gruntfile.js in place, executing the tests from the command line can by done with the following command:

$ grunt test

This will run the server side unit tests with Jasmine, the client side unit tests with Karma, and the end to end tests with Protractor. Pretty cool stuff!

Protractor is a [Node.js](http://nodejs.org/) program. To run, you will need to have Node.js installed. You will download Protractor package using [npm](https://www.npmjs.org/), which comes with Node.js. Check the version of Node.js you have by running node --version. It should be greater than v0.10.0.

By default, Protractor uses the [Jasmine](http://jasmine.github.io/1.3/introduction.html) test framework for its testing interface. This tutorial assumes some familiarity with Jasmine.

This tutorial will set up a test using a local standalone Selenium Server to control browsers. You will need to have the [Java Development Kit (JDK)](http://www.oracle.com/technetwork/java/javase/downloads/index.html)installed to run the standalone Selenium Server. Check this by running java -version from the command line.

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 athttp://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**.

Let's start with a simple test that navigates to an example AngularJS application and checks its title. We’ll use the Super Calculator application at<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 = {

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 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 usesendKeys 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}}

[Learn more about locators and ElementFinders](https://angular.github.io/protractor/#/locators).

Run the tests with

protractor conf.js

You should see the page enter 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.

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');

});

});

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.

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 = {

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. For a full list of options, see [the reference config file](https://github.com/angular/protractor/blob/master/docs/referenceConf.js).

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

// conf.js

exports.config = {

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 theby.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 wrong!

});

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".

# [grunt-protractor-runner](https://www.npmjs.com/package/grunt-protractor-runner)

A Grunt plugin for running protractor runner.

npm install grunt-protractor-runner --save-dev

This plugin will install protractor module locally as a normal dependency. Once the plugin has been installed, it may be enabled inside your Gruntfile with this line of JavaScript:

grunt.loadNpmTasks('grunt-protractor-runner');

Finally you need a Selenium server. If you don't have one set up already, you can install a local standalone version with this command:

./node\_modules/grunt-protractor-runner/node\_modules/protractor/bin/webdriver-manager update

## [The "protractor" task](https://www.npmjs.com/package/grunt-protractor-runner#the-protractor-task)

### [Overview](https://www.npmjs.com/package/grunt-protractor-runner#overview)

In your project's Gruntfile, add a section named protractor to the data object passed into grunt.initConfig().

grunt.initConfig({

  protractor**:** {

    options**:** {

      configFile**:** "node\_modules/protractor/example/conf.js", *// Default config file*

      keepAlive**:** true, *// If false, the grunt process stops when the test fails.*

      noColor**:** false, *// If true, protractor will not use colors in its output.*

      args**:** {

*// Arguments passed to the command*

      }

    },

    your\_target**:** {   *// Grunt requires at least one target to run so you can simply put 'all: {}' here too.*

      options**:** {

        configFile**:** "e2e.conf.js", *// Target-specific config file*

        args**:** {} *// Target-specific arguments*

      }

    },

  },

})