PROTRACTOR - TESTING ANGULAR AND NON ANGULAR SITES

So what if at work you need to write an automated functional regression for non angular site? If you're a lucky guy/gal you spend your days dealing with AngularJS and you have convinced your manager to use protractor to write the much needed automated functional suite. Now your manager comes back with a smile on his face asking you to create a new set of tests for this other non angularized web application.

Some would be scared, somebody would run, not you! That's because you know how this thing actually works. You know you can always get the webDriver instance and do some vanilla selenium things. That's why they call you 'The Selenium dude' and sometimes other names which are outside the article's scope.

Well we have this at my job as well. We got two web applications we provide support to.

A brand new one, written in AngularJs and covered by lots of e2e tests (jasmine/karma) from a past time when protractor was not around.

And then we have "THE LEGACY APP". A 10+ years legacy codebase. This application still delivers but there is almost no test harness and making any change is as risky as blind folded ice driving. (I'm feeling dramatic today)

THE LIST OF DEMANDS

When I instrumented Protractor at work, I already knew I had to tackle a few goals:

Ability to test both AngularJS and non AngularJS sites.
Ability to use the same DSL for both AngularJS and non AngularJS tests
Use the same language we use to write unit and e2e tests
Use the same tools/frameworks we use to write unit and e2e tests
Fast to write tests
Fast to learn
Ability to execute tests locally and in Sauce Labs.
Enforce PageObject pattern

MEET PROTRACTOR

1- Protractor provides the means to test angularjs and non angularjs out of the box

2- The dsl to write each kind of test is not the same.

The dsl dedicated for angularjs sites wraps selenium's dsl and enhances it with angularjs specific knowledge. Example:

```
element.find(By.id('example'))
    element.find(By.model('trip'))
```

If you need to interact with a non-Angular page, you may access the wrapped webdriver instance directly by using 'browser.driver'.

```
browser.driver.find(By.id('example'))
```

The 'element' keyword is exposed to all tests via the global. We can observe this looking into protractor's gears in the runner.js file.

```
// Export protractor to the global namespace to be used in tests.
    global.protractor = protractor;
    global.browser = browser;
    global.$ = browser.$;
```

```
    global.protractor = protractor;
    global.browser = browser;
    global.$ = browser.$;
    global.$$ = browser.$$;
    global.element = browser.element;
```

So my first approach was - following the same standard - exposing the browser.driver via an alias. A shorthand if you will. I did this in our protractor configuration, at the onPrepare method.

```
onPrepare: function(){
      global.dv = browser.driver;
      }
```

Having done this, I was able to say

```
dv.find(By.id('.example'))
```

This was shorter but we were still not using the same DSL. This would take developers off, increase learning times, confusion, etc.

After all, protractor is basically a wrapper around selenium. Protractor's charm comes from waiting on angular to finish its work and from knowing what to look in our html to identify our models, repeaters, etc.

Having this in mind and after some code source, github issues light reading, I updated my tests to tell protractor not to be that smart about my non angular site.

```
beforeEach(function() {
      return browser.ignoreSynchronization = true;
    });
```

This forces protractor to forget about his dedicated patience for angular to load and finish its tasks. This allowed us to start using the same DSL

```
element(By.id('example'))
```

Unfortunately the scope of this flag (ignoreSynchronization) affects the whole suite of test, it's a global change every time we set to false or true. We still want to test our AngularSite with all protractor's smartness and patience. So this change would now push our developers to include the same flag on those angular tests, so the flag gets updated accordingly in every test.

```
beforeEach(function() {
      return browser.ignoreSynchronization = false;
    });
```

Far from ideal and error prone, I needed to change this. I needed to provide a more semantic expression so devs could determine whether to set this flag true or false without the need to understand what is going on under the hood.

Once again, the onPrepare method comes really handy. I added a method simply provides a more semantic method to set the flag.

```
onPrepare: function(){
        global.isAngularSite = function(flag){
            browser.ignoreSynchronization = !flag;
        };
    }
```

Now our AngularJS tests start like this:

```
beforeEach(function() {
      isAngularSite(true);
    });
```

And naturally, our non AngularJS tests start like this:

```
beforeEach(function() {
      isAngularSite(false);
    });
```

```
beforeEach(function() {
      isAngularSite(false);
    });
```

It would be great (I have not done a line of code about it) to provide the same flag via annotations instead. Something like:

```
@isAngularSite
    describe "my test", function(){

    }

    @isNonAngularSite
    describe "my test", function(){

    }
```

3- Use the same language we use to write unit and e2e tests

We write CoffeeScript but with a small grunt task we generated all our js files into a temp folder and with a few small changes in protractor config file we were done.

```
specs: ['.tmp/specs/*.js']
```

In CoffeeScript by is a keyword. Therefore we cannot use 'element by.model'.

```
onPrepare: function(){
      global.by_ = protractor.By;
      }
```

Protractor 0.17 has now exposed global.By to solve this issue. This hack is no longer needed.

4- Use the same tools/frameworks we use to write unit and e2e tests

We use Jasmine as our testing framework. We really enjoy the BDD style. As soon as we added protractor to our packages.json, grunt retrieved protractor and its dependencies. Among the dependencies we can find jasmine, mocha, chai, saucelabs, etc.

5- Fast to write tests

I think this quality is not actually provided by protractor itself but the pageObject pattern. Separating the concerns, giving classes/objects only one reason to change makes it easy to write tests. Mandatory reading if you only have an idea of what I'm talking about.

Here is a very simple overview.

You basically represent each page or module or widget by a unique class/object/file. This object has methods/functions that corresponds to this page domains. So a Login page only knows how to 'completeLoginForm(username, password)' Now your 100 tests can actually say LoginPage.completeLoginForm(myUser, myPass123).

If you have done it right each test should be only a couple of lines long, hiding all its webdriver's interactions on the pageObjects, example:

```
beforeEach(function() {
      isAngularSite(true);
      builtWithAngularPage.go("http://builtwith.angularjs.org/");
    });

    it("should be able to search projects", function() {
      builtWithAngularPage.search("airline");
      var projectsList = builtWithAngularPage.getProjectsList();
      expect(projectsList.count()).toBe(1);
```

```
    builtWithAngularPage.search("airline");
    var projectsList = builtWithAngularPage.getProjectsList();
    expect(projectsList.count()).toBe(1);
  });
```

6- Fast to learn

I'm going to speak based on my experience. A year ago, I collaborated on a project to automate that legacy web application. At that time, we were using Java instead of javascript, TestNg and selenium web driver. We were applying the pageObject pattern just the same.

Part of my job was teaching how to write tests to a team coming from QTP sequential programming style, not very familiar with Java. The assignment was over when this team was able to take over the project.

It was a very slow learning curve. Even when we got a core knowledgeable group, it was hard to transmit the knowledge to the next guy. The pageObject pattern and unfamiliarity to Java prevented this project from escalating.

Forward 12 months and here I was facing the same issue; attempting to setup an automated regression but this time with a group of semi to senior Java developers. This group was being introduced to our new UI built with AngularJS, coffeescript, unit tested with jasmine, e2es, etc…the whole thing!

It was now arguable that this automation suite should be written were we had more core knowledge (Java). But everybody was still learning how to write unit and e2e tests in CoffeeScript.

I felt very strongly about using the same tools and language that we had selected to move forward.

So after a discussion I got the green light and move forward with protractor using CoffeeScript.

After setting a new project and making the first commit, I picked an automation story and paired with a senior Java developer who had no javascript/coffeescript/selenium experience. We wrote a test and a couple of page objects. We spent that morning together and when we came back from lunch he took over, I split.

Next day he got a new pair and same thing happened. After a couple of hours, the just arrived dev was ready to be the anchor on the story.

There were times when the pair was stuck but it was always related to a selenium interaction they were doing for the first time; such as: "how do I select a value from this dropdown?". These were expected problems, developers building experience on selenium interactions, the waiting dance, etc.

A week later, with only one pair of rotating Java devs (new to js, cs, wd), we had written 30 end to end tests and around 60 page objects. I have only participated on the first test.

7- Ability to execute tests locally and in Sauce Labs

Protractor allows you to run your tests locally by using chrome driver or using any browser if you download and run the selenium web server. I've set it up both options so when devs are creating tests they can easily test those in firefox, chrome, etc.

Our protractor.conf.js looks like this (extract):

```
exports.config = {
        // The address of a running selenium server.
        seleniumAddress: 'http://localhost:4444/wd/hub',
```

```
        // The address of a running selenium server.
        seleniumAddress: 'http://localhost:4444/wd/hub',

        // If chromeOnly is true, we dont need to stand the selenium s
        // If you want to test with firefox, then set this to false an
        chromeOnly: true,

        // Capabilities to be passed to the webdriver instance.
        capabilities: {
            'browserName': 'chrome'
        },
```

In our Jenkins, I have setup a couple of parametrized jobs that executes different test suites against different browsers. I basically have a shell script that passes protractor a different configuration, protractor-sauce.conf.js and it looks like this (extract):

```
exports.config = {

        // If sauceUser and sauceKey are specified, seleniumServerJar
        // The tests will be run remotely using SauceLabs.
        sauceUser: 'myUSer',
        sauceKey: 'myKey',
```

And this is what my shell script looks like:

```
#!/bin/sh
    NODE_VERSION=0.10.5

    [[ -s ${HOME}/.nvm/nvm.sh ]] && . ${HOME}/.nvm/nvm.sh
    nvm use $NODE_VERSION

    # echo "cleaning previous tests"
    # rm -r test-results/*.xml

    echo "creating tests"
    grunt test-setup

    echo "running regression suite in saucelabs: " + $1 + "-" + $2
    ./node_modules/.bin/protractor protractor-sauce.conf.js --browser=
```

8- Enforce PageObject pattern

Pairing, lunch and learn and code reviews sessions are the most effective ways I found to share and reinforce this approach. Nevertheless, I wanted to provide some kind of boilerplate generation for the developer. I have learned and loved yeoman as a prototype builder and a script generator.

ENTER "THE PATH OF LEAST RESISTANCE"

I created a very simple yeoman generator: generator-ptor. It has several goals such as creating the project with proper dependencies (protractor, selenium server, coffeescript and other needed grunt tasks). This is mainly useful at the beginning of the project or if you plan to create multiple projects.

For daily usage, it provides commands to generate PageObjects and Specs in javascript or coffeescript.

This one creates a file called PageName.js under the pageObjects folder.

```
~/project/functional-regression$ yo ptor:page "TestName"
    You called the page subgenerator with the argument TestName.
       create pageObjects/TestName.js
```

The file would look like this

```
var Factory = require("../lib/pageObject.js").PageObjectFactory;
```

```
var Factory = require("../lib/pageObject.js").PageObjectFactory;

    module.exports = Factory.create({
    #  Examples:

    #  search: function(param){
    #    return element(By.model("query")).sendKeys(param);
    #  }

    #  getProjectsAmount: function(){
    #    element By.binding("projects.length");
    #  }

    })
```

The command creates a file called TestNameSpec.js under the specs folder.

```
~/project/labradoodle/functional-regression$ yo ptor:spec "TestName"
    You called the spec subgenerator with the argument TestName.
       create specs/TestNameSpec.js
```

The file would look like this

```
describe "TestName Test", function(){

      var page = require("../pageObjects/builtWithAngular.js");

      beforeEach function(){
        #    Example
        #    isAngularSite(true);
        #    page.go("http://builtwith.angularjs.org/");
      }

      it "should", function(){
        # page interactions here

        # expectations here
      }

    }
```

If your project already has other tests or pageObjects in CS it will create the CS version; otherwise the JS version will be generated.

Or you can force the CS version like this

```
yo ptor:spec "TestName" --coffee
    yo ptor:page "PageName" --coffee
```

It has many imperfections and it is still crawling but it attempts to aid devs to avoid copy/pasting and promoting to start clean.

Opinions expressed are solely my own and do not express the views or opinions of my employer.

Author: Santiago Esteva, Tags: Protractor , comments