END-TO-END TESTING WITH PROTRACTOR

Lessons learned about
e2e test automation

Walmyr Lima e Silva Filho

End to end testing with Protractor

Lessons learned about e2e test automation

Walmyr Lima e Silva Filho

This book is for sale at http://leanpub.com/end-to-end-testing-with-protractor

This version was published on 2016-11-27

I dedicate this book to my parents and my wife, who always supported me in my endeavors.

I thank all colleagues who have accompanied me in my software development career and that were part of my learnings as a professional focused on software quality.

I thank Dionatan Moura for their support and encouragement in writing this book, in addition to several tips that helped me make this happen.

I thank the beta readers who helped me reviewing the book's content.

I also thank you, reader, for your interest in learning about end to end test automation with the Protractor framework.

Contents

Preface

By Carmen Popoviciu

Testing is an important part of software development. It is our main tool, as developers, in ensuring that the applications we are building are error free and working as intended.

I like to believe that the art of writing good tests lies in a good understanding of what testing actually is, in grasping its basic concepts and paradigms and ultimately in applying those and other best practices wisely. Just like with writing code, testing takes time and effort to get right. It takes reading and practice to fully understand what to test where, to understand the different types of tests or which to use when and just to get to an overall level of proficiency in it. But once you get it right, that knowledge is there to stay, and there is no other feeling like it!

The main focus of this book is Protractor, an e2e testing tool, used mostly in the angular ecosystem. The tool was developed by the Angular Core Team and has been around from pretty much the early beginnings of the framework. Despite that fact, there was a long time when Protractor-related literature and resources were scarce and not as well represented as those related to unit testing for instance. Even to this day I still see some confused faces when talking about e2e tests and Protractor. This was one of the main reasons why Andres Dominguez and myself set out to write the Protractor Styleguide, a collection of rules and best practices, which later became the official styleguide of the framework. Our goal was to share our experience with using Protractor in large scale projects, and provide a set of guidelines to help other developers get started easily and write efficient e2e tests for their applications.

This book is a great continuation of the Protractor Styleguide, not just because it reinforces the best practices described there, but because it takes them a step further and discusses the entire Protractor workflow more in depth and tackles different common scenarios and corner use cases. This can be a great entry point for those just starting out with Protractor, but also a valuable resource for developers who are already familiar with the tool, but want to get more insights into it.

Carmen Popoviciu

@CarmenPopoviciu[1]

Front-End developer

[1]https://twitter.com/@CarmenPopoviciu

By Stefan Teixeira

My first contact with test automation was in 2011, and since then this has changed my career. Being able to use programming and testing skills together, reduce repetitive work and provide fast feedback to the team is fascinating. Nowadays, with the growing popularity of the DevOps culture and the search for continuous delivery, test automation is absolutely necessary. It is impossible to have a reliable deployment process without a good test automation strategy. When we talk about web applications, it is essential to have tests that simulate users interacting with them. These tests are usually called UI tests or end-to-end tests, and they are the focus of this book.

In the last Github Octoverse, it was possible to see JavaScript prevailing as the most used programming language in the open source world. In the web development world, there are many JavaScript frameworks being extensively used, such as AngularJS. Protractor, which was created by the AngularJS team, appears as one of the most complete e2e JavaScript testing frameworks, given its simple learning curve and the features it provides. Besides that, Protractor can also be perfectly used with non-AngularJS applications, which favors its usage.

I follow Walmyr's work since 2014, when I knew his blog (Talking About Testing[2]). Since October 2014, Walmyr has created quality content about Protractor, like blog posts, presentations and videos of the excellent "Learning Protractor" initiative. This book consolidates, in a concise way, all experience acquired in the last years by Walmyr about Protractor and web application testing. This book is extremely recommended for both beginners and experienced professionals, serving as a great reference book on the subject. Good reading!

Stefan Teixeira

@stefan_teixeira[3]

QA Automation Engineer at Toptal

http://stefanteixeira.com.br[4]

[2]http://talkingabouttesting.com
[3]http://twitter.com/@stefan_teixeira
[4]http://stefanteixeira.com.br

Introduction

The idea of writing this book came as a way to gathering a collection of learnings during my career using the Protractor framework, to serve as a source of research for professionals that already use the tool or professionals and students that are interested in learning.

Before going into the details related to the framework, I would like to address some issues that I consider important for having better benefit of what will come next.

Automated tests are a very important part in the software development process, being the basis to ensure quick feedback after application changes, beyond helping on application design and also as source of documentation.

When I started working with software development, focused in software testing, in 2004, I was working in a manual testing approach, where test analysts wrote extensive test cases, which were later manually executed by software testers, who helped on updating these documents, and also in reporting defects into bug tracking tools.

After some time I realized that running tests in a manual way, beyond being a tiring process, repetitive and boring, it was also a not trustable process, since in almost all cases, new changes were added to the application even before the manual test cycle was finished, cycles that sometimes could take a whole week for having all the regression test cases being executed. In other words, in the middle of the regression tests execution a new application version were available for testing, and if the regression tests had to be executed since the beginning, we were running the risk of the same thing happening again, I mean, that before finishing running all tests, a new version would be available again, and if we decided just to continue running the tests from where we had stopped, we could not ensure that the already tested features (in the last version) were still working.

So I decided to start automating these manual tests, which then could be ran as many times as needed, in a quick and trustable way, and against different browsers and different operation system's version.

Also, there is a subject in software development called test pyramid, that was developed by Mike Cohn, where he defines that for ensuring quality in software applications, those applications need a good base of unit testing (the base of the pyramid), some integration tests, to check that the application parts work as expected when integrated (middle of the pyramid), and a smaller number of UI (user interface) tests, that are the kind of tests that the Protractor framework intends to test, also known as end-to-end (e2e) tests.

Below there is an illustration of the test pyramid:

```
                    Ë†
                   / \
                  /   \
                 / E2E \
                /_______\
               /         \
              /Integration\
             /_____________\
            /               \
           /      Unit       \
          /___________________\
```

Therefore, I would like to point out that, even the book dealing with techniques for tests in the top of the pyramid (e2e tests), we as software engineering professionals cannot forget the tests that support the pyramid, once those are tests that can be executed in a much fast way when compared with e2e tests, beyond ensuring that each part of the application works as designed, when isolated tested, and also when integrated with each other.

Now that we already have some fundamentals, let's go to the Protractor e2e tests.

What is Protractor?

Protractor is an open source end-to-end test framework for AngularJS applications (created by the same team that maintain the AngularJS framework), which is used for running tests in web applications as real users would do, in real browsers, such as Chrome and Firefox.

Even being created with some specific features for AngularJS apps, with a single line of code it can be used to create test for non Angular apps.

Protractor is based on WebDriverJS, but as mentioned, it has some specific features for AngularJS apps and a different syntax, and it works with Selenium to provide this test infrastructure to simulate real web applications usage.

The Selinium server interpret test commands and forward it to the browser, and Protractor cares about the test scripts, written in JavaScript, using Node.js.

To exemplify, it works like this:

First, Protractor communicates with Selenium through the WebDriver API, using HTTP, then Selenium communicates with the browser using the JSON Webdriver Wire Protocol, and finally, the browser runs the received commands as a real user.

Pre-requirements to start

Protractor is a program base on Node.js, so, for using it you will need Node.js installed, in a version higher than v.0.10.0.

To check which version of Node.js is installed in your computer you just need to run the following command in your console:

```
node --version
```

Installation

Since Protractor is based on Node.js, this uses NPM (node package manager) for its installation. NPM is a package manager for Node.js based projects.

To install Protractor you just need to run the following command in the console:

```
npm install -g protractor
```

This command will install Protractor globally in your computer.

After the installation you can check which version of Protractor is installed using the below command:

```
protractor --version
```

It must show something like this:

```
Version 4.0.2
```

After the Protractor installation you will need to update the webdriver-manager. Use the following command:

```
webdriver-manager update
```

You can also install Protractor and the webdriver-manager as development dependencies of your project, instead of installing them globally. For doing this you just need to run the following commands:

```
npm install protractor --save-dev
npm install webdriver-manager --save-dev
```

Basic configurations

With Protractor installed and the webdriver-manager updated, the next step is to created a configuration file, which will define things such as: the address where the Selenium server will be running, in which browser the tests will run against, which tests will be executed, which base URL each test will access when starting, beyond many other possible configurations. For more knowing more about other configurations take a look at the **Advanced configurations** chapter.

Creating the configuration file

Inside the directory's structure of your project, if there is already a directory where tests are stored, such as unit tests and integration tests, I recommend you to create a directory called **e2e**. If there is no **tests** directory created yet, create it and then create the **e2e** directory inside it. For more details take a look at the **Project structure** section of the **Good practices** chapter.

Inside the **e2e** directory, create a file called **protractor.conf.js**..

Open the **protractor.conf.js** file in you code editor and then add the following baisc configurations:

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

Obs.: The value of the baseUrl attribute must be later changed by the base URL of the application project that will be tested using the Protractor framework. For now look at this URL as an example.

With this you already have the minimum necessary to start writing your tests.

Writing the first test

With the basic configuration for running tests with the Protractor framework, the next step is to create the tests themselves.

From the **e2e** directory (previously created), which contains the Protractor configuration file, create a sub-direcotry called **specs**, and inside it, create a file called **homepage.spec.js**.

Open the **homepage.spec.js** file in your code editor and add the following code into it, which will be detailed explained soon:

```
// homepage.spec.js

describe('Homepage', function() {
  it('perform a search into the api page', function() {
    browser.get('#/api');

    element(by.model('searchTerm')).sendKeys('restart');
    element(by.css('.depth-1')).click();

    expect(element(by.css('.api-title')).getText()).toContain('browser.restart');
  });
});
```

Before running the just created test, let's understand the structure of a test written using Protractor.

First a **describe** is defined, which is the test suite definition. This received a description (in this case 'Homepage'), and then executes a function (or a callback).

Then a **it** is defined, which is the test case itself. This also receives a description, that is exactly what the tests is supposed to test (in this case 'perform a search into the api page'), and then executes a function (or callback).

And then the test steps are defined, which will be explained next, line by line:

This first steps is the following:

```
browser.get('#/api');
```

This step basically access a **#/api** relative URL through the browser, but, since we defined a base URL (baseUrl) in the Protractor configuration file (protractor.conf.js), then this URL is concatenated with the base URL, generating the following absolute URL: **http://www.protractortest.org/#/api**.

In the second step the value 'restart' is typed in a field defined by its model (by.model('searchTerm')).

```
element(by.model('searchTerm')).sendKeys('restart');
```

In the third step a link with a css class 'depth-1' is clicked.

```
element(by.css('.depth-1')).click();
```

And finally the assertion is done:

```
expect(element(by.css('.api-title')).getText()).toContain('browser.restart');
```

In this case it is verified that an element with a css class 'api-title' contains the text 'browser.restart'.

Running the first test

Now that Protractor already have the basic configurations to run the tests and it also has a first test written, it is time to run the test, to see the application behavior.

In the console, type the following command to start the Selenium server:

```
webdriver-manager start
```

And in a second tab of the console, from the directory where the Protractor configuration file (protractor.conf.js) is, execute the below command:

```
protractor
```

With webdriver-manager started and the command for Protractor run the tests, the browser defined in the Protractor configuration file will be opened (in this case the Chrome browser), and then the defined test steps will be executed. After the test execution the browser will be automatically closed.

In case of success running the tests, you will see a result like this:

```
$ protractor
[17:26:44] I/hosted - Using the selenium server at http://localhost:4444/wd/hub
[17:26:44] I/launcher - Running 1 instances of WebDriver
Started
.


1 spec, 0 failures
Finished in 1.724 seconds
[17:26:48] I/launcher - 0 instance(s) of WebDriver still running
[17:26:48] I/launcher - chrome #01 passed
```

See that the Selenium address is shown, the same defined in the Protractor configuration file (http://localhost:4444/wd/hub), and after approximately 2 seconds the test is finished, without any failure. Also, it shows the information that the test was executed against the Chrome browser.

The importance of test results

When writing automated test scripts it is important to ensure not just that the tests will pass, but also that they will fail if the application doesn't behave as expected.

A technique I use to use is to check if my tests are really testing something is to modify the assertion (expect) to simulate a failure, and then see that the test will fail if the expected result is different from what was defined.

Simulating a failure

Modify the last test step by the following:

```
expect(element(by.css('.api-title')).getText()).not.toContain('browser.restart');
```

See that now the assertion is that the element with the css class 'api-title' **does not** contain the text 'browser.restart'.

Execute the test again and take a look at the result.

After the test execution, the result must be something like this:

```
$ protractor
[17:57:42] I/hosted - Using the selenium server at http://localhost:4444/wd/hub
[17:57:42] I/launcher - Running 1 instances of WebDriver
Started
F

Failures:
1) Homepage perform a search into the api page
  Message:
    Expected 'browser.restart View code' not to contain 'browser.restart'.
  Stack:
    Error: Failed expectation
        at Object.<anonymous> (/Users/Walmyr/www/Protractor-eBook/e2e/specs/home\
page.spec.js:8:57)
        at /usr/local/lib/node_modules/protractor/node_modules/jasminewd2/index.\
js:96:23
        at new Promise (/usr/local/lib/node_modules/protractor/node_modules/sele\
nium-webdriver/lib/promise.js:1043:7)
        at controlFlowExecute (/usr/local/lib/node_modules/protractor/node_modul\
es/jasminewd2/index.js:82:18)
        at TaskQueue.execute_ (/usr/local/lib/node_modules/protractor/node_modul\
es/selenium-webdriver/lib/promise.js:2790:14)
        at TaskQueue.executeNext_ (/usr/local/lib/node_modules/protractor/node_m\
odules/selenium-webdriver/lib/promise.js:2773:21)
        at asyncRun (/usr/local/lib/node_modules/protractor/node_modules/seleniu\
m-webdriver/lib/promise.js:2697:25)
        at /usr/local/lib/node_modules/protractor/node_modules/selenium-webdrive\
r/lib/promise.js:639:7
        at process._tickCallback (internal/process/next_tick.js:103:7)

1 spec, 1 failure
Finished in 2.049 seconds
[17:57:45] I/launcher - 0 instance(s) of WebDriver still running
[17:57:45] I/launcher - chrome #01 failed 1 test(s)
[17:57:45] I/launcher - overall: 1 failed spec(s)
[17:57:45] E/launcher - Process exited with error code 1
```

See that and error happened related to the expected result, where it expected that the valued ‘browser.restart’ does not contained the value ‘browser.restart’.

It is extremely important to force failures in the written tests, to ensure that they will really fail when there is a real application failure. Without this guarantee it may happen what is called false negatives, in other words, tests that passes when they should actually fail.

Correct the test so it can pass again and run it one more time to ensure that the behavior is ok.

The AAA standard (Arrange, Act, Assert)

To finish the introductory chapter I decided to bring a widely used standard, not just in writing e2e tests, but also when writing unit tests and integration tests. This standard is called AAA (arrange, act e assert).

Arrange is the test pre-condition. It is what needs to happen for the test being in the needed condition to be executed.

In this example’s chapter, the arrange is the navigation until the ‘#/api’ relative URL.

In an example of a content management system, where to create a content the user needs to be logged in, for example, the arrange would be to perform the login process in the application.

Act are the steps executed in the application (the inputs).

In this example’s chapter, the act are the steps where the value ‘restart’ is typed in the element defined by de model ‘searchTerm’, and when the field defined by the css class ‘depth-1’ is clicked.

In an example of a content management system, in the create content test case, the act would be the steps to create the content, such as navigating until the desired content creation page, filling the form and them submitting the same form.

Assert is the verification (the output). It is what turns this automation in a real test.

In this example's chapter, the assert is the assertion that the element with the css class 'api-title' contains the text 'browser.restart'.

One more time in the content management system example, the assert in a test case of a content creation could be an assertion that a success message is displayed after creating the content, or even better, an assertion that the element was really created, such as verifying that the content is available in a list of contents.

Good practices

I decided to address good practices on e2e testing development early as the second chapter of the book, as these will be the basis for many of the other items discussed in the next chapters, and also the foundation for the development of robust and easy to maintain automated tests.

In this chapter we will see:

- Some general rules
- Issues related to project structure
- Locators strategies
- The Page Object standard
- And test suites

General rules

In this section are some general rules regarding good practices in writing e2e automated test scripts.

Don't write e2e tests for functionalities already covered by unit tests or integration tests

E2e tests that simulate the real usage of an application as a real user does have a high cost on time execution when compared with unit tests and integration tests, since in this UI (user interface) tests it needs to open the browser, visit web pages (which can be a bit slow), fill forms, click in buttons, etc. So, if there are already tests for such feature in one of theses other layers, these will be executed much faster.

Beyond that, when dealing with software development using clean code practices, for example, it is suggested not to duplicated code, and e2e tests for features already covered by tests in other layers (unit or integration) is test duplication, in other words, something unnecessary and that just adds complexity, without adding value.

Use only on configuration file

As I had just said, code duplication is something we must avoid, and the same is true for configurations, once if there is a change in the application or its architecture, for example, that needs to change a configuration, then this change would need to be done in more than one configuration file, and this is something that may be forgotten. So, in some cases you will change the configuration in one file and will forget in others.

Also, there are build and task automation tools, such as Gulp and Grunt that can help you in the task for overwritten such configurations at runtime, for example, for enabling tests execution in different environments, such as local development environment, QA environment, UAT (user acceptance testing) environment, performance environment, or even in production.

So,

Avoid something like this:

- protractor.conf.local.js
- protractor.conf.qa.js
- protractor.conf.uat.js
- protractor.conf.perf.js
- protractor.conf.prod.js

And instead, have only one file:

- protractor.conf.js

But then, with different tasks to overwrite the baseUrl, for example, to run the same tests in different environments. Something like:

- gulp e2e-test-local
- gulp e2e-test-qa
- gulp e2e-test-uat

And son on.

The same can be done to overwrite, for example, the browser in which you wan to run the tests. Something like:

- gulp e2e-test-chrome
- gulp e2e-test-firefox
- gulp e2e-test-ie

Project structure

Well structured projects bring several advantages in software development, these advantages will be presented below, along with some practices I used successfully on real projects.

Group the e2e tests in a way that make sense for you project

Software projects, including automated tests sub-projects, must be well structured for organizational reasons, ease location of each of its parts, separation relative to other types of testing, such as unit tests, integration tests, performance tests, etc, beyond making possible to have clean structure where things make sense.

An example of a e2e test project structure I like to use is the following:

```
|-- project-root-directory
    |-- client
    |-- server
    |-- tests
        |-- e2e
            |-- page-objects
                  contact.po.js
                  homepage.po.js
                  signIn.po.js
                  signUp.po.js
            |-- specs
                  contact.spec.js
                  hompepage.spec.js
                  signIn.spec.js
                  signUp.spec.js
            helper.js
            protractor.conf.js
            README.md
        |-- integration
        |-- unit
    .gitignore
    gulpfile.js
    package.json
    README.md
```

Notice that in the above example, inside the e2e directory there is a README.md file, which I use to document briefly important questions related to the purpose of these tests and explanations about the structure of this sub-project, the pre-requirements and initial setup to start, along with useful tips and any other information that is convenient.

Locators strategies

Choosing good locators is one of the secrets for writing robust and easy to maintain e2e tests. In this section I will show you some strategies recommended to choose the HTML element locators, beyond some practices that must not be followed.

Avoid using xpath

As per the style guide from the official Protractor website, it is not recommended to use xpath to locate HTML elements for e2e testing. Some reasons are:

- Markup liable to change
- Performance Issues
- Bad for readability

An example of an element located using xptah (from the Protractor official web site) is shown below:

```
var elem = element(by.xpath('/*/p[2]/b[2]/following-sibling::node()' +
   '[count(.|/*/p[2]/b[2]/following-sibling::br[1]/preceding-sibling::node())' +
   '=' +
   ' count((/*/p[2]/b[2]/following-sibling::br[1]/preceding-sibling::node()))' +
   ']'));
```

Realize the difficulty to understand what HTML element such locator aims to identify.

So, **don't use xpath** to locate elements.

Prefer to use Protractor specific locator when possible

Since Protractor is the official framework for e2e testing for AngularJS applications, it has a series of locators that are specific for this kind of application, such as:

- by.binding
- by.model
- by.repeater

With that in mind, it is recommended that these are the first option when choosing locators. Some reasons are:

- Easy access to elements
- The code is less likely to change relative to the markup
- There are more readable locators

Here are some examples of elements located through Protractor specific locators:

```
var personField = element(by.binding('person.name'));
var searchField = element(by.model('ctrl.findBy'));
var firstDocumentLink = element(by.repeater('item in trail')).row(0);
```

Prefer by.id and by.css when there are no Protractor specific locators available

Not always is possible to locate elements in the application through Protractor specific locators. In theses cases, it is recommended, first, identify them by id, and if there is not id, then use css locator (css selector), preferably by class.

Some advantages of using such elements are:

- Easy access to elements
- Good performance in locating elements in the DOM (document object model)
- Use of markup less likely to change
- Readability

Here are some examples of elements located by id and css class:

```
var footer = element(by.id('footer'));
var backToTop = footer.element(by.css('.back-to-top')):
```

Avoid locating elements by their texts

In some projects I worked I made the mistake of locating elements by their text, and suddenly, the e2e tests started to fail, causing false negative results, which is totally detrimental to software projects, since the team must trust the test results, and when try fail, they should be indicating real bugs in the application, and not false negatives, due to poorly written tests.

In the case of the projects where I witnessed such problems, I had located some elements through their texts, and when the application went through internationalization, where many texts were translated, tests started failing and had to be fixed.

Some reasons to avoid locators by text are:

- Text button, links and labels tend to change over time
- E2e tests cannot fail due to text changes
- Internationalization (translation)

Finally, if there is no good locator (Protractor specific, by id or by css class, for example) to the element which the test needs to interact with or even to do a verification, it is wiser to modify the application to add an id or a css class to the element, than using a fragile locator. This is called testability.

Page Objects

As described in the Protractor official website, it is common to use Page Objects in writing e2e tests, since such a patter helps writing clean tests, once the application components are encapsulated in such objects, and when for some particular reason an application template changes, only the Page Object that defines that element needs to be changed, instead of the tests that use it, these remaining intact. Furthermore, Page Objects can be used across different tests, where appropriate.

Use Page Objects to interact with the page in test

Just reinforcing, the advantages of using the Page Object standards are:

- Reuse of code
- Clean an readable tests

See an example of a test written using the Page Objects standard:

```
// contactPage.spec.js

var ContactPage = require('../page-objects/contactPage.po.js');

describe('contact page', function() {
  var contactPage = new ContactPage();

  it('successfull contact form submission', function() {
      contactPage.visit();

      contactPage.fillForm('walmyr', 'walmyr@email.com', 'My contact message.');
      contactPage.submitButton.click();

      expect(contactPage.successMessage.isDisplayed()).toBe(true);
  });
});
```

Notice how easy is to understand what the test is proposed to make/test.

Obs.: The contactPage.po.js file is shown later.

Declare a Page Object per file

Since a Page Object proposes to encapsulate the elements of a specific page of the application (or part of it, in some situations), it is recommended to declare only one Page Object per file. So the code is cleaner and it is easier to find what you are looking for.

See an example, from the project structure section of this book:

```
|-- page-objects
      contact.po.js
      homepage.po.js
      signIn.po.js
      signUp.po.js
```

In this case, there are four Page Objects, each one encapsulating elements of different parts of the application.

Use only one module.exports at the end of the Page Object file

As per the last recommendation, having only one Page Object per file, there is only one class to be exported.

See below the contactPage.po.js file that encapsulate the elements shown in the **Use Page Objects to interact with the page in test** section:

```
// contactPage.po.js

var ContactPage = function() {
  this.emailField = element(by.id('email-field'));
  this.messageTextAreaField = element(by.id('message-text-area-field'));
  this.nameField = element(by.id('name-field'));
  this.submitButton = element(by.id('submit'));
  this.successMessage = element(by.css('.success-message'));
};

ContactPage.prototype.fillForm = function(name, email, message) {
  this.nameField.sendKeys(name);
  this.emailField.sendKeys(email);
  this.messageTextAreaField.sendKeys(message):
};

ContactPage.prototype.visit = function() {
  browser.get('contact');
};

module.exports = ContactPage;
```

Notice that the ContactPage Page Object has five elements exposed in a public way, which can be directly used in the tests, even only two of them are being used directly in this case. Later there is a section the explains better this approach. The elements are:

- emailField
- messageTextAreaField
- nameField
- submitButton
- successMessage

Obs.: I use to organize elements in an alphabetical way when they are part of the same context.

Furthermore, the ContactPage Page Object has two functions (methods in object orientation), which are isolated explained below:

Method of function **fillForm**:

It receives three arguments (name, email and message), and fill three elements with these values, in this order. The elements filled are exactly the ones that are not directly used in the test, since in this case they are used by this method or function. Later there is a section that explains in depth this approach.

Method of function **visit**:

It basically access a page through a relative URL 'contact'.

Require and instantiate all node modules at the top

Just for clarification purpose, a Page Object in this case is considered a node module.

In the example shown in the **Use Page Objects to interact with the page in test** section, it is possible to notice that in the first line of the file, the ContactPage Page Object is required (**var ContactPage = require('../page-objects/contactPage.po.js')**), and then an object of type ContactPage is instantiated (**var contactPage = new ContactPage();**), right after the describe.

An example of a test that requires more than one Page Object is shown below:

```
// contactPageWithPageObjectWrapper.spec.js

var ContactPage = require('../page-objects/contactPage.po.js');
var MessagesWrapper = require('..page-objects/messagesWrapper.po.js');

describe('contact page', function(){
  var contactPage = new ContactPage();
  var messagesWrapper = new MessagesWrapper();

  it('successfull contact form submission', function() {
      contactPage.visit();

      contactPage.fillForm('walmyr', 'walmyr@email.com', 'My contact message.');
      contactPage.submitButton.click();

      expect(messagesWrapper.successMessage.isDisplayed()).toBe(true);
  });
});
```

See that the test is actually the same already shown, but refactored to use a Page Object called MessageWrapper, since there are other pages in the same application that also have these messages, so it is something like a generic Page Object. In this case, now the verification (expect) is done over the messageWrapper instance of the MessageWrapper Page Object, instead of the contactPage instance of the ContactPage Page Object. This way, other pages that also needs to verify a success message can use the exact same line of code, that is:

```
expect(messagesWrapper.successMessage.isDisplayed()).toBe(true);
```

Declare all the constructor's elements as public

As mentioned in the section **Use only one module.exports at the end of the Page Object file**, in the Page Object example, I commented that of the five elements exposed as public, only two were being used directly by the test, but even so all were exposed in a public way. This is explained since the user of a Page Object (the developer or tester that will write the tests) must have quick access to the elements available in the page. In the test shown, not all the elements were being used, but in other tests, these may be necessary in isolation, instead of in the method that used them (fillForm).

Take a look at an example:

```
// contactPageWithPageObjectWrapper.spec.js

var ContactPage = require('../page-objects/contactPage.po.js');
var MessagesWrapper = require('..page-objects/messagesWrapper.po.js');

describe('contact page', function(){
  var contactPage = new ContactPage();
  var messagesWrapper = new MessagesWrapper();

  it('try to submit contact form filling only the email field', function() {
      contactPage.visit();

      contactPage.emailField.sendKeys('email@example.com');
      contactPage.submitButton.click();

      expect(messagesWrapper.errorMessage.isDisplayed()).toBe(true);
  });
});
```

In this case, since the element emailField is also exposed in a public way, it can be directly used in the test.

As an aditional and informative information (not related to this context), notice that now the messageWrapper instance (from the MessageWrapper Page Object) also has an element exposed in a public way, called errorMessage, which is being used in the assertion (expect).

Declare function for operations that need more than one step

As mentioned in the **Use only one module.exports at the end of the Page Object file** section, in the example of the Page Object shown, there is a method called **fillForm**. See below this method:

```
ContactPage.prototype.fillForm = function(name, email, message) {
  this.nameField.sendKeys(name);
  this.emailField.sendKeys(email);
  this.messageTextAreaField.sendKeys(message):
};
```

This method aims exactly in meet the good practice **Declare function for operations that need more than one step**, in other words, since three steps are necessary to fill the contact form, these steps are encapsulated in a function that receives three arguments (name, email and message), and then it fills the three elements with the values received as arguments, in this order.

This approach is very important for code reuse issues and elimination of duplication.

See below a new test, which uses such method:

```
it('try to submit contact form with invalid email format', function() {
    contactPage.visit();

    contactPage.fillForm('walmyr', 'invalidformat', 'My contact message');
    contactPage.submitButton.click();

    expect(messagesWrapper.errorMessage.isDisplayed()).toBe(true);
});
```

Notice that the same method was used, but just changing the second argument, then enabling to test the application's behavior in another test scenario.

Don’t make assertions in the Page Objects

I consider this a very important rule when talking about automated tests. See some reasons that justify it:

- It is responsibility of the tests doing assertions
- The people who will read the test scripts should be able to understand the application’s expected behavior by just reading the test.

That is, the **expect** must always be inside the file with the extension .spec.js. This is related to the question of separation of responsibilities, where the responsibility of the test is: from a context, perform actions, and then check the result, while the responsibility of the Page Object is: to encapsulate elements, and functions or methods for operations that need more than one step, as explained in the last section.

Add *wrappers* for directives, dialogs and common elements

As shown in the **Require and instantiate all node modules at the top** section, in this, the test used a Page Object called MessageWrapper, which was created from a refactoring, since the messages are displayed in different pages. This is the reason of this good practice. That is, if there are elements that are repeated throughout the application in different pages, instead of duplicating such elements in different Page Objects, it is recommended to create these wrappers, that are more generic Page Objects, which can be used in different contexts.

Some reasons to use Page Objects of the type wrappers:

- They can be used in multiple tests
- Avoid code duplication
- When a directive, dialog, or common element changes, you just need to change the wrapper, and only once.

More on Page Objects is addressed in the next chapter, **Page Objects* ___

Test suites

To finish the good practices chapter, I will talk about some important points when dealing with e2e automated test suites.

Do not use mock unless it is totally necessary

It is not recommended to use mocks in e2e tests, exactly because e2e tests are supposed to test the application as a real user would do, and when a real user utilizes the application, it does this by interacting with it as a whole, with all its integrations.

Some reasons about why not to use mocks in e2e automated tests are:

- Using the real application with all its dependencies provides high reliability
- In the case where one part of the application that is being tested using mock is changed and we forget to change the mock, the test that uses it will be still passing, even if the change has broken the application. This is called false positive. That is, the test will be still passing when it should be failing, since certain part of the application stopped behaving as expected, but since the mock is outdated it causes the impression that the application is still working

Use Jasmine2

By default the Protractor framework uses the syntax of the Jasmine framework, but in its initial version, it used Jasmine1, and when it evolved, it started to support Jasmine2, which can be defined in the configuration file, in the framework configuration, subject that is addressed in the **Advanced configurations** chapter.

Some advantages of using Jasmine2 are:

- It is well documented
- It is also supported by the Protractor team
- It has the beforeAll and afterAll functions, not existent in the previous version

See an example of the beforeAll function usage:

```
beforeAll(function() {
  homepage.visit();
  homepage.accessNewForm();
  newAccountForm.createAccount(accountData);
});
```

See an example of the afterAll function usage:

```
afterAll(function() {
  dashboard.delete(accountData);
});
```

Notice that such functions can be used for pre and post conditions of the test suite.

Make the tests independent at least at the file level

It is ideal that each test is totally independent, but when this is not possible, due to issues related to the cost of time execution, due to pre and post conditions too costly, make them independent at least the the file level (file with .spec.js extension).

Some advantages of this approach are:

- Parallel test execution using sharding (this subject is addressed in the **Advanced configurations** chapter)
- The order of the test execution is not always guaranteed
- It is possible to execute test suites in an isolated way

Make the test cases totally independent of each other

Independent tests is a good practice not just on writing e2e tests, but also when writing unit tests and integration tests.

Some advantages of creating totally independent tests are:

- Possibility of running tests in an isolated way
- Easy do debug

Exception:

When the operations to initialize the tests are too costly, it is recommended that the tests are independent in the file level. Otherwise they would take to long to run, not providing quick feedback, which is one of the principles behind automated tests.

Navigate to the page in test before each test

When navigating to the page that will be tested before each test case, we ensure that the application is in a clean state, where there is not risk of previous executed tests after the current test.

One practice way to use this approach is by using the beforeEach function.

See an example:

```
beforeEach(function() {
  homepage.visit();
});
```

Have a test suite that navigate through the main application’s routes

It is interesting to have a suite focused on navigating through the main routes of the application, such as navigating through menus, to ensure that when a real user is using such application, he will be directed to the right page.

In this kind of tests we can have assertions that the user was taken to the right URL, as well as assertions that the right elements are being displayed (elements that are specific from that page, such as titles, for example). My recommendation in this kind of tests is to use both assertions combined, that is, besides verifying that the user is taken to the right URL, verifies also that the right elements are being shown.

Furthermore, tests related to navigation are usually faster, when compared with tests that interact more with the application.

Some advantages of this approach are:

- Ensuring that the main parts of the application are well connected
- Users don’t navigate through applications typing URLs
- Trust related to access permissions issues

The definition of test suites is detailed in the **Advanced configurations** chapter.

Have a smoke test suite

A smoke test suite intends to be a quick test suite that ensure that the happy path test scenarios are working as expected.

Also, when using practices like continuous delivery, a smoke test suite may be useful when using deployment pipelines, avoiding that more slow test suites, such as a e2e regression test suite, is executed if there is a failure in the smoke test suite, for example.

As already mentioned in the last section, the definition of test suites in the configuratoin file is detailed in the **Advanced configurations** chapter.

Page Objects

As already mentioned in the **Good practices** chapter, the Page Objects standard helps on writing clean tests with reusable code.

In this chapter I will demonstrate with real examples using the Page Objects standard, so that you reader can realize its advantages.

Do you remember the below test, shown in the introductory chapter of the book?

```
// homepage.specs.js

describe('Homepage', function() {
  it('perform a search into the api page', function() {
    browser.get('#/api');

    element(by.model('searchTerm')).sendKeys('restart');
    element(by.css('.depth-1')).click();

    expect(element(by.css('.api-title')).getText()).toContain('browser.restart');
  });
});
```

Below is a refactoring version of the same test, but now using the Page Objects standard.

```
// homepageWithPageObjects.specs.js

var ApiPage = require('../page-objects/apiPage.po.js');

describe('Homepage', function() {
  var apiPage = new ApiPage();

  it('perform a search into the api page', function() {
    apiPage.visit();

    apiPage.searchField.sendKeys('restart');
    apiPage.firstLinkOnLeftSide.click();

    expect(apiPage.itemTitle.getText()).toContain('browser.restart');
  });
});
```

Notice how the test itself is now easier to read, without expression dificult to understand, such as element, browser, by.id, by.model and by.css.

The idea of using Page Objects is to make tests readable and to encapsulate elements and functions from specific pages in Page Object files, in this case, files with the extension .po.js.

See below the Page Object from the ApiPage used in the test shown above:

```
// apiPage.po.js

var ApiPage = function() {
  this.firstLinkOnLeftSide = element(by.css('.depth-1'));
  this.itemTitle = element(by.css('.api-title'));
  this.searchField = element(by.model('searchTerm'));
};

ApiPage.prototype.visit = function() {
  browser.get('#/api');
};

module.exports = ApiPage;
```

Refactoring tests to use Page Objects

In case you already use the Protractor framework for writing e2e automated tests, but you are not yet using the Page Objects standard, I would recommend you to start doing this, because it will ease your life or the life of who will maintain these tests in the future, besides facilitating on creating new test cases when new features are added into the application.

The way I use to refactor tests that don't use Page Objects to start using it is the following.

First, I rewrite the test as if the Page Objects already exist.

This was exactly what I did to refactor the tests shown in the introductory chapter.

First of all, I required the Page Object before the describe's definition, even before the existance of this file.

```
var ApiPage = require('../page-objects/apiPage.po.js');
```

After that I created an instance of this object, right after the describe, so that the elements I would implement in the Page Object would be already available for usage through the tests.

```
var apiPage = new ApiPage();
```

And then I replaced the browser.get and the elements definitions into the test by the following:

```
apiPage.visit()
apiPage.searchField
apiPage.firstLinkOnLeftSide
apiPage.itemTitle
```

With that I already knew exactly what the Page Object would need to have.

So I created the page-objects directory inside the e2e directory, and inside it I created the apiPage.po.js file.

And then I implemented the **visit** function this way:

```
ApiPage.prototype.visit = function() {
  browser.get('#/api');
};
```

I like to have a function called visit to navigate until the page in test, for readability questions.

I implemented also the public elements to the ApiFunction, this way:

```
this.firstLinkOnLeftSide = element(by.css('.depth-1'));
this.itemTitle = element(by.css('.api-title'));
this.searchField = element(by.model('searchTerm'));
```

And finally I exported such Page Object, so that it could be instanciated inside any test, after required:

```
module.exports = ApiPage;
```

This same process can be used to refactor any test that does not uses the Page Objects standard, to start using it, and it makes the process of refactoring easier and intuitive, since when you are going to implement the Page Object, you already know exactly what it will need to make your test to work. This is similar of doing TDD (test-driven development), where you first develop the test and just after that you develop the feature. When refactoring e2e tests to use the Page Objects standard, you first modify the test as if a Page Object already exist, and then you implement the appropriate Page Object for it.

Another example of Page Objects

In the **Good practices** chapter I talked about Page Objects and how it helps on eliminating duplicity of code in different test cases, or even in different test suites, in the cases where we use Page Objects of type wrappers.

See below an example of a test suite that don't use Page Objects and try to find the duplicated code.

```
// todoMvc.spec.js

describe('Todo MVC Angular', function() {
  var newTodoField = element(by.id('new-todo'));

  it('add an item in the todo list', function() {
    browser.get('http://todomvc.com/examples/angularjs/#/');

    newTodoField.sendKeys('Create test without page object');
    newTodoField.sendKeys(protractor.Key.ENTER);

    expect(element.all(by.css('.view')).count()).toEqual(1);
  });

  it('add new item in the todo list', function() {
    browser.get('http://todomvc.com/examples/angularjs/#/');

    newTodoField.sendKeys('Create new test without page object');
    newTodoField.sendKeys(protractor.Key.ENTER);

    expect(element.all(by.css('.view')).count()).toEqual(2);
  });
});
```

Besides the code not being the more readable, since the exposed elements definitions inside the tests, also, the first and the second test cases execute the same actions when typing an item in the newTodoField and then simulating the ENTER key being pressed.

See how the test suite is readable after is refactored using the Page Objects pattern:

```
// todoMvcWithPageObjects.spec.js

var TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Angular', function() {
  var todoMvc = new TodoMvc();

  it('add an item in the todo list', function() {
    todoMvc.visit();

    todoMvc.addItemOnTodoList('Create test without page object');

    expect(todoMvc.listOfItems.count()).toEqual(1);
  });

  it('add new item in the todo list', function() {
    todoMvc.visit();

    todoMvc.addItemOnTodoList('Create new test without page object');

    expect(todoMvc.listOfItems.count()).toEqual(2);
  });
});
```

Notice that both tests, that before had four steps each, now have just three, since the repeated steps were now transformed in a function, called addItemOnTodoList, which received as parameter a string with the item value that you need to add to the list.

Notice also the AAA (arrange, act, assert) standard, mentioned in the introductory chapter.

See below the implementation of this Page Object:

```
// todoMvc.po.js

var TodoMvc = function() {
  this.listOfItems = element.all(by.css('.view'));
  this.newTodoField = element(by.id('new-todo'));
};

TodoMvc.prototype.addItemOnTodoList = function(item) {
  this.newTodoField.sendKeys(item);
  this.newTodoField.sendKeys(protractor.Key.ENTER);
};

TodoMvc.prototype.visit = function() {
  browser.get('http://todomvc.com/examples/angularjs/#/');
};

module.exports = TodoMvc;
```

Note that in addition to the addItemOnTodoList method, a visit method was created too, which encapsulate the code **browser.get('http://todomvc.com/examples/angularjs/#/');**, making the test itself more readable and without unnecessary information.

Creating and using wrapper Page Objects

As already mentioned in the **Good practices** chapter, in the section about **Page Objects**, it is recommended to create wrapper Page Objects for elements that are common in different application's pages.

Below there is an example of tests for two different pages of the same application, but that share some common elements, and then they need a wrapper Page Object.

```
// choko.specs.js

var CreateAccountPage = require('../page-objects/chokoCreateAccount.po.js');
var MessagesWrapper = require('../page-objects/chokoMessagesWrapper.po.js');
var SignInPage = require('../page-objects/chokoSignIn.po.js');

var messageWrapper = new MessagesWrapper();

describe('Choko - Sign in', function() {
  var signInPage = new SignInPage();

  it('try to sign in without filling any field', function() {
    signInPage.visit();

    signInPage.signInButton.click();

    expect(messageWrapper.errorMessage.isDisplayed()).toBe(true);
  });
});

describe('Choko - Create account', function() {
  var createAccountPage = new CreateAccountPage();

  it('try to create account without filling any field', function() {
    createAccountPage.visit();

    createAccountPage.createAccountButton.click();

    expect(messageWrapper.errorMessage.isDisplayed()).toBe(true);
  });
});
```

Since the error message shown in both tests is encapsulated in the same element, it is defined in a wrapper Page Object.

See the mentioned Page Object below (chokoMessagesWrapper.po.js):

```
// chokoMessagesWrapper.po.js

var MessageWrapper = function() {
  this.errorMessage = element(by.css('.alert-danger'));
};

module.exports = MessageWrapper;
```

Notice that this is a very simple Page Object, which just has on element exposed in a public way, but that may be used in different tests.

Just as additional examples, see also the Page Objects **chokoCreateAccount.po.js** and **chokoSignIn.po.js**:

```
// chokoCreateAccount.po.js

var chokoCreateAccountPage = function() {
  this.createAccountButton = element(by.id('element-create-account-submit'));
};

chokoCreateAccountPage.prototype.visit = function() {
  browser.get('http://choko.org/create-account');
};

module.exports = chokoCreateAccountPage;
```

```
// chokoSignIn.po.js

var ChokoSignInPage = function() {
  this.signInButton = element(by.id('element-sign-in-submit'));
};

ChokoSignInPage.prototype.visit = function() {
  browser.get('http://choko.org/sign-in');
};

module.exports = ChokoSignInPage;
```

As mentioned in the beginning of this chapter, all examples shown here are real examples, so, if you have interest, copy the code and test it yourself.

Helpers

Beyond the usage of Page Objects to encapsulate specific elements from the pages of the application in test, and for the creation of functions for operations that need more than one step, or even for functions that helps on readability, in some projects I also use to use helpers.

Helpers can exist for diverse reasons, and as the name already says, they exist to help. That is, we can create functions inside helpers, for example, to help us in certain operations that may be repeated over the tests, avoid code duplication and facilitating maintenance, besides helping in the test readability.

One case of helper I use to use is, for example, a helper for creating random strings that can be used in sign up forms, for example, where if an account was already created for a user, it would not be able to be used for the same test, in a future execution. In this case I can generate a random string that will be used as the user's name, ensuring that it will not be repeated in the next test. Another approach for the same issue can be the creation of a routine that clean all the data created during the tests. This routine can also be a helper.

See below an example of a helper to random strings creation, as mentioned:

```
// helper.js

var uuid = require('node-uuid');

var Helper = function() {};

Helper.prototype.generateRandomString = function() {
  return uuid.v4();
};

module.exports = Helper;
```

The helper shown here uses a node module called node-uuid. For now, just know that this is used for generating random strings. More details about this node module will be discussed in the next chapter, **Useful node modules**.

And now see an example of a test using this function **generateRandomString**, from the helper file:

```
// todoMvcWithPageObjects.spec.js

var Helper = require('../helper')
var TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Anguar', function() {
  var helper = new Helper();
  var todoMvc = new TodoMvc();

  it('add random value in the todo list', function() {
    var randomString = helper.generateRandomString();

    todoMvc.visit();

    todoMvc.addItemOnTodoList(randomString);

    expect(todoMvc.listOfItems.getText()).toContain(randomString);
  });
});
```

See that the helper is also required as a node module, in the top of the file, and then it is instantiated right after the describe.

The above shown test uses the helper to generate a random string, which is stored in a variable called **randomString**, then the test visits the TODO MVC application page, adds the generated random string to the TODO list, and finally verifies that the list contain the random string's text just added.

See that the helper is not a Page Object, since it has no relation with the application, it is just a helper, to be consumed by the tests.

See one more example:

```
it('try to sign in with a random email and random password', function() {
  var randomEmail = helper.generateRandomEmail();
  var randomPassword = helper.generateRandomString();

  signInPage.visit();

  signInPage.usernameField.sendKeys(randomEmail);
  signInPage.passwordField.sendKeys(randomPassword);
  signInPage.signInButton.click();

  expect(messageWrapper.errorMessage.isDisplayed()).toBe(true);
});
```

This new test simulates a user trying to sign in in an application as random user and with a random password.

Notice that to generate the random password, the same helper from the previous example is used (**generateRandomString**).

But for the random email generation a new helper is used.

See below the refactored helper.js file, now also containing the **generateRandomEmail** function:

```
// helper.js

var shortid = require('shortid');
var uuid = require('node-uuid');

var Helper = function() {};

Helper.prototype.generateRandomEmail = function() {
  return shortid.generate() + '@email.com';
};

Helper.prototype.generateRandomString = function() {
  return uuid.v4();
};

module.exports = Helper;
```

The **generateRandomEmail** function basically returns an id, concatenated with the string '@email.com', to ensure that it is a valid email format.

Notice that in this new function a new node module is been used (shortid). The same way as the node-uuid node module, this will also be seen in details in the next chapter, **Useful node modules**.

Taking advantage that the just shown test needs an operation that needs more than one step (fill the user, fill the password, and then click in the sign in button), see how it looks like after a refactoring, using a function for trying to login, which can be also used for a successful login, if the provided user and password are valid:

```
it('try to sign in with a random email and random password - refactored', functi\
on() {
  var randomEmail = helper.generateRandomEmail();
  var randomPassword = helper.generateRandomString();

  signInPage.visit();

  signInPage.signIn(randomEmail, randomPassword);

  expect(messageWrapper.errorMessage.isDisplayed()).toBe(true);
});
```

Obs.: This refactoring is not related to the usage of helpers, but since this is a good practice, I decided to show it here.

That is, now instead of the user fill the fields and click the button in different steps, all of them are encapsulated in the **signIn** function.

See the function below:

```
ChokoSignInPage.prototype.signIn = function(email, password) {
  this.usernameField.sendKeys(email);
  this.passwordField.sendKeys(password);
  this.signInButton.click();
};
```

Let’s now see a helper a little bit different those already seen.

Some time ago I wrote some automated tests that were failing even without something wrong in the application, causing false negative results.

As already mentioned in the **Good practices** chapter, test cases that results in false negatives are harmful to the team. Automated tests must provide real results about the application’s behavior in test, and must be trustable.

The case was that when it tried to click in such element, it was not available yet.

The fact was that in the accessed page, some elements were reflowed, causing this moment where the element that I needed to interact with, was behind another.

At first, webdriver should deal with these issues, but this is not always what happens, and in such cases we can use expected conditions (EC).

To solve such problem I created a helper:

See below the code of this helper (it is actually the same helper already shown, but with a new function, the last helper’s function).

```
// helper.js

var EC = protractor.ExpectedConditions;
var shortid = require('shortid');
var uuid = require('node-uuid');

var Helper = function() {};

Helper.prototype.generateRandomEmail = function() {
  return shortid.generate() + '@email.com';
};

Helper.prototype.generateRandomString = function() {
  return uuid.v4();
};

Helper.prototype.waitElementVisibility = function(element) {
  browser.wait(EC.visibilityOf(element), 3000);
};

module.exports = Helper;
```

That is, I created a helper that received an element as an argument and then waited until 3000 milliseconds (3 seconds) for the element being visible, only interacting with it when it was already available for this.

An interesting point about using this approach is that if the element is visible right in the first second, for example, then the test will not wait two more seconds to proceed with the next step, usually being interacting with such element, or an assertion.

In the below example, imagine that the usernameField is not available immediately right after visiting the Sign in page. In this case such helper comes in handy. That is, before interacting with such element, it waits for the element being visible.

```
it('try to sign in just filling the email field', function() {
  var randomEmail = helper.generateRandomEmail();

  signInPage.visit();
  helper.waitElementVisibility(signInPage.usernameField);

  signInPage.usernameField.sendKeys(randomEmail);
  signInPage.signInButton.click();

  expect(messageWrapper.errorMessage.isDisplayed()).toBe(true);
});
```

Furthermore, you can create helper for more complex issues, depending on your needs.

Useful node modules

One of the good parts of developing e2e automated tests with Protractor is that you can benefit with many existing node modules to supplement this activity, since Protractor is a framework based on Node.js.

In this chapter you will find a list of useful node modules which you can use to solve different issues when writing e2e automated tests.

But before going into details, if your project still don't have a file to manage the project dependencies, execute the following command to generate the **package.json** file and accept all the default options:

```
npm init
```

With this, always you will install a node module using NPM, the **package.json** file will be updated with the new dependencies.

jasmine-spec-reporter

jasmine-spec-reporter improves the feedback provided in your console.

This means that after running the tests, instead of see something like this:

```
[22:10:31] I/hosted - Using the selenium server at http://localhost:4444/wd/hub
[22:10:31] I/launcher - Running 1 instances of WebDriver
Started
...


3 specs, 0 failures
Finished in 4 seconds
[22:10:37] I/launcher - 0 instance(s) of WebDriver still running
[22:10:37] I/launcher - chrome #01 passed
```

Using the **jasmine-spec-reporter** node module you will have a more detailed feedback, like this:

```
[22:13:20] I/hosted - Using the selenium server at http://localhost:4444/wd/hub
[22:13:20] I/launcher - Running 1 instances of WebDriver
Spec started
Started

  1 Todo MVC Anguar
    ✓ add an item in the todo list (3 secs)
.     ✓ add new item in the todo list (0.958 sec)
.     ✓ add random value in the todo list (1 sec)
.
Executed 3 of 3 specs SUCCESS in 5 secs.


3 specs, 0 failures
Finished in 4.623 seconds
[22:13:26] I/launcher - 0 instance(s) of WebDriver still running
[22:13:26] I/launcher - chrome #01 passed
```

Using jasmine-spec-reporter

To start using **jasmine-spec-reporter** follow these steps:

Run the following command, from the project's root directory, to install **jasmine-spec-reporter** as a development dependency of your project:

```
npm install jasmine-spec-reporter --save-dev
```

Then update the Protractor configuration file adding the following to it:

Before the module.exports.config:

```
var SpecReporter = require('jasmine-spec-reporter');
```

And inside the module.exports.confg:

```
onPrepare: function() {
  jasmine.getEnv().addReporter(new SpecReporter({
    displayFailuresSummary: true,
    displayFailedSpec: true,
    displaySuiteNumber: true,
    displaySpecDuration: true
  }));
}
```

With the node module installed and the new configuration you will have a better feedback after running tests in your console, as already shown.

Some details about this configuration:

```
displayFailuresSummary: true, // display summary of all failures after execution
displayFailedSpec: true, // display each failed spec
displaySuiteNumber: true, // display each suite number (hierarchical)
displaySpecDuration: true // display each spec duration
```

All available configurations for the **jasmine-spec-reporter** node module can be found through the following URL: https://www.npmjs.com/package/jasmine-spec-reporter[5].

More details about the **onPrepare** configuration can be found in the **Advanced configurations** chapter.

---

protractor-jasmine2-html-reporter

protractor-jasmine2-html-reporter is an alternative for a test report generation after testing execution, in HTML format, with the ability of adding screenshots of each test on it.

Using protractor-jasmine2-html-reporter

To instal **protractor-jasmine2-html-reporter** as a development dependency of your project, run the following command:

```
npm install protractor-jasmine2-html-reporter --save-dev
```

After the node module installation, update the Protractor configuration as follows:

[5]https://www.npmjs.com/package/jasmine-spec-reporter

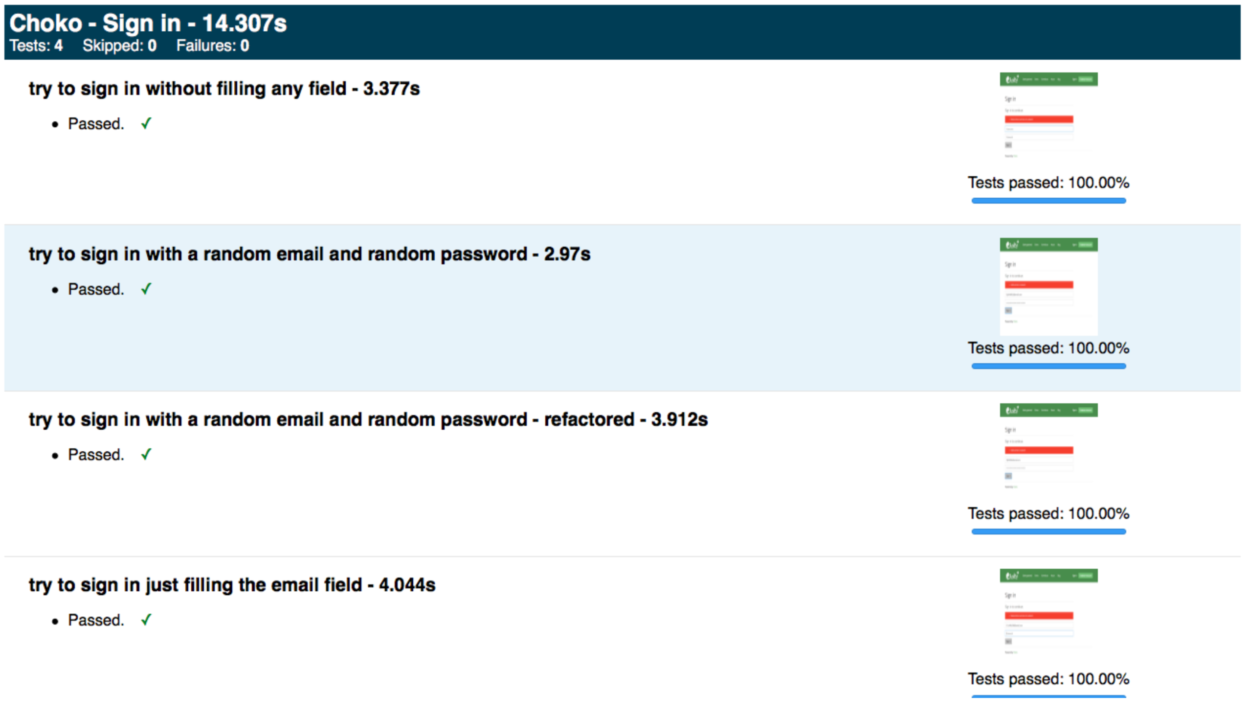
```
// protractor.conf.js

var Jasmine2HtmlReporter = require('protractor-jasmine2-html-reporter');
var SpecReporter = require('jasmine-spec-reporter');

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/',
  onPrepare: function() {
    jasmine.getEnv().addReporter(new SpecReporter({
      displayFailuresSummary: true,
      displayFailedSpec: true,
      displaySuiteNumber: true,
      displaySpecDuration: true
    }));

    jasmine.getEnv().addReporter(new Jasmine2HtmlReporter({
      takeScreenshots: true,
      fixedScreenshotName: true
    }));
  }
};
```

Pay attention to the line in the beginning of the file, where the **protractor-jasmine2-html-reporter** node module is required, and the final lines, where a new reporter is added, instantiating a **Jasmine2HtmlReporter** object, and defining two attributes to it: **takeScreenshots: true**, **fixedScreenshotName: true**, being the first to take screenshots after each test, and the second one to have fixed names for the screenshots, instead of random names, as default.

All available configurations for the **protractor-jasmine2-html-reporter** node module can be found through the following URL https://www.npmjs.com/package/protractor-jasmine2-html-reporter[6].

Run the tests with this new configuration, and then open the **htmlReport.html** generated, which will show you something like this:

[6]https://www.npmjs.com/package/protractor-jasmine2-html-reporter



Choko - Sign in - 14.307s
Tests: 4 Skipped: 0 Failures: 0

try to sign in without filling any field - 3.377s
- Passed. ✓

Tests passed: 100.00%

try to sign in with a random email and random password - 2.97s
- Passed. ✓

Tests passed: 100.00%

try to sign in with a random email and random password - refactored - 3.912s
- Passed. ✓

Tests passed: 100.00%

try to sign in just filling the email field - 4.044s
- Passed. ✓

Tests passed: 100.00%

html_report

With this configuration, for each new test execution, the report and screenshots will be overwritten.

---

shortid

Wit **shortid** it is possible to generated short and sequential IDs.

Using shortid

To install the **shortid** node module as a development dependency of your project, run the following command:

```
npm install shortid --save-dev
```

After the installation the node module will be ready to be used.

Below is an example of this node module, from the **helpers** chapter:

```
// helper.js

var shortid = require('shortid');

var Helper = function() {};

Helper.prototype.generateRandomEmail = function() {
  return shortid.generate() + '@email.com';
};

module.exports = Helper;
```

Notice that after requiring the node module, with only one line of code a short id can be generated:

```
shortid.generate();
```

Below is the list of the default characters used by **shortid**:

```
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-_'
```

More details about the **shortid** node module can be through the following URL: https://www.npmjs.com/package/sh

---

node-uuid

With **node-uuid** it is possible to create IDs based on timestamp or random IDs.

Using node-uuid

To install the **node-uuid** node module as a development dependency of your project, run the following command:

```
npm install node-uuid --save-dev
```

After the installation it will be ready to be used.

Below is an example, also from the **helpers** chapter:

[7] https://www.npmjs.com/package/shortid

```
// helper.js

var uuid = require('node-uuid');

var Helper = function() {};

Helper.prototype.generateRandomString = function() {
  return uuid.v4();
};

module.exports = Helper;
```

After requiring the **node-uuid** node module with a single line of code you can generate random IDs:

```
uuid.v4();
```

To generate IDs based on timestamp, use the following:

```
uuid.v1();
```

More details about the **node-uuid** node module can be found through the following URL: https://www.npmjs.com/pa uuid[8].

---

fs

The **fs** node module enables interacting with the file system.

Using fs

To install the **fs** node module as a development dependency of your project, run the following command:

```
npm install fs --save-dev
```

After the installation it will be ready to be used.

Following is an example of the usage of this node module, which I used in some projects to enable me overwriting some Protractor configurations based in a local configuration file (not tracked by git):

[8]https://www.npmjs.com/package/node-uuid

```
// protractor.conf.js

var fs = require('fs');

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/',
};

// Local config customization.
try {
  fs.statSync(__dirname + '/config.local.js');
  require(__dirname + '/config.local.js')(module.exports.config);
} catch(error) {
  return;
}
```

In the beginning the **fs** node module is required.

And after the configuration's definitions, a **try catch** checks for the existence of a file called **config.local.js**, in the same directory where the **protractor.conf.js** file is, and in case the file is found, it is required, overwriting the **protractor.conf.js** configurations from the configurations defined in this file. If it is not found, the **catch** simply executes a **return** statement.

See below the **config.local.js** file, which overwrites the **baseUrl** property by the value '**http://localhost:8000/**', and adds a new property **directConnect** with value **true**, which disregard the **seleniumAddress** property, using the browser's own WebDriver (in case of using Chrome or Firefox).

```
// config.local.js

/**
 * Configuration alter method.
 * @param {object} Current Protractor configuration, as defined in
 * protractor.conf.js file.
 */

module.exports = function (config) {
  config.baseUrl = 'http://localhost:8000/';
  config.directConnect = true;
};
```

More details about the **fs** node module can be found through the following URL: https://nodejs.org/api/fs.html[9].

---

browserstack-local

BrowserStack[10] is a SaaS (Software as a Service) which provides a combination of different operating systems, browsers and devices, for manual and automated testing.

Also, with this service it is possible to run tests against local development environment.

For this we use the **browserstack-local** node module.

That is, with this node module it is possible to run automated tests in browsers in the cloud, but against you local development environment.

More details about BrowserStack can be found in the **Cloud testing** chapter.

Using browserstack-local

To install the **browserstack-local** node module as a development dependency of you project, run the following command:

```
npm install browserstack-local --save-dev
```

With the **browserstack-local** node module installed, update the Protractor configuration file, requiring the **browserstack-local** node module and creating an instance of it. Also, add the following capabilities: rowserstack.user, browserstack.key and browserstack.local, with the following values: process.env.BROWSERSTACK_USERNAME, process.env.BROWSERSTACK_ACCESS_KEY and true, respectively, and then add the **beforeLaunch** and **onComplete** attributed as follow:

```
// protractor.conf.js

var browserstack = require('browserstack-local');
var browserstackLocal = new browserstack.Local();

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome',
    'browserstack.user': process.env.BROWSERSTACK_USERNAME,
    'browserstack.key': process.env.BROWSERSTACK_ACCESS_KEY,
```

[9] https://nodejs.org/api/fs.html
[10] https://www.browserstack.com/

```
    'browserstack.local': 'true'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://localhost:8080/',
  beforeLaunch: function() {
    return new Promise(function(resolve, reject) {
      const browserstackLocalArgs = {
        'key': process.env.BROWSERSTACK_ACCESS_KEY,
        'force': 'true',
        'forcelocal': 'true'
      };
      browserstackLocal.start(browserstackLocalArgs, function(error) {
        if (error) {
          reject(error);
          return;
        }
        resolve();
      });
    });
  },
  onComplete: function() {
    return new Promise(function(resolve, reject) {
      browserstackLocal.stop(function(error) {
        if (error) {
          reject(error);
          return;
        }
        resolve();
      });
    });
  }
};
```

This code depends on the definition of the following environment variables, which must contain the BrowserStack credentials from your account in the service:

- BROWSERSTACK_USERNAME
- BROWSERSTACK_ACCESS_KEY

The BrowserStack documentation for Node.js projects may be found in the following URL: https://www.browserstack.com/automate/node[11].

[11]https://www.browserstack.com/automate/node

Details about the **beforeLaunch** and **onComplete** functions are presented in the **Advanced configurations** chapter.

For now, just know that the **browserstack-local** is initiated before tests are started and it is finished when tests end, enabling tests to be run in your local environment (notice that the **baseUrl** is set to 'http://localhost:8080', to exemplify).

More details about the **browserstack-local** node module can be found through the following URL: https://www.npmjs.com/package/browserstack-local[12].

---

faker.js

Faker.js is a library to help you on generating fake data for usage in your tests.

With faker.js you may create fake data for:

- fake addresses
- fake data related to commerce, such as: prices or products
- fake data related to companies, such as company name or company suffixes
- fake dates
- fake data related to finance, such as amount or a currency code
- and many other kind of fake data, such as 'hacker', 'helpers', 'images', 'internet', 'lorem', 'name', 'phone', 'random', and 'system'

Using faker.js

To start using faker.js you need to install it on your project as a dev dependencies.

Use the below npm command to install faker.js as a dev dependency of you project:

```
npm install faker.js --save-dev
```

After installing faker.js, you need to require it inside your test files, to use it.

Below is the code you need to add in the top of your spec files, to create fake data for your tests:

```
var faker = require('faker');
```

And then you can start creating your fake data, such as below:

[12] https://www.npmjs.com/package/browserstack-local

```
var name: faker.name.findName(), // generate a random name
var email: faker.internet.email(), // generate a random email
var address: faker.address.streetName() + ', ' + faker.random.number(), // gener\
ate a random address and concatenate it with a coma and a random number
var city: faker.address.city(), // generate a random city
var country: faker.address.country() // generate a random country
```

And this are just a few examples of what kind of fake data you can create for using in your tests.

More details about the **faker.js** node module can be found through the following URL: https://github.com/marak/Fak

Also, if you want to see some examples of the library usage, take a look in the below git repository:

https://github.com/wlsf82/faker-experiments[14]

Various other node modules can be found from the following URL: https://www.npmjs.com/[15], so, if you have a specific need, remember to search for something that may be ready before implementing your own solution. Probably someone has already solved such problem, saving you time.

[13]https://github.com/marak/Faker.js/
[14]https://github.com/wlsf82/faker-experiments
[15]https://www.npmjs.com/

Actions and verifications

Actions and verifications are an important part of writing e2e automated tests, and are also part of the AAA standards (arrange, act and assert), mentioned in the introductory chapter, where actions are the A of act and the verifications are the A or assert.

The Protractor framework has a list of actions and verification that can be used when writing automated testing scripts, which will be discussed below.

Actions

Actions are intended to simulate users interacting with the application, such as, navigation, clicks, filling fields, clearing fields, among others.

Following are presented some common actions that can be performed using the Protractor framework.

browser.get(url); –> Navigation action

Navigate to a page through the URL.

Obs.: In case a **baseUrl** is defined in the configuration file, simply pass a relative URL as an argument to the **get()** function, which will be concatenated with the **baseUrl**, forming the absolute URL you want to navigate.

See some examples:

```
browser.get('https://leanpub.com/testes-e2e-com-protractor'); // navigation thro\
ugh absolute URL

browser.get('testes-e2e-com-protractor'); // navigation through relative URL. Im\
agine that the defined **baseUrl** is 'https://leanpub.com/'
```

element.click(); –> Click action

Clicks in a particular HTML element.

See some examples:

```
var button = element(by.id('submit'));
button.click(); // clicking in a button with id 'submit'

element(by.css('a .my-link')).click(); // clicking in a link with a css class 'm\
y-link'
```

element.sendKeys('text'); –> Filling a field action

The **sendKeys()** function is usually used for filling **input** HTML elements.

This action may be used to fill text fields, to simulate pressing keyboard keys, or even to send files to **file input** fields.

See some examples:

```
element(by.model('searchTerm')).sendKeys('restart'); // typing the value 'restar\
t' in a fields with model 'searchTerm'

var newTodoField = element(by.id('new-todo'));
newTodoField.sendKeys(protractor.Key.ENTER); // simulating pressing the ENTER ke\
y in a fields with id 'new-todo'

var element = element(by.css('input[type=file]'));
element.sendKeys('/path/to/file.txt'); // simulating sending a file to a **file*\
* **input** field.
```

element.clear(); –> Clearing field action

Sometimes, before typing in a field it is necessary to clear it. For doing this Protractor has the **clear()** function.

See an example:

```
var searchField = element(by.model('searchTerm'));
var text = 'restart';
searchTerm.sendKeys(text);
searchField.clear(); // Clearing an element defined by the model 'searchField' r\
ight after this field received a string value (a text)
expect(searchField.getText()).not.toContain(text)
```

browser.refresh(); –> Refresh screen action

In a particular test case you may need to simulate the browser page being refreshed. For doing this Protractor has the **refresh()** function.

An example could be to verify that a TODO list maintain the previous values filled, even after the browser screen is refreshed.

See an example:

```
var newTodoField = element(by.id('new-todo'));
var text = 'Create test without page object';

newTodoField.sendKeys(text);
newTodoField.sendKeys(protractor.Key.ENTER);
browser.refresh(); // refreshing the browser's page

expect(element.all(by.css('.view')).getText()).toContain(text);
```

Verifications

Verifications are what make automation really tests, because after a pre condition and one or more actions, we verify for an expected result. Therefore, verifications are an important part of writing e2e automated testing scripts.

Following are presented some common verifications that can be performed using the Protractor framework.

toEqual()

The equality verification may be used, for example, to check that the text of a particular HTML element is exactly the same text passed as argument.

See an example:

```
var foo = element(by.id('my-id'));
expect(foo.getText()).toEqual('some text'); // verification that an element loca\
ted by the id 'my-id' has exactly the text 'some text'
```

This verification may be also used together with the **count()** function, for example.

See some examples:

```
expect(element.all(by.css('.view')).count()).toEqual(1); // verification that an\
 array of elements identified by the css class 'view' contains only one element

expect(element.all(by.css('.view')).count()).toEqual(3); // verification that an\
 array of elements identified by the css class 'view' contains three elements
```

toContain()

The **toContain()** verification may be used to verify that a particular HTML element contains a particular text, enabling verifying only part of the text contained in such element.

See an example:

```
var foo = element(by.id('my-id'));
expect(foo.getText()).toContain('some text'); // verification that an element lo\
cated by the id 'my-id' contains the text 'some text'
```

toBe()

Verification usually used for returns of promises and boolean values.

See some examples:

```
expect(messageWrapper.errorMessage.isDisplayed()).toBe(true); // verification th\
at an error message is being displayed

expect(browser.getCurrentUrl()).toBe('https://leanpub.com/testes-e2e-com-protrac\
tor'); // verification that the return of the current URL promise is 'https://le\
anpub.com/testes-e2e-com-protractor'
```

not

Negation verification. The negation verification may be used together with any other of the already seen verifications, to deny such expectation.

See some examples:

```
var foo = element(by.id('my-id'));
expect(foo.getText()).not.toEqual('some text'); // verification that an element \
located by the id 'my-id' does not has the exact text 'some text'

var foo = element(by.id('my-id'));
expect(foo.getText()).not.toContain('some text'); // verification that an elemen\
t located by the id 'my-id' does not contain the value 'some text'

expect(messageWrapper.errorMessage.isDisplayed()).not.toBe(true); // verificatio\
n that an error message is not being displayed

expect(browser.getCurrentUrl()).not.toBe('https://leanpub.com/testes-e2e-com-pro\
tractor'); // verification that the return of the current URL promise is not 'ht\
tps://leanpub.com/testes-e2e-com-protractor'
```

Visual review testing

During the software development, even automating tests, still are necessary some manual verifications, like exploratory testing and tests to check issues related to style, like CSS changes, that maybe can break the app visually.

One alternative of integration for the Protractor framework, to help in such activity is the **VisualReview-protractor**[16], that is an API of **VisualReview**[17] that intends to offer a productive and friendly workflow for reviewing web app's layouts during regression testing.

The idea is the following: based on your already existing e2e regression tests, adding some steps for taking screenshots of each page, for later visual reviewing it, with the ease of a user friendly interface that helps on this workflow.

Visual Review works with a concept of screenshots previously reviewed, which are used as base for comparing with the screenshots taken at the moment that the e2e regression tests are executed. If some difference is found between the actual screenshot and the base one, then you can decide if it will pass or fail, with the help of a graphical user interface. When a screenshot with some difference is approved, then it is substituted by the old one, and then it starts being used as the base screenshot for the next test executions. If the screenshot is rejected, then the base screenshot is still valid and you probably found a visual issue, needing some action for solving it. If no difference is found between the base screenshot and the one taken during the e2e test execution, then the visual review test automatically passes, and there is not need for visual review.

Integrating VisualReview with Protractor

To install **VisualReview-protractor** as a development dependency of your project, run the following command:

```
npm install visualreview-protractor --save-dev
```

After the installation it is necessary to add some configurations. In the **protractor.conf.js** file, before the module.exports.config, add the following lines of code:

[16]https://github.com/xebia/VisualReview-protractor
[17]https://github.com/xebia/VisualReview

```
const VisualReview = require('visualreview-protractor');
var vr = new VisualReview ({
  hostname: 'localhost',
  port: 7000
});
```

Then, inside the module.exports.config, add the following configurations (the project and test suite's names may be changes as you need):

```
beforeLaunch: function () {
  vr.initRun('Visual-Review-Sample-Project', 'visualReviewSuite');
},

afterLaunch: function (exitCode) {
  return vr.cleanup(exitCode);
},

params: {
  visualreview: vr
}
```

With this configurations you just need to add two lines of code in your tests (.spec.js files) to make **Visual Review** works. Below is an example:

```
// sample.spec.js

var vr = browser.params.visualreview;

describe ('Sample', function () {
  it ('title is correct', function () {
    browser.get('/#');
    expect(element(by.id('title')).getText()).toEqual('Sample');
    vr.takeScreenshot('sample-home');
  });
});
```

In the beginning of the file a vr variable is defined with the value of the **visualreview** attribute, in the **params** configuration.

And inside the test, right after the verification (expect), a screenshot is taken and named.

In addition to the **VisualReview-protractor** installation, before running the tests it is necessary to download and uncompress the last version of **Visual Review**.

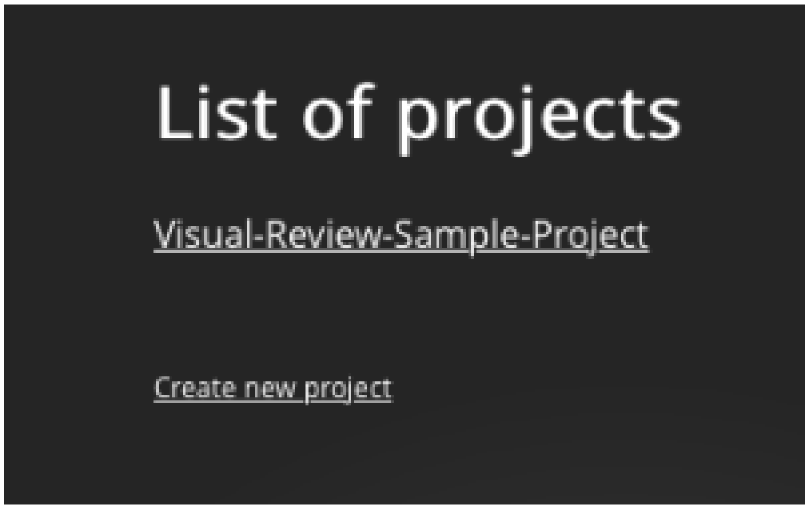
To find the last version of **Visual Review**, see the following URL: https://github.com/xebia/VisualReview/releases[18]

Then run the following command from the directory where **Visual Review** was uncompressed, to start it:

```
./start.sh
```

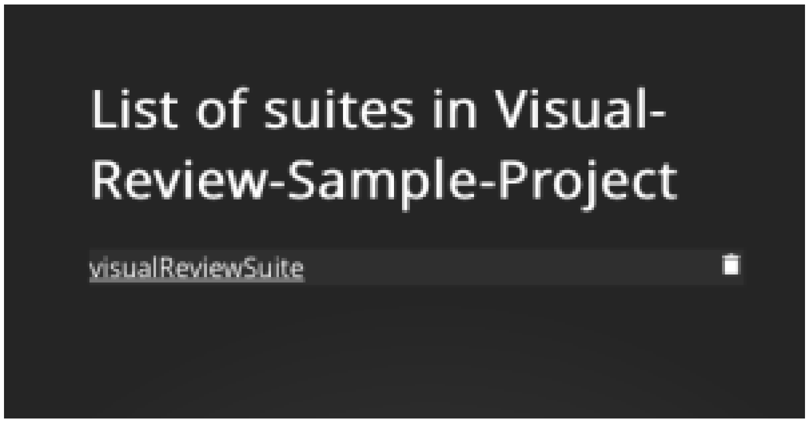
After that you just need to run Protractor to have your e2e tests integrated with **Visual Review**.

After the testing execution, accessing the address http://localhost:7000 through the browser, you will see something like this:



visualreview-screen1

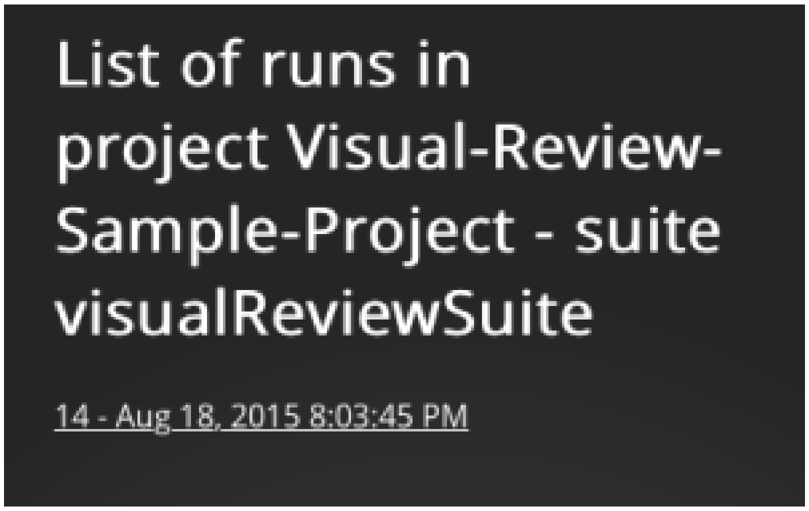
Accessing the link **Visual-Review-Sample-Project**, the following page is shown:



visualreview-screen2

[18] https://github.com/xebia/VisualReview/releases

Clicking in the **visualReviewSuite** link you will have access to the list of the last tests execution. Something like this:

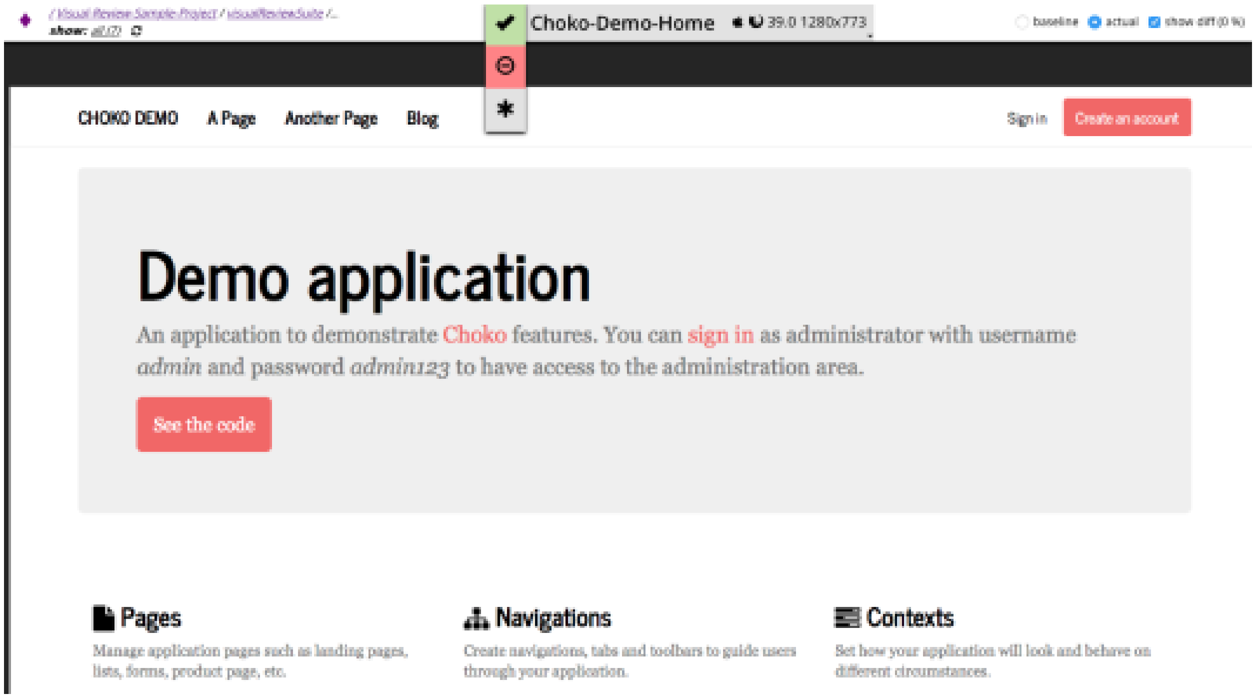


List of runs in project Visual-Review-Sample-Project - suite visualReviewSuite

14 - Aug 18, 2015 8:03:45 PM

visualreview-screen3

And finally, accessing the link form the last test execution you will have access to the graphical user interface to approve or reject the screenshots. Something like this:



Demo application

An application to demonstrate Choko features. You can sign in as administrator with username *admin* and password *admin123* to have access to the administration area.

Pages
Manage application pages such as landing pages, lists, forms, product page, etc.

Navigations
Create navigations, tabs and toolbars to guide users through your application.

Contexts
Set how your application will look and behave on different circumstances.

visualreview-screen4

In the first test execution, if all the screenshots are showing the application as expected, these should be approved, for being used as comparison base for the next test executions, and in the next executions, if some difference is found between the base screenshot and the one taken during the test execution, you will be able to approve or reject it, in the visual review moment.

With that, easily you can increment the test process of your software development project that already uses the framework Protractor, to ensure even more quality in the delivered applications.

What to test and what not to test with Visual Review

- Analyze your e2e tests before adding visual review steps to them.
- Try to understand which tests are more suitable for this approach. Visual review testing exists to save you time, so, if in all your tests executions there are visual differences, maybe you're not choosing the right tests cases for it.
- If the layout of a certain screen is still in an experimentation phase, do not use visual review testing.
- Add visual review steps for already stable pages. In other words, pages that are not planned to change in a near future.
- Add visual review steps to pages that don't have dynamic elements, such as advertising, date and time, and so on.
- Also, it is not recommended using this kind of testing in pages that have animated gifs and auto executable videos, since it will show visual differences for each test execution, even when the application is not visually "broken".

As a additional source of reading, I recommend you the following web post: https://medium.com/@walmyrlimaesilv read-property-platform-of-undefined-8848acc8bb3#.722l41ea5[19]

[19]https://medium.com/@walmyrlimaesilv/cannot-read-property-platform-of-undefined-8848acc8bb3#.722l41ea5

Cloud testing

What we can do when it is necessary to test the application in different browsers? Or even in different combinations of browsers and operation systems? And what if we need to test the same application in browsers and mobile devices?

One of the advantages of e2e testing is the ability to run test suites in different combinations of operation systems, browsers and devices.

In our own computers can have more than one browser to run tests, or we can even have virtual machines with different operation systems, but there are services that are specialized in offering a variety of these combinations, including old versions of browsers, for cases when the developed application needs to attend specific needs.

Also, using such service together with continuous integration tools decreases the feedback loop after application changes.

Some of these services for cloud testing are BrowserStack[20] and SauceLabs[21], both available for integration with Protractor.

Let's see each of them:

BrowserStack

BrowserStack is a cloud testing service based on **Selenium** and it offers more than 1000 combinations of different operation systems, browsers and devices. It is a paid service, but offers a trial version which includes 100 minutes for running automated tests.

Integrating Protractor test scripts with BrowserStack

To enable running tests written with the Protractor framework on BrowserStack it is necessary to make some changes to the Protractor configuration file.

See an example:

[20]http://browserstack.com/
[21]http://saucelabs.com/

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://hub.browserstack.com/wd/hub',
  capabilities: {
    'browserName': 'chrome',
    'browser_version': '52',
    'os': 'OS X',
    'os_version': 'El Capitan',
    'browserstack.user': process.env.BROWSERSTACK_USERNAME,
    'browserstack.key': process.env.BROWSERSTACK_ACCESS_KEY
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

Basically, what needs to be changed is the seleniumAddress, which now receives the BrowserStack's Selenium address, and the capabilities, which beyond defining the browser version, operation system and its version, also received two new attributes ('browserstack.user' and 'browserstack.key').

Obs.: See that the BrowserStack's credential values ('browserstack.user' and 'browserstack.key') point to the environment variables BROWSERSTACK_USERNAME and BROWSERSTACK_ACCESS_KEY. This is done this way so such sensitive data is not hard coded in the configuration file, so, to make it work it is necessary that these environment variables are created with the correct values (user and key of your BrowserStack account[22]).

With that, when running the Protractor tests they will be executed in a Chrome browser in the BrowserStack's cloud.

You can also run the same tests using the **multiCapabilities** configuration, so trigger tests in different combinations of browser, operation systems and devices.

See an example:

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://hub.browserstack.com/wd/hub',
  browserstack.user: process.env.BROWSERSTACK_USERNAME,
  browserstack.key: process.env.BROWSERSTACK_ACCESS_KEY,
  multiCapabilities: [
    {
      'browserName': 'chrome',
```

[22]https://www.browserstack.com/accounts/settings

```
      'browser_version': '52.0',
      'os': 'OS X',
      'os_version': 'El Capitan',
    },
    {
       browserName: 'firefox',
       browser_version: '47.0',
       os: 'Windows',
       os_version: '7'
    },
  ]
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

In this example, when running the tests, these will execute in parallel in a OS X El Capitan in a Chrome browser version 52, and in a Windows 7 in a Firefox browser version 47, in the BrowserStack's cloud.

Nevertheless, as shown in the **Useful node modules** chapter, it is possible to run tests in the BrowserStack's cloud against a development local environment, through a tunnel, using the **browserstack-local** node module.

SauceLabs

SauceLabs is another cloud testing service also based on **Selenium** and offers more than 700 combinations of different operation systems, browsers and devices. It is a paid service and has a trial version for two weeks, 90 minutes for automated test execution and up to 8 test suites running in parallel.

Integrating Protractor test scripts with SauceLabs

To enable running tests written with the Protractor framework on SauceLabs it is necessary to make some changes to the Protractor configuration file.

See an example:

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://ondemand.saucelabs.com:80/wd/hub',
  username: process.env.SAUCELABS_USERNAME,
  accessKey: process.env.SAUCELABS_ACCESS_KEY,
  multiCapabilities: [
    {
      'browserName': 'chrome',
      'browser_version': '52.0',
      'os': 'OS X',
      'os_version': 'El Capitan',
    },
    {
       browserName: 'firefox',
       browser_version: '47.0',
       os: 'Windows',
       os_version: '7'
    },
  ]
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

As with BrowserStack, what needs to be adapted in the protractor.conf.js file is the seleniumAddress, which points to the SauceLabs' Selenium server, and the credentials access, which are also obtained through environment variables, since these are sensitive data. Also, it is possible to execute tests with different capabilities to run the same test suite using different combinations of operation systems, browsers and devices.

SauceLabs has the ability to run tests in local development environment as well, using Sauce Connect.

Details about Sauce Connect can be found through the following URL: https://wiki.saucelabs.com/display/DOCS/Set

---

Both services (BrowserStack and SauceLabs) support different programming languages, including **JavaScript**, the programming language used by Protractor. These services can also be integrated with continuous integration tools such as **Jenkins**, **SemaphoreCI**, **Travis CI**, etc. And they also have support service for helping on problems resolutions.

[23] https://wiki.saucelabs.com/display/DOCS/Setting+Up+Sauce+Connect

Some interesting features available in both services are the screenshots taken during test execution (that may be used as evidence of tests), and videos recorded during test execution, which can be reproduced later, to help identifying issues when tests fail.

Continuous integration

Continuous integration, or just CI, is a software development practice of integrating code continuously (at least once a day—per developer), and in an automated way.

Also, it is about verifying if the new code you just wrote broke or not the code that was already working, since the automated tests and other tasks (like syntax verification) are executed after the new code is integrated (with the new build or within the repository).

Using this approach allows software development teams to have a very fast feedback loop about the changes they are doing in an specific application, and this is a cheaper way of solving issues when they are found, because the newly changed code is still fresh on the developers' mind.

This is also one of the practices from the eXtreme programming discipline, created by Kent Beck and Ron Jeffries, in 1997. After using CI (and other XP practices) and having good success in the projects they were working, they decided to write about it, as a way of sharing this knowledge with the world, so we could have better software. Software that matters!

One of the important points of using CI is about having less conflicts when integrating code. Once the code is frequently merged (from an specific branch, for example, to the trunk branch), it has less chances of breaking what already exists. And even if it breaks what was already working, it is easier to solve.

Another very important aspect when talking about CI is that it needs to be supported by a suite of automated tests (not only unit tests, but also by integration tests, and even better, if possible, by end-to-end tests).

E2e testing in the continuous integration process

With the help of tools for conducting cloud testing it is possible to add e2e tests in the process of continuous integration, along with the use of a specific tool for this, such as **SemaphoreCI**[24], which will be shown below.

Integrating e2e tests written using the Protractor framework with SemaphoreCI

SemaphoreCI is a continuous integration tool to streamline the software development cycles and enable a scalable workflow.

[24] http://semaphoreci.com/

With **SemaphoreCI** it is possible to test your code after each change, perform deploy quickly and securely, and it is easily customizable for different needs.

To integrate e2e tests written using Protractor with SemaphoreCI to run tests in the SauceLabs cloud, for example, the following changes are needed in the Protractor configuration file.

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://ondemand.saucelabs.com:80/wd/hub',
  username: process.env.SAUCELABS_USERNAME,
  accessKey: process.env.SAUCELABS_ACCESS_KEY,
  capabilities: {
      'browserName': 'chrome',
      'name': 'SemaphoreCI, SauceLabs and Protractor test suite'
  }
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

See that the needed changes are basically the same already mentioned in the **Cloud testing** chapter, with one extra attribute for the test suite name.

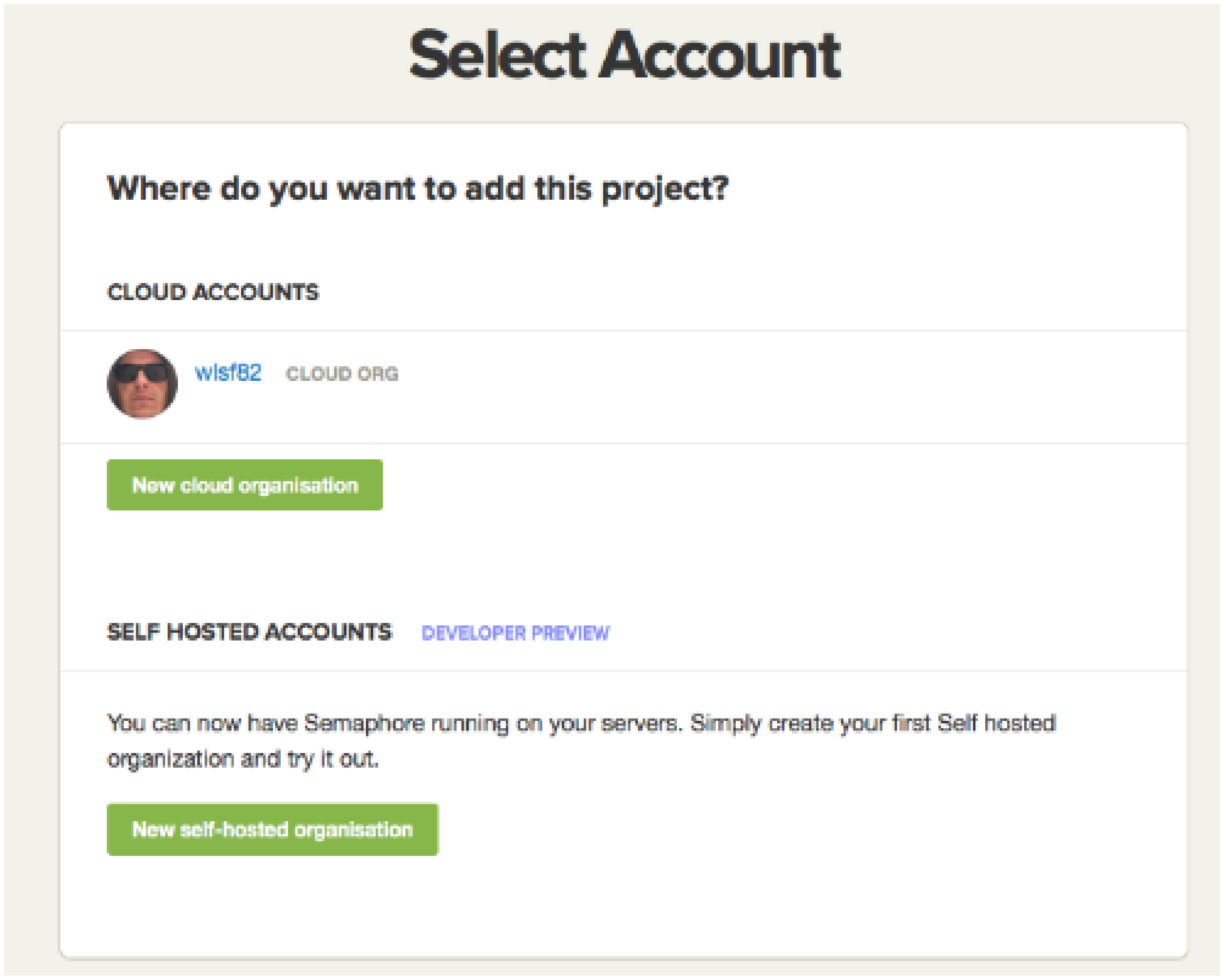
The steps to start the integration are:

1 - Create an account on **SemaphoreCI** through the following URL: http://semaphoreci.com/[25].

2 - Create a project:

In case your project uses **Github** for versioning the code, after creating your **SemaphoreCI** account it is possible to create a new project through your **Github** account. See the image below:

[25]http://semaphoreci.com/


Select Account

Where do you want to add this project?

CLOUD ACCOUNTS

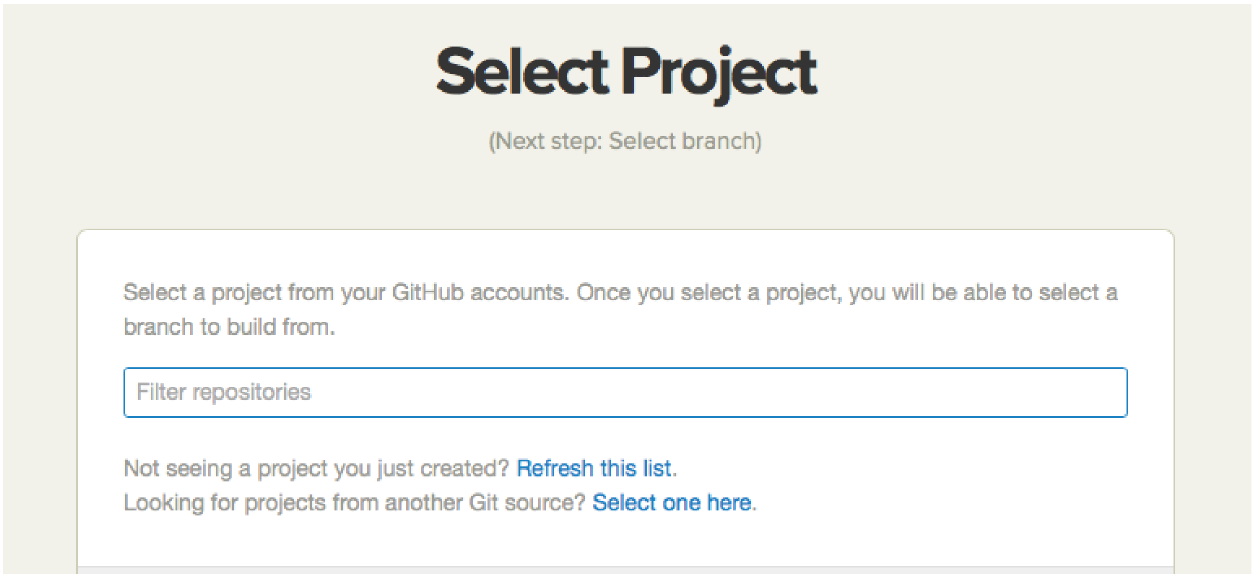
wlsf82 CLOUD ORG

SELF HOSTED ACCOUNTS DEVELOPER PREVIEW

You can now have Semaphore running on your servers. Simply create your first Self hosted organization and try it out.

semaphoreci-screen1

Clicking in the **Github** user link you will be directed to the next screen to search for the project that you want to integrate with **SemaphoreCI**. See:



Select Project

(Next step: Select branch)

Select a project from your GitHub accounts. Once you select a project, you will be able to select a branch to build from.

Filter repositories

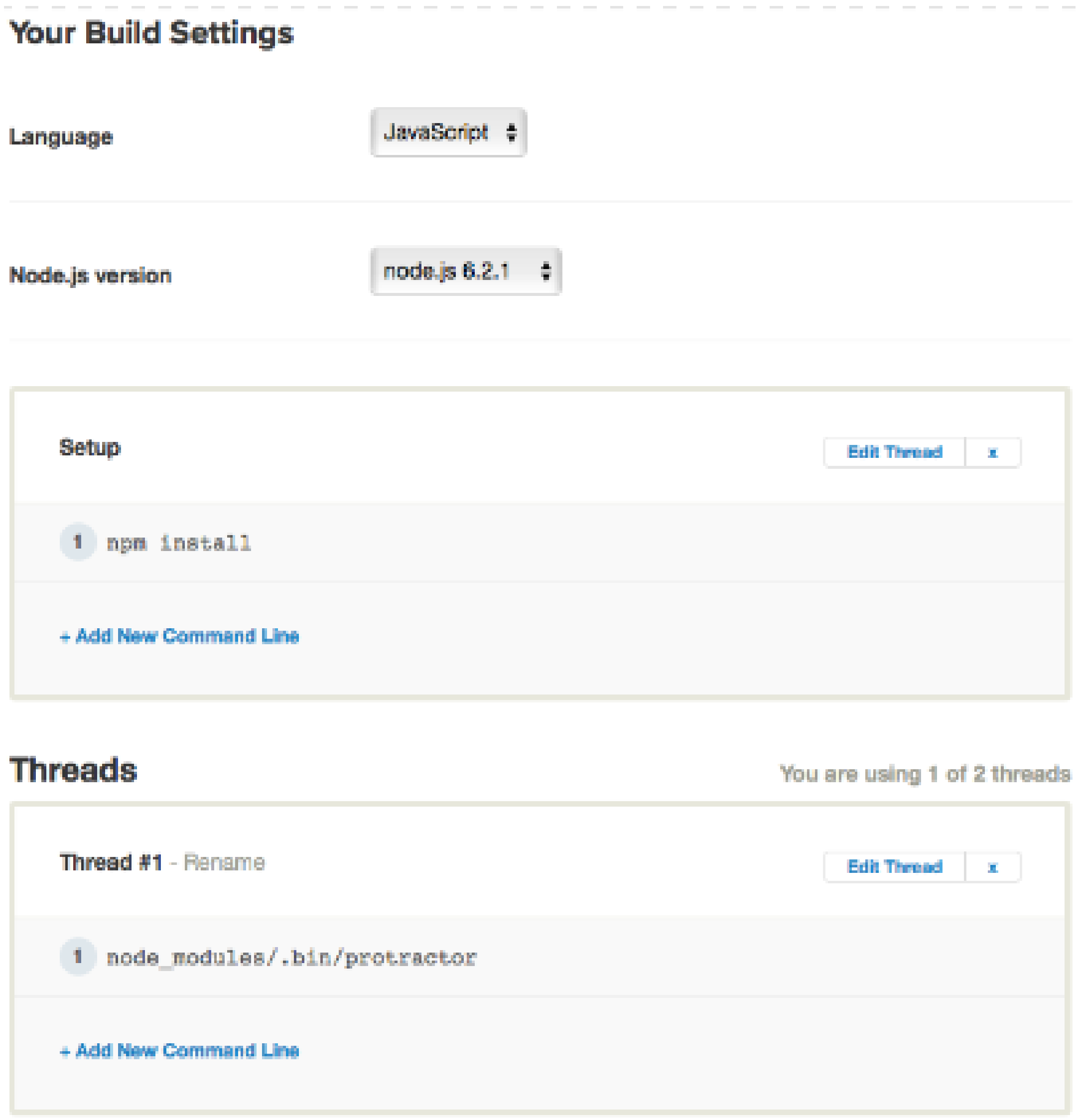
Not seeing a project you just created? Refresh this list.
Looking for projects from another Git source? Select one here.

semaphoreci-screen2

Search and select the project.

3 - Configure the build:

After searching and selecting the project it is necessary to define the build configurations. See:


Your Build Settings

Language: JavaScript

Node.js version: node.js 6.2.1

Setup

```
1 npm install
```

Threads — You are using 1 of 2 threads

Thread #1

```
1 node_modules/.bin/protractor
```

semaphoreci-screen3

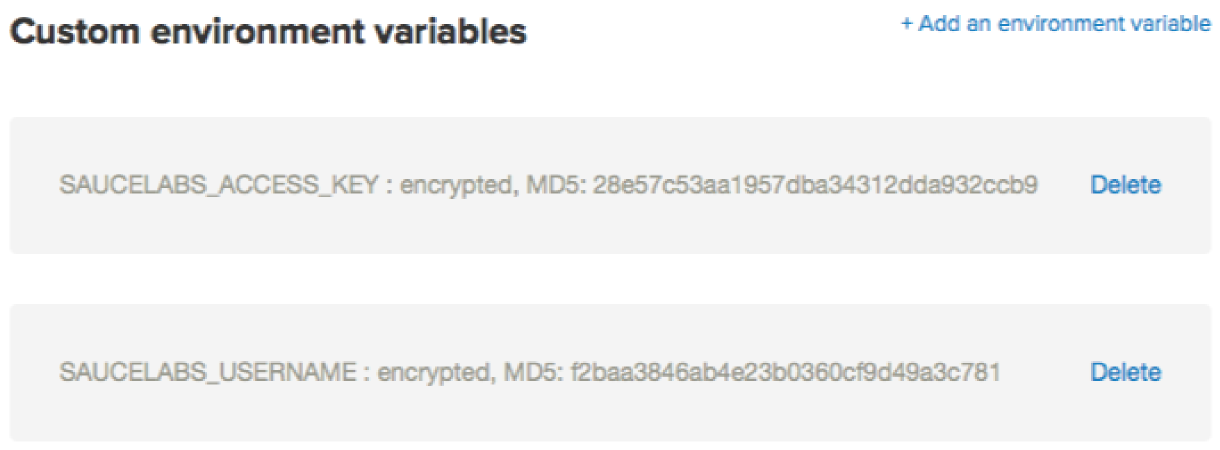
Since Protractor is Node.js based, the selected language should be JavaScript, and it is also necessary to select a Node.js version.

In the build configurations, in the setup the **npm install** command is filled, to install the project dependencies.

And in the thread the Protractor command is filled, from the installed binary in the node modules directory.

4 - Define the environment variables:

Since the **SauceLabs** access credentials are sensitive data, it is possible to store them into **SemaphoreCI** environment variables, encrypted. See:



SAUCELABS_ACCESS_KEY : encrypted, MD5: 28e57c53aa1957dba34312dda932ccb9

SAUCELABS_USERNAME : encrypted, MD5: f2baa3846ab4e23b0360cf9d49a3c781

semaphoreci-screen4

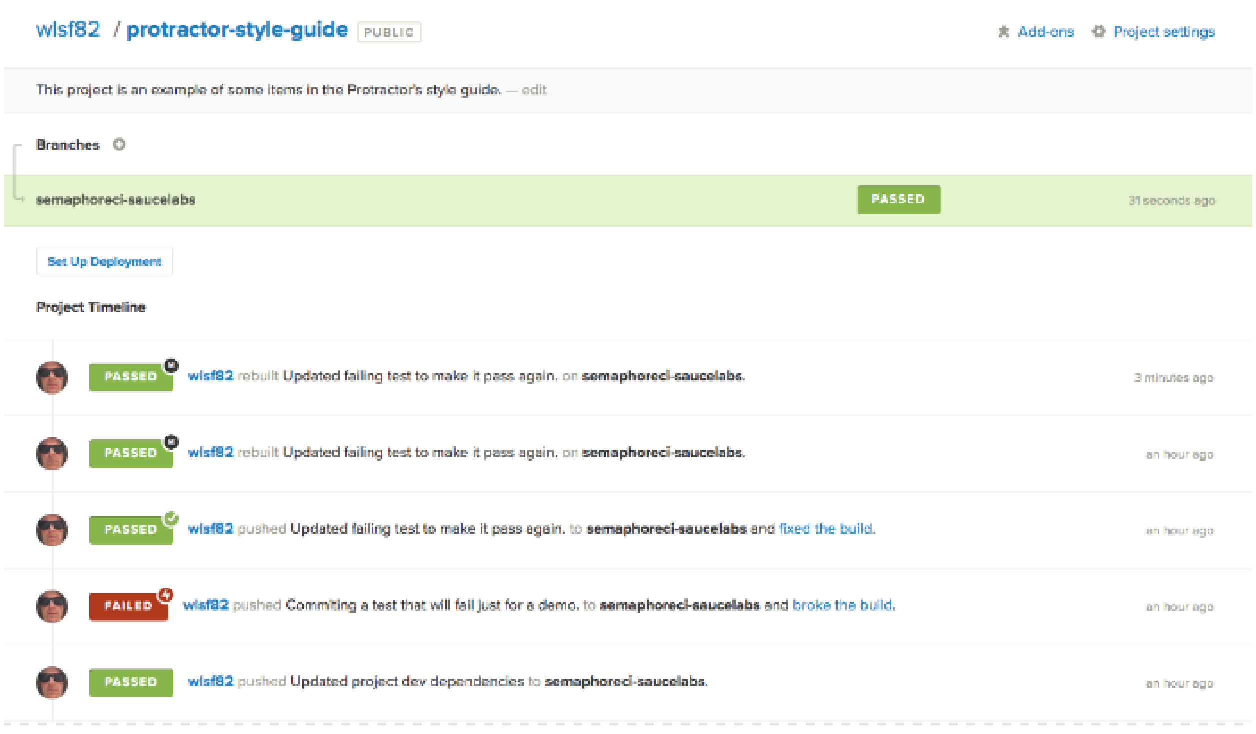
It is also possible to define different tasks to be run for different repository branches and to define email or chat notifications for when the tests fail, for example.

More information about **SemaphoreCI** may be obtained through the following URL: https://semaphoreci.com/docs/j continuous-integration.html[26].

With these configurations, after a git push in the configured branch to trigger the tests by **SemaphoreCI**, this will automatically run the tests in the **SauceLabs** cloud (in this example) against the application defined by the **baseUrl** in the Protractor configuration file, in this case the **Protractor** official web site.

See below an example of the **SemaphoreCI** dashboard after some build executions, which resulted in green build (the tests have passed), a red build (the tests have failed). Also, it is possible to identify builds that were automatically triggered (through a git push in the configured branch) or manually triggered (also available through the **SemaphoreCI** interface).

[26]https://semaphoreci.com/docs/javascript-continuous-integration.html


wlsf82 / protractor-style-guide PUBLIC

This project is an example of some items in the Protractor's style guide. — edit

Branches

semaphoreci-saucelabs PASSED 31 seconds ago

Project Timeline

PASSED wlsf82 rebuilt Updated failing test to make it pass again. on semaphoreci-saucelabs. 3 minutes ago

PASSED wlsf82 rebuilt Updated failing test to make it pass again. on semaphoreci-saucelabs. an hour ago

PASSED wlsf82 pushed Updated failing test to make it pass again. to semaphoreci-saucelabs and fixed the build. an hour ago

FAILED wlsf82 pushed Commiting a test that will fail just for a demo. to semaphoreci-saucelabs and broke the build. an hour ago

PASSED wlsf82 pushed Updated project dev dependencies to semaphoreci-saucelabs. an hour ago

semaphoreci-screen5

In conclusion, with just some steps it is possible to add e2e tests written using the **Protractor** framework to be run in the cloud after each moment the code is changed and sent to a remote repository, with help of and CI tool, what helps for a quick feedback to the development team, specially when application changes broke something that was working, enabling the team to act quickly to resolve the problem, making a hotfix, or even rolling back the application to the previous version.

Mobile testing

When talking about software test automation nowadays, besides thinking on testing web applications that run in conventional computers or notebooks, we also need to think on testing for mobile devices, since now they are so used or even more than conventional computers.

What to do when we need to test an application in mobile devices, such as a smart phone?

In this chapter will be shown some alternatives for testing applications for mobile devices.

Redefining the browser's size

One way to test a responsive application, which in the same version is differently presented depending on the browser's dimension, is to resize the browser's window.

When resizing a browser to 320 per 568, for example, it is possible to simulate the dimensions of an iPhone 5.

See below how to resize the browser's window to simulate the dimensions of a mobile device:

```
browser.driver.manage().window().setSize(320, 568);
```

This may be a valid approach when talking about responsive applications.

Simulating a mobile device into the browser

While creating test scripts, sometimes it is not good enough just to resize the browser's window to simulate a mobile device. This approach may not work, for example, if the application that you are testing has a specific version for mobile, different from a just responsive application.

In cases like this, one way that may be used it to simulate a mobile device through the capabilities configuration. See an example:

```
capabilities: {
  'browserName': 'chrome',
  'chromeOptions': {
    'mobileEmulation': {
      'deviceName': 'Google Nexus 5'
    }
  }
}
```

Using this configuration, tests will be run in a Chrome browser, but using a Google Nexus 5 device simulator, available through the the own browser.

Using mobile devices simulators in the cloud

Another way to test applications on mobile devices is through the resources available on cloud testing services, such as **BrowserStack** or **SauceLabs**, services that were already mentioned in the **Cloud testing** chapter.

To create mobile tests we can use a web server called Appium[27], which is based on **Selenium**, being able to drive mobile devices. **Appium** is a test automation framework for use with native, hybrid and mobile web apps, and it drives iOS and Android apps using the WebDriver protocol.

See below a configuration example to run tests on **SauceLabs** simulating an iPhone 4s:

```
module.exports.config = {
  seleniumAddress: 'http://ondemand.saucelabs.com:80/wd/hub',
  specs: ['*.spec.js'],
  capabilities: {
    browserName: 'safari',
    'appium-version': '1.5',
    platformName: 'iOS',
    platformVersion: '9.2',
    deviceName: 'iPhone 4s',

    username: process.env.SAUCELABS_USERNAME,
    accessKey: process.env.SAUCELABS_ACCESS_KEY,

    'name': 'iOS simulation'
  },
  baseUrl: 'http://appium.io/'
};
```

[27]http://appium.io

Notice that besides the seleniumAddress that is pointing to the **SauceLabs** server and the capabilities already seen in the **Cloud testing** chapter, now there is also a property called 'appium-version', with value '1.5'. This is the only needed change, which will allow the test execution on mobile devices.

Obs.: As already mentioned in other chapters, the access credentials from the cloud service testing are passed through environment variables, since these are sensitive data, so, they need to be defined with the right values into the server that will run the tests.

See one more example, now simulating an Android mobile phone:

```
module.exports.config = {
  seleniumAddress: 'http://ondemand.saucelabs.com:80/wd/hub',
  specs: ['*.spec.js'],
  capabilities: {
    browserName: 'chrome',
    'appium-version': '1.5',
    platformName: 'Android',
    platformVersion: '4.4',
    deviceName: 'Samsung Galaxy S4 Emulator',

    username: process.env.SAUCELABS_USERNAME,
    accessKey: process.env.SAUCELABS_ACCESS_KEY,

    'name': 'Android simulation'
  },
  baseUrl: 'http://appium.io/'
};
```

There are many ways to run tests for mobile applications, you just need to understand which one is the best for your need.

For more information about mobile testing, see the Protractor official documentation, through the following URL: http://www.protractortest.org/#/mobile-setup[28]

[28]http://www.protractortest.org/#/mobile-setup

ECMAScript 2015

ECMAScript is the JavaScript base programming language, the same language used for writing automated test scripts with the **Protractor** framework, but until now the shown code in this book was written using the version 5 of ECMAScript (ES5).

In June of 2015 ECMAScript version 6 was launched, named as ECMAScript 2015, which will be referred in this chapter as **ES2015**.

Some of the goals for **ES2015** include providing better support for large applications, library creation, and for use of ECMAScript as a compilation target for other languages. Some of its major enhancements include modules, class declarations, lexical block scoping, iterators and generators, promises for asynchronous programming, destructuring patterns, and proper tail calls.

The majority of utilities of **ES2015** are supported by Node.js version 6, and since **Protractor** is Node.js based, it is possible to take advantage of these changes to write more readable code.

See some changes when writing **Protractor** tests with **ES2015**.

Configuration file in ES2015

Below is an example of a Protractor configuration file, using a good practice recommended when using **ES2015**.

```
// protractor.conf.js ES2015

'use strict';

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

In this example of a **Protractor** configuration file, the only change is the use of '**use strict**', which is used to ensure that JavaScript is executed only in strict mode, so that, for example, variables cannot be used if they are not defined.

See an example:

```
'use strict';

a = 123; // This causes an error saying that 'a' is not defined.
```

It is also possible to use the strict mode in ES5, but when talking about **ES2015**, the use of such mode is considered a good practice.

See another configuration file, now also using a **onPrepare** function, defined in a different way when in **ES2015**, when compared to ES5:

```
// protractor.conf.js ES2015

'use strict';

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
  onPrepare() {
    browser.driver.manage().window().maximize();
  },
};
```

The **onPrepare** function used in this example maximizes the browser's window before the tests start, and in **ES2015** it is defined as shown is this example. Take a look at how to define the same function in ES5:

```
// onPrepare function defined using  ES5

onPrepare: function() {
  browser.driver.manage().window().maximize();
}
```

Notice that in **ES2015** the **onPrepare** function definition is simplified, not needing the attribute definition, which has a function (*function() {}*) as value of it.

More about the **onPrepare** function may be seen in the **Advanced configurations** chapter.

Test files (spec files) in ES2015

See below an example of a test file using **ES2015**:

```
'use strict';

const Helper = require('../helper');
const TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Angular', () => {
  const helper = new Helper();
  const todoMvc = new TodoMvc();

  it('add random value in the todo list', () => {
    const randomString = helper.generateRandomString();

    todoMvc.visit();

    todoMvc.addItemOnTodoList(randomString);

    expect(todoMvc.listOfItems.getText()).toContain(randomString);
  });
});
```

The first change when comparing with the already seen codes in other chapter is again the 'use strict', already explained.

The second change now is that the helper, the page object, its dependencies and a random string are defined using **const**, instead of **var**.

In **ES2015**, **const** is used for defining values that cannot be changed during the rest of the code in the same scope. See some examples to ease the understanding:

```
'use strict';

const a = 'text'

a = 'another text' // this is not allowed and causes a compilation error
```

In **ES2015**, for values that need to be changed along the code, differently of ES5, where we use **var**, we use **let**. See:

```
'use strict';

let a = 'text'

a = 'another text' // change allowed!
```

The third and last change is in the **it** function's callback, which in **ES2015** uses what is called array functions. See below:

```
'use strict';

it('add random value in the todo list', () => {});
```

The same code in ES5 would be this way:

```
it('add random value in the todo list', function() {});
```

In other words, it is necessary to write less code for the callback definition. In **ES2015** no named function are defined as array functions.

The same is valid for no named functions with arguments. See an example below:

```
'use strict';

const sayHello = (name) => {
    console.log('Hello ' + name + '!');
}

sayHello('John'); // Prints 'Hello John!' in the console
```

Page Objects and *Helpers* in ES2015

The most notable change when writing **Protractor** tests using **ES2015**, in my opinion, is in writing Page Objects or Helpers. See some examples:

Helper file:

```
// helper.js

'use strict';

const shortid = require('shortid');
const uuid = require('node-uuid');

class Helper {
  generateRandomEmail() {
    return shortid.generate() + '@email.com';
  }

  generateRandomString() {
    return uuid.v4();
  }
}

module.exports = Helper;
```

Notice that:

- The helper uses the strict mode good practice
- The helper defines node modules using **const**
- The helper is defined as a class, instead of a simple function
- The helper has methods, instead of defining functions by its prototype function. In this case the methods are:**generateRandomEmail** and **generateRandomString**.

Page Objects file:

```
// todoMvc.po.js

'use strict';

class TodoMvc {
  constructor() {
    this.listOfItems = element.all(by.css('.view'));
    this.newTodoField = element(by.id('new-todo'));
  }

  addItemOnTodoList(item) {
    this.newTodoField.sendKeys(item);
    this.newTodoField.sendKeys(protractor.Key.ENTER);
  }

  visit() {
    browser.get('http://todomvc.com/examples/angularjs/#/');
  }
}

module.exports = TodoMvc;
```

In the Page Object, notice that:

- The Page Object uses the strict mode good practice
- The Page Object is defined as a class, instead of a simple function
- The Page Object defines a constructor for the public elements definition
- The Page Object has methods, in this case the methods **addItemOnTodoList** and **visit**.

Using **ES2015** for writing Page Objects and Helpers, in my opinion, makes the code more readable, facilitating even more the test maintenance, any time it is necessary.

In this link[29] there is a sample project with tests written using the **Protractor** framework and **ES2015**.

For more information about **ES2015** I recommend reading the official documentation, through the following URL: http://www.ecma-international.org/ecma-262/6.0/[30]

[29]https://github.com/wlsf82/protractor-es6

[30]http://www.ecma-international.org/ecma-262/6.0/

Advanced configurations

In the introductory chapter we saw the basic configurations to execute automated tests with the **Protractor** framework, however, many other configurations may be done, enabling other facilities. Some of this configurations enable: using the browser's own webdriver, defining different frameworks (such as Jasmine and Mocha), running tests in parallel, defining test suites, and so on. In this chapter we will se these other configurations.

directConnect - Using the browser's own webdriver

The **directConnect** configuration may be used with Firefox or Chrome browsers.

This configuration disregard the need for a running Selenium server, once using the already mentioned browsers it is possible to use their own webdriver.

See an example:

```
// protractor.conf.js

module.exports.config = {
  directConnect: true,
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

By default the **directConnect** configuration has a **false** value, but when defined with a **true** value, the **seleniumAddress** configuration is ignored, or as it is shown in this example, it is not even necessary. In this case **Protractor** uses the Chrome's webdriver to run the tests.

This is a useful configuration when running tests in a local development environment.

---

framework - Defining a base framework to write tests

With the **Protractor** framework it is possible to define which base framework will be used to write the test scripts. Jasmine version 2.x is the default base framework.

Jasmine

Jasmine is a behavior-driven development framework for testing JavaScript code with simple and easy to understand syntax.

Furthermore, as already mentioned, **Jasmine** is the default base framework for the **Protractor** framework, so, if you have read the book until here, you are already used with the **Jasmine** syntax.

Updating Jasmine from version 1.x to 2.x

In the **Protractor** older versions the **Protractor**'s default base framework was **Jasmine** version 1.3, however, even using an older version of **Protractor**, it is possible to use an actual version of **Jasmine**.

One of the advantages of using **Jasmine** version 2.x is the possibility of using the **beforeAll** and **afterAll** functions, not available in the version 1.x. These configurations may be used to define pre and post conditions for a test suite.

To use **Jasmine** version 2.x with an older version of **Protractor**, define the framework as shown below:

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/',
  framework: 'jasmine2'
};
```

Note: Although it is possible to use **Jasmine** version 2.x in an older version of **Protractor**, the recommendation is to use the **Protractor** latest stable version, in which **Jasmine** version 2.x is already default.

More information about the **Jasmine** framework can be found in the tool's official website, through the following URL: http://jasmine.github.io[31]

Mocha

Mocha is a feature-rich unit testing framework for JavaScript code that simplify writing asynchronous tests.

[31]http://jasmine.github.io

To use the **Protractor** framework with **Mocha** as the base framework the following node modules need to be installed: mocha, chai and chai-as-promised.

It is recommended installing such modules as development dependencies, so, use the bellow commands:

```
npm install mocha --save-dev
npm install chai --save-dev
npm install chai-as-promised --save-dev
```

After installing the dependencies, to use the **Mocha** framework this needs to be defined in the **Protractor** configuration file, as shown below:

```
// protractor.conf.js

module.exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  capabilities: {
    'browserName': 'chrome'
  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/',
  framework: 'mocha'
};
```

And to use the **Mocha** framework in the **Protractor** test files it is necessary to require and configure **Chai**. See:

```
var chai = require('chai');
var chaiAsPromised = require('chai-as-promised');

chai.use(chaiAsPromised);
var expect = chai.expect;
```

Then it is possible to use **Chai as Promised** this way:

```
expect(myElement.getText()).to.eventually.equal('some text');
```

Below it is shown a simple example of a test using **Mocha** as the base framework:

```
var chai = require('chai');
var chaiAsPromised = require('chai-as-promised');

chai.use(chaiAsPromised);
var expect = chai.expect;

describe('Protractor website', function() {
  it('logo', function() {
    var logo = element(by.css('.protractor-logo'));

    browser.get('#');

    expect(logo.getAttribute('class')).to.eventually.equal('protractor-logo');
  });
});
```

More information about the **Mocha** framework can be found in the tool's official website, through the following URL: http://mochajs.org[32]

For more details about choosing a base framework for using the the **Protractor** framework, see the following URL: http://www.protractortest.org/#/frameworks[33] ___

shardTestFiles - Running tests in parallel

In running the automated tests (or suites of tests) developed in the project, even early in the process you will discover that the test execution is slow, due to the large number of tests that cover different scenarios.

As already seen, automated tests exist to provide quick feedback to the team after application changes, but when tests are too slow to provide such feedback, some action needs to be taken.

One way of solving such problem is parallelization of tests.

Since **Protractor** is based on **Selenium**, it is possible to use the **grid** resources to run tests in parallel.

To run tests in parallel with **Protractor** you just need to add some properties to the defined capabilities, in the configuration file. See:

[32]http://mochajs.org
[33]http://www.protractortest.org/#/frameworks

```
// protractor.conf.js

module.exports.config = {
  directConnect: true,
  capabilities: {
    'browserName': 'chrome'
    shardTestFiles: true,
    maxInstances: 2

  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/'
};
```

In this configuration, the **shardTestFiles: true** property is used to allow parallelization of tests, and the **maxInstances: 2** property is used to define how many browsers will be available for running tests in parallel (in this case 2).

In other words, with this configuration the tests will be parallelized in two different browsers. For example, in case there are two files with the extension .spec.js, each one will be ran in parallel in one browser, halving the time of testing.

If necessary, to run parallel tests in more than two browsers, you just need to define the desired value in the **maxInstances** property.

Additional video suggestions (in Portuguese):

Some time ago I recorded two videos showing in a "hands on" section how to run tests in parallel, on development environment and in the cloud. Both videos can be found through the following URLs, respectively:

https://www.youtube.com/watch?v=KllelI_Cd30[34] https://www.youtube.com/watch?v=rhWT3Ev1ROA[35]

Additional reading suggestion:

Paul Yoder wrote a very well explained post about test parallelization with **Protractor**. The post can be found through the following URL: http://blog.yodersolutions.com/run-protractor-tests-in-parallel/[36]

[34]https://www.youtube.com/watch?v=KllelI_Cd30
[35]https://www.youtube.com/watch?v=rhWT3Ev1ROA
[36]http://blog.yodersolutions.com/run-protractor-tests-in-parallel/

suites - Test suites

A test suite may be defined by a file with the extension .spec.js, for example.

In case the test files are separated by functionalities (which is also recommended), we could say that there are different test suites for different application features.

However, we can also define different kind of test suites, such as a smoke test suite (to verify the main application functionalities and only the happy path scenarios), or a test suite for the main application routes (to check that users are directed to the right routes when navigating through the application), or even a test suite that run only the mobile tests, for example.

With **Protractor** it is possible to define test suites in the configuration file and then it is possible to run an specific test suite from the command line.

See below an example from a definition of a smoke test suite:

```
// protractor.conf.js

module.exports.config = {
  directConnect: true,
  capabilities: {
    'browserName': 'chrome'

  },
  specs: ['specs/*.spec.js'],
  baseUrl: 'http://www.protractortest.org/',
  suites: {
    smoke: 'specs/smokeTests.spec.js',
  },
};
```

The **suites** property receives as value an object, which may define one or more test suites.

In the above example a smoke test suite is defined, which points to a file located in the **specs** directory with name **smokeTests.spec.js**.

To run a specific test suite, such as the smoke test suite just defined, the following command can be executed through the command line, which will run only the files defined for this suite, disregarding any other file defined in the array of specs:

```
protractor --suite smoke
```

This configuration may be useful when using deployment pipelines, where before running the regression test suite, the smoke test suite may be ran, ensuring that, in this case, there will not be a time misused if something fails in a previous phase.

beforeLaunch - Before any environment configuration

The **beforeLaunch** function may be useful for defining necessary things before any environment configuration, such as starting a tunnel for running tests on cloud services against a local development environment, or starting some other service, like **Visual Review**, for visual review testing. This function is executed only once and before the **are** function.

See the below two examples:

Cloud testing against a local development environment:

```
beforeLaunch: function() {
  return new Promise(function(resolve, reject) {
    var browserstackLocalArgs = {
      'key': process.env.BROWSERSTACK_ACCESS_KEY,
      'force': 'true',
      'forcelocal': 'true'
    };
    browserstackLocal.start(browserstackLocalArgs, function(error) {
      if (error) {
        reject(error);
        return;
      }
      resolve();
    });
  });
}
```

In the above example the **beforeLaunch** configuration starts a tunnel on the **BrowserStack** service before any other environment configuration.

Visual review testing:

```
beforeLaunch: function () {
  vr.initRun('Visual-Review-Sample-Project', 'visualReviewSuite');
}
```

In the above example **Visual Review** is started enabling visual review testing after the automated tests execution.

onPrepare - Before running tests

The **onPrepare** function is executed as soon as **Protractor** is ready and available. This function is ran before any test script will be effectively executed.

A simple use for such configuration can be, for example, maximizing the browser's window before testing. See:

```
onPrepare: function () {
        browser.driver.manage().window().maximize()
}
```

Another examples os using the **onPrepare** functions were already seen in the **Useful node modules** chapter, to define different test reports. See:

jasmine-spec-reporter:

```
onPrepare: function() {
  jasmine.getEnv().addReporter(new SpecReporter({
    displayFailuresSummary: true,
    displayFailedSpec: true,
    displaySuiteNumber: true,
    displaySpecDuration: true
  }));
}
```

In the above example the **onPrepare** function is defined to use the **jasmine-spec-reporter** node module for a better feedback in the console after running tests.

---

protractor-jasmine2-html-reporter:

```
onPrepare: function() {
  jasmine.getEnv().addReporter(new SpecReporter({
    displayFailuresSummary: true,
    displayFailedSpec: true,
    displaySuiteNumber: true,
    displaySpecDuration: true
  }));

  jasmine.getEnv().addReporter(new Jasmine2HtmlReporter({
    takeScreenshots: true,
    fixedScreenshotName: true
  }));
}
```

In the above example the **onPrepare** function is defined to use the **protractor-jasmine2-html-reporter** node module, enabling an HTML test report with screenshots of each executed test, that may be used as test evidences.

---

onComplete - As soon as tests are finished

The **onComplete** function is executed as soon as tests are finished.

Such function may be used, for example, to stop a tunnel with e cloud service, as you can see below:

```
onComplete: function() {
  return new Promise(function(resolve, reject) {
    browserstackLocal.stop(function(error) {
      if (error) {
        reject(error);
        return;
      }
      resolve();
    });
  });
}
```

In this example the tunnel with the **BrowserStack** service is stopped after running tests to avoid that other tests are triggered against the local development environment.

Such configuration is useful when using continuous integration.

---

afterLaunch - After running tests

The **afterLaunch** function is executed after the tests execution and when webdriver is already shut down. This function pass an **exitCode** parameter, which has value equal to `0` when tests passed.

An example of using such function is when cleaning up temporary files generated by **Visual Review** after running e2e tests using such integration. See:

```
afterLaunch: function (exitCode) {
  return vr.cleanup(exitCode);
}
```

For the complete list of available configurations, look the the official **Protractor** documentation, through the following URL: https://github.com/angular/protractor/blob/master/docs/referenceConf.js[37]

[37]https://github.com/angular/protractor/blob/master/docs/referenceConf.js

The software test creative process

When talking about the software test creative process, in the context of e2e testing, the idea is to answer some of the following questions:

- Which scenarios/test cases to create?
- How to evolve the test suite?
- How to organize the test project to help on its maintenance?
- How to evolve even more?

In this chapter will be presented some ideas to answer these questions.

Defining test cases

To start writing automated e2e tests it is important to ensure that the main application functionalities are working, so a good idea when defining the test cases is to start by the happy path scenarios, that is, the scenarios that the user will follow to successfully accomplish a procedure in the application under test, such as a successfully login, adding a product in the cart with success, or the successful user account creation, for example.

Also, it is interesting to write the test descriptions even before implementing its logic. See some examples:

```
describe('Test system', function() {
  it('successful account creation', function() {});

  it('successfully login', function() {});

  it('add product to the cart', function() {});
});
```

Note that the tests has not implementation, but, such technique helps on structuring the test suite for later implementation and improvement. Besides that, such visualization helps in new test cases creation and on removal of redundant or unnecessary tests.

After thinking in the happy path scenarios it is easier to think in the alternate paths. See some examples:

```
describe('Test system', function() {
  it('try to create account without filling all mandatory fields', function() {}\
);

  it('try to create account with password that does not match', function() {});

  it('try to create account using password that does not respect the rules', fun\
ction() {});

  it('try to login with invalid user', function() {});

  it('try to login without filling any field', function() {});
});
```

With an idea of the test scenarios it is possible to evolve the test suite and go to its implementation.

Evolving the test suite

A good tactic to start is to follow the red, green, refactoring technique, from TDD (test-driven development). In other words, first we create a failing test, after that we make the minimum to make the test pass, and then we refactor it to make it robust.

See the below example:

```
describe('Test system', function() {
  it('successful account creation', function() {
    browser.get('http://choko.org/sign-in');

    element(by.id('element-sign-in-username')).sendKeys('validuser');
    element(by.id('element-sign-in-password')).sendKeys('validpassword');
    element(by.id('element-sign-in-submit')).click();

    expect(element(by.css(.authenticated)).isPresent()).not.toBe(true);
  });
});
```

In case the user and password of the above test are valid, such test should fail, because the user must be authenticated and the application must have an element with a CSS class 'authenticated', but the test expect that such element is not present. It is good to remember that is always to make tests fail, to ensure that when passing it is not a false positive, that is, a test that always pass, even when it should be failing.

Later, following the red, green, refactoring technique, we must do the minimum to make the test pass. See:

```
describe('Test system', function() {
  it('successful account creation', function() {
    browser.get('http://choko.org/sign-in');

    element(by.id('element-sign-in-username')).sendKeys('validuser');
    element(by.id('element-sign-in-password')).sendKeys('validpassword');
    element(by.id('element-sign-in-submit')).click();

    expect(element(by.css(.authenticated)).isPresent()).toBe(true);
  });
});
```

With the test passing it is then time to refactor it to make it more robust. See an alternative:

```
describe('Test system', function() {
  function login(user, password) {
    element(by.id('element-sign-in-username')).sendKeys(user);
    element(by.id('element-sign-in-password')).sendKeys(password);
    element(by.id('element-sign-in-submit')).click();
  }

  it('successful account creation', function() {
    browser.get('http://choko.org/sign-in');

    login('validuser', 'validpassword');

    expect(element(by.css(.authenticated)).isPresent()).toBe(true);
  });
});
```

Note that the login function was create, which receives as parameters a user and password, fill the needed fields with these values, and clicks in the login button.

With this simple function, what before were three steps is not just one. Also, such function may be reused for other login scenarios, such as the login tentative without filling user and password. See:

```
it('try to login without filling any field', function() {
  browser.get('http://choko.org/sign-in');

  login('', '');

  expect(element(by.css(.authenticated)).isPresent()).not.toBe(true);
});
```

But it could be even better. Until now the shown tests are mixing the elements' definitions and function with the test case's steps itself.

As already seen in other chapter, one way to organize tests to ease its maintenance is using the Page Objects pattern.

Organizing the test project for a evolving maintenance

Organize tests is a primordial task to help on their maintenance, such as make then readable.

Using the Page Objects pattern helps in this sense, separating the elements's definitions and functions in specific files, which may be changed once and every tests that use them can enjoy such changes, leaving the tests themselves with a more business focused language, where the necessary steps to run the test cases are defined, helping in the readability.

See the login test already shown, but now refactored to use the Page Objects pattern:

```
var LoginPage = require('../page-objects/loginPage.po.js');

describe('Test system', function() {
  var loginPage = new LoginPage();

  it('successful account creation', function() {
    loginPage.visit();

    loginPage.login('validuser', 'validpassword');

    expect(loginPage.isLoggedIn.isPresent()).toBe(true);
  });
});
```

Note that reading such test is easier and closer to a business language, where there are no unnecessary informations, such and IDs of elements, etc.

Below is shown the Page Object:

```
var LoginPage = function() {
  this.usernameField = element(by.id('element-sign-in-username'));
  this.passwordField = element(by.id('element-sign-in-password'));
  this.loginButton = element(by.id('element-sign-in-submit'));
};

LoginPage.prototype.login = function(user, password) {
  this.usernameField.sendKeys(user);
  this.passwordField.sendKeys(password);
  this.loginButton.click();
};

LoginPage.prototype.visit = function() {
  browser.get('http://choko.org/sign-in');
};

module.exports = LoginPage;
```

In this case, what the Page Object does is to define the elements identified on the page under test, and also is defining the functions required for executing more than one step, such as the login scenario, or even a function is created just to ease the readability of the test case, such as visiting a web page (URL).

In this manner the test suite is getting more organized and taking shape, but it does not stop there.

Evolving even more

Software may always be improved, and once a test suite starts to grow its architecture must evolve.

In the first section of this chapter test scenarios of three different application's functionalities were shown (account creation, login and shopping cart).

One way to evolve such tests for a better organization, for example, is to separate them in different test files, where each one intent to test different features, in other words, the tests become to be separated depending on their contexts. See an example of such test evolution:

Tests for user account creation:

```
describe('Account creation', function() {
  it('successful account creation', function() {});

  it('try to create account without filling all mandatory fields', function() {}\
);

  it('try to create account with password that does not match', function() {});

  it('try to create account using password that does not respect the rules', fun\
ction() {});
});
```

Tests for login:

```
describe('Login', function() {
  it('successfully login', function() {});

  it('try to login with invalid user', function() {});

  it('try to login without filling any field', function() {});
});
```

Tests for shopping cart:

```
describe('Shopping cart', function() {
  it('add product to the cart', function() {});
});
```

And each test may have its own specific Page Objects, and also some generic ones.

Obviously there is the possibility of more improvements, which will depend of different contexts, such as organizing Page Objects per functionalities and sub directories (for too big applications), helpers creation for generic things along the application, etc, but the idea here was at least to show a bit about the creative process involving e2e tests creation.

Suggested further reading:

In my Medium[38] account I wrote a post that I will leave as suggested reading, since this is related to the topic of this chapter. I hope it helps! You can find it through the following URL:

https://medium.com/the-making-of-appear-in/automated-acceptance-tests-from-where-to-start-bb747c49a401#.s0

[38]https://medium.com/@walmyrlimaesilv

[39]https://medium.com/the-making-of-appear-in/automated-acceptance-tests-from-where-to-start-bb747c49a401#.s0qvwv2kk

Useful tips

This chapter is a collection of useful tips for using with the **Protractor** framework to ease the software development united to the benefits of automated tests, such as a test structure generator, some **Jasmine** facilities, debug related issues, tests for non-AngularJS apps, tips for application demos, and how to overwrite configurations by command line.

Test structure generator

For each new web project some steps needs to be repeated for using **Protractor** as the e2e test framework, such as defining the configuration file and its configurations (as **baseUrl** and the browser in which the tests will be executed against), the **package.json** file to define the test dependencies, and the test directory.

To ease this initial configuration it is possible to use the generator-protractor, which will be explained below.

The generator-protractor is a code generator for **Protractor** tests and can be installed following the below steps:

```
npm install -g yo
npm install -g generator-protractor
```

After installed it may be used in the following way:

From the root's project directory that will be tested using **Protractor**, create a directory called **tests**, and inside it create a directory called **e2e**.

Access the just created **e2e** directory from the command line:

```
cd tests/e2e
```

And run the following command:

```
yo protractor
```

Some questions will follow, which may be answered with the default values suggested or as your choice. Following are the questions with their default values:

```
Welcome to the protractor code generator.
? Choose a name for the protractor configuration file (protractor.conf.js)
? Choose a base URL (http://localhost:8000)
? Which browsers do you want to run? (Use arrow keys)
â□¯ Chrome
  Firefox
  Both, at the same time
```

After answering the questions the basic to start is generated, including a sample test file, and then it is just to customize it as your specific project needs. An additional suggestion is the ***page-objects** directory creation, since this is a good practice to help on maintainability and readability.

More details about the generator-protractor may be found through the following URL: https://www.npmjs.com/pack
protractor[40].

Jasmine facilities

When creating automated tests, sometimes we need to run only a new test just created, without the needs of running all the others, or even, we may want to run just a specific file, without the need of running all the files, or, we may want to skip a specific test without affect the others, when such test is failing and needs some maintenance because it is not robust enough and is causing false negative test results, for example.

The **Jasmine** base framework, used as default by **Protractor** has some facilities to solve such issues and they will be addressed below:

Running only one test case

Let's look the following example:

```
var TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Angular', function() {
  var todoMvc = new TodoMvc();

  it('add an item in the todo list', function() {
    todoMvc.visit();

    todoMvc.addItemOnTodoList('Create test without page object');

```

[40]https://www.npmjs.com/package/generator-protractor

```
    expect(todoMvc.listOfItems.count()).toEqual(1);
  });

  it('add new item in the todo list', function() {
    var text = 'Create new test without page object';

    todoMvc.visit();

    todoMvc.addItemOnTodoList(text);

    expect(todoMvc.listOfItems.getText()).toContain(text);
  });
});
```

The above shown test has two test cases. In case there is a need to run only one of them (the second one, for example), it is only needed to modify the **it** definition for **fit**, then, when **Protractor** is run, only the **fit** test will be executed, and any other will not be considered. See:

```
var TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Angular', function() {
  var todoMvc = new TodoMvc();

  it('add an item in the todo list', function() {
    todoMvc.visit();

    todoMvc.addItemOnTodoList('Create test without page object');

    expect(todoMvc.listOfItems.count()).toEqual(1);
  });

  fit('add new item in the todo list', function() {
    var text = 'Create new test without page object';

    todoMvc.visit();

    todoMvc.addItemOnTodoList(text);

    expect(todoMvc.listOfItems.getText()).toContain(text);
  });
});
```

Since the second test case is defined as **fit**, only this one is run.

Skipping a specific test case

The opposite is also possible. Let's say you want to skip a specific test case for further maintenance to make it more reliable, such as a test that sometimes fails and sometimes passes, even without any application change.

Such objective is hit using another **Jasmine** facility. It is just to substitute the **it** for **xit**. See:

```
var TodoMvc = require('../page-objects/todoMvc.po.js');

describe('Todo MVC Angular', function() {
  var todoMvc = new TodoMvc();

  it('add an item in the todo list', function() {
    todoMvc.visit();

    todoMvc.addItemOnTodoList('Create test without page object');

    expect(todoMvc.listOfItems.count()).toEqual(1);
  });

  xit('add new item in the todo list', function() {
    var text = 'Create new test without page object';

    todoMvc.visit();

    todoMvc.addItemOnTodoList(text);

    expect(todoMvc.listOfItems.getText()).toContain(text);
  });
});
```

In the same example, now only the first test is executed, because the test with the **xit** definition is not considered in runtime execution. Furthermore, when a test is skipped it is possible to add an explanation about why such test is been skipped, which is available after the test execution in the list of pending tests (tests defined with **xit**). See:

```
xit('add new item in the todo list', function() {
  todoMvc.visit();

  todoMvc.addItemOnTodoList('Create new test without page object');

  expect(todoMvc.listOfItems.count()).toEqual(2);
}).pend('This test needs refactoring, because it is not independent');
```

Note that the test has a **.pend();** in the end, which receives a string that is shown in the result of the test execution, in the list of pending tests. See an example:

```
$ protractor
[18:49:48] I/direct - Using ChromeDriver directly...
[18:49:48] I/launcher - Running 1 instances of WebDriver
Started
.*.

Pending:

1) angularjs homepage todo list should list todos
  Needs refactoring

3 specs, 0 failures, 1 pending spec
Finished in 5.761 seconds
[18:49:55] I/launcher - 0 instance(s) of WebDriver still running
[18:49:55] I/launcher - chrome #01 passed
```

Running only one describe

Another facility available when using **Jasmine** is to run only a specific **describe**. Such option may be useful when there are many **describes** for different functionalities in the application, and there is a need of running only the tests for certain functionality. Such thing is hit in the following way:

```
var TodoMvc = require('../page-objects/todoMvc.po.js');

fdescribe('Todo MVC Angular', function() {
  var todoMvc = new TodoMvc();

  it('add an item in the todo list', function() {
    todoMvc.visit();

    todoMvc.addItemOnTodoList('Create test without page object');

    expect(todoMvc.listOfItems.count()).toEqual(1);
  });

  it('add new item in the todo list', function() {
    var text = 'Create new test without page object';

    todoMvc.visit();

    todoMvc.addItemOnTodoList(text);

    expect(todoMvc.listOfItems.getText()).toContain(text);
  });
});
```

Note that the **describe** definition is modified to use **fdescribe** instead. By defining a **describe** as **fdescribe**, when **Protractor** is run it only run the **fdescribe** and do not consider the **describes**.

Debugging tests

Many times when writing or running tests it is necessary some way to debug them, mainly when they are failing and we don't know why.

Protractor has the ability to pause a test during its execution to help on debugging.

Let's see how to pause a test during test execution, as below:

```
it('add new item in the todo list', function() {
  var text = 'Create new test without page object';

  todoMvc.visit();

  todoMvc.addItemOnTodoList(text);
  browser.pause();

  expect(todoMvc.listOfItems.getText()).toContain(text);
});
```

Note that there is a code **browser**.**pause()**; right after the step where an item is added in the TODO list.

When **Protractor** run the tests and finds the pause code the following is shown in the console and the browser stays opened in this exact point:

```
[19:00:09] I/protractor - Encountered browser.pause(). Attaching debugger...
[19:00:09] I/protractor -
[19:00:09] I/protractor - ------- WebDriver Debugger -------
[19:00:09] I/protractor - Starting WebDriver debugger in a child process. Pause \
is still beta, please report issues at github.com/angular/protractor
[19:00:09] I/protractor -
[19:00:09] I/protractor - press c to continue to the next webdriver command
[19:00:09] I/protractor - press ^D to detach debugger and resume code execution
[19:00:09] I/protractor - type "repl" to enter interactive mode
[19:00:09] I/protractor - type "exit" to break out of interactive mode
```

With that it is possible to analyze the application where the browser stopped to understand some specific problem, such as why certain element is not being used, for example. In this case we may discover that the selector being used to choose the element which we want to interact with is not suitable, since there are more than one elements with the same selector, let's say.

Testing non-AngularJS applications

Even **Protractor** being the official e2e test framework for AngularJS applications, with only one line of code it is possible to use it for creating and running tests for any other web application.

Since it is so easy to create tests with **Protractor**, non-AngularJS projects may use it, where only the AngularJS specific locators will not be available, such as: by.binding, by.model e by.repeater, but all the rest will be available.

See below the needed change in the **Protractor** configuration file to use it for testing non-AngularJS applications:

```
onPrepare() {
  browser.ignoreSynchronization = true;
},
```

As seen in the **Advanced configurations** chapter, many configurations may be done when preparing the tests, before their execution. One of this configurations is the **browser.ignoreSynchronization = true**;.

By default the value of this configuration is **false**, which means that **Protractor** will always understand that the application under test is an AngularJS application.

By setting such configuration with **true** value, in the moment **Protractor** run the tests it will not look for AngularJS specific things, enabling it to create and execute tests for any kind of web application, such as Ember, Drupal, etc.

Tips for demos

Automated tests are usually very fast when executed when compared with humans running the same test cases, which is one of the benefits of their usage, but automated tests may be used for other things, such as documenting the application behavior in different situations, or even for demos.

Let's say an application is been developed for an external customer, and in the end of each week (Fridays) the new features are demonstrated to the customer.

One way to demo such functionality to the customer may be running the e2e tests, once they are created exactly thinking in the application usage by the end user perspective. But, since such tests are so fast executed, it may be difficult to the customer to understand what each this is really doing.

Protractor has a **sleep** functionality, which defines a wait time between each test steps when tests are executed. Using sleeps is not recommended for running tests in their natural way, since it can make them slower and not always trustable, but it may be useful for demonstrating features.

See below a test case sample, where there is a 5 seconds of wait time is added between each test step, so that the customer can see the test being executed in a "paused" way, to understand the application behavior step-by-step:

```
it('add new item in the todo list', function() {
  var text = 'Create new test without page object';

  todoMvc.visit();
  browser.sleep(5000);

  todoMvc.addItemOnTodoList(text);
  browser.sleep(5000);

  expect(todoMvc.listOfItems.getText()).toContain(text);
});
```

The **sleep** function receives as argument a value in milliseconds and the next test steps is executed after this time is hit.

Obs.: Remember that it is not recommended using sleep times unless for demos, in other words, avoid using sleep times in any other situation.

Overwriting configurations via command line

In some moments may be interesting to overwrite the configurations defined in the **Protractor** configuration file when running tests, for example, to run the same test in a different browser, to run the same test against a different base URL, to run a specific test file, to run tests using the browser's own webdriver, etc.

With **Protractor** it is possible to overwrite configurations directly in the command line. See:

Running tests in a browser different from the one configured

Let's say the browser defined in the **protractor.conf.js** file is Chrome, and in the moment of running the tests it is necessary to run the same tests, but against Firefox.

In the console, if you are already in the directory where the configuration file is located, run the following command:

```
protractor --browserName firefox
```

When passing to protractor the argument –browserName firefox, such capability is overwritten in the configuration file with the value defined in the command line, in this case the capability **browserName** receives the value **firefox**.

Running tests in a different base URL

In the same way, the base URL may be overwritten. Such approach is useful to run the same tests in different environments, such as QA environment, UAT environment (user acceptance testing), or even in production environment. See:

```
protractor --baseUrl http://qa-sample.io
```

With the above command the base URL is overwritten to run the tests in the QA environment, for example.

```
protractor --baseUrl http://uat-sample.io
```

With the above command the base URL is overwritten to run the tests in the UAT environment, for example.

```
protractor --baseUrl http://sample.io
```

With the above command the base URL is overwritten to run the tests in the production environment, for example.

Running only tests of a specific file

In case there is a need to run a specific file from the **specs** directory, for example, you just need to run the following command:

```
protractor --specs specs/specific-test.spec.js
```

By running such command the array of specs configuration is overwritten by the file defined directly in the command line and only this file is executed.

Running tests using the own browser's webdriver

Usually **Protractor** is configured to use a Selenium server to run the tests. Such approach is widely used when running tests in continuous integration servers, for example. However, in some moments a developer (or tester) may want to run the tests directly in their computers, and in case the computer uses Chrome or Firefox, such tests can be executed without the need of a selenium server, since with **Protractor** it is possible to use the webdriver of these browsers.

To overwrite such configuration, run the following command:o:

```
protractor --directConnect true
```

When running the above command, even if there is a **seleniumAddress** set in the **protractor.conf.js** file, this will not be considered and the browser's webdriver will be used.

Obs.: Remember that this is only possible when using Chrome or Firefox.

I hope such tips are useful!

Going further

While learning about e2e test automation we saw the importance of using good practices, the usage of the Page Objects design pattern, to help on maintainability and readability, and the usage of helpers, with generic functions to ease writing tests.

Also, we knew some node modules that may be used to complement the tests and how to make actions and verifications while testing.

We knew different techniques that may be used together with e2e tests as well, such as visual review testing, cloud testing, mobile testing, and how to integrate some of these routines in the continuous integration process, to provide a quick feedback after any application change.

Finally, were presented questions about writing e2e tests using ECMAScript 2015, some advanced configurations, the creative process behind automated tests, and some tips to help in different moments when performing software test related activities.

All the content addressed in this book may evolve even more with practice, so, my suggestion is that you start to implement automated e2e tests right now, in your personal software projects, in your job, or in your academic projects, because as software development professionals, all days we needs to glimpse developing better softwares, and practice is what will lead to perfection.

Walmyr Lima e Silva Filho, book's author

http://walmyr-filho.com[41]

[41]http://walmyr-filho.com