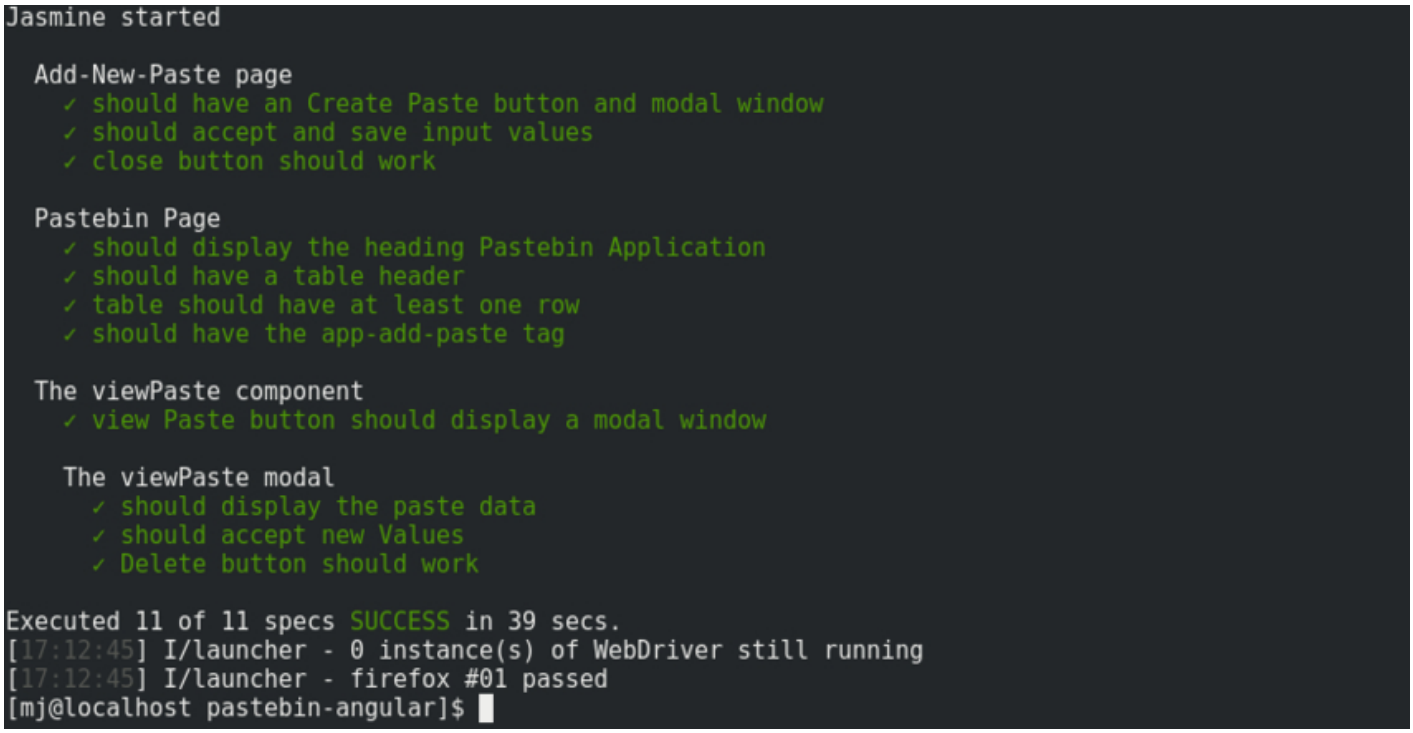
Getting Started With End-to-End Testing in Angular Using Protractor

by Manjunath M 14 Sep 2017

Difficulty: Intermediate Length: Long Languages: English

Angular Angular 2+ Testing Web Development



What You'll Be Creating

Protractor is a popular end-to-end test framework that lets you test your Angular application on a real browser simulating the browser interactions just the way that a real user would interact with it. End-to-end tests are designed to ensure that the application behaves as expected from a user's perspective. Moreover, the tests are not concerned about the actual code implementation.

Protractor runs on top of the popular Selenium WebDriver, which is an API for browser automation and testing. In addition to the features provided by Selenium WebDriver, Protractor offers locators and methods for capturing the UI components of the Angular application.

In this tutorial, you will learn about:

- setting up, configuring and running Protractor
- writing basic tests for Protractor
- page objects and why you should use them
- guidelines to be considered while writing tests
- writing E2E tests for an application from start to finish

Doesn't that sound exciting? However, first things first.

Do I Need to Use Protractor?

If you've been using Angular-CLI, you might know that by default, it comes shipped with two frameworks for testing. They are:

- unit tests using Jasmine and Karma
- end-to-end tests using Protractor

The apparent difference between the two is that the former is used to test the logic of the components and services, while the latter is used to ensure that the high-level functionality (which involves the UI elements) of the application works as expected.

If you are new to testing in Angular, I'd recommend reading the Testing Components in Angular Using Jasmine series to get a better idea of where to draw the line.

In the former's case, you can leverage the power of Angular testing utilities and Jasmine to write not just unit tests for components and services, but basic UI tests also. However, if you need to test the front-end functionality of your application from start to end, Protractor is the way to go. Protractor's API combined with design patterns such as page objects make it easier to write tests that are more readable. Here's an example to get things rolling.

```
/*
  1. It should have a create Paste button
  2. Clicking the button should bring up a modal window
*/

it('should have a Create Paste button and modal window', () => {

    expect(addPastePage.isCreateButtonPresent()).toBeTruthy("The button should e
    expect(addPastePage.isCreatePasteModalPresent()).toBeFalsy("The modal window

    addPastePage.clickCreateButton();

    expect(addPastePage.isCreatePasteModalPresent()).toBeTruthy("The modal windc
});
```

Configuring Protractor

Setting up Protractor is easy if you are using Angular-CLI to generate your project. The directory structure created by `ng new` is as follows.

```
.
├── e2e
|   ├── app.e2e-spec.ts
|   ├── app.po.ts
|   └── tsconfig.e2e.json
├── karma.conf.js
├── package.json
├── package-lock.json
├── protractor.conf.js
├── README.md
├── src
```

```
|   ├── app
|   ├── assets
|   ├── environments
|   ├── favicon.ico
|   ├── index.html
|   ├── main.ts
|   ├── polyfills.ts
|   ├── styles.css
|   ├── test.ts
|   ├── tsconfig.app.json
|   ├── tsconfig.spec.json
|   └── typings.d.ts
├── tsconfig.json
└── tslint.json

5 directories, 19 files
```

The default project template created by Protractor depends on two files to run the tests: the spec files that reside inside the **e2e** directory and the configuration file (**protractor.conf.js**). Let's see how configurable **protractor.conf.js** is:

```
/* Path: protractor.conf.ts*/

// Protractor configuration file, see link for more information
// https://github.com/angular/protractor/blob/master/lib/config.ts

const { SpecReporter } = require('jasmine-spec-reporter');

exports.config = {
  allScriptsTimeout: 11000,
  specs: [
    './e2e/**/*.e2e-spec.ts'
  ],
  capabilities: {
    'browserName': 'chrome'
  },
  directConnect: true,
  baseUrl: 'http://localhost:4200/',
  framework: 'jasmine',
  jasmineNodeOpts: {
    showColors: true,
    defaultTimeoutInterval: 30000,
    print: function() {}
  },
  onPrepare() {
    require('ts-node').register({
      project: 'e2e/tsconfig.e2e.json'
    });

```

```
    jasmine.getEnv().addReporter(new SpecReporter({ spec: { displayStacktrace: t
  }
};
```

If you are ok with running the test on Chrome web browser, you can leave this as is and skip the rest of this section.

Setting Up Protractor With Selenium Standalone Server

The `directConnect: true` lets Protractor connect directly to the browser drivers. However, at the moment of writing this tutorial, Chrome is the only supported browser. If you need multi-browser support or run a browser other than Chrome, you will have to set up Selenium standalone server. The steps are as follows.

Install Protractor globally using npm:

```
npm install -g protractor
```

This installs the command-line tool for webdriver-manager along with that of protractor. Now update the webdriver-manager to use the latest binaries, and then start the Selenium standalone server.

```
webdriver-manager update

webdriver-manager start
```

Finally, set the `directConnect: false` and add the `seleniumAddress` property as follows:

```
capabilities: {
    'browserName': 'firefox'
  },
  directConnect: false,
  baseUrl: 'http://localhost:4200/',
  seleniumAddress: 'http://localhost:4444/wd/hub',
  framework: 'jasmine',
  jasmineNodeOpts: {
    showColors: true,
    defaultTimeoutInterval: 30000,
    print: function() {}
  },
```

The config file on GitHub provides more information about the configuration options available on Protractor. I will be using the default options for this tutorial.

Running the Tests

`ng e2e` is the only command you need to start running the tests if you are using Angular-CLI. If the tests appear to be slow, it's because Angular has to compile the code every time you run `ng e2e`. If you want to speed it up a bit, here's what you should do. Serve the application using `ng serve`.

Then fire up a new console tab and run:

```
ng e2e -s false
```

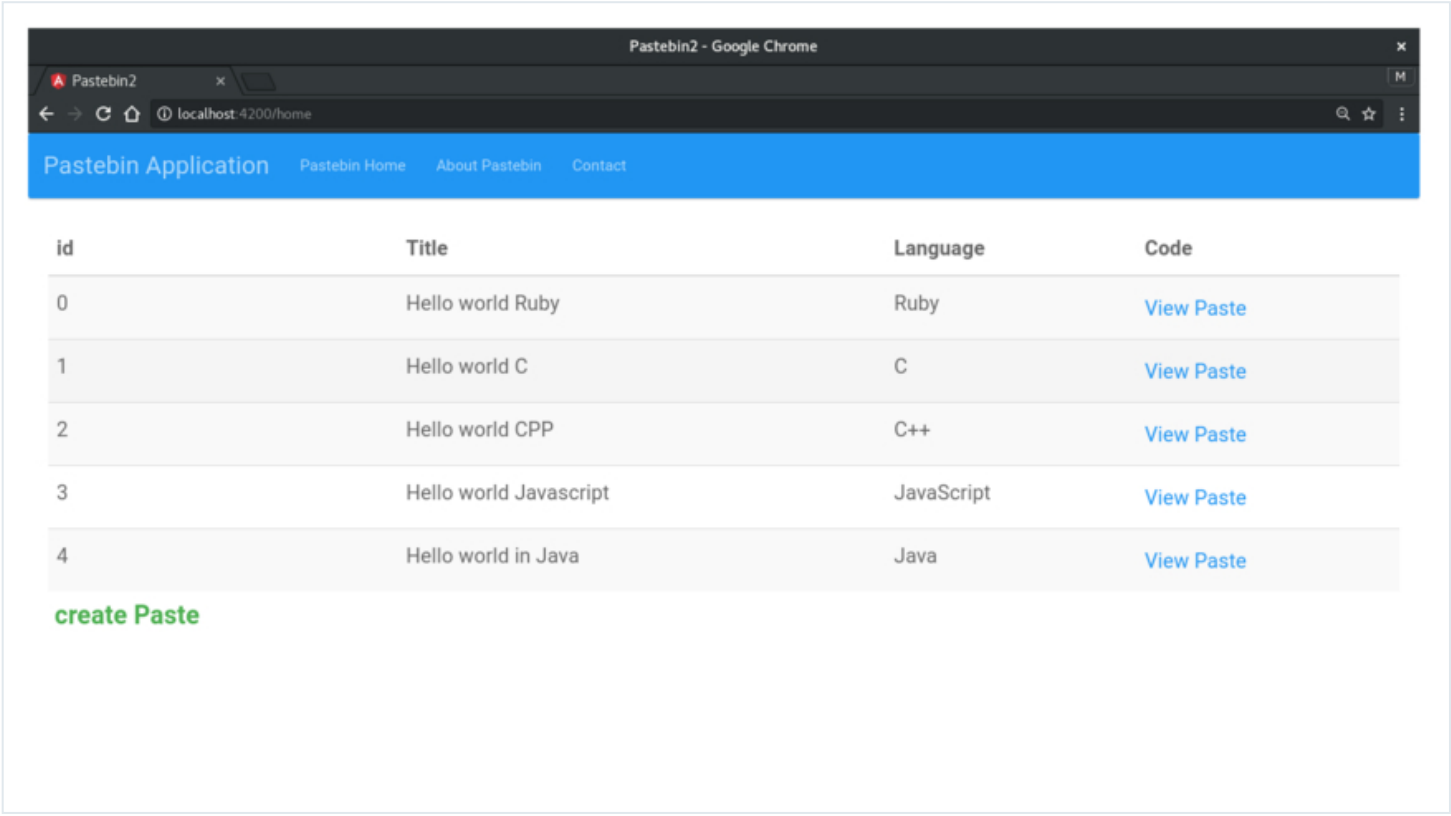
The tests should load faster now.

Our Goal

We will be writing E2E tests for a basic Pastebin application. Clone the project from the GitHub repo.

Both the versions, the starter version (the one without the tests) and the final version (the one with the tests), are available on separate branches. Clone the starter branch for now. Optionally, serve the project and go through the code to get acquainted with the application at hand.

Let's describe our Pastebin application briefly. The application will initially load a list of pastes (retrieved from a mock server) into a table. Each row in the table will have a **View Paste** button which, when clicked on, opens up a bootstrap modal window. The modal window displays the paste data with options to edit and delete the paste. Towards the end of the table, there is a **Create Paste** button which can be used to add new pastes.



The sample application.

The rest of the tutorial is dedicated to writing Protractor tests in Angular.

Protractor Basics

The spec file, ending with **.e2e-spec.ts**, will host the actual tests for our application. We will be placing all the test specs inside the **e2e** directory since that's the place we've

configured Protractor to look for the specs.

There are two things you need to consider while writing Protractor tests:

- Jasmine Syntax
- Protractor API

Jasmine Syntax

Create a new file called **test.e2e-spec.ts** with the following code to get started.

```
/* Path: e2e/test.e2e-spec.ts */

import { browser, by, element } from 'protractor';

describe('Protractor Demo', () => {

  beforeEach(() => {
    //The code here will get executed before each it block is called
    //browser.get('/');
  });

  it('should display the name of the application',() => {
   /*Expectations accept parameters that will be matched with the real value
   using Jasmine's matcher functions. eg. toEqual(),toContain(), toBe(), toBeTru
   expect("Pastebin Application").toEqual("Pastebin Application");

  });

  it('should click the create Paste button',() => {
    //spec goes here

  });
});
```

This depicts how our tests will be organized inside the spec file using Jasmine's syntax. `describe()`, `beforeEach()` and `it()` are global Jasmine functions.

Jasmine has a great syntax for writing tests, and it works just as well with Protractor. If you are new to Jasmine, I would recommend going through Jasmine's GitHub page first.

The **describe** block is used to divide the tests into logical test suites. Each **describe** block (or test suite) can have multiple **it** blocks (or test specs). The actual tests are defined

inside the test specs.

"Why should I structure my tests this way?" you may ask. A test suite can be used to logically describe a particular feature of your application. For instance, all the specs concerned with the Pastebin component should ideally be covered inside a describe block titled Pastebin Page. Although this may result in tests that are redundant, your tests will be more readable and maintainable.

A describe block can have a `beforeEach()` method which will be executed once, before each spec in that block. So, if you need the browser to navigate to a URL before each test, placing the code for navigation inside `beforeEach()` is the right thing to do.

Expect statements, which accept a value, are chained with some matcher functions. Both the real and the expected values are compared, and a boolean is returned which determines whether the test fails or not.

Protractor API

Now, let's put some flesh on it.

```
01 /* Path: e2e/test.e2e-spec.ts */
02
03 import { browser, by, element } from 'protractor';
04
05 describe('Protractor Demo', () => {
06
07   beforeEach(() => {
08     browser.get('/');
09   });
10
11   it('should display the name of the application',() => {
12
13     expect(element(by.css('.pastebin')).getText()).toContain('Pastebin Applicati
14
15   });
16
17   it('create Paste button should work',() => {
18
19     expect(element(by.id('source-modal')).isPresent()).toBeFalsy("The modal winc
20     element(by.buttonText('create Paste')).click();
21     expect(element(by.id('source-modal')).isPresent()).toBeTruthy('The modal wir
22
23   });
```

```
24 });
```

`browser.get('/')` and `element(by.css('.pastebin')).getText()` are part of the Protractor API. Let's get our hands dirty and jump right into what Protractor has to offer.

The prominent components exported by Protractor API are listed below.

1. `browser()`: You should call `browser()` for all the browser-level operations such as navigation, debugging, etc.
2. `element()`: This is used to look up an element in the DOM based on a search condition or a chain of conditions. It returns an ElementFinder object, and you can perform actions such as `getText()` or `click()` on them.
3. `element.all()`: This is used to look for an array of elements that match some chain of conditions. It returns an ElementArrayFinder object. All the actions that can be performed on ElementFinder can be performed on ElementArrayFinder also.
4. locators: Locators provide methods for finding an element in an Angular application.

Since we will be using locators very often, here are some of the commonly used locators.

- `by.css('selector-name')`: This is by far the commonly used locator for finding an element based on the name of the CSS selector.
- `by.name('name-value')`: Locates an element with a matching value for the name attribute.
- `by.buttonText('button-value')`: Locates a button element or an array of button elements based on the inner text.

Note: The locators by.model, by.binding and by.repeater do not work with Angular 2+ applications at the time of writing this tutorial. Use the CSS-based locators instead.

Let's write more tests for our Pastebin application.

```
01 it('should accept and save input values', () => {
02       element(by.buttonText('create Paste')).click();
03
04       //send input values to the form using sendKeys
05
06       element(by.name('title')).sendKeys('Hello world in Ruby');
07       element(by.name('language')).element(by.cssContainingText('option', 'Ruby'
```

```
        element(by.name('paste')).sendKeys("puts 'Hello world';");

        element(by.buttonText('Save')).click();

        //expect the table to contain the new paste
        const lastRow = element.all(by.tagName('tr')).last();
        expect(lastRow.getText()).toContain("Hello world in Ruby");
});
```

The code above works, and you can verify that yourself. However, wouldn't you feel more comfortable writing tests without the Protractor-specific vocabulary in your spec file? Here's what I am talking about:

```
it('should have an Create Paste button and modal window', () => {

   expect(addPastePage.isCreateButtonPresent()).toBeTruthy("The button should ex
   expect(addPastePage.isCreatePasteModalPresent()).toBeFalsy("The modal window

   addPastePage.clickCreateButton();

   expect(addPastePage.isCreatePasteModalPresent()).toBeTruthy("The modal windo


 });

it('should accept and save input values', () => {

   addPastePage.clickCreateButton();

   //Input field should be empty initially
   const emptyInputValues = ["","",""];
   expect(addPastePage.getInputPasteValues()).toEqual(emptyInputValues);

   //Now update the input fields
   addPastePage.addNewPaste();

   addPastePage.clickSaveButton();

   expect(addPastePage.isCreatePasteModalPresent()).toBeFalsy("The modal window
   expect(mainPage.getLastRowData()).toContain("Hello World in Ruby");

 });
```

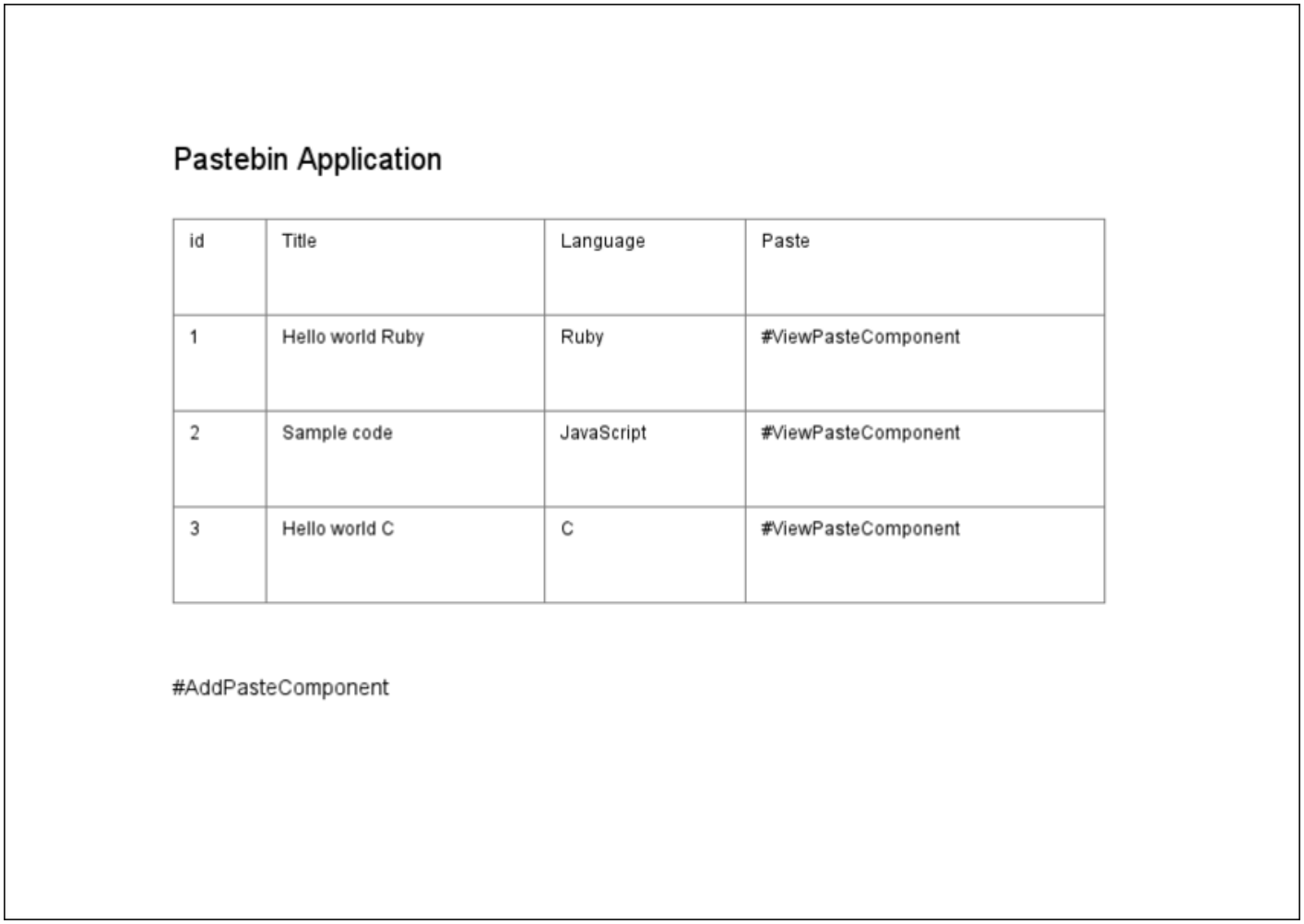
The specs appear more straightforward without the extra Protractor baggage. How did I do that? Let me introduce you to Page Objects.

Page Objects

Page Object is a design pattern which is popular in the test automation circles. A page object models a page or part of an application using an object-oriented class. All the objects (that are relevant to our tests) like text, headings, tables, buttons, and links can be captured in a page object. We can then import these page objects into the spec file and invoke their methods. This reduces code duplication and makes maintenance of code easier.

Create a directory named **page-objects** and add a new file inside it called **pastebin.po.ts**. All the objects concerned with the Pastebin component will be captured here. As previously mentioned, we divided the whole app into three different components, and each component will have a page object dedicated to it. The naming scheme **.po.ts** is purely conventional, and you can name it anything you want.

Here is a blueprint of the page we are testing**.**



Here is the code.

pastebin.po.ts

```
01 /* Path e2e/page-objects/pastebin.po.ts*/
02
03 import { browser, by, element, promise, ElementFinder, ElementArrayFinder } from
04
05
06 export class Pastebin extends Base {
07
08     navigateToHome():promise.Promise<any> {
09         return browser.get('/');
10     }
11
12     getPastebin():ElementFinder {
13         return element(by.css('.pastebin'));
14     }
15
16     /* Pastebin Heading */
17     getPastebinHeading(): promise.Promise<string> {
```

```
        return this.getPastebin().element(by.css("h2")).getText();
    }

    /*Table Data */

    getTable():ElementFinder {
        return this.getTable().element(by.css('table'));

    }

    getTableHeader(): promise.Promise<string> {
        return this.getPastebin().all(by.tagName('tr')).get(0).getText();
    }

    getTableRow(): ElementArrayFinder {
        return this.getPastebin().all(by.tagName('tr'));
    }


    getFirstRowData(): promise.Promise<string> {
        return this.getTableRow().get(1).getText();
    }

    getLastRowData(): promise.Promise<string> {
        return this.getTableRow().last().getText();
    }

    /*app-add-paste tag*/

    getAddPasteTag(): ElementFinder {
        return this.getPastebin().element(by.tagName('app-add-paste'));
    }

    isAddPasteTagPresent(): promise.Promise<boolean> {
        return this.getAddPasteTag().isPresent();
    }

}
```

Let's go over what we've learned thus far. Protractor's API returns objects, and we've encountered three types of objects thus far. They are:

- promise.Promise
- ElementFinder
- ElementArrayFinder

In short, `element()` returns an ElementFinder, and `element().all` returns an ElementArrayFinder. You can use the locators (`by.css`, `by.tagName`, etc.) to find the location of the element in the DOM and pass it to `element()` or `element.all()`.

ElementFinder and ElementArrayFinder can then be chained with actions, such as `isPresent()`, `getText()`, `click()`, etc. These methods return a promise that gets resolved when that particular action has been completed.

The reason why we don't have a chain of `then()`s in our test is because Protractor takes care of it internally. The tests appear to be synchronous even though they are not; therefore, the end result is a linear coding experience. However, I recommend using async/await syntax to ensure that the code is future proof.

You can chain multiple `ElementFinder` objects, as shown below. This is particularly helpful if the DOM has multiple selectors of the same name and we need to capture the right one.

```
getTable():ElementFinder {
    return this.getPastebin().element(by.css('table'));

}
```

Now that we have the code for the page object ready, let's import it into our spec. Here's the code for our initial tests.

```
/* Path: e2e/mainPage.e2e-spec.ts */

import { Pastebin } from './page-objects/pastebin.po';
import { browser, protractor } from 'protractor';


/* Scenarios to be Tested
  1. Pastebin Page should display a heading with text Pastebin Application
  2. It should have a table header
  3. The table should have rows
  4. app-add-paste tag should exist
*/

describe('Pastebin Page', () => {

  const mainPage: Pastebin = new Pastebin();

```

```
    beforeEach(() => {
        mainPage.navigateToHome();
    });

    it('should display the heading Pastebin Application', () => {

        expect(mainPage.getPastebinHeading()).toEqual("Pastebin Application");


    });

     it('should have a table header', () => {

        expect(mainPage.getTableHeader()).toContain("id Title Language Code");

    })
    it('table should have at least one row', () => {

        expect(mainPage.getFirstRowData()).toContain("Hello world");
    })

    it('should have the app-add-paste tag', () => {
        expect(mainPage.isAddPasteTagPresent()).toBeTruthy();
    })
});
```

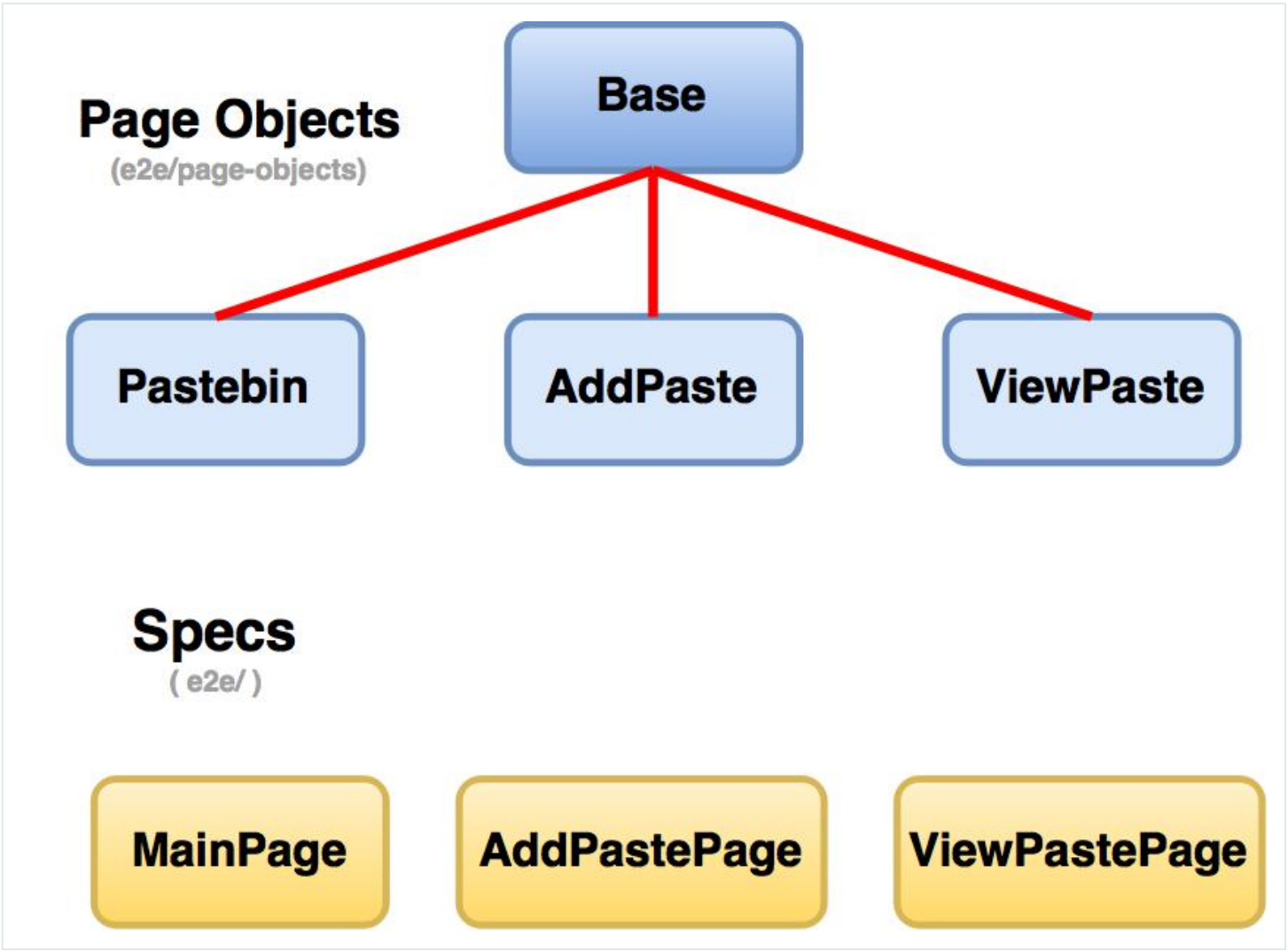
Organizing Tests and Refactoring

Tests should be organized in such a way that the overall structure appears meaningful and straightforward. Here are some opinionated guidelines that you should keep in mind while organizing E2E tests.

- Separate E2E tests from unit tests.
- Group your E2E tests sensibly. Organize your tests in a way that matches the structure of your project.
- If there are multiple pages, page objects should have a separate directory of their own.
- If the page objects have some methods in common (such as `navigateToHome()`), create a base page object. Other page models can inherit from the base page model.
- Make your tests independent from each other. You don't want all your tests to fail because of a minor change in the UI, do you?

- Keep the page object definitions free of assertions/expectations. Assertions should be made inside the spec file.

Following the guidelines above, here's what the page object hierarchy and the file organization should look like.



We've already covered **pastebin.po.ts** and **mainPage.e2e-spec.ts**. Here are the rest of the files.

Base Page Object

```
01 /* path: e2e/page-objects/base.po.ts */
02
03 import { browser, by, element, promise, ElementFinder, ElementArrayFinder } from
04
05 export class Base {
06
```

```
    /* Navigational methods */
    navigateToHome():promise.Promise<any> {
        return browser.get('/');
    }

    navigateToAbout():promise.Promise<any>  {
        return browser.get('/about');
    }

    navigateToContact():promise.Promise<any>  {
        return browser.get('/contact');
    }

    /* Mock data for creating a new Paste and editing existing paste */

    getMockPaste(): any {
        let paste: any = { title: "Something  here",language: "Ruby",paste: "Tes
        return paste;
    }

    getEditedMockPaste(): any {
        let paste: any = { title: "Paste 2", language: "JavaScript", paste: "Tes
        return paste;
    }

    /* Methods shared by addPaste and viewPaste */

    getInputTitle():ElementFinder {
        return element(by.name("title"));
    }

    getInputLanguage(): ElementFinder {
        return element(by.name("language"));
    }

    getInputPaste(): ElementFinder {
        return element(by.name("paste"));

    }
}
```

Add Paste Page Object

Blueprint for the AddPaste component

```
/* Path: e2e/page-objects/add-paste.po.ts */

import { browser, by, element, promise, ElementFinder, ElementArrayFinder } from
import { Base } from './base.po';
export class AddPaste extends Base  {

    getAddPaste():ElementFinder {
        return element(by.tagName('app-add-paste'));
    }

    /* Create Paste button */
    getCreateButton(): ElementFinder {
        return this.getAddPaste().element(by.buttonText("create Paste"));
    }

    isCreateButtonPresent() : promise.Promise<boolean> {
        return this.getCreateButton().isPresent();
    }

    clickCreateButton(): promise.Promise<void> {

```

```
        return this.getCreateButton().click();
    }

    /*Create Paste Modal */

    getCreatePasteModal(): ElementFinder {
        return this.getAddPaste().element(by.id("source-modal"));
    }

    isCreatePasteModalPresent() : promise.Promise<boolean> {
        return this.getCreatePasteModal().isPresent();
    }

    /*Save button */
    getSaveButton(): ElementFinder {
        return this.getAddPaste().element(by.buttonText("Save"));
    }

    clickSaveButton():promise.Promise<void> {
        return this.getSaveButton().click();
    }

    /*Close button */

    getCloseButton(): ElementFinder {
        return this.getAddPaste().element(by.buttonText("Close"));
    }

    clickCloseButton():promise.Promise<void> {
        return this.getCloseButton().click();
    }


    /* Get Input Paste values from the Modal window */
    getInputPasteValues(): Promise<string[]> {
        let inputTitle, inputLanguage, inputPaste;

        // Return the input values after the promise is resolved
        // Note that this.getInputTitle().getText doesn't work
        // Use getAttribute('value') instead
        return Promise.all([this.getInputTitle().getAttribute("value"), this.get
        .then( (values) => {
            return values;
        });

    }

    /* Add a new Paste */

    addNewPaste():any {
        let newPaste: any = this.getMockPaste();
```

```
73
74          //Send input values
75          this.getInputTitle().sendKeys(newPaste.title);
76          this.getInputLanguage()
77              .element(by.cssContainingText('option', newPaste.language)).click();
78          this.getInputPaste().sendKeys(newPaste.paste);
79
80          //Convert the paste object into an array
81          return Object.keys(newPaste).map(key => newPaste[key]);
82
83      }
84
}
```

Add Paste Spec File

```
01  /* Path: e2e/addNewPaste.e2e-spec.ts */
02
03  import { Pastebin } from './page-objects/pastebin.po';
04  import { AddPaste } from './page-objects/add-paste.po';
05  import { browser, protractor } from 'protractor';
06
07  /* Scenarios to be Tested
08    1. AddPaste Page should have a button when clicked on should present a modal w
09    2. The modal window should accept the new values and save them
10    4. The saved data should appear in the MainPage
11    3. Close button should work
12  */
13
14  describe('Add-New-Paste page', () => {
15
16    const addPastePage: AddPaste = new AddPaste();
17    const mainPage: Pastebin = new Pastebin();
18
19    beforeEach(() => {
20
21      addPastePage.navigateToHome();
22    });
23
24    it('should have an Create Paste button and modal window', () => {
25
26      expect(addPastePage.isCreateButtonPresent()).toBeTruthy("The button should e
27      expect(addPastePage.isCreatePasteModalPresent()).toBeFalsy("The modal windov
28
29      addPastePage.clickCreateButton();
30
31      expect(addPastePage.isCreatePasteModalPresent()).toBeTruthy("The modal windc
32
33
34    });
```

```
35
36    it("should accept and save input values", () => {
37
38      addPastePage.clickCreateButton();
39
40      const emptyInputValues = ["","",""];
41      expect(addPastePage.getInputPasteValues()).toEqual(emptyInputValues);
42
43      const newInputValues = addPastePage.addNewPaste();
44      expect(addPastePage.getInputPasteValues()).toEqual(newInputValues);
45
46      addPastePage.clickSaveButton();
47
48      expect(addPastePage.isCreatePasteModalPresent()).toBeFalsy("The modal window
49      expect(mainPage.getLastRowData()).toContain("Something here");
50
51    });
52
53    it("close button should work", () => {
54
55      addPastePage.clickCreateButton();
56      addPastePage.clickCloseButton();
57
58      expect(addPastePage.isCreatePasteModalPresent()).toBeFalsy("The modal window
59
60    });
61
62  });
```

Exercises

There are a couple of things missing, though: the tests for the **View Paste** button and the modal window that pops up after clicking the button. I am going to leave this as an exercise for you. However, I will drop you a hint.

The structure of the page objects and the specs for the ViewPastePage are similar to that of the AddPastePage.

Blueprint for the ViewPaste component

Here are the scenarios that you need to test:

1. ViewPaste Page should have a button, and on click, it should bring up a modal window.
2. The modal window should display the paste data of the recently added paste.
3. The modal window should let you update values.
4. The delete button should work.

Try to stick to the guidelines wherever possible. If you're in doubt, switch to the final branch to see the final draft of the code.

Wrapping It Up

So there you have it. In this article, we've covered writing end-to-end tests for our Angular application using Protractor. We started off with a discussion about unit tests vs. e2e tests, and then we learned about setting up, configuring and running Protractor. The rest of the tutorial concentrated on writing actual tests for the demo Pastebin application.

Please let me know your thoughts and experiences about writing tests using Protractor or writing tests for Angular in general. I would love to hear them. Thanks for reading!

Manjunath M
Kerala, India

Founder of Storylens - A content publishing platform for the devs. Amateur musician. Favorite quote? "Being a jack of all trades doesn't mean you're a master at none."

blizzerand

Cji • 10 months ago

Hi there,
I have found this page is very useful for my current project. But I have some extra questions if you could help.
1. I have used selenium/testNG a lot, regarding the protractor, what is the way to generate the report regarding passed and failed test cases ?
2. How about screenshot ?
3. Do you know a group where we can always discuss the protractor problems ?

Thanks a lot for the help.

CJI