Protractor is an end-to-end test framework for Angular and AngularJS applications. Protractor runs tests against your application running in a real browser, interacting with it as a user would.

**Test Like a User**

Protractor is built on top of WebDriverJS, which uses native events and browser-specific drivers to interact with your application as a user would.

**For Angular Apps**

Protractor supports Angular-specific locator strategies, which allows you to test Angular-specific elements without any setup effort on your part.

**Automatic Waiting**

You no longer need to add waits and sleeps to your test. Protractor can automatically execute the next step in your test the moment the webpage finishes pending tasks, so you don’t have to worry about waiting for your test and webpage to sync.

**Setup**

Use npm to install Protractor globally with:

npm install -g protractor

This will install two command line tools, protractor and webdriver-manager. Try running protractor --version to make sure it's working.

The webdriver-manager is a helper tool to easily get an instance of a Selenium Server running. Use it to download the necessary binaries with:

webdriver-manager update

Now start up a server with:

webdriver-manager start

This will start up a Selenium Server and will output a bunch of info logs. Your Protractor test will send requests to this server to control a local browser. You can see information about the status of the server at <http://localhost:4444/wd/hub>.

**Write a test**

Open a new command line or terminal window and create a clean folder for testing.

Protractor needs two files to run, a spec file and a configuration file.

Let's start with a simple test that navigates to the todo list example in the AngularJS website and adds a new todo item to the list.

Copy the following into todo-spec.js:

describe('angularjs homepage todo list', function() {

it('should add a todo', function() {

browser.get('https://angularjs.org');

element(by.model('todoList.todoText')).sendKeys('write first protractor test');

element(by.css('[value="add"]')).click();

var todoList = element.all(by.repeater('todo in todoList.todos'));

expect(todoList.count()).toEqual(3);

expect(todoList.get(2).getText()).toEqual('write first protractor test');

// You wrote your first test, cross it off the list

todoList.get(2).element(by.css('input')).click();

var completedAmount = element.all(by.css('.done-true'));

expect(completedAmount.count()).toEqual(2);

});

});

The describe and it syntax is from the Jasmine framework. browser is a global created by Protractor, which is used for browser-level commands such as navigation with browser.get.

**Configuration**

Now create the configuration file. Copy the following into conf.js:

exports.config = {

seleniumAddress: 'http://localhost:4444/wd/hub',

specs: ['todo-spec.js']

};

This configuration tells Protractor where your test files (specs) are, and where to talk to your Selenium Server (seleniumAddress). It will use the defaults for all other configuration. Chrome is the default browser.

**Run the test**

Now run the test with:

protractor conf.js

You should see a Chrome browser window open up and navigate to the todo list in the AngularJS page, then close itself (this should be very fast!). The test output should be 1 test, 3 assertions, 0 failures. Congratulations, you've run your first Protractor test!

# **Testing Classes & Pipes**

## Sample class & test suite

We’ll start our unit testing journey with all you will ever need to know, how to test a class.

#### Tip

Everything in Angular is an instance of a class, be it a Component, Directive, Pipe and so on. So once you know how to test a basic class you can test everything.

Let’s imagine we have a simple class called AuthService it’s something we want to provide to Angulars DI framework but that doesn’t play a part in how we want to test it.

Listing 1. app/auth.service.ts

TypeScript

export class AuthService {

isAuthenticated(): boolean {

return !!localStorage.getItem('token');

}

}

It has one function called isAuthenticated which returns true if there is a token stored in the browsers localStorage.

To test this class we create a test file called auth.service.spec.ts that sits next to our auth.service.ts file, like so:

Listing 2. app/auth.service.spec.ts

TypeScript

import {AuthService} from './auth.service'; (1)

describe('Service: Auth', () => { (2)

});

|  |  |
| --- | --- |
|  | We first import the AuthService class we want to run our tests against. |
|  | We add a describe test suite function to hold all our individual test specs. |

## Setup & teardown

We want to run our test specs against fresh instances of AuthService so we use the beforeEach and afterEach functions to setup and clean instances like so:

Listing 3. app/auth.service.spec.ts

TypeScript

describe('Service: Auth', () => {

let service: AuthService;

beforeEach(() => { (1)

service = new AuthService();

});

afterEach(() => { (2)

service = null;

localStorage.removeItem('token');

});

});

|  |  |
| --- | --- |
|  | Before each test spec is run we create a new instance of AuthService and store on the service variable. |
|  | After each test spec is finished we null out our service and also remove any tokens we stored in localStorage. |

## Creating test specs

Now we create some test specs, the first spec I want to create should check if the isAuthenticated function returns true when there is a token.

TypeScript

it('should return true from isAuthenticated when there is a token', () => { (1)

localStorage.setItem('token', '1234'); (2)

expect(service.isAuthenticated()).toBeTruthy(); (3)

});

|  |  |
| --- | --- |
|  | We pass to the it function a human readable description of what we are testing. This is shown in the test report and makes it easy to understand what feature isn’t working. |
|  | We setup some spec only data in local storage which should trigger the effect we want. |
|  | We test an expectation that the service.isAuthenticated() function returns something that resolves to true. |

We also want to test the reverse case, when there is no token the function should return false:

TypeScript

it('should return false from isAuthenticated when there is no token', () => {

expect(service.isAuthenticated()).toBeFalsy();

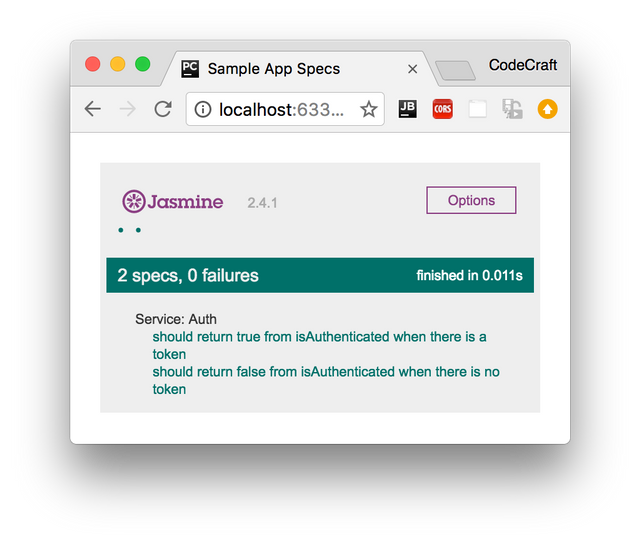
});

We know that in this function the token isn’t set since we make sure to clear out the token in the afterEach function.

We now test an expectation that the service.isAuthenticated() function returns something that resolves to false.

## Running the tests

To run our tests we simply open up the HTML file in the browser, you can just click the plunker link and make sure to press run in the toolbar.



## Pipes

Pipes are by far the simplest part of Angular, they can be implemented as a class with one function and therefore can be tested with just Jasmine and the knowledge we’ve gained so far.

In the section on pipes we built one called DefaultPipe, this pipe lets us provide default values for variables in templates like so:

{{ image | default:"http://example.com/default-image.png" }}

The code for this pipe looked like so:

TypeScript

import {Pipe, PipeTransform} from '@angular/core';

@Pipe({

name: 'default'

})

export class DefaultPipe implements PipeTransform {

transform(value: string, fallback: string, forceHttps: boolean = false): string {

let image = "";

if (value) {

image = value;

} else {

image = fallback;

}

if (forceHttps) {

if (image.indexOf("https") == -1) {

image = image.replace("http", "https");

}

}

return image;

}

}

Our starting test suite file looks like so:

TypeScript

describe('Pipe: Default', () => {

let pipe: DefaultPipe;

beforeEach(() => {

pipe = new DefaultPipe();

});

});

In our setup function we create an instance of our pipe class.

Pipe classes have one function called transform so in order to test pipes we just need to test this one function, passing inputs and expecting outputs.

Our first test spec checks to see that if the pipe doesn’t recieve an input it returns the default value, like so:

TypeScript

it('providing no value returns fallback', () => {

expect(pipe.transform('', 'http://place-hold.it/300')).toBe('http://place-hold.it/300');

});

We pass in empty string as the input to the transform function and therefore it returns the second argument back to us.

For testing pipes there isn’t much else to it, we simply check the various inputs and expected ouputs of our transform function.

#### Note

In order to run this test spec file in our test Plunker remember to add it to the list of test spec files in the spec\_files array.

#### Tip

If your Pipe requires dependencies to be injected into the constructor it might be better to use the Angular Test Bed which we cover later on in this section.

## Summary

That’s it really, we can test any isolated class that doesn’t require anything else with a simple jasmine spec file, nothing more complex required.

Since everything in Angular is represented as classes, we could stop here — you have most of the tools already to write tests for directives, components, pipes and so on.

However our code often requires other code to work, it has dependencies. So how we write isolated tests for pieces of code which by nature are not isolated and need dependencies is the topic of the next lecture.

## Listing

Listing 4. auth.service.ts

TypeScript

export class AuthService {

isAuthenticated(): boolean {

return !!localStorage.getItem('token');

}

}

Listing 5. auth.service.spec.ts

TypeScript

import { AuthService } from './auth.service';

describe('Service: Auth', () => {

let service: AuthService;

beforeEach(() => {

service = new AuthService();

});

afterEach(() => {

service = null;

localStorage.removeItem('token');

});

it('should return true from isAuthenticated when there is a token', () => {

localStorage.setItem('token', '1234');

expect(service.isAuthenticated()).toBeTruthy();

});

it('should return false from isAuthenticated when there is no token', () => {

expect(service.isAuthenticated()).toBeFalsy();

});

});

Listing 6. default.pipe.ts

TypeScript

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'default'

})

export class DefaultPipe implements PipeTransform {

transform(value: string, fallback: string, forceHttps: boolean = false): string {

let image = "";

if (value) {

image = value;

} else {

image = fallback;

}

if (forceHttps) {

if (image.indexOf("https") == -1) {

image = image.replace("http", "https");

}

}

return image;

}

}

Listing 7. default.pipe.spec.ts

TypeScript

/\* tslint:disable:no-unused-variable \*/

import { DefaultPipe } from './default.pipe';

describe('Pipe: Default', () => {

let pipe: DefaultPipe;

beforeEach(() => {

pipe = new DefaultPipe();

});

it('providing no value returns fallback', () => {

expect(pipe.transform('', 'http://place-hold.it/300')).toBe('http://place-hold.it/300');

});

it('providing a value returns value', () => {

expect(pipe.transform('http://place-hold.it/300', 'fallback')).toBe('http://place-hold.it/300');

});

it('asking for https returns https', () => {

expect(pipe.transform('', 'http://place-hold.it/300', true)).toBe('https://place-hold.it/300');

});

});