**Unit Testing in Angular**

We can test our Angular applications from scratch by writing and executing pure javascript functions. Creating instances of the relevant classes, calling functions and checking the actual versus expected result.

But since testing is such a common activity with javascript there are a number of testing libraries and frameworks we can use which reduce the amount of time it takes to write tests.

Two such tools and frameworks that are used when testing Angular is Jasmine and Karma and a discussion of those is the topic of this lecture.

Learning Objectives

* What is the Jasmine test framework?
* How to write tests in Jasmine?
* What is the Karma test runner?
* How to create and run tests using the Angular CLI?
* How to create and run tests in Plunker?

## Jasmine

Jasmine is a javascript testing framework that supports a software development practice called Behaviour Driven Development, or BDD for short. It’s a specific flavour of Test Driven Development (TDD).

Jasmine, and BDD in general, attempts to describe tests in a human readable format so that non-technical people can understand what is being tested. However even if you are technical reading tests in BDDformat makes it a lot easier to understand what’s going on.

For example if we wanted to test this function:

**function** helloWorld() {

**return** 'Hello world';

}

We would write a jasmine test spec like so:

describe('hello check', () => {

it('says hello', () => {

expect(helloWorld()).toEqual('Hello world!');

});

});

|  |  |
| --- | --- |
|  | The describe(string, function) function defines what we call a Test Suite, a collection of individual Test Specs. |
|  | The it(string, function) function defines an individual Test Spec, this contains one or more Test Expectations. |
|  | The expect(actual) expression is what we call an Expectation. In conjunction with a Matcher it describes an expected piece of behaviour in the application. |
|  | The matcher(expected) expression is what we call a Matcher. It does a boolean comparison with the expected value passed in vs. the actual value passed to the expect function, if they are false the spec fails. |

### Built-in matchers

Jasmine comes with a few pre-built matchers like so:

expect(array).toContain(member);

expect(fn).toThrow(string);

expect(fn).toThrowError(string);

expect(test()).toBe(instance);

expect(mixed).toBeDefined();

expect(mixed).toBeFalsy();

expect(mixed).toBeNull();

expect(mixed).toBeTruthy();

expect(mixed).toBeUndefined();

expect(mixed).toEqual(mixed);

expect(mixed).toMatch(pattern);

expect(number).toBeCloseTo(number, decimalPlaces);

expect(number).toBeGreaterThan(number);

expect(number).toBeLessThan(number);

expect(number).toBeNaN();

expect(spy).toHaveBeenCalled();

expect(spy).toHaveBeenCalledTimes(number);

expect(spy).toHaveBeenCalledWith(...arguments);

You can see concrete examples of how these matchers are used by looking at the Jasmine docs here: http://jasmine.github.io/edge/introduction.html#section-Included\_Matchers

### Setup and teardown

Sometimes in order to test a feature we need to perform some setup, perhaps it’s creating some test objects. Also we may need to perform some cleanup activities after we have finished testing, perhaps we need to delete some files from the hard drive.

These activities are called setup and teardown (for cleaning up) and Jasmine has a few functions we can use to make this easier:

**beforeAll**

This function is called **once**, before all the specs in describe test suite are run.

**afterAll**

This function is called **once** after all the specs in a test suite are finished.

**beforeEach**

This function is called before **each** test specification, it function, has been run.

**afterEach**

This function is called after **each** test specification has been run.

We might use these functions like so:

describe('Hello world', () => {

**let** expected = "";

beforeEach(() => {

expected = "Hello World";

});

afterEach(() => {

expected = "";

});

it('says hello', () => {

expect(helloWorld()).toEqual(expected);

});

});

### Running Jasmine tests

To manually run Jasmine tests we would create a HTML file and include the required jasmine javascript and css files like so:

<**link** rel="stylesheet" href="jasmine.css">

<**script** src="jasmine.js"></**script**>

<**script** src="jasmine-html.js"></**script**>

<**script** src="boot.js"></**script**>

We then load in the parts of our application code that we want to test, for example if our hello worldfunction was in main.js:

<**link** rel="stylesheet" href="jasmine.css">

<**script** src="jasmine.js"></**script**>

<**script** src="jasmine-html.js"></**script**>

<**script** src="boot.js"></**script**>

<**script** src="main.js"></**script**>

We then would load each individual test suite file, for example if we placed our test suite code above in a file called test.js we would load it in like so:

Copy<**link** rel="stylesheet" href="jasmine.css">

<**script** src="jasmine.js"></**script**>

<**script** src="jasmine-html.js"></**script**>

<**script** src="boot.js"></**script**>

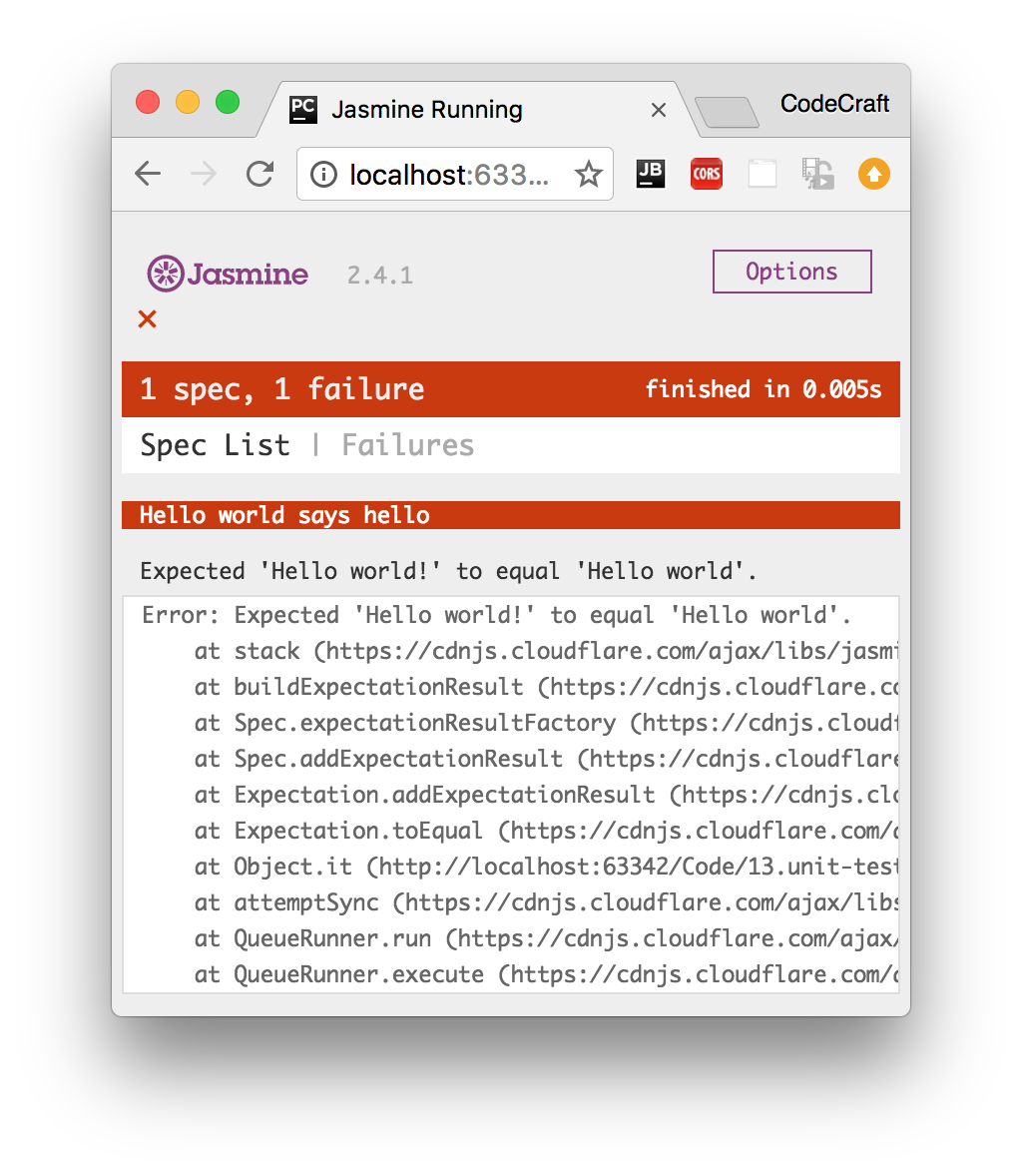
<**script** src="main.js"></**script**>

<**script** src="test.js"></**script**>

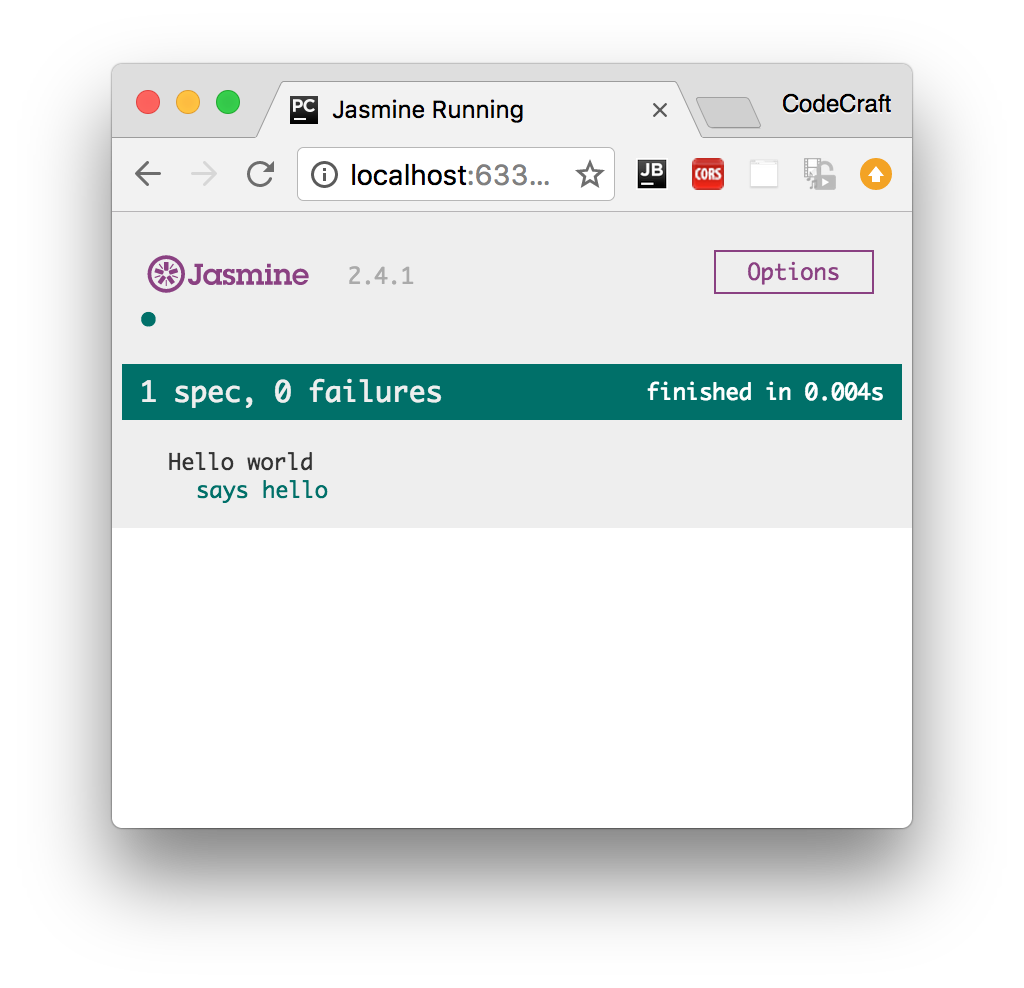
To run the tests we simply open up the HTML file in a browser.

Once all the files requested via script and link are loaded by a browser the function window.onload is called, this is when Jasmine actually runs the tests.

The results are displayed in the browser window, a failing run looks like:



A passing run looks like:



If we wanted to test our code in different browsers we simply load up the HTML file in the browser we want to test in.

If we wanted to debug the code we would use the developer tools available to us in that browser.

## Karma

Manually running Jasmine tests by refreshing a browser tab repeatedly in different browsers every-time we edit some code can become tiresome.

Karma is a tool which lets us spawn browsers and run jasmine tests inside of them all from the command line. The results of the tests are also displayed on the command line.

Karma can also watch your development files for changes and re-run the tests automatically.

Karma lets us run jasmine tests as part of a development tool chain which requires tests to be runnable and results inspectable via the command line.

It’s not necessary to know the internals of how Karma works. When using the Angular CLI it handles the configuration for us and for the rest of this section we are going to run the tests using only Jasmine.

## Angular CLI

When creating Angular projects using using the Angular CLI it defaults to creating and running unit tests using Jasmine and Karma.

Whenever we create files using the CLI as well as creating the main code file it also creates simple jasmine spec file named the same as the main code file but ending in .spec.ts, like so:

If we create a Pipe using the CLI like so:

ng generate pipe My

This would create two files:

* my-pipe.ts — This is the main code file where we put the code for the pipe.
* my-pipe.spec.ts — This is the jasmine test suite for the pipe.

The spec file will have some code already bootstrapped, like so:

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

**import** { TestBed, async } from '@angular/core/testing';

**import** { MyPipe } from './my.pipe';

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

it('create an instance', () => {

**let** pipe = **new** MyPipe();

expect(pipe).toBeTruthy();

});

});

#### Note

The code that gets bootstrapped depends on the item that we are creating.

To run all the tests in our application we simply type ng test in our project root.

This runs all the tests in our project in Jasmine via Karma.

It watches for changes to our development files, bundles all the developer files together and re-runsthe tests automatically.

## Angular Plunker

When building real Angular applications I recommend sticking with the file and folder structure defined by the Angular CLI as well as using the built-in test runner.

However for this section, to give you an easy way to view and play with the code we are going to use onlyJasmine and execute tests by refreshing a browser.

This so we can easily share code via a Plunker like we have done for all the other sections in this course.

An Angular Jasmine Plunker looks very similar to a normal Jasmine Plunker appart from a few key differences:

1. We also include the required Angular libraries and some patches for Jasmine so it works better with Angular.

Copy<**script** src="https://unpkg.com/systemjs@0.19.27/dist/system.src.js"></**script**>

<**script** src="https://unpkg.com/reflect-metadata@0.1.3"></**script**>

<**script** src="https://unpkg.com/zone.js@0.6.25?main=browser"></**script**>

<**script** src="https://unpkg.com/zone.js/dist/long-stack-trace-zone.js?main=browser"></**script**>

<**script** src="https://unpkg.com/zone.js/dist/proxy.js?main=browser"></**script**>

<**script** src="https://unpkg.com/zone.js/dist/sync-test.js?main=browser"></**script**>

<**script** src="https://unpkg.com/zone.js/dist/jasmine-patch.js?main=browser"></**script**>

<**script** src="https://unpkg.com/zone.js/dist/async-test.js?main=browser"></**script**>

<**script** src="https://unpkg.com/zone.js/dist/fake-async-test.js?main=browser"></**script**>

1. We then add the spec files we want to test into a special array called spec\_files.

Copyvar \_\_spec\_files\_\_ = [

'app/auth.service.spec'

];

1. We then load a shim javascript file which triggers running the test specs once we have finished transpiling and loading the transpiled files in the browser.

Copy<**script** src="browser-test-shim.js"></**script**>

### Disabled and focused tests

You can disable tests without commenting them our by just pre-pending x to the describe or itfunctions, like so:

Copyxdescribe('Hello world', () => {

xit('says hello', () => {

expect(helloWorld())

.toEqual('Hello world!');

});

});

|  |  |
| --- | --- |
|  | These tests will not be run. |

Conversely you can also focus on specific tests by pre-pending with f, like so:

Copyfdescribe('Hello world', () => {

fit('says hello', () => {

expect(helloWorld())

.toEqual('Hello world!');

});

});

|  |  |
| --- | --- |
|  | Out of all the tests in all the tests suites and tests specs, these are the only ones that will be run. |

## Summary

Jasmine is a testing framework that supports Behavior Driven Development. We write tests in Test Suiteswhich are composed of one or more Test Specs which themselves are composed of one or more Test Expectations.

We can run Jasmine tests in a browser ourselves by setting up and loading a HTML file, but more commonly we use a command line tool called Karma. Karma handles the process of creating HTML files, opening browsers and running tests and returning the results of those tests to the command line.

If you use the Angular CLI to manage projects it automatically creates stub jasmine spec files for you when generating code. It also handles the Karama configuration, transpilation and bundling of your files so all you need to do in order to run your tests is type the command ng test.

For the purposes of this section we will be using a simple browser based Jasmine test runner so we can share the code easily via plunker.

## Listing

http://plnkr.co/edit/8ApdkvletoEc4Q0maXSN?p=preview

*Listing 1. index.html*

Copy*<!-- Run application specs in a browser -->*

<!DOCTYPE html>

<**html**>

<**head**>

<**title**>Jasmine Running</**title**>

<**meta** charset="UTF-8">

<**meta** name="viewport" content="width=device-width, initial-scale=1">

<**link** rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/jasmine.css">

</**head**>

<**body**>

<**script** src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/jasmine.js"></**script**>

<**script** src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/jasmine-html.js"></**script**>

<**script** src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/boot.js"></**script**>

<**script** src="main.js"></**script**>

<**script** src="test.js"></**script**>

</**body**>

</**html**>

*Listing 2. main.js*

Copy**function** helloWorld() {

**return** 'Hello world!';

}

*Listing 3. test.js*

Copydescribe('Hello world', () => {

**let** expected = "";

beforeEach(() => {

expected = "Hello world!";

});

afterEach(() => {

expected = "";

});

it('says hello', () => {

expect(helloWorld())

.toEqual(expected);

});

});

# 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*

Copy**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.tsfile, like so:

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

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

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

});

|  |  |
| --- | --- |
|  | 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 beforeEachand afterEach functions to setup and clean instances like so:

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

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

**let** service: AuthService;

beforeEach(() => {

service = **new** AuthService();

});

afterEach(() => {

service = **null**;

localStorage.removeItem('token');

});

});

|  |  |
| --- | --- |
|  | Before each test spec is run we create a new instance of AuthService and store on the servicevariable. |
|  | 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 isAuthenticatedfunction returns true when there is a token.

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

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

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

});

|  |  |
| --- | --- |
|  | 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:

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

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

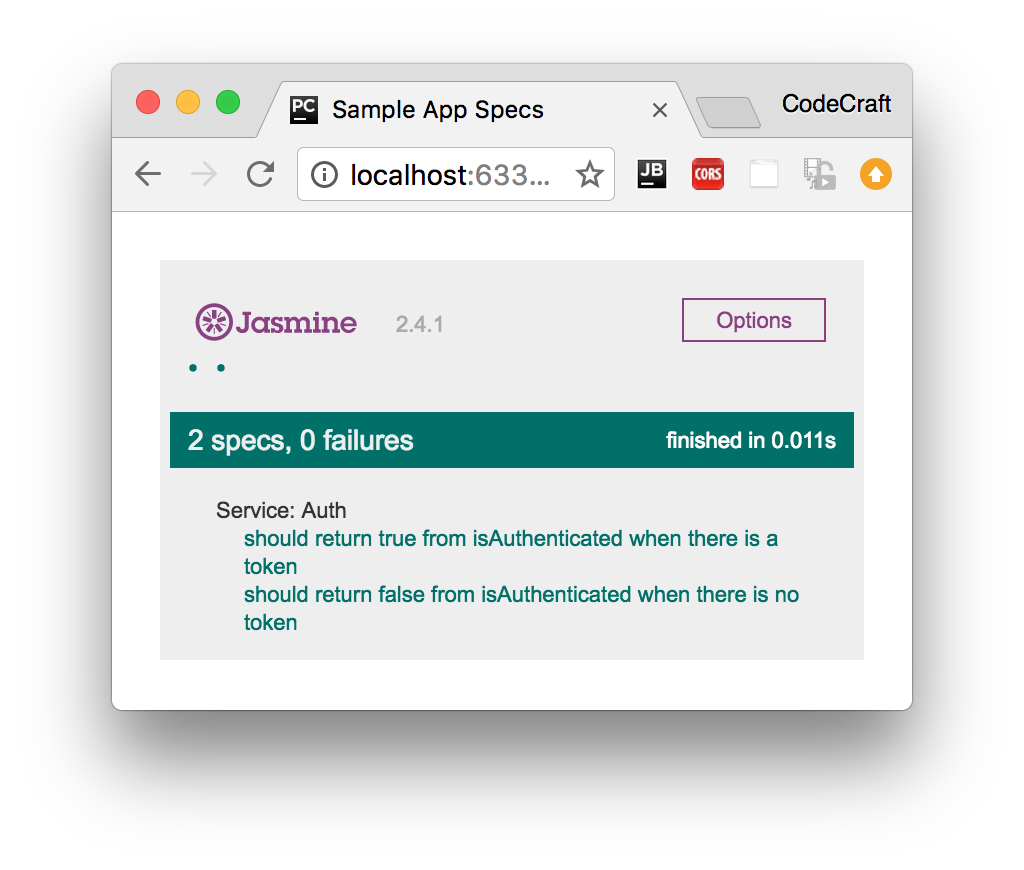
});

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:

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

The code for this pipe looked like so:

Copy**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:

Copydescribe('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:

Copyit('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*

Copy**export** **class** AuthService {

isAuthenticated(): **boolean** {

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

}

}

*Listing 5. auth.service.spec.ts*

Copy**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*

Copy**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*

Copy*/\* 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');

});

});

# Testing with Mocks & Spies

## Sample code

Let’s imagine we have a LoginComponent which works with the AuthService we tested in the previous lecture, like so:

*Listing 1. login.component.ts*

Copy**import** {Component} from '@angular/core';

**import** {AuthService} from "./auth.service";

**@Component**({

selector: 'app-login',

template: `<a [hidden]="needsLogin()">Login</a>`

})

**export** **class** LoginComponent {

**constructor**(**private** auth: AuthService) {

}

needsLogin() {

**return** !**this**.auth.isAuthenticated();

}

}

We inject the AuthService into the LoginComponent and the component shows a Login button if the AuthService says the user isn’t authenticated.

The AuthService is the same as the previous lecture:

*Listing 2. auth.service.ts*

Copy**export** **class** AuthService {

isAuthenticated(): **boolean** {

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

}

}

## Testing with the real AuthService

We could test the LoginComponent by using a real instance of AuthService but if you remember to trickAuthService into returning true for the isAuthenticated function we needed to setup some data via localStorage.

Copy**import** {LoginComponent} from './login.component';

**import** {AuthService} from "./auth.service";

describe('Component: Login', () => {

**let** component: LoginComponent;

**let** service: AuthService;

beforeEach(() => {

service = **new** AuthService();

component = **new** LoginComponent(service);

});

afterEach(() => {

localStorage.removeItem('token');

service = **null**;

component = **null**;

});

it('canLogin returns false when the user is not authenticated', () => {

expect(component.needsLogin()).toBeTruthy();

});

it('canLogin returns false when the user is not authenticated', () => {

localStorage.setItem('token', '12345');

expect(component.needsLogin()).toBeFalsy();

});

});

|  |  |
| --- | --- |
|  | We create an instance of AuthService and inject it into out LoginComponent when we create it. |
|  | We clean up data and localStorage after each test spec has been run. |
|  | We setup some data in localStorage in order to get the behaviour we want from AuthService. |

So in order to test LoginComponent we would need to know the inner workings of AuthService.

That’s not very isolated but also not too much to ask for in this scenario. However imagine if LoginComponent required a number of other dependencies in order to run, we would need to know the inner workings of a number of other classes just to test our LoginComponent.

This results in Tight Coupling and our tests being very Brittle, i.e. likely to break easily. For example if the AuthService changed how it stored the token, from localStorage to cookies then the LoginComponent test would break since it would still be setting the token via localStorage.

This is why we need to test classes in isolation, we just want to worry about LoginComponent and not about the myriad of other things LoginComponent depends on.

We achieve this by Mocking our dependencies. Mocking is the act of creating something that looks like the dependency but is something we control in our test. There are a few methods we can use to create mocks.

## Mocking with fake classes

We can create a fake AuthService called MockedAuthService which just returns whatever we want for our test.

We can even remove the AuthService import if we want, there really is no dependency on anything else. The LoginComponent is tested in isolation:

Copy**import** {LoginComponent} from './login.component';

**class** MockAuthService {

authenticated = **false**;

isAuthenticated() {

**return** **this**.authenticated;

}

}

describe('Component: Login', () => {

**let** component: LoginComponent;

**let** service: MockAuthService;

beforeEach(() => {

service = **new** MockAuthService();

component = **new** LoginComponent(service);

});

afterEach(() => {

service = **null**;

component = **null**;

});

it('canLogin returns false when the user is not authenticated', () => {

service.authenticated = **false**;

expect(component.needsLogin()).toBeTruthy();

});

it('canLogin returns false when the user is not authenticated', () => {

service.authenticated = **true**;

expect(component.needsLogin()).toBeFalsy();

});

});

|  |  |
| --- | --- |
|  | We create a class called MockAuthService which has the same isAuthenticated function as the real AuthService class. The one difference is that we can control what isAuthenticated returns by setting the value of the authenticated property. |
|  | We inject into our LoginComponent an instance of the MockAuthService instead of the real AuthService. |
|  | In our tests we trigger the behaviour we want from the service by setting the authenticatedproperty. |

By using a fake MockAuthService we:

* Don’t depend on the real AuthService, in fact we don’t even need to import it into our specs.
* Make our code less brittle, if the inner workings of the real AuthService ever changes our tests will still be valid and still work.

## Mocking by overriding functions

Sometimes creating a complete fake copy of a real class can be complicated, time consuming and unnecessary.

We can instead simply extend the class and override one or more specific function in order to get them to return the test responses we need, like so:

Copy**class** MockAuthService **extends** AuthService {

authenticated = **false**;

isAuthenticated() {

**return** **this**.authenticated;

}

}

In the above class MockAuthService extends AuthService. It would have access to all the other functions and properties that exist on AuthService but only override the isAuthenticated function so we can easily control it’s behaviour and isolate our LoginComponent test.

#### Note

The rest of the test suite using mocking via overriding functions is the same as the previous version with fake classes.

## Mock by using a real instance with Spy

A Spy is a feature of Jasmine which lets you take an existing class, function, object and mock it in such a way that you can control what gets returned from functions.

Let’s re-write our test to use a Spy on a real instance of AuthService instead, like so:

Copy**import** {LoginComponent} from './login.component';

**import** {AuthService} from "./auth.service";

describe('Component: Login', () => {

**let** component: LoginComponent;

**let** service: AuthService;

**let** spy: any;

beforeEach(() => {

service = **new** AuthService();

component = **new** LoginComponent(service);

});

afterEach(() => {

service = **null**;

component = **null**;

});

it('canLogin returns false when the user is not authenticated', () => {

spy = spyOn(service, 'isAuthenticated').and.returnValue(**false**);

expect(component.needsLogin()).toBeTruthy();

expect(service.isAuthenticated).toHaveBeenCalled();

});

it('canLogin returns false when the user is not authenticated', () => {

spy = spyOn(service, 'isAuthenticated').and.returnValue(**true**);

expect(component.needsLogin()).toBeFalsy();

expect(service.isAuthenticated).toHaveBeenCalled();

});

});

|  |  |
| --- | --- |
|  | We create a real instance of AuthService and inject it into the LoginComponent. |
|  | In our teardown function there is no need to delete the token from localStorage. |
|  | We create a spy on our service so that if the isAuthenticated function is called it returns false. |
|  | We can even check to see if the isAuthenticated function was called. |

By using the spy feature of jasmine we can make any function return anything we want:

CopyspyOn(service, 'isAuthenticated').and.returnValue(**false**);

In our example above we make the isAuthenticated function return false or true in each test spec according to our needs.

## Summary

Testing with real instances of dependencies causes our test code to know about the inner workings of other classes resulting in tight coupling and brittle code.

The goal is to test pieces of code in isolation without needing to know about the inner workings of their dependencies.

We do this by creating Mocks; we can create Mocks using fake classes, extending existing classes or by using real instances of classes but taking control of them with Spys.

## Listing

http://plnkr.co/edit/08ppx8olCnTMpkPdW3eC?p=preview

*Listing 3. login.component.ts*

Copy**import** {Component} from '@angular/core';

**import** {AuthService} from "./auth.service";

**@Component**({

selector: 'app-login',

template: `<a [hidden]="needsLogin()">Login</a>`

})

**export** **class** LoginComponent {

**constructor**(**private** auth: AuthService) {

}

needsLogin() {

**return** !**this**.auth.isAuthenticated();

}

}

*Listing 4. login.component.spec.ts*

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

**import** {LoginComponent} from './login.component';

**import** {AuthService} from "./auth.service";

describe('Component: Login', () => {

**let** component: LoginComponent;

**let** service: AuthService;

**let** spy: any;

beforeEach(() => {

service = **new** AuthService();

component = **new** LoginComponent(service);

});

afterEach(() => {

service = **null**;

component = **null**;

});

it('canLogin returns false when the user is not authenticated', () => {

spy = spyOn(service, 'isAuthenticated').and.returnValue(**false**);

expect(component.needsLogin()).toBeTruthy();

expect(service.isAuthenticated).toHaveBeenCalled();

});

it('canLogin returns false when the user is not authenticated', () => {

spy = spyOn(service, 'isAuthenticated').and.returnValue(**true**);

expect(component.needsLogin()).toBeFalsy();

expect(service.isAuthenticated).toHaveBeenCalled();

});

});

# Angular Test Bed

## Configuring

Lets demonstrate how to use the ATB by converting the component we tested with plain vanilla Jasmine to one that uses the ATB.

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

**import** {TestBed, ComponentFixture} from '@angular/core/testing';

**import** {LoginComponent} from './login.component';

**import** {AuthService} from "./auth.service";

describe('Component: Login', () => {

beforeEach(() => {

TestBed.configureTestingModule({

declarations: [LoginComponent],

providers: [AuthService]

});

});

});

In the beforeEach function for our test suite we configure a testing module using the TestBed class.

This creates a test Angular Module which we can use to instantiate components, perform dependency injection and so on.

We configure it in exactly the same way as we would configure a normal NgModule. On this case we pass in the LoginComponent in the declarations and the AuthService in the providers.

## Fixtures and DI

Once the ATB is setup we can then use it to instantiate components and resolve dependencies, like so:

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

**import** {TestBed, ComponentFixture} from '@angular/core/testing';

**import** {LoginComponent} from './login.component';

**import** {AuthService} from "./auth.service";

describe('Component: Login', () => {

**let** component: LoginComponent;

**let** fixture: ComponentFixture<LoginComponent>;

**let** authService: AuthService;

beforeEach(() => {

TestBed.configureTestingModule({

declarations: [LoginComponent],

providers: [AuthService]

});

*// create component and test fixture*

fixture = TestBed.createComponent(LoginComponent);

*// get test component from the fixture*

component = fixture.componentInstance;

*// UserService provided to the TestBed*

authService = TestBed.get(AuthService);

});

});

|  |  |
| --- | --- |
|  | A fixture is a wrapper for a component and it’s template. |
|  | We create an instance of a component fixture through the TestBed, this injects the AuthServiceinto the component constructor. |
|  | We can find the actual component from the componentInstance on the fixture. |
|  | We can get resolve dependencies using the TestBed injector by using the get function. |

#### Note

Since the LoginComponent doesn’t have it’s own child injector the AuthService that gets injected in is the same one as we get from the TestBed above.

## Test specs

Now we’ve configured the TestBed and extracted the component and service we can run through the same test specs as before:

Copyit('canLogin returns false when the user is not authenticated', () => {

spyOn(authService, 'isAuthenticated').and.returnValue(**false**);

expect(component.needsLogin()).toBeTruthy();

expect(authService.isAuthenticated).toHaveBeenCalled();

});

it('canLogin returns false when the user is not authenticated', () => {

spyOn(authService, 'isAuthenticated').and.returnValue(**true**);

expect(component.needsLogin()).toBeFalsy();

expect(authService.isAuthenticated).toHaveBeenCalled();

});

## When to use ATB

We will continue to use ATB for the rest of this section because:

* It allows us to test the interaction of a directive or component with it’s template.
* It allows us to easily test change detection.
* It allows us to test and use Angulars DI framework,
* It allows us to test using the NgModule configuration we use in our application.
* It allows us to test user interaction via clicks & input fields

## Summary

The ATB lets us test parts of our code as if it is being run in the context of a real Angular app.

It’s usefulness will become more apparent in future lectures, the next one being how to use the ATB to test change detection and property binding.

## Listing

http://plnkr.co/edit/jqw3ddMQU7zPQg9KBXJE?p=preview

*Listing 1. login.component.spec.ts*

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

**import** {TestBed, ComponentFixture} from '@angular/core/testing';

**import** {LoginComponent} from './login.component';

**import** {AuthService} from "./auth.service";

describe('Component: Login', () => {

**let** component: LoginComponent;

**let** fixture: ComponentFixture<LoginComponent>;

**let** authService: AuthService;

beforeEach(() => {

*// refine the test module by declaring the test component*

TestBed.configureTestingModule({

declarations: [LoginComponent],

providers: [AuthService]

});

*// create component and test fixture*

fixture = TestBed.createComponent(LoginComponent);

*// get test component from the fixture*

component = fixture.componentInstance;

*// UserService provided to the TestBed*

authService = TestBed.get(AuthService);

});

it('canLogin returns false when the user is not authenticated', () => {

spyOn(authService, 'isAuthenticated').and.returnValue(**false**);

expect(component.needsLogin()).toBeTruthy();

expect(authService.isAuthenticated).toHaveBeenCalled();

});

it('canLogin returns false when the user is not authenticated', () => {

spyOn(authService, 'isAuthenticated').and.returnValue(**true**);

expect(component.needsLogin()).toBeFalsy();

expect(authService.isAuthenticated).toHaveBeenCalled();

});

});

# Testing Change Detection

## Setup

We’ll continue testing our LoginComponent from previous lectures but this time we’ll update the template so we have both a Login and Logout button like so:

Copy**@Component**({

selector: 'app-login',

template: `

<a>

<span \*ngIf="needsLogin()">Login</span>

<span \*ngIf="!needsLogin()">Logout</span>

</a>

`

})

**export** **class** LoginComponent {

**constructor**(**private** auth: AuthService) {

}

needsLogin() {

**return** !**this**.auth.isAuthenticated();

}

}

Our test spec file starts close to the version we had in the last lecture like so:

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

**import** {TestBed, async, ComponentFixture} from '@angular/core/testing';

**import** {LoginComponent} from './login.component';

**import** {AuthService} from "./auth.service";

**import** {DebugElement} from "@angular/core";

**import** {By} from "@angular/platform-browser";

describe('Component: Login', () => {

**let** component: LoginComponent;

**let** fixture: ComponentFixture<LoginComponent>;

**let** authService: AuthService;

**let** el: DebugElement;

beforeEach(() => {

*// refine the test module by declaring the test component*

TestBed.configureTestingModule({

declarations: [LoginComponent],

providers: [AuthService]

});

*// create component and test fixture*

fixture = TestBed.createComponent(LoginComponent);

*// get test component from the fixture*

component = fixture.componentInstance;

*// UserService provided to the TestBed*

authService = TestBed.get(AuthService);

*// get the "a" element by CSS selector (e.g., by class name)*

el = fixture.debugElement.query(By.css('a'));

});

});

|  |  |
| --- | --- |
|  | We’ve imported a few more classes that are needed when interacting with a components view, DebugElement and By. |
|  | We have another variable called el which holds something called a DebugElement. |
|  | We store a reference to a DOM element in our el variable. |

The fixture as well as holding an instance of the component also holds a reference to something called a DebugElement, this is a wrapper to the low level DOM element that represents the components view, via the debugElement property.

We can get references to other child nodes by querying this debugElement with a By class. The By class lets us query using a a number of methods, one is via a css class like we have in our example another way is to request by a type of directive like By.directive(MyDirective).

We request a reference to the a tag that exists in the components view, this is the button which either says Login or Logout depending on whether the AuthService says the user is authenticated or not.

We can find out the text content of the tag by calling el.nativeElement.textContent.trim(), we’ll be using that snippet in the test specs later on.

Lets now add a basic test spec like so:

Copyit('login button hidden when the user is authenticated', () => {

*// TODO*

});

## Detect Changes

The first expectation we place in our test spec might look a bit strange

Copyit('login button hidden when the user is authenticated', () => {

expect(el.nativeElement.textContent.trim()).toBe('');

});

We initially expect the text inside the a tag to be blank.

That’s because when Angular first loads no change detection has been triggered and therefore the view doesn’t show either the Login or Logout text.

fixture is a wrapper for our components environment so we can control things like change detection.

To trigger change detection we call the function fixture.detectChanges(), now we can update our test spec to:

Copyit('login button hidden when the user is authenticated', () => {

expect(el.nativeElement.textContent.trim()).toBe('');

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

});

Once we trigger a change detection run Angular checks property bindings and since the AuthServicedefaults to not authenticated we show the text Login.

Now lets change the AuthService so it now returns authenticated, like so:

Copyit('login button hidden when the user is authenticated', () => {

expect(el.nativeElement.textContent.trim()).toBe('');

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

spyOn(authService, 'isAuthenticated').and.returnValue(**true**);

expect(el.nativeElement.textContent.trim()).toBe('Login');

});

But at this point the button content still isn’t Logout, we need to trigger another change detection run like so:

Copyit('login button hidden when the user is authenticated', () => {

expect(el.nativeElement.textContent.trim()).toBe('');

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

spyOn(authService, 'isAuthenticated').and.returnValue(**true**);

expect(el.nativeElement.textContent.trim()).toBe('Login');

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Logout');

});

Now we’ve triggered a second change detection run Angular detected that the AuthService returns true and the button text updated to Logout accordingly.

## Summary

By using the ATB and fixtures we can inspect the components view through fixture.debugElementand also trigger a change detection run by calling fixture.detectChanges().

Next up we’ll look at how to can test asynchronous functions in Angular.

## Listing

*Listing 1. login.component.ts*

Copy**import** {Component} from '@angular/core';

**import** {AuthService} from "./auth.service";

**@Component**({

selector: 'app-login',

template: `

<a>

<span \*ngIf="needsLogin()">Login</span>

<span \*ngIf="!needsLogin()">Logout</span>

</a>

`

})

**export** **class** LoginComponent {

**constructor**(**private** auth: AuthService) {

}

needsLogin() {

**return** !**this**.auth.isAuthenticated();

}

}

*Listing 2. login.component.spec.ts*

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

**import** {TestBed, async, ComponentFixture} from '@angular/core/testing';

**import** {LoginComponent} from './login.component';

**import** {AuthService} from "./auth.service";

**import** {DebugElement} from "@angular/core";

**import** {By} from "@angular/platform-browser";

describe('Component: Login', () => {

**let** component: LoginComponent;

**let** fixture: ComponentFixture<LoginComponent>;

**let** authService: AuthService;

**let** el: DebugElement;

beforeEach(() => {

*// refine the test module by declaring the test component*

TestBed.configureTestingModule({

declarations: [LoginComponent],

providers: [AuthService]

});

*// create component and test fixture*

fixture = TestBed.createComponent(LoginComponent);

*// get test component from the fixture*

component = fixture.componentInstance;

*// UserService provided to the TestBed*

authService = TestBed.get(AuthService);

*// get the "a" element by CSS selector (e.g., by class name)*

el = fixture.debugElement.query(By.css('a'));

});

it('login button hidden when the user is authenticated', () => {

*// To being with Angular has not done any change detection so the content is blank.*

expect(el.nativeElement.textContent.trim()).toBe('');

*// Trigger change detection and this lets the template update to the initial value which is Login since by*

*// default we are not authenticated*

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

*// Change the authetication state to true*

spyOn(authService, 'isAuthenticated').and.returnValue(**true**);

*// The label is still Login! We need changeDetection to run and for angular to update the template.*

expect(el.nativeElement.textContent.trim()).toBe('Login');

*// Which we can trigger via fixture.detectChange()*

fixture.detectChanges();

*// Now the label is Logout*

expect(el.nativeElement.textContent.trim()).toBe('Logout');

});

});

# Testing Asynchronous Code

## Test setup

We want to see how we can test asynchronous functions.

So we change our AuthService.isAuthenticated() function to an asynchronous one that return a promise which resolves into a boolean at a later time.

Copy**export** **class** AuthService {

isAuthenticated(): Promise<**boolean**> {

**return** Promise.resolve(!!localStorage.getItem('token'));

}

}

We also then change our LoginComponent:

Copy**export** **class** LoginComponent **implements** OnInit {

needsLogin: boolean = **true**;

**constructor**(**private** auth: AuthService) {

}

ngOnInit() {

**this**.auth.isAuthenticated().then((authenticated) => {

**this**.needsLogin = !authenticated;

})

}

}

We’ve changed needsLogin from a function into a property and we set the value of this property in the then callback from the promise returned from AuthService.

Importantly we’ve done the above in the ngOnInit() lifecycle function. Probably not the best place to put this functionality given that the value might change over time but good for demonstration purposes.

## No asynchronous handling

Our first attempt might be to try to test our application without taking into account the asynchronous nature of our app, like so:

Copy it('Button label via jasmine.done', () => {

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(**true**));

component.ngOnInit();

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Logout');

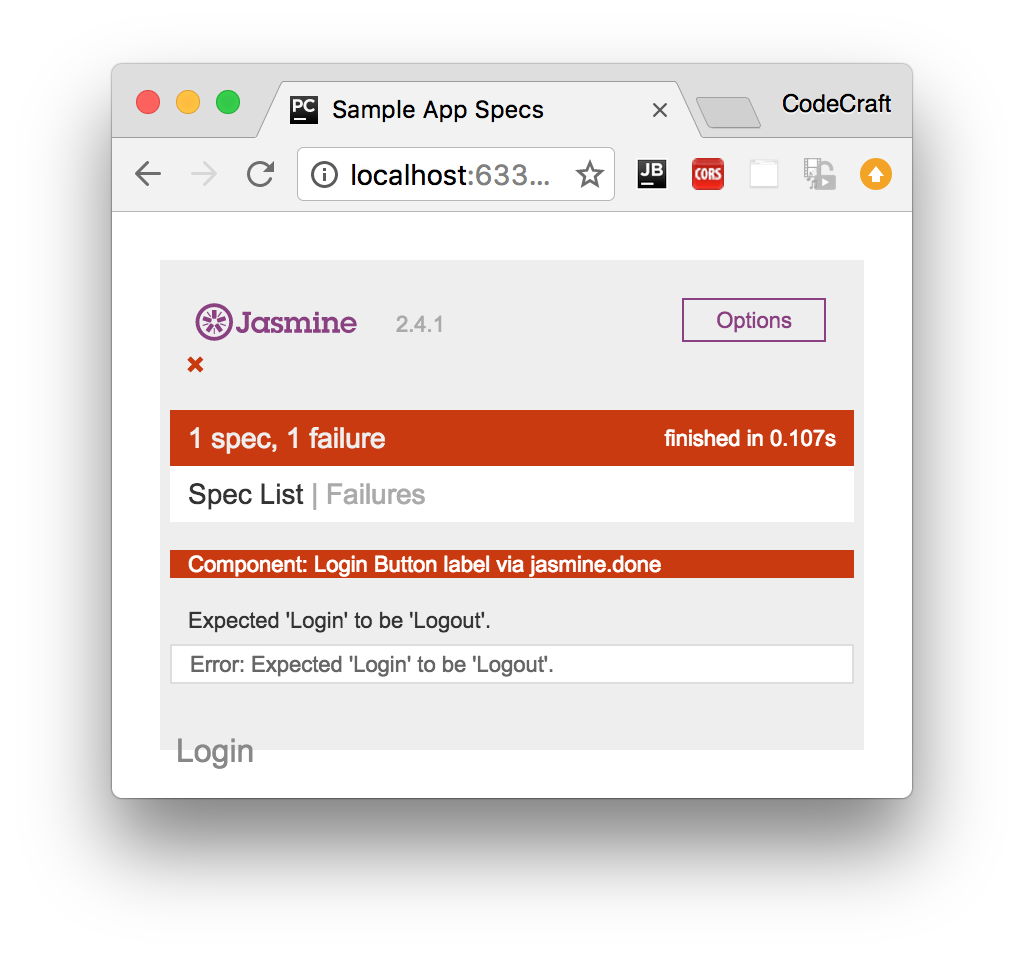
});

|  |  |
| --- | --- |
|  | We issue our first change detection run so the view does it’s initial update. |
|  | We expect the button text to display Login |
|  | We change our AuthService so it returns a promise resolved to true. |
|  | We call component.ngOnInit(). |
|  | We issue our second change detection run. |
|  | We now expect the button text to read Logout. |

#### Important

When performing testing we need to call component lifecycle hooks ourselves, like ngOnInit(). Angular won’t do this for us in the test environment.

If we ran the above code we would see it doesn’t pass:



It’s failing on the last expectation. By the time we run the last expectation the AuthService.isAuthenticated() function hasn’t yet resolved to a value. Therefore the needsLoginproperty on the LoginComponent hasn’t been updated.

There are a few ways we can handle asynchronous code in our tests, one is the Jasmine way and two are Angular specific, lets start with the Jasmine way.

## Jasmines done function

Jasmine has a built-in way to handle async code and that’s by the passed in done function in the test specs.

So far we’ve been defining our test specs without any parameters but it can take a parameter, a done function which we call when all the async processing is complete, like so:

Copyit('Button label via jasmine.done', (done) => {

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

**let** spy = spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(**true**));

component.ngOnInit();

spy.calls.mostRecent().returnValue.then(() => {

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Logout');

done();

});

});

|  |  |
| --- | --- |
|  | The jasmine test spec function is passed a function as the first param, we usually call this parameter done. |
|  | We can add a callback function (using the spy) which is called when the promise returned from isAuthenticated function resolved. In this function we know that the component has the new value of needsLogin and we can add our additional expectation here. |
|  | When we are done with our asynchronous tasks we tell Jasmine via the done function. |

Jasmine lets us create asynchronous tests by giving us an explict done function which we call when the test is complete.

Although it works trying to understand the code can be difficult as it jumps about and is not executed in the order it’s written in.

## async and whenStable

Angular has another method for us to test asynchronous code via the async and whenStable functions.

Let’s rewrite the above test to use these and then we will explain the differences.

Copyit('Button label via async() and whenStable()', async(() => {

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(**true**));

fixture.whenStable().then(() => {

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Logout');

});

component.ngOnInit();

}));

|  |  |
| --- | --- |
|  | We wrap our test spec function in another function called async. |
|  | We place the tests we need to run after the isAuthenticated promise resolves inside this function. |

This async function executes the code inside it’s body in a special async test zone. This intercepts and keeps track of all promises created in it’s body.

Only when all of those pending promises have been resolved does it then resolves the promise returned from whenStable.

So by using the async and whenStable functions we now don’t need to use the Jasmine spy mechanism of detecting when the isAuthenticated promise has been resolved, like the previous example.

This mechanism is slightly better than using the plain Jasmine solution but there is another version which gives us fine grained control and also allows us to lay out our test code as if it were synchronous.

## fakeAsync and tick

Copyit('Button label via fakeAsync() and tick()', fakeAsync(() => {

expect(el.nativeElement.textContent.trim()).toBe('');

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(**true**));

component.ngOnInit();

tick();

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Logout');

}));

|  |  |
| --- | --- |
|  | Like async we wrap the test spec function in a function called fakeAsync. |
|  | We call tick() when there are pending asynchronous activities we want to complete. |

Like the async function the fakeAsync function executes the code inside it’s body in a special fake async test zone. This intercepts and keeps track of all promises created in it’s body.

The tick() function blocks execution and simulates the passage of time until all pending asynchronous activities complete.

So when we call tick() the application sits and waits for the promise returned from isAuthenticatedto be resolved and then lets execution move to the next line.

The code above is now layed our linearly, as if we were executing synchronous code, there are no callbacks to confuse the mind and everything is simpler to understand.

#### Important

fakeAsync does have some drawbacks, it doesn’t track XHR requests for instance.

## Summary

If the code we are testing is asynchronous then we need to take this into account when writing our tests.

There are three mechanisms we can use.

The jasmine done function and spy callbacks. We attach specific callbacks to spies so we know when promises are resolves, we add our test code to thos callbacks and then we call the done function. This works but means we need to know about all the promises in our application and be able to hook into them.

We can use the Angular async and whenStable functions, we don’t need to track the promises ourselves but we still need to lay our code out via callback functions which can be hard to read.

We can use the Angular fakeAsync and tick functions, this additionally lets us lay out our async test code as if it were synchronous.

## Listing

http://plnkr.co/edit/83TAHD1hE3s7XBhz5sL3?p=preview

*Listing 1. auth.service.ts*

Copy**export** **class** AuthService {

isAuthenticated(): Promise<**boolean**> {

**return** Promise.resolve(!!localStorage.getItem('token'));

}

}

*Listing 2. login.component.ts*

Copy**import** {Component} from '@angular/core';

**import** {AuthService} from "./auth.service";

**@Component**({

selector: 'app-login',

template: `

<a>

<span \*ngIf="needsLogin">Login</span>

<span \*ngIf="!needsLogin">Logout</span>

</a>

`

})

**export** **class** LoginComponent **implements** OnInit {

needsLogin: boolean = **true**;

**constructor**(**private** auth: AuthService) {

}

ngOnInit() {

**this**.auth.isAuthenticated().then((authenticated) => {

**this**.needsLogin = !authenticated;

})

}

}

*Listing 3. login.component.spec.ts*

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

**import** {TestBed, async, whenStable, fakeAsync, tick, ComponentFixture} from '@angular/core/testing';

**import** {LoginComponent} from './login.component';

**import** {AuthService} from "./auth.service";

**import** {DebugElement} from "@angular/core";

**import** {By} from "@angular/platform-browser";

describe('Component: Login', () => {

**let** component: LoginComponent;

**let** fixture: ComponentFixture<LoginComponent>;

**let** authService: AuthService;

**let** el: DebugElement;

beforeEach(() => {

*// refine the test module by declaring the test component*

TestBed.configureTestingModule({

declarations: [LoginComponent],

providers: [AuthService]

});

*// create component and test fixture*

fixture = TestBed.createComponent(LoginComponent);

*// get test component from the fixture*

component = fixture.componentInstance;

*// UserService provided to the TestBed*

authService = TestBed.get(AuthService);

*// get the "a" element by CSS selector (e.g., by class name)*

el = fixture.debugElement.query(By.css('a'));

});

it('Button label via fakeAsync() and tick()', fakeAsync(() => {

expect(el.nativeElement.textContent.trim()).toBe('');

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(**true**));

component.ngOnInit();

*// Simulates the passage of time until all pending asynchronous activities complete*

tick();

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Logout');

}));

it('Button label via async() and whenStable()', async(() => {

*// async() knows about all the pending promises defined in it's function body.*

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(**true**));

fixture.whenStable().then(() => {

*// This is called when ALL pending promises have been resolved*

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Logout');

});

component.ngOnInit();

}));

it('Button label via jasmine.done', (done) => {

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

*// Make the authService return a promise that resolves to true*

**let** spy = spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(**true**));

*// We trigger the component to check the authService again*

component.ngOnInit();

*// We now want to call a function when the Promise returned from authService.isAuthenticated() is resolved*

spy.calls.mostRecent().returnValue.then(() => {

*// The needsChanged boolean has been updated on the Component so to update the template we trigger change detection*

fixture.detectChanges();

*// Now the label is Logout*

expect(el.nativeElement.textContent.trim()).toBe('Logout');

*// We tell jasmine we are done with this test spec*

done();

});

});

});

# Testing Components

## Test setup

We’ll continue with our example of testing a LoginComponent. We are going to change our component into a more complex version with inputs, outputs, a domain model and a form, like so:

Copy**import** {Component, EventEmitter, Input, Output} from '@angular/core';

**export** **class** User {

**constructor**(**public** email: string, **public** password: string) {

}

}

**@Component**({

selector: 'app-login',

template: `

<form>

<label>Email</label>

<input type="email"

#email>

<label>Password</label>

<input type="password"

#password>

<button type="button"

(click)="login(email.value, password.value)"

[disabled]="!enabled">Login

</button>

</form>

`

})

**export** **class** LoginComponent {

**@Output**() loggedIn = **new** EventEmitter<User>();

**@Input**() enabled = **true**;

login(email, password) {

console.log(`Login **${**email**}** **${**password**}**`);

**if** (email && password) {

console.log(`Emitting`);

**this**.loggedIn.emit(**new** User(email, password));

}

}

}

|  |  |
| --- | --- |
|  | We create a User class which holds the model of a logged in user. |
|  | The button is sometimes disabled depending on the enabled input property value and on clicking the button we call the login function. |
|  | The component has an output event called loggedIn. |
|  | The component has an input property called enabled. |
|  | In the login function we emit a new user model on the loggedIn event. |

The component is more complex and uses inputs, outputs and emits a domain model on the output event.

#### Note

We are not using the AuthService any more.

We also bootstrap our test suite file like so:

Copydescribe('Component: Login', () => {

**let** component: LoginComponent;

**let** fixture: ComponentFixture<LoginComponent>;

**let** submitEl: DebugElement;

**let** loginEl: DebugElement;

**let** passwordEl: DebugElement;

beforeEach(() => {

TestBed.configureTestingModule({

declarations: [LoginComponent]

});

*// create component and test fixture*

fixture = TestBed.createComponent(LoginComponent);

*// get test component from the fixture*

component = fixture.componentInstance;

submitEl = fixture.debugElement.query(By.css('button'));

loginEl = fixture.debugElement.query(By.css('input[type=email]'));

passwordEl = fixture.debugElement.query(By.css('input[type=password]'));

});

});

We just have a few more debug elements stored on our test suite which we’ll inspect or interact with in our test specs.

## Testing @Inputs

To test inputs we need to do things:

1. We need to be able to change the input property enabled on our component.
2. We need to check that the button is enabled or disabled depending on the value of our input property.

Solving the first is actually very easy.

Just because it’s an @Input doesn’t change the fact it’s a still just a simple property which we can change like any other property, like so:

Copyit('Setting enabled to false disables the submit button', () => {

component.enabled = **false**;

});

For the second we need to check the disabled property value of the buttons DOM element like so:

Copyit('Setting enabled to false disables the submit button', () => {

component.enabled = **false**;

fixture.detectChanges();

expect(submitEl.nativeElement.disabled).toBeTruthy();

});

#### Note

We also need to call fixture.detectChanges() to trigger change detection and update the view.

## Testing @Outputs

Testing outputs it somewhat trickier, especially if we want to test from the view.

Firstly lets see how we can track what gets emmitted by the output event and add some expectations for it.

Copyit('Entering email and password emits loggedIn event', () => {

**let** user: User;

component.loggedIn.subscribe((value) => user = value);

expect(user.email).toBe("test@example.com");

expect(user.password).toBe("123456");

});

The key line above is

Copycomponent.loggedIn.subscribe((value) => user = value);

Since the output event is actually an Observable we can subscribe to it and get a callback for every item emitted.

We store the emitted value to a user object and then add some expectations on the user object.

How do we actually trigger an event to be fired? We could call the component.login(…​) function ourselves but for the purposes of this lecture we want to trigger the function from the view.

Firstly lets set some values to our email and password input controls in the view. We’ve already got references to both those fields in our setup function so we just set the values like so:

CopyloginEl.nativeElement.value = "test@example.com";

passwordEl.nativeElement.value = "123456";

Next we trigger a click on the submit button, but we want to do that after we’ve subscribed to our observable like so:

Copyit('Entering email and password emits loggedIn event', () => {

**let** user: User;

loginEl.nativeElement.value = "test@example.com";

passwordEl.nativeElement.value = "123456";

component.loggedIn.subscribe((value) => user = value);

submitEl.triggerEventHandler('click', **null**);

expect(user.email).toBe("test@example.com");

expect(user.password).toBe("123456");

});

|  |  |
| --- | --- |
|  | Setup data in our input controls. |
|  | Trigger a click on our submit button, this synchronously emits the user object in the subscribe callback! |

## Summary

We can test inputs by just setting values on a components input properties.

We can test outputs by subscribing to an EventEmitters observable and storing the emitted values on local variables.

In combination with the previous lectures and the ability to test inputs and outputs we should now have all the information we need to test components in Angular.

In the next lecture we will look at how to test Directives.

## Listing

http://plnkr.co/edit/N95Scj9LcUbxaLhZismT?p=preview

*Listing 1. login.component.ts*

Copy**import** {Component, EventEmitter, Input, Output} from '@angular/core';

**export** **class** User {

**constructor**(**public** email: string, **public** password: string) {

}

}

**@Component**({

selector: 'app-login',

template: `

<form>

<label>Email</label>

<input type="email"

#email>

<label>Password</label>

<input type="password"

#password>

<button type="button"

(click)="login(email.value, password.value)"

[disabled]="!enabled">Login

</button>

</form>

`

})

**export** **class** LoginComponent {

**@Output**() loggedIn = **new** EventEmitter<User>();

**@Input**() enabled = **true**;

login(email, password) {

console.log(`Login **${**email**}** **${**password**}**`);

**if** (email && password) {

console.log(`Emitting`);

**this**.loggedIn.emit(**new** User(email, password));

}

}

}

*Listing 2. login.component.spec.ts*

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

**import** {TestBed, ComponentFixture, inject, async} from '@angular/core/testing';

**import** {LoginComponent, User} from './login.component';

**import** {Component, DebugElement} from "@angular/core";

**import** {By} from "@angular/platform-browser";

describe('Component: Login', () => {

**let** component: LoginComponent;

**let** fixture: ComponentFixture<LoginComponent>;

**let** submitEl: DebugElement;

**let** loginEl: DebugElement;

**let** passwordEl: DebugElement;

beforeEach(() => {

*// refine the test module by declaring the test component*

TestBed.configureTestingModule({

declarations: [LoginComponent]

});

*// create component and test fixture*

fixture = TestBed.createComponent(LoginComponent);

*// get test component from the fixture*

component = fixture.componentInstance;

submitEl = fixture.debugElement.query(By.css('button'));

loginEl = fixture.debugElement.query(By.css('input[type=email]'));

passwordEl = fixture.debugElement.query(By.css('input[type=password]'));

});

it('Setting enabled to false disabled the submit button', () => {

component.enabled = **false**;

fixture.detectChanges();

expect(submitEl.nativeElement.disabled).toBeTruthy();

});

it('Setting enabled to true enables the submit button', () => {

component.enabled = **true**;

fixture.detectChanges();

expect(submitEl.nativeElement.disabled).toBeFalsy();

});

it('Entering email and password emits loggedIn event', () => {

**let** user: User;

loginEl.nativeElement.value = "test@example.com";

passwordEl.nativeElement.value = "123456";

*// Subscribe to the Observable and store the user in a local variable.*

component.loggedIn.subscribe((value) => user = value);

*// This sync emits the event and the subscribe callback gets executed above*

submitEl.triggerEventHandler('click', **null**);

*// Now we can check to make sure the emitted value is correct*

expect(user.email).toBe("test@example.com");

expect(user.password).toBe("123456");

});

})

;

# Testing Http

## Test setup

To demonstrate how to test http requests we will add a test for our iTunes SearchService which we created in the section on Http.

We will use the promise version of the search service that uses JSONP to get around the issue of CORS.

Copy**import** {Injectable} from '@angular/core';

**import** {Jsonp} from '@angular/http';

**import** 'rxjs/add/operator/toPromise';

**class** SearchItem {

**constructor**(**public** name: string,

**public** artist: string,

**public** thumbnail: string,

**public** artistId: string) {

}

}

**@Injectable**()

**export** **class** SearchService {

apiRoot: string = 'https://itunes.apple.com/search';

results: SearchItem[];

**constructor**(**private** jsonp: Jsonp) {

**this**.results = [];

}

search(term: string) {

**return** **new** Promise((resolve, reject) => {

**this**.results = [];

**let** apiURL = `**${this**.apiRoot**}**?term=**${**term**}**&media=music&limit=20&callback=JSONP\_CALLBACK`;

**this**.jsonp.request(apiURL)

.toPromise()

.then(

res => { *// Success*

**this**.results = res.json().results.map(item => {

console.log(item);

**return** **new** SearchItem(

item.trackName,

item.artistName,

item.artworkUrl60,

item.artistId

);

});

resolve(**this**.results);

},

msg => { *// Error*

reject(msg);

}

);

});

}

}

#### Note

Although we are using JSONP here, testing Http and Jsonp is exactly the same. We just replace instances of Jsonp with Http.

### Configuring the test suite

We want the Jsonp and Http services to use the MockBackend instead of the real Backend, this is the underling code that actually sends and handles http.

By using the MockBackend we can intercept real requests and simulate responses with test data.

The configuration is slightly more complex since we are using a factory provider:

Copy{

provide: Http,

useFactory: (backend, options) => **new** Http(backend, options),

deps: [MockBackend, BaseRequestOptions]

}

|  |  |
| --- | --- |
|  | We are configuring a dependency for the token Http. |
|  | The injector calls this function in order to return a new instance of the Http class. The arguments to the useFactory function are themselves injected in, see (3). |
|  | We define the dependencies to our useFactory function via the deps property. |

For our API however we are using Jsonp, we can just replace all mention of Http with Jsonp like so:

Copy{

provide: Jsonp,

useFactory: (backend, options) => **new** Jsonp(backend, options),

deps: [MockBackend, BaseRequestOptions]

}

The above configuration ensures that the Jsonp service is constructed using the MockBackend so we can control it later on in testing.

Together with the other providers and modules we need our initial test suite file looks like so:

Copydescribe('Service: Search', () => {

**let** service: SearchService;

**let** backend: MockBackend;

beforeEach(() => {

TestBed.configureTestingModule({

imports: [JsonpModule],

providers: [

SearchService,

MockBackend,

BaseRequestOptions,

{

provide: Jsonp,

useFactory: (backend, options) => **new** Jsonp(backend, options),

deps: [MockBackend, BaseRequestOptions]

}

]

});

backend = TestBed.get(MockBackend);

service = TestBed.get(SearchService);

});

});

|  |  |
| --- | --- |
|  | We grab a reference to the mock backend so we can control the http responses from our test specs. |
|  | We grab a reference to the SearchService, this has been created using the MockBackend above. |

## Using the MockBackend to simulate a response

Just by using the MockBackend instead of the real Backend we have stopped the tests from triggering real http requests from being sent out.

Now we need to configure the MockBackend to return dummy test data instead, like so:

Copyit('search should return SearchItems', fakeAsync(() => {

**let** response = {

"resultCount": 1,

"results": [

{

"artistId": 78500,

"artistName": "U2",

"trackName": "Beautiful Day",

"artworkUrl60": "image.jpg",

}]

};

backend.connections.subscribe(connection => {

connection.mockRespond(**new** Response(<ResponseOptions>{

body: JSON.stringify(response)

}));

});

}));

|  |  |
| --- | --- |
|  | We create some fake data we want the API to response with. |
|  | The mock backend connections property is an observable that emits an connection every time an API request is made. |
|  | For every connection that is requested we tell it to mockRespond with our dummy data. |

The above code returns the same dummy data for every API request, regardless of the URL.

## Testing the response

Using HTTP is asynchronous so in order to test we need to use one of the asynchronous testing methods, we’ll use the fakeAsync method.

Copyit('search should return SearchItems', fakeAsync(() => {

**let** response = {

"resultCount": 1,

"results": [

{

"artistId": 78500,

"artistName": "U2",

"trackName": "Beautiful Day",

"artworkUrl60": "image.jpg",

}]

};

*// When the request subscribes for results on a connection, return a fake response*

backend.connections.subscribe(connection => {

connection.mockRespond(**new** Response(<ResponseOptions>{

body: JSON.stringify(response)

}));

});

*// Perform a request and make sure we get the response we expect*

service.search("U2");

tick();

expect(service.results.length).toBe(1);

expect(service.results[0].artist).toBe("U2");

expect(service.results[0].name).toBe("Beautiful Day");

expect(service.results[0].thumbnail).toBe("image.jpg");

expect(service.results[0].artistId).toBe(78500);

}));

|  |  |
| --- | --- |
|  | We use the fakeAsync method to execute in the special fake async zone and track pending promises. |
|  | We make the asynchronous call to service.search(…​) |
|  | We issue a tick() which blocks execution and waits for all the pending promises to be resolved. |
|  | We now know that the service has received and parsed the response so we can write some expectations. |

## Summary

We can test code that makes Http requests by using a MockBackend.

This requires that we configure our TestBed so that the Jsonp or Http services are created using the MockBackend.

We grab a reference to the instance of MockBackend that was injected and use it to simulate responses.

Since Http is asynchronous we use of one of the async testing mechanisms so we can write tests specs for our code.

## Listing

http://plnkr.co/edit/FEQLRETfKMlgSTDm2XMI?p=preview

*Listing 1. search.service.ts*

Copy**import** {Injectable} from '@angular/core';

**import** {Jsonp} from '@angular/http';

**import** 'rxjs/add/operator/toPromise';

**class** SearchItem {

**constructor**(**public** name: string,

**public** artist: string,

**public** thumbnail: string,

**public** artistId: string) {

}

}

**@Injectable**()

**export** **class** SearchService {

apiRoot: string = 'https://itunes.apple.com/search';

results: SearchItem[];

**constructor**(**private** jsonp: Jsonp) {

**this**.results = [];

}

search(term: string) {

**return** **new** Promise((resolve, reject) => {

**this**.results = [];

**let** apiURL = `**${this**.apiRoot**}**?term=**${**term**}**&media=music&limit=20&callback=JSONP\_CALLBACK`;

**this**.jsonp.request(apiURL)

.toPromise()

.then(

res => { *// Success*

**this**.results = res.json().results.map(item => {

console.log(item);

**return** **new** SearchItem(

item.trackName,

item.artistName,

item.artworkUrl60,

item.artistId

);

});

resolve(**this**.results);

},

msg => { *// Error*

reject(msg);

}

);

});

}

}

*Listing 2. script.ts*

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

**import** {

JsonpModule,

Jsonp,

BaseRequestOptions,

Response,

ResponseOptions,

Http

} from "@angular/http";

**import** {TestBed, fakeAsync, tick} from '@angular/core/testing';

**import** {MockBackend} from "@angular/http/testing";

**import** {SearchService} from './search.service';

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

**let** service: SearchService;

**let** backend: MockBackend;

beforeEach(() => {

TestBed.configureTestingModule({

imports: [JsonpModule],

providers: [

SearchService,

MockBackend,

BaseRequestOptions,

{

provide: Jsonp,

useFactory: (backend, options) => **new** Jsonp(backend, options),

deps: [MockBackend, BaseRequestOptions]

}

]

});

*// Get the MockBackend*

backend = TestBed.get(MockBackend);

*// Returns a service with the MockBackend so we can test with dummy responses*

service = TestBed.get(SearchService);

});

it('search should return SearchItems', fakeAsync(() => {

**let** response = {

"resultCount": 1,

"results": [

{

"artistId": 78500,

"artistName": "U2",

"trackName": "Beautiful Day",

"artworkUrl60": "image.jpg",

}]

};

*// When the request subscribes for results on a connection, return a fake response*

backend.connections.subscribe(connection => {

connection.mockRespond(**new** Response(<ResponseOptions>{

body: JSON.stringify(response)

}));

});

*// Perform a request and make sure we get the response we expect*

service.search("U2");

tick();

expect(service.results.length).toBe(1);

expect(service.results[0].artist).toBe("U2");

expect(service.results[0].name).toBe("Beautiful Day");

expect(service.results[0].thumbnail).toBe("image.jpg");

expect(service.results[0].artistId).toBe(78500);

}));

});

# Testing Routing

Test setup

To test routing we need a few components and a route configuration:

Copy**import** {Component} from "@angular/core";

**import** {Routes} from "@angular/router";

**@Component**({

template: `Search`

})

**export** **class** SearchComponent {

}

**@Component**({

template: `Home`

})

**export** **class** HomeComponent {

}

**@Component**({

template: `<router-outlet></router-outlet>`

})

**export** **class** AppComponent {

}

**export** **const** routes: Routes = [

{path: '', redirectTo: 'home', pathMatch: 'full'},

{path: 'home', component: HomeComponent},

{path: 'search', component: SearchComponent}

];

We create three components HomeComponent, SearchComponent and an AppComponent with a <router-outlet>.

We also create a route configuration where '' redirects you to home and home and search show their respective components.

Our basic test suite looks like so:

Copy**import** {Location} from "@angular/common";

**import** {TestBed, fakeAsync, tick} from '@angular/core/testing';

**import** {RouterTestingModule} from "@angular/router/testing";

**import** {Router} from "@angular/router";

**import** {

HomeComponent,

SearchComponent,

AppComponent,

routes

} from "./router"

describe('Router: App', () => {

beforeEach(() => {

TestBed.configureTestingModule({

declarations: [

HomeComponent,

SearchComponent,

AppComponent

]

});

});

});

|  |  |
| --- | --- |
|  | We import and declare our components in the test bed configuration. |

Router setup

Normally to setup routing in an Angular application we import the RouterModule and *provide* the routes to the NgModule with RouterModule.withRoutes(routes).

However when testing routing we use the RouterTestingModule instead. This modules sets up the router with a *spy* implementation of the *Location Strategy* that doesn’t actually change the URL.

We also need to get the injected Router and Location so we can use them in the test specs.

Our test suite file now looks like:

Copydescribe('Router: App', () => {

**let** location: Location;

**let** router: Router;

**let** fixture;

beforeEach(() => {

TestBed.configureTestingModule({

imports: [RouterTestingModule.withRoutes(routes)],

declarations: [

HomeComponent,

SearchComponent,

AppComponent

]

});

router = TestBed.get(Router);

location = TestBed.get(Location);

fixture = TestBed.createComponent(AppComponent);

router.initialNavigation();

});

});

|  |  |
| --- | --- |
|  | We import our RouterTestingModule with our routes. |
|  | We grab a reference to the injected Router. |
|  | We grab a reference to the injected Location. |
|  | We ask the test bed to create an instance of our root AppComponent. We don’t need this reference in our test specs but we do need to create the root component with the router-outlet so the router has somewhere to insert components. |
|  | This sets up the location change listener and performs the initial navigation. |

We are now ready to create our test specs.

Testing routing

In our configuration we’ve set it so if you land on the root *empty* url you will be redirected to /home, lets add a test spec for this:

Copyit('navigate to "" redirects you to /home', fakeAsync(() => {

router.navigate(['']);

tick();

expect(location.path()).toBe('/home');

}));

|  |  |
| --- | --- |
|  | Routing is an asynchronous activity so we use one of the asynchronous testing methods at our disposal, in this case the fakeAsync method. |
|  | We trigger the router to navigate to the empty path. |
|  | We wait for all pending promises to be resolved. |
|  | We can then inspect the *path* our application should be at with location.path() |

Lets also add a test spec for navigating to the search route, like so:

Copyit('navigate to "search" takes you to /search', fakeAsync(() => {

router.navigate(['search']);

tick();

expect(location.path()).toBe('/search');

}));

The spec is exactly the same as the previous one, our link params array is different since we are triggering a different route and our expectation is again different, but the rest is the same.

Summary

We can test routing in Angular by using RouterTestingModule instead of RouterModule to provide our routes.

This uses a *spy* implementation of Location which doesn’t trigger a request for a new URL but does let us know the target URL which we can use in our test specs.

Listing

http://plnkr.co/edit/mTYEDNRlgGjdwv3M5b28?p=preview

*Listing 1. router.ts*

Copy**import** {Component} from "@angular/core";

**import** {Routes} from "@angular/router";

**@Component**({

template: `Search`

})

**export** **class** SearchComponent {

}

**@Component**({

template: `Home`

})

**export** **class** HomeComponent {

}

**@Component**({

template: `<router-outlet></router-outlet>`

})

**export** **class** AppComponent {

}

**export** **const** routes: Routes = [

*// {path: '', redirectTo: 'home', pathMatch: 'full'},*

{path: 'home', component: HomeComponent},

{path: 'search', component: SearchComponent}

];

*Listing 2. router.spec.ts*

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

**import** {Location} from "@angular/common";

**import** {TestBed, fakeAsync, tick} from '@angular/core/testing';

**import** {RouterTestingModule} from "@angular/router/testing";

**import** {Router} from "@angular/router";

**import** {

HomeComponent,

SearchComponent,

AppComponent,

routes

} from "./router"

describe('Router: App', () => {

**let** location: Location;

**let** router: Router;

**let** fixture;

beforeEach(() => {

TestBed.configureTestingModule({

imports: [ RouterTestingModule.withRoutes(routes)],

declarations: [

HomeComponent,

SearchComponent,

AppComponent

]

});

router = TestBed.get(Router);

location = TestBed.get(Location);

fixture = TestBed.createComponent(AppComponent);

router.initialNavigation();

});

it('fakeAsync works', fakeAsync(() => {

**let** promise = **new** Promise((resolve) => {

setTimeout(resolve, 10)

});

**let** done = **false**;

promise.then(() => done = **true**);

tick(50);

expect(done).toBeTruthy();

}));

it('navigate to "" redirects you to /home', fakeAsync(() => {

router.navigate(['']);

tick(50);

expect(location.path()).toBe('/home');

}));

it('navigate to "search" takes you to /search', fakeAsync(() => {

router.navigate(['/search']);

tick(50);

expect(location.path()).toBe('/search');

}));

});

import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { ParentComponent } from './parent.component';

import { DataService } from 'src/app/data.service';

describe('ParentComponent', () => {

beforeAll(()=>{

console.log("before all in parent component");

});

afterAll(()=>{

console.log('after all - parent component');

});

beforeEach(()=>{

console.log('before each - parent component');

});

afterEach(()=>{

console.log('after each - parent component');

});

it("addition check",()=>{

//expect(10+5).toEqual(15);

var d = new DataService();

var o = new ParentComponent( d);

expect(o.add(5,5)).toEqual(10);

});

it("addition check2",()=>{

//expect(10+5).toEqual(15);

var d = new DataService();

var o = new ParentComponent( d);

expect(o.add(5,6)).toEqual(11);

});

it("substruction check",()=>{

expect(10-5).toEqual(2);

});

it("multiplication check",()=>{

expect(10\*5).toEqual(40);

});

it("division check",()=>{

expect(10/5).toEqual(2);

});

});