Unit testing in Angular

Inhoud

[Introduction ( 1 - 2) 2](#_Toc47982281)

[Naming 2](#_Toc47982282)

[AAA 2](#_Toc47982283)

[Select or skip tests 2](#_Toc47982284)

[Testing principals 2](#_Toc47982285)

[Debugging tests in Karma 2](#_Toc47982286)

[Isolated Unit tests 2](#_Toc47982287)

[Mocking a service 2](#_Toc47982288)

[State based- and interaction tests 3](#_Toc47982289)

[Shallow Integration Tests 5](#_Toc47982290)

[TestBed 5](#_Toc47982291)

[A simple component 5](#_Toc47982292)

[A complex component 6](#_Toc47982293)

[Deep Integration Tests 8](#_Toc47982294)

[Integration test for components 8](#_Toc47982295)

[Integration tests for (HTTP) services 9](#_Toc47982296)

[DOM interaction and Routing 10](#_Toc47982297)

[Triggering an event on an element 10](#_Toc47982298)

[Emitting events from children 11](#_Toc47982299)

[Raising events from children 11](#_Toc47982300)

[Interacting with input boxes 11](#_Toc47982301)

[Testing with activated route 12](#_Toc47982302)

[Mocking a directive (routerLink) 13](#_Toc47982303)

[Testing the RouterLink (Stub) 13](#_Toc47982304)

[Asynchronous code testing 14](#_Toc47982305)

[Code coverage 15](#_Toc47982306)

[Appendix 16](#_Toc47982307)

[Sources 16](#_Toc47982308)

# Introduction ( 1 - 2)

## Naming

When in the code snippets in this test name like Shallow test or Isolate test are used this is only for the purpose of this course. Normally Isolate and shallow are not used file or test names.

## AAA

Unit test should have the following structure:

* Arrange: all necessary preconditions and inputs
* Act: on the object or class under test
* **A**ssert: that the expected results have occurred.

This is analog to given, when, then

## Select or skip tests

To select one or more tests and skip all others, put an f before it(‘should….. to select an entire test suite put the f before the describe(‘….

To skip one or more tests, put a x before it(‘should….. to skip an entire test suite put the x before the describe(‘….

## Testing principals

* DRY (don’t repeat yourself) vs DAMP (repeat yourself if necessary)  
  Repeat if necessary applies to situations where, for instance the setup is very critical to the story. In that case the setup is not done in the beforeEach but in the it(‘shoul…’, () => {}
* Don’t test the framework (i.e. routing) but test your code.

## Debugging tests in Karma

There are two features which can help you debugging tests in Karma:

* First one is to open the development console of Chrome and select the console tab. This might give extra info if tests are failing
* Karma and zone.js sometimes do not interact correct. To avoid this issue change the scrips > tests in package.json from: "test": "ng test" to: "test": "ng test --source-map=false"

# Isolated Unit tests

You can look at isolated test as if it’s not an angular class but just a piece of javascript. (in your mind) remove all the things like @Component, @Pipe, @Input etc.

## Mocking a service

The test for the delete method of the HeroComponent is depending on a service.

See: the Herocomponent.

* The HeroComponent has a heroes property. To test the delete function we have to populate heroes first.
* The HeroComponent is depending on the HeroService which is injected into the contructor.
* To resolve this independency of the HeroService we have to Mock it. Jasmine provides craeateSpyObj() which takes an array of methodnames of the HeroService we want to use.
* As the heroService.deleteHero(…) returns an observable, so should our mockHeroService. We can achieve that by returning a subject when calling deleteHero using the “of” method, like so:  
  mockHeroService.deleteHero.and.returnValue(**of(**true**)**);
* The ‘should delete …..’ test is setup following the AAA principle.

The code for this test with some highlights:

import { Hero } from "../hero";

import { HeroesComponent } from "./heroes.component";

import { of } from "rxjs";

describe('HeroesComponent (Isolated tests)', () => {

let heroesComponent: HeroesComponent;

let HEROES;

let mockHeroService;

beforeEach(() => {

HEROES = [

{ id: 1, name: 'pietje', strength: 25 },

{ id: 2, name: 'jantje', strength: 75 },

{ id: 3, name: 'klaasje', strength: 10 }

]

mockHeroService = jasmine.createSpyObj(['addHero', 'getHeroes', 'deleteHero'])

heroesComponent = new HeroesComponent(mockHeroService);

});

it('should delete a hero', () => {

// **Arrange**: init HEROES: is already done by beforeEach

heroesComponent.heroes = HEROES;

// make the mockHeroService.delete retrurn a observable

mockHeroService.deleteHero.**and.returnValue**(**of**(true));

// **Act**: delete a hero

heroesComponent.delete(HEROES[2]);

// **Assert**: HEROES contains two elements now

expect(heroesComponent.heroes.length).toBe(2);

expect(heroesComponent.heroes[0].id).toBe(1);

expect(heroesComponent.heroes[1].id).toBe(2);

expect(heroesComponent.heroes[2]).toBeUndefined();

})

})

## State based- and interaction tests

The above test checks if the state of the component has changed. It’s a **state based** test. What it doesn’t do is check if certain parts of the code were executed. That kind of tests is called **interaction** tests.

An example of an interaction test which checks if a method (of the mocked service) was called:

it('should call heroService.deleteHero with correct hero', () => {

// Arrange

heroesComponent.heroes = HEROES;

mockHeroService.deleteHero.and.returnValue(of(true));

// Act

heroesComponent.delete(HEROES[1]);

// Assert

// check that delleteHero was called

expect(mockHeroService.deleteHero).**toHaveBeenCalled()**;

// or even better... check that delleteHero was called with the correct parameter

expect(mockHeroService.deleteHero).**toHaveBeenCalledWith**(HEROES[1]);

})

# Shallow Integration Tests

A shallow integration test means we’re only going to test a component (.ts and template) and **none** of its child components.

## TestBed

The TestBed object makes it possible to test a component, its typescript and template, together.

TestBed.configureTestingModule(…): configures the module. Here the definition of the module is done. Like: declarations: [], providers: [] …. Etc.

TestBed.createComponent(….): constructs the component of type ComponentFixture<> (ie the (HeroComponent) mentioned in the declarations section of configureTestingModule

fixture.componentInstance; hold the actual instance of the component class (ie heroComponent.ts)

To ignore unknown elements and attributes of our template we can add a property “schemas” in our testingModule like so: schemas: **[NO\_ERRORS\_SCHEMA]**  
The drawback of this approach is that all issues on the template are hidden. You will not be warrened by Angular if something is wrong (ie <buttons> instead of <button>

fixture.**nativeElement** gets a handle to the DOM element which represents the container of the template

nativeElement.**querySelector**('a').textContent) gets the text content of an element. In this case element <a>

fixture.**detectChanges**() runs change detection which will update any bindings that exist on the component (ie hero)

fixture.**debugElement** is more or less the same as native element but it’s got some extra features (ie query (single node) and queryAll (all nodes) and also the possibility to get directives ). Parameter for the query is:

**By.css()** which can query the DOM by elements css**(‘a’)**, just like querySelector. But it can also query the DOM by className: css**(‘.a’)** or by id: css**(‘#a‘)**

## A simple component

Let’s take a look at a simple component first. The component does not have a constructor, no dependencies and the template doesn’t have child components.

The code for a test (without injection and without child elelements) containing the above:

import { TestBed, ComponentFixture } from "@angular/core/testing"

import { HeroComponent } from "./hero.component"

import { NO\_ERRORS\_SCHEMA } from "@angular/core";

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

describe('HeroComponent (Shallow)', () => {

let fixture: ComponentFixture<HeroComponent>;

let heroComponent;

beforeEach(() => {

TestBed.configureTestingModule({

declarations: [HeroComponent],

schemas: [NO\_ERRORS\_SCHEMA]

}

)

fixture = TestBed.createComponent(HeroComponent);

heroComponent = fixture.componentInstance;

})

it('should have the correct hero', () => {

heroComponent.hero = { id: 1, name: "SuperJantje", strength: 44 };

expect(heroComponent.hero.id).toEqual(1);

})

it('should render the hero name in the <a> tag', () => {

heroComponent.hero = { id: 1, name: "SuperJantje", strength: 44 };

fixture.detectChanges();

expect(fixture.nativeElement.**querySelector**('a').textContent).toContain('Super')

// or

expect(fixture.debugElement.**query(By.css('a'))**

.nativeElement.textContent).toContain('Jantje')

// or

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

expect(debugElementA.nativeElement.textContent).toContain('Jantje')

})

})

## A complex component

Let’s have a look at a little more complex component. The Hero**es**Component. This one does have a dependency: the HeroService, and it does have a child component: the HeroComponent. Let’s start with an example which ignores the child component (NO\_ERRORS\_SCHEMA)

How to create a mockHeroService see the above chapter “Mocking a service”.

If you need an instance of a service (ie it’s injected in the constructor of the component, then

the mockHeroService should be added in the providers array of the module configuration (using the longhand provider syntax). It should contain the HeroService and the mockHeroService that will be used.

The code for the test (with injection of the mocked service in the constructor of the component:

import { HeroesComponent } from "./heroes.component";

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

import { NO\_ERRORS\_SCHEMA } from "@angular/core";

import { HeroService } from "../hero.service";

import { of } from "rxjs";

describe('HeroesComponent (shallow)', () => {

let fixture: ComponentFixture<HeroesComponent>;

let heroesComponent;

let **mockHeroService**;

let HEROES;

beforeEach(() => {

HEROES = [

{ id: 1, name: 'pietje', strength: 25 },

{ id: 2, name: 'jantje', strength: 75 },

{ id: 3, name: 'klaasje', strength: 10 }

]

**mockHeroService = jasmine.createSpyObj(['getHeroes', 'addHero', 'deleteHero']);**

TestBed.configureTestingModule({

declarations: [

HeroesComponent

],

providers: [

{ **provide: HeroService**, **useValue: mockHeroService** }

],

schemas: [NO\_ERRORS\_SCHEMA]

})

fixture = TestBed.createComponent(HeroesComponent);

heroesComponent = fixture.componentInstance;

})

it('should populate the heroes array when heroService.getHeroes is called', () => {

// Arrange

mockHeroService.getHeroes.and.returnValue(**of(HEROES));**

// Act (actually do nothing as the constructor will be called automatically)

fixture.detectChanges();

// Assert

expect(heroesComponent.heroes.length).toBe(3);

})

});

Now let’s also solve the issue with unknown child elements (Hero) in the template. Let’s remove the NO\_ERRORS\_SCHEMA from the module schemas: . Now the error is back. Let’s fix, using the code of the previous example above, by creating a mock for the HeroComponent which has a very simple template.

The highlighted parts contain the changes to implement and use a mocked HeroComponent: fakeHeroComponent.

import { HeroesComponent } from "./heroes.component";

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

import { NO\_ERRORS\_SCHEMA, Input, Output, Component } from "@angular/core";

import { HeroService } from "../hero.service";

import { of } from "rxjs";

import { Hero } from "../hero";

describe('HeroesComponent (shallow)', () => {

let fixture: ComponentFixture<HeroesComponent>;

let heroesComponent;

let mockHeroService;

let HEROES;

@Component({

selector: 'app-hero',

**template: '<div></div>'**,

styleUrls: []

})

class **FakeHeroComponent** {

@Input() hero: Hero;

// @Output() delete = new EventEmitter();

}

beforeEach(() => {

HEROES = [

{ id: 1, name: 'pietje', strength: 25 },

{ id: 2, name: 'jantje', strength: 75 },

{ id: 3, name: 'klaasje', strength: 10 }

]

mockHeroService = jasmine.createSpyObj(['getHeroes', 'addHero', 'deleteHero']);

TestBed.configureTestingModule({

declarations: [

HeroesComponent,

**FakeHeroComponent**

],

providers: [

{ provide: HeroService, useValue: mockHeroService }

]

})

fixture = TestBed.createComponent(HeroesComponent);

heroesComponent = fixture.componentInstance;

})

it('should populate the heroes array when heroService.getHeroes is called', () => {

// Arrange

mockHeroService.getHeroes.and.returnValue(of(HEROES));

// Act (actually do nothing as the constructor will be called automatically)

fixture.detectChanges();

// Assert

expect(heroesComponent.heroes.length).toBe(3);

})

});

Now we have a shallow integration test, but… it doesn’t really take advantage of the benefits of having access to the template too. So let’s add one more test to de above HeroesComponent test.

In the last test we’ve counted the number of elements in the heroes array of the .ts. Now we’re go to count the number of <li> elements in the template, queried by element (‘li’) and by id (“#hero).

it('should create a <li> element for each hero', () => {

// Arrange

mockHeroService.getHeroes.and.returnValue(of(HEROES));

// Act

fixture.detectChanges();

// Assert

// query for all li elements

let debugElementsLi = fixture.debugElement.queryAll(By.css(**'li'**));

expect(debugElementsLi.length).toBe(3);

// query for all elements with id 'hero'

let debugElementsIdHero = fixture.debugElement.queryAll(By.css('**#hero**'));

expect(debugElementsLi.length).toBe(3);

})

# Deep Integration Tests

A deep integration test means we will test a component (.ts and template) and its child components, and also a service and it’s dependencies.

## Integration test for components

First an example of a deep integration test with the HeroesComponent which contains child component HeroComponent.

To get access to child elements we use a queryAll, like we did before, but now we’re going to query on directives instead of on css.

This way we can access the directive HeroComponent. Yes, yes, Hero is a directive. Actually it’s a component, but Component is a subclass of Directive, just like attributes.

Let’s have a look at the code. We will implement expectations on the Component and template.

Notice it doesn’t contain a fakeHeroComponent anymore. But the real HeroComponent instead.

import { HeroesComponent } from "./heroes.component";

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

import { NO\_ERRORS\_SCHEMA, Input, Output, Component, DebugElement } from "@angular/core";

import { HeroService } from "../hero.service";

import { of } from "rxjs";

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

import { HeroComponent } from "../hero/hero.component";

describe('HeroesComponent (Deep)', () => {

let fixture: ComponentFixture<HeroesComponent>;

let heroesComponent;

let mockHeroService;

let HEROES;

beforeEach(() => {

HEROES = [

{ id: 1, name: 'pietje', strength: 25 },

{ id: 2, name: 'jantje', strength: 75 },

{ id: 3, name: 'klaasje', strength: 10 }

]

mockHeroService = jasmine.createSpyObj(['getHeroes', 'addHero', 'deleteHero']);

TestBed.configureTestingModule({

declarations: [

HeroesComponent,

**HeroComponent**

],

providers: [

{ provide: HeroService, useValue: mockHeroService }

],

schemas: [NO\_ERRORS\_SCHEMA]

})

fixture = TestBed.createComponent(HeroesComponent);

})

it('should render the HeroComponet template for each hero', () => {

// Arrange

mockHeroService.getHeroes.and.returnValue(of(HEROES));

// Act: run ngOnInit

fixture.detectChanges();

// Assert

const heroComponentDebugElement =

fixture.debugElement.queryAll(**By.directive(HeroComponent)**);

expect(heroComponentDebugElement.length).toEqual(3);

for (let i = 0; i < heroComponentDebugElement.length; i++) {

const element = heroComponentDebugElement[i];

expect(element.nativeElement.textContent).toContain(HEROES[i].name);

expect(element.query(By.css('a')).nativeElement.textContent)

.toContain(HEROES[i].name);

}

})

it('should create a HeroComponent for each hero', () => {

// Arrange

mockHeroService.getHeroes.and.returnValue(of(HEROES));

// Act: run ngOnInit

fixture.detectChanges();

// Assert

let heroComponentDebugElement =

fixture.debugElement.queryAll(**By.directive(HeroComponent)**);

for (let i = 0; i < heroComponentDebugElement.length; i++) {

expect(heroComponentDebugElement[i].**componentInstance.hero)**.toEqual(HEROES[i]);

}

})

});

## Integration tests for (HTTP) services

Most of the time an isolated test would be sufficient for services. An exception to this rule is a HTTP-service. It’s possible to mock a HTTP-service but it’s complex. That’s why the Angular developers helped us and provided a mock for HTTP. Now we can write integration test for HTTP-services.

The module **HttpClientTestingModule,** provides the mock for doing so. To get a handle to the httpMock there is a special controller: **HttpTestingController**.

Used methods:

* httpTestingController.expectOne('api/heroes/4') => to check that a request to the given url is done exactly once
* request.flush({….}) => actually executes the request.
* httpTestingController.verify() => verify we get exactly what we expect: just one request with correct url

The code for testing a HttpService:

it('should getHero with correct url:', () => {

// mockMessageService.add.and.returnValue();

heroService.getHero(4).subscribe();

// heroService.getHero(5).subscribe();

//expect exactly one call is done to the given url

const request = httpTestingController.**expectOne**('api/heroes/4');

// check if it a GET request

expect(request.**request.method).**toBe("**GET"**);

request.**flush(**{id: 4, name: "Pietje", stregth: 44});

// verify we get exactly what we expect => just one request with correct url

httpTestingController.verify();

})

it('should postHero with correct url and postBody:', () => {

let postBody = "{id: 2}";

// mockMessageService.add.and.returnValue();

heroService.postHero(postBody).subscribe();

// heroService.getHero(5).subscribe();

//expect exactly one call is done to the given url

const request = httpTestingController.expectOne('api/heroes');

// check Post method with body use:

expect(request.**request.method**).toBe("**POST**");

expect(request.**request.body**).toBe("{id: 2}");

request.flush({id: 4, name: "Pietje", stregth: 44});

// verify we get exactly what we expect => just one request with correct url

httpTestingController.verify();

})

# DOM interaction and Routing

## Triggering an event on an element

For this example we will click the ‘delete button’ on the HeroComponent and the we’ll check that the delete method of the Hero**es**Component was called with the right Hero.

So let’s add a test to the heroes.component.deep.spec.ts:

Arrange:

* + populate the HEROES array
  + set a spy on the HeroesComponent delete method
  + detectChanges();

Act:

* + query all child directives “HeroComponent”: queryAll(By.directive(HeroComponent)
  + query button: query(By.css(‘button’)
  + trigger event: triggerEventHandler(*event, eventObject*)  
    event is the event to trigger, which is ‘click’ (of the button)  
    the eventObject, as you can see in the hero.component.ts is a stopPropagation method of the event. Because we’re not interested in what this method does we can just pass in a fake / mock for this method. Like so: { stopPropagation () => {} }

Assert:

* + assert that the delete method of the HeroesComponent was called with the right Hero

The code:

it(`should call the delete method with the right hero,

when the delete button of child HeroComponent was clicked`, () => {

mockHeroService.getHeroes.and.returnValue(of(HEROES));

**spyOn**(fixture.componentInstance, **'delete'**);

fixture.detectChanges();

const heroComponents = fixture.debugElement.queryAll(By.**directive**(HeroComponent));

const button = heroComponents**[0]**.query(By.**css**('button'));

button.**triggerEventHandler**('**click**', { **stopPropagation: () => {}** })

expect(fixture.componentInstance.delete).**toHaveBeenCalledWith(HEROES[0]**);

})

## Emitting events from children

In the previous test we triggered on event by clicking the button. It is possible to skip the button click and trigger the event by calling the delete property (which is an eventEmitter).

This is the code:

Notice:

1. the emit function does not need a value for the parameter, because the Hero to delete is already bound to this instance of the HeroComponent by the \*ngFor in the HeroesComponent  [hero]="hero".
2. The casting to HeroComponent which is not necessary but does result in a nice list of methods in the outcomplet. Try this at home.
3. Insteads of delete.emit(undefined) you can also use: delete.next();

it(`should call the delete method with the right hero,

when the delete.emit() property of child HeroComponent was called`, () => {

mockHeroService.getHeroes.and.returnValue(of(HEROES));

spyOn(fixture.componentInstance, 'delete');

fixture.detectChanges();

const heroComponents = fixture.debugElement.queryAll(By.directive(HeroComponent));

(<HeroComponent>heroComponents[0].componentInstance).**delete.next();** // or **emit**(undefined);

expect(fixture.componentInstance.delete).toHaveBeenCalledWith(HEROES[0]);

})

## Raising events from children

In the previous test we emitted the event from the child component HeroComponent. As the debugElement of this component has a triggerEventHandler (like all debugElements), it’s also possible to call this trigger eventHandler (instead of calling delete.emit(undefined)’.

The code looks a little smaller, like so:

    it(`should call the delete method with the right hero,

    when the triggerEventHandler on the HeroComponent was called`, () => {

       mockHeroService.getHeroes.and.returnValue(of(HEROES));

       spyOn(fixture.componentInstance, 'delete');

       fixture.detectChanges();

       const heroComponents = fixture.debugElement.queryAll(By.directive(HeroComponent));

       heroComponents[0].**triggerEventHandler('delete', null);**

       expect(fixture.componentInstance.delete).toHaveBeenCalledWith(HEROES[0]);

   })

## Interacting with input boxes

To enter data into in input field and click the add button:

* we have to query the input field and get its nativeElement. Then set the value of that element.
* then query for the button (which is a debugElement) and then call triggerEventHandler(‘click’, null) on it.
* Query listItem (<li> for item 3 (which is the 4th element and check the textContent to contain the added hero

The code:

    it(`should add a new hero, with the inputted name to the heroes list

        when add button is clicked` , () => {

        const heroName = "**Mr. Hans**";

        mockHeroService.getHeroes.and.returnValue(of(HEROES));

        fixture.detectChanges();

        mockHeroService.addHero.and.returnValue(of({ id: **5**, name: **heroName**, strength: 33 }));

        const inputElement = fixture.debugElement.query(By.css('**input**')).nativeElement;

        const addButton = fixture.debugElement.query(By.css('**#addButton**'));

        inputElement.**value** = heroName;

        addButton.triggerEventHandler('click', null);

        fixture.detectChanges();

        // expect(fixture.debugElement.nativeElement.textContent).toContain("Mr. Hans");

        const listElement4 = fixture.debugElement.queryAll(**By.css('li'))[3].nativeElement**;

        expect(listElement4.textContent).toContain("**5 Mr. Hans**");

    })

## Testing with activated route

For this scenario we write a test for the HeroDetailComponent. It uses three services that needs to be mocked. One of them is the ActivatedRoute. Mocking the route could be fairly complex. The route has a snapshot property, which has a paramMap property, which has a get() function: +this.route.**snapshot.paramMap.get('id'); `**

Instead of using jasmine.createSpyObject we create our mockActivateRoute;

**Notice** the import of FormsModule. This is necessary because the template use ngModel which is part of FormsModule.

The code:

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

import { ActivatedRoute } from "@angular/router";

import { HeroService } from "../hero.service";

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

import { HeroDetailComponent } from "./hero-detail.component";

import { of } from "rxjs";

import { FormsModule } from "@angular/forms";

describe("HeroDetailService", () => {

    let mockActivatedRoute;

    let mockHeroService;

    let mockLocation;

    let fixture: ComponentFixture<HeroDetailComponent>;

    beforeEach(() => {

        mockHeroService = jasmine.createSpyObj(["getHero", "updateHero"]);

        mockLocation = jasmine.createSpyObj(["back"]);

        mockActivatedRoute = {

**snapshot**: **{ paramMap: { get: () => { return "1234"; } }** }

        }

        TestBed.configureTestingModule({

            imports: [

**FormsModule**

            ],

            declarations: [

                HeroDetailComponent

            ],

            providers: [

                { provide: ActivatedRoute, useValue: **mockActivatedRoute** },

                { provide: HeroService, useValue: mockHeroService },

                { provide: Location, useValue: mockLocation }

            ],

        })

        fixture = TestBed.createComponent(HeroDetailComponent);

    })

    it('should render hero name in H2 tag', () => {

        mockHeroService.getHero.and.returnValue(of({ id: 1, name: "hans", strength: 32 }));

        fixture.detectChanges();

        expect(fixture.nativeElement.querySelector('h2').textContent).toContain("HANS");

    })

})

## Mocking a directive (routerLink)

If we remove “schemas: “NO\_ERROR\_SCHEMA” from the heroes.component.deep.spec.ts, the test will fail because the child template hero.component.html has a routerLink. Our test doesn’t know a directive “routerLink”.

Let’s deal with this routerLink issue by creating a fake routerLink directive.

This directive should listen to a click event on the ‘host’ (hero.component.html), and then call its onClick() method. The fake directive should be added to the directives in configureTestingModule. Notice that you could put the directive in its own file. But it’s also possible to put it after the describe(…. , in that case the export should be removed.

The code for the directive is added before the describe(….

@Directive({

    selector: "[routerLink]",

    host: {"(click)": "onClick()"}

})

export class RouterLinkDirectiveStub{

    @Input('routerLink') linkParams: any;

    navigatedTo: any = null;

    onClick(){

        this.navigatedTo = this.linkParams;

    }

}

…

Describe(‘HeroesComponent (Deep)…

…

  TestBed.configureTestingModule({

            declarations: [

                HeroesComponent,

                HeroComponent,

                RouterLinkDirectiveStub

            ],

…

## Testing the RouterLink (Stub)

Now let’s test the routerlink directive created in the previous chapter by clicking the link to the first hero. Follow the next steps:

* Get a handle to the first hero (query by directive HeroComponent)
* Get a handle to the anchor tag <a> (query by directive HeroComponent)
* Click the link
* Get the routerlink instance on the <a> tag:
  + Query the heroComponent for the routerlink directive and then
  + Inject the actual RouterLinkDirectiveStub
* Assert the navigateTo property

The code:

   it('should have the correct route for the first hero', () => {

        mockHeroService.getHeroes.and.returnValue(of(HEROES));

        fixture.detectChanges();

        const firstHero = fixture.debugElement.queryAll(By.directive(HeroComponent))[0];

        const debugElementA = firstHero.query(By.css('a'));

        debugElementA.triggerEventHandler('click', null);

        fixture.detectChanges();

        const **routerLink** = firstHero

             .query(By.directive(RouterLinkDirectiveStub))

             .**injector.get**(RouterLinkDirectiveStub);

        expect(routerLink.navigatedTo).toBe("/detail/1");

    })

## Asynchronous code testing

For asynchronous testing we use the save() method of HeroDetailComponent. We will simulate async by calling heroService.saveHero with a timeout of 250 millies.

First approach is using the **done** function. You put the expect in a timeout which is long enough to process all your asynchronous code, and then you call done()

Disadvantage:

* We have to add code (timeout and done) which makes the test more complex.
* We have to build in a delay (for how long?) which makes testing slow.

Code:

    fit('should call updateHero when save is called', (done) => {

        mockHeroService.getHero.and.returnValue(of({id: 1, name: "pietje", strength: 33}));

        mockHeroService.updateHero.and.returnValue(of({}));

        fixture.detectChanges();

        fixture.componentInstance.save();

        setTimeout(() => {

            expect(mockHeroService.updateHero).toHaveBeenCalled();

**done()**;

        }, **300**)

    })

Second approach is using the **fakeAsync** function.

Wrap the whole test a fakeAsync(….) function. Before the expect call tick(nnn) or flush(). tick(nnn) will fastforward the clock by the number of seconds you specifiy. Flush will also do that but determines the number of seconds to forward itself. In most of the cases flush() will be sufficient.

fakeAsync handles observables and promises.

Code:

    fit('should call updateHero when save is called (with done)', fakeAsync**(**() => {

        mockHeroService.getHero.and.returnValue(of({ id: 1, name: "pietje", strength: 33 }));

        mockHeroService.updateHero.and.returnValue(of({}));

        fixture.detectChanges();

        fixture.componentInstance.save();

        flush(); // **or** tick(250);

        expect(mockHeroService.updateHero).toHaveBeenCalled();

    })**)**

Third approach is using the async function. This looks a lot like fakeAsync, but it’s only suitable for promises, not for observables. Async is used combine with whenStable:

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

expect(mockHeroService.updateHero).toHaveBeenCalled();

}

**Conslusion**: Most of the time fakeAsync with flush() or tick(nnn) will be the best option as:

* The code is more simple/easier to read
* Handles both promises and observables
* Does not delay the test by x \* nnn millieseconds

## Code coverage

Run it with: ng test –code-coverage

Reports only over the .spec files. So if you have 100% coverage that means the spec files you have cover 100%. But if a component or a service does not have a spec file coverage will still be 100%

For output open the index.html file in the coverage dir on the root of the project.

# Appendix

## Sources

Cource code: <https://github.com/joeeames/PSAngularUnitTestingCourse>

My code: <https://github.com/drummer0417/ngUnitTesting>

Course: <https://app.pluralsight.com/library/courses/unit-testing-angular/table-of-contents>