**Unit testing in Angular**

Inhoud

[Introduction ( 1 - 2) 2](#_Toc46941718)

[Naming 2](#_Toc46941719)

[AAA 2](#_Toc46941720)

[Testing principals 2](#_Toc46941721)

[Debugging tests in Karma 2](#_Toc46941722)

[Isolated Unit tests 2](#_Toc46941723)

[Mocking a service 2](#_Toc46941724)

[State based- and interaction tests 3](#_Toc46941725)

[Shallow Integration Tests 4](#_Toc46941726)

[TestBed 4](#_Toc46941727)

[A simple component 4](#_Toc46941728)

[A complex component 5](#_Toc46941729)

[Appendix 6](#_Toc46941730)

[Sources 6](#_Toc46941731)

# 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 anolog to given, when, then

## 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…’, () => {}

## 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('shouild 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 (ie heroComponent)

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.

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

If you really need an instance of the 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. 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);

})

});

# Appendix

## Sources

Code source: <https://github.com/joeeames/PSAngularUnitTestingCourse>

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