Testing:

* Unit test / tests in isolation – single unit of code. ( single class )
* E2E testing. – Run against live running application. Performance is poor and less reliable.
* Integration tests – components & its class are tested. Some call it unit tests itself, some call it intergration tests
* functional testing – certain parts of app works with other parts of the application.
* Test async code.
* Maintainable tests.

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

Mocking: - test a single unit of code at a time.

A component can have service dependency – Instead of using real service, we mock them.

Mock is just like a real class, but we control what it returns, what methods were called.

Types of mock – dummy, ( just objects )

Stub – has controllable behavior. If we call a certain method on a stub, we can decide what value on that method call will return.

Spies: - keep track of which of its methods are called, how many times, what parameters were used.

True mocks – used specifically, eg: Http

Types of unit test in angular – isolated, integration ( shallow only that component & deep – parent & child components. )

Tools – karma, jasmine, jest, mocha/chai, sinon mocking library, wallaby is paid, Cyprus e2e testing tool,

// first-test.spec.ts

describe("#my first test", () => {

let sut; // system under test

beforeEach(() => {

sut = {};

});

it("should be true if true", () => {

// arrange

sut.a = false;

// act

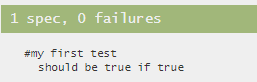
sut.a = true;

// assert

expect(sut.a).toBe(true);

});

});



We run ng test or npm test

AAA pattern – arrange, act, assert.

Damp vs dry - Damp – repeat if necessary.

A test should tell a story and within the it() block. Use setup in beforeEach(), critical seup in it(), AAA inside the it()

**Isolated unit tests:**

Component – plain class if we remove the @Component() decorator.

Pipe also is just a plain class.

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

@Pipe({

name: 'strength'

})

export class StrengthPipe implements PipeTransform {

transform(value: number): string {

if(value < 10) {

return value + " (weak)";

} else if(value >= 10 && value < 20) {

return value + " (strong)";

} else {

return value + " (unbelievable)";

}

}

}

The spec file:

import { StrengthPipe } from "./strength.pipe";

describe("StrengthPipe", () => {

it("should display weak if strenth is 5", () => {

const pipe = new StrengthPipe();

expect(pipe.transform(5)).toEqual("5 (weak)");

});

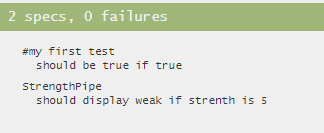
it("should display strong if strenth is 10", () => {

const pipe = new StrengthPipe();

expect(pipe.transform(10)).toEqual("10 (strong)");

});

});



**Test a service:** Message.service.ts

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

@Injectable()

export class MessageService {

messages: string[] = [];

add(message: string) {

this.messages.push(message);

}

clear() {

this.messages = [];

}

}

Spec file:

import { MessageService } from "./message.service";

describe("MessageService", () => {

let service: MessageService;

beforeEach(() => {

service = new MessageService();

});

it("should have no message to start", () => {

expect(service.messages.length).toBe(0);

});

it("should add a message to when add is called", () => {

service.add("Added one more message");

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

});

it("should reset messages on clear", () => {

service.add("Added one more message before reset");

service.clear();

expect(service.messages.length).toBe(0);

});

});

Testing a component with a dependency ( removed @Component decorator )

export class HeroesComponent implements OnInit {

heroes: Hero[];

constructor(private heroService: HeroService) { }

ngOnInit() {

this.getHeroes();

}

getHeroes(): void {

this.heroService.getHeroes()

.subscribe(heroes => this.heroes = heroes);

}

add(name: string): void {

name = name.trim();

var strength = 11

if (!name) { return; }

this.heroService.addHero({ name, strength } as Hero)

.subscribe(hero => {

this.heroes.push(hero);

});

}

delete(hero: Hero): void {

this.heroes = this.heroes.filter(h => h !== hero);

this.heroService.deleteHero(hero).subscribe();

}

}

The component calls only get , add & delete methods in service.

Spec file. ( The service has a dependency on another service ) – **Mocks can isolate code**.

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

import { of } from "rxjs/internal/observable/of";

describe("HeroesComponent", () => {

let component: HeroesComponent;

let HEROES;

let mockHeroService;

beforeEach(() => {

HEROES = [

{ id: 1, name: "A", strength: 8 },

{ id: 2, name: "B", strength: 28 },

{ id: 3, name: "C", strength: 17 }

];

// creates a mock object that we can control & we define its methods

// write the methods we need.

mockHeroService = jasmine.createSpyObj([

"getHeroes",

"addHero",

"deleteHero"

]);

// we need a service as DI here.We cant pass in empty objects.

// We need to pass in an object that look like a HeroService.

// component = new HeroesComponent({}) 🡨 this wont work.

component = new HeroesComponent(mockHeroService);

});

describe("delete", () => {

it("should remove hero from list", () => {

// we need to have an observable to be returned after deleting.

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

component.heroes = HEROES;

component.delete(HEROES[2]);

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

});

});

});

Its trying to delete heroes & then trying to subscribe to the result. Method return an observable which we subscribe to. We need to tell the deleteHero method that we need to return an observable. We need the mock object to return an observable when delete is called. **But we do not check if the third item itself is removed or not.** We just checked if one was removed, not the exact one or not. We arrange in such a way that whenever delete is called, once its done, return an Observable. We did a state based test now.

**Testing interactions:** ( component – service interaction )

it("should call delete hero", () => {

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

component.heroes = HEROES;

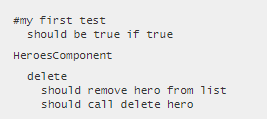
// component.ngOnInit() if we need

component.delete(HEROES[2]);

expect(mockHeroService.deleteHero).toHaveBeenCalled();

expect(mockHeroService.deleteHero).toHaveBeenCalledWith(HEROES[2]);

});



**Shallow integration test:** ( we only test a single component & none of its child comp/directives )

For debugging, open console window open.

TestBed, ignore child directives, testing templates, Dealing with child components.

In package.json

"scripts": {

"ng": "ng",

"build": "ng build --prod",

"start": "ng serve",

"test": "ng test --source-map=false" ,

Karma & zone interact with certain kinds of failing tests, turning off the source map can help us see whats really going on.

What we are going to test is HeroComponent.

export class HeroComponent {

@Input() hero: Hero;

@Output() delete = new EventEmitter();

onDeleteClick($event): void {

$event.stopPropagation();

this.delete.next();

}

}

And in its template, we have a router directive and a button

<a routerLink="/detail/{{hero.id}}">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</a>

<button class="delete" (click)="onDeleteClick($event)">x</button>

**TestBed:**

We begin with shallow test, ignoring other components. In hero.component.shallow.spec.ts

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

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

// please ignore unknown attributes/elements

// but it may hide other issues.

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

describe("HeroComponent -shallow test", () => {

let fixture: ComponentFixture<HeroComponent>;

beforeEach(() => {

// TestBed - allows us to test component & its template.

// We create a module just for testing.

TestBed.configureTestingModule({

declarations: [HeroComponent],

schemas: [NO\_ERRORS\_SCHEMA]

});

// We create a component once module is created.

// Please use the testing module that we made above

// to create the component.

// The createComponent returns a component fixture

// which is a wrapper for a component that has a few

// more properties on top of what a component has.

fixture = TestBed.createComponent(HeroComponent);

// Once we have the fixture, we can grab the component instance

// which is the class part

// or the DOM which is the HTML part. Now if we drill in,

// fixture.componentInstance.delete,

// fixture.componentInstance.hero,

// fixture.componentInstance.onDeleteClick,

});

it("should have the correct Hero", () => {

fixture.componentInstance.hero = { id: 1, name: "SuperDude", strength: 3 };

expect(fixture.componentInstance.hero.name).toEqual("SuperDude");

});

});

Testing rendered HTML: ( nativeElement gives us a handle to DOM element )

it("should render Hero name in anchor tag", () => {

fixture.componentInstance.hero =

{ id: 1, name: "SuperDude", strength: 3 };

// please run change detection & update binding. {{ hero.name }}

fixture.detectChanges();

expect(fixture.nativeElement.querySelector("a").textContent).toContain(

"SuperDude"

);

});

**NativeElement vs DebugElement:**

Debug element is like the native element, it’s a way to access the root element in our template, but instead of the native element which exposes underlying DOM API, debug element is just a wrapper that has a set of functionality similar to nativeElement.

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

expect(

fixture.debugElement.query(By.css("a")).nativeElement.textContent

).toContain("SuperDude");

expect(fixture.nativeElement.querySelector("a").textContent).toContain(

"SuperDude"

);

Or

const de = fixture.debugElement.query(By.css("a"));

expect(de.nativeElement.textContent).toContain("SuperDude");

if we have a debugElement and that has directives, then debugElement can have access to those directives. ( eg: link with routerLink ) But with native element, we cannot. We also can access the componentInstance from debugElement if we wish to.

**More complex shallow integration tests:**

In Heroes.component.html does have other components inside with ngFor. The component has a dependency on a service which again depends on an HTTP and another service.

<div>

<label>Hero name:

<input #heroName />

</label>

<button (click)="add(heroName.value); heroName.value=''">

add

</button>

</div>

<ul class="heroes">

<li \*ngFor="let hero of heroes">

<app-hero [hero]="hero" (delete)="delete(hero)"></app-hero>

</li>

</ul>

And in its class,

export class HeroesComponent implements OnInit {

heroes: Hero[];

constructor(private heroService: HeroService) {}

Spec file: we need to deal with a Service as well as another component.

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

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

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

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

import { of } from "rxjs/internal/observable/of";

// heroes.component.shallow.spec.ts

// We have an injected service.

describe("Heroes shallow test", () => {

let fixture: ComponentFixture<HeroesComponent>;

let mockHeroService;

let HEROES;

beforeEach(() => {

HEROES = [

{ id: 1, name: "A", strength: 8 },

{ id: 2, name: "B", strength: 28 },

{ id: 3, name: "C", strength: 17 }

];

// In the component, what service methods are called.

// Look for service.method() calls that need to be mocked.

// Now we have a mockHero that looks like a real service.

mockHeroService = jasmine.createSpyObj([

"getHeroes",

"addHero",

"deleteHero"

]);

TestBed.configureTestingModule({

declarations: [HeroesComponent],

providers: [{ provide: HeroService, useValue: mockHeroService }],

schemas: [NO\_ERRORS\_SCHEMA] // <app-hero

});

fixture = TestBed.createComponent(HeroesComponent);

});

it("should set heroes correctly from service", () => {

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

// call ngOnInit()

fixture.detectChanges();

expect(fixture.componentInstance.heroes.length).toBe(3);

});

});

**Mocking child components:**

With NO\_ERROR\_SCHEMA, the error will be suppressed even if typo.

We can create a mock child component that looks just like the original component.

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

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

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

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

import { of } from "rxjs/internal/observable/of";

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

@Component({

selector: "app-hero",

template: "<div></div>"

})

class FakeHeroComponent {

@Input() hero: Hero;

}

describe("Heroes shallow test", () => {

let fixture: ComponentFixture<HeroesComponent>;

let mockHeroService;

let HEROES;

beforeEach(() => {

HEROES = [

{ id: 1, name: "A", strength: 8 },

{ id: 2, name: "B", strength: 28 },

{ id: 3, name: "C", strength: 17 }

];

mockHeroService = jasmine.createSpyObj([

"getHeroes",

"addHero",

"deleteHero"

]);

TestBed.configureTestingModule({

declarations: [HeroesComponent, FakeHeroComponent],

providers: [{ provide: HeroService, useValue: mockHeroService }]

});

fixture = TestBed.createComponent(HeroesComponent);

});

it("should set heroes correctly from service", () => {

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

// call ngOnInit()

fixture.detectChanges();

expect(fixture.componentInstance.heroes.length).toBe(3);

});

});

**Dealing with list of elements:** ( ie the template )

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

it("shoule create a list item for each heroes", () => {

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

fixture.detectChanges();

expect(fixture.debugElement.queryAll(By.css("li")).length).toBe(3);

});

**Deep integration tests:**

Test components with live chidren. The same heroes.component.html. We test interaction b/w heroes & hero component. In heroes.component.deep.spec.ts

// heroes.component.deep.spec.ts

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

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

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

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

import { of } from "rxjs/internal/observable/of";

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

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

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

// we need to see hero & heroes interact, but

// without live service being injected.

describe("Heroes deep test", () => {

let fixture: ComponentFixture<HeroesComponent>;

let mockHeroService;

let HEROES;

beforeEach(() => {

HEROES = [

{ id: 1, name: "A", strength: 8 },

{ id: 2, name: "B", strength: 28 },

{ id: 3, name: "C", strength: 17 }

];

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 each hero as hero component", () => {

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

fixture.detectChanges();

// get every one of <app-hero> elements that is looped with \*ngFor

// queryAll returns debugElement. One eg where we use

// componentInstance along with debugElement is below.

const heroCompDEs = fixture.debugElement.queryAll(

By.directive(HeroComponent)

);

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

expect(heroCompDEs[0].componentInstance.hero.name).toEqual("A");

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

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

}

});

});

**Integration testing of Services:**

With http, in service, we call this.http.get() . We need the module first, and then the controller.

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

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

import { MessageService } from "./message.service";

// creates a mock http client testing module

// We need a handle to Mock http client service to adjust

// it & control it over the test - HttpTestingController

import {

HttpClientTestingModule,

HttpTestingController

} from "@angular/common/http/testing";

// we make http.get() requests.

// We can provide a special mock http service.

describe("HeroService", () => {

let mockMessageService;

let httpTestingController: HttpTestingController;

let service: HeroService;

beforeEach(() => {

// In HeroService, we only have add method

mockMessageService = jasmine.createSpyObj(["add"]);

TestBed.configureTestingModule({

imports: [HttpClientTestingModule],

providers: [

HeroService,

{

provide: MessageService,

useValue: mockMessageService

}

]

});

// Look inside DI registry,find the service & give is a handle to it.

httpTestingController = TestBed.get( HttpTestingController );

service = TestBed.get(HeroService);

});

describe("getHero()", () => {

it("shud call get with correct URL", () => {

// step 1 – console.log() is executed after the flush() only.

service.getHero(4).subscribe(() => {

console.log("fullfilled.");

});

// step 2: Create a request to this endpoint

const req = httpTestingController.expectOne("api/heroes/4");

// step 3: data we need to come back

req.flush({ id: 4, name: "SuperDude", strength: 100 });

// step 4: - verifies exactly what we expected.

httpTestingController.verify();

});

});

});

If we make multiple calls, ie, service.getHero(4).subscribe() multiple times, or getHero(3) with another parameter, test may pass, but if we make a call to correct URL and incorrect URL, test passes, which is an issue.

The service method under test is:

getHero(id: number): Observable<Hero> {

const url = `${this.heroesUrl}/${id}`;

return this.http.get<Hero>(url).pipe(

tap(\_ => this.log(`fetched hero id=${id}`)),

catchError(this.handleError<Hero>(`getHero id=${id}`))

);

}

We could have the service inside the beforeEach too with service = TestBed.get(HeroService) and do let service: HeroService. One approach of DI the service inside the test is,

it("shud call get with correct URL", inject(

[HeroService, HttpTestingController],

(service: HeroService, ctrl: HttpTestingController) => {

service.getHero(4).subscribe();

}

));

**Testing DOM interaction & routing components:**

Trigger event on an element:

In heroes.component.html,

<li \*ngFor="let hero of heroes">

<app-hero [hero]="hero" (delete)="delete(hero)"></app-hero>

</li>

When delete() is called in child, the same is handled in parent.

In hero.component.html,

<button class="delete" (click)="onDeleteClick($event)">x</button>

@Output() delete = new EventEmitter();

onDeleteClick($event): void {

$event.stopPropagation();

this.delete.next();

}

In heroes.component.deep.spec.ts

it(`should call heroService.deleteHero() when

heroComponent delete button is clicked`, () => {

spyOn(fixture.componentInstance, "delete");

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

fixture.detectChanges();

const heroComponents = fixture.debugElement.queryAll(

By.directive(HeroComponent)

);

heroComponents[0].query(By.css("button")).triggerEventHandler("click", {

stopPropagation: () => {}

});

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

});

});

**Emitting events from children:**

// Here we clicked the button

// heroComponents[0].query(By.css("button")).triggerEventHandler("click", {

// stopPropagation: () => {}

// });

// Here, we raised the delete event

(<HeroComponent>heroComponents[0].componentInstance).delete.emit(undefined);

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

Here we have seen 2 variations, one via the DOM and the other via the component instance.

**Raising events on child directives:**

// Here, we raised the delete event

// (<HeroComponent>heroComponents[0].componentInstance)

.delete.emit(undefined);

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

we tell the debugElement to raise that event. We don’t even know if the child has an event emitter called delete()

**Interact with input boxes:**

Adding a new item in input and clicking button will add. In heroes.component.ts

<div>

<label>Hero name:

<input #heroName />

</label>

<button (click)="add(heroName.value); heroName.value=''">

add

</button>

In hero.component.deep.spec.ts

it("should add a new hero to a list when the add btn is clicked", () => {

// we need an initial set of stuff from service

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

// ngOnInit()

fixture.detectChanges();

// we type in a new item

const name = "Mr.Ice";

// when we call add, we should return the new Hero object

mockHeroService.addHero.and.returnValue(

of({ id: 5, name: name, strength: 4 })

);

// get the input

const inputEl = fixture.debugElement.query(By.css("input")).nativeElement;

// get add button

const addBtn = fixture.debugElement.queryAll(By.css("button"))[0];

// we typed Mr.Ice

inputEl.value = name;

// we clicked add buton

addBtn.triggerEventHandler("click", null);

fixture.detectChanges();

const heroText = fixture.debugElement.query(By.css("ul")).nativeElement

.textContent;

expect(heroText).toContain(name);

});

A new hero component is added to the list.

**Testing ActivatedRoute:**

**Don’t test framework when testing component**. We just test our code. Assume framework works correctly. Don’t do angular routing, but just interactions.

In hero-detail.component.ts,

export class HeroDetailComponent implements OnInit {

@Input() hero: Hero;

constructor(

private route: ActivatedRoute,

private heroService: HeroService,

private location: Location

) {}

ngOnInit(): void {

this.getHero();

}

getHero(): void {

const id = +this.route.snapshot.paramMap.get("id");

this.heroService.getHero(id).subscribe(hero => (this.hero = hero));

}

In template,

<div \*ngIf="hero">

<h2>{{ hero.name | uppercase }} Details</h2>

<div><span>id: </span>{{hero.id}}</div>

<div>

<label>name:

<input [(ngModel)]="hero.name" placeholder="name"/>

</label>

</div>

<button (click)="goBack()">go back</button>

<button (click)="save()">save</button>

</div>

In hero-detail.component.spec.ts

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

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

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

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

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

import { of } from "rxjs/internal/observable/of";

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

// we need to mock 3 services.

describe("Herodetail", () => {

// Activated route API is complex.

let fixture: ComponentFixture<HeroDetailComponent>;

let mockActivatedRoute, mockHeroService, mockLocation;

beforeEach(() => {

mockActivatedRoute = {

snapshot: {

paramMap: {

get: () => {

return "3";

}

}

}

};

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

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

TestBed.configureTestingModule({

imports: [FormsModule],

declarations: [HeroDetailComponent],

providers: [

{ provide: ActivatedRoute, useValue: mockActivatedRoute },

{ provide: HeroService, useValue: mockHeroService },

{ provide: Location, useValue: mockLocation }

]

});

fixture = TestBed.createComponent(HeroDetailComponent);

mockHeroService.getHero.and.returnValue(

of({

id: 3,

name: "Superdude",

strength: 100

})

);

});

it("should render hero name in h2 tag", () => {

// act

fixture.detectChanges();

expect(fixture.nativeElement.querySelector("h2").textContent).toContain(

"SUPERDUDE"

);

});

});

**Mock routerLink:**

In heroes.component.deep.spec.ts, if we take out schema, then we will get the error.

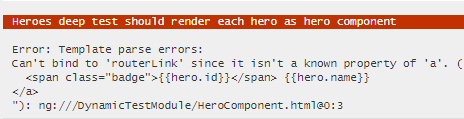
TestBed.configureTestingModule({

declarations: [HeroesComponent, HeroComponent],

providers: [{ provide: HeroService, useValue: mockHeroService }]

// schemas: [NO\_ERRORS\_SCHEMA]

});



@Directive({

selector: "[routerLink]",

host: {

"(click)": "onClick()" // listen for a click event

}

})

export class RouterLinkStub {

@Input("routerLink") linkParams: any; // routerLink=”/detail/4”

navigatedTo: any = null;

onClick() {

this.navigatedTo = this.linkParams;

}

}

And we need to register the stub:

TestBed.configureTestingModule({

declarations: [HeroesComponent, HeroComponent, RouterLinkStub],

providers: [{ provide: HeroService, useValue: mockHeroService }]

// schemas: [NO\_ERRORS\_SCHEMA]

});

Now that we have stubbed the router link, we need to test the routerLink: In heroes.deep.spec.ts,

it("shud have correct route for first hero", () => {

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

fixture.detectChanges();

const heroComps = fixture.debugElement.queryAll(

By.directive(HeroComponent)

);

// get routerLink for the first <app-hero>

const routerLink = heroComps[0]

// gives debugElement for <a routerLink=/detail/1 /> anchor tag that

// the routerLink on it.

.query(By.directive(RouterLinkStub))

// handle to the RouterLinkStub class that has navigatedTo ppty

.injector.get(RouterLinkStub); // need handle to class

heroComps[0].query(By.css("a")).triggerEventHandler("click", null);

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

});

**Advanced topics:**

Dealing with async code in tests. CLI to generate code coverage reports.

save(): void {

debounce(

() => {

this.heroService.updateHero(this.hero).subscribe(() => this.goBack());

},

250, false)();

}

In hero-detail.component.ts,

function debounce(func, wait, immediate) {

let timeout;

return function() {

const context = this,

args = arguments;

const later = function() {

timeout = null;

if (!immediate) {

func.apply(context, args);

}

};

const callNow = immediate && !timeout;

clearTimeout(timeout);

timeout = setTimeout(later, wait);

if (callNow) {

func.apply(context, args);

}

};

}

In hero-detail.component.spec.ts, Our test is sync, but code is not. So we need to add timeout as an option 1

it("shud call updateHero when save is called", () => {

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

fixture.detectChanges();

fixture.componentInstance.save();

setTimeout(() => {

expect(mockHeroService.updateHero).toHaveBeenCalled();

}, 300);

});

Another option is to use done() given by jasmine.

// test waits until async code is done

it("shud call updateHero when save is called", done => {

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

fixture.detectChanges();

fixture.componentInstance.save();

setTimeout(() => {

expect(mockHeroService.updateHero).toHaveBeenCalled();

done();

}, 300);

});

But this will make testing slow with the setTimeout() method.

We can deal by using fakeAsync helper.

import {

TestBed, ComponentFixture, fakeAsync, tick, flush} from "@angular/core/testing";

it("shud call updateHero when save is called", fakeAsync(() => {

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

fixture.detectChanges();

fixture.componentInstance.save();

tick(250);

expect(mockHeroService.updateHero).toHaveBeenCalled();

}));

// if we do not know the exact time, instead of tick

flush();

**Use async helper:**

Similar to fake async, but works differently. If there are promises in code, we can try this.

import {TestBed,ComponentFixture,fakeAsync,tick,flush,async} from "@angular/core/testing";

it("should render hero name in h2 tag", async(() => {

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

fixture.detectChanges();

fixture.componentInstance.save();

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

expect(mockHeroService.updateHero).toHaveBeenCalled();

});

}));

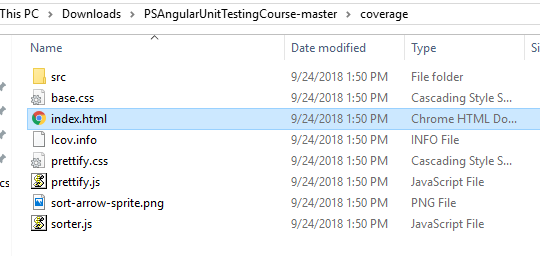
fakeAsync can work with promise as well as setTimeOut. This can be a default option.

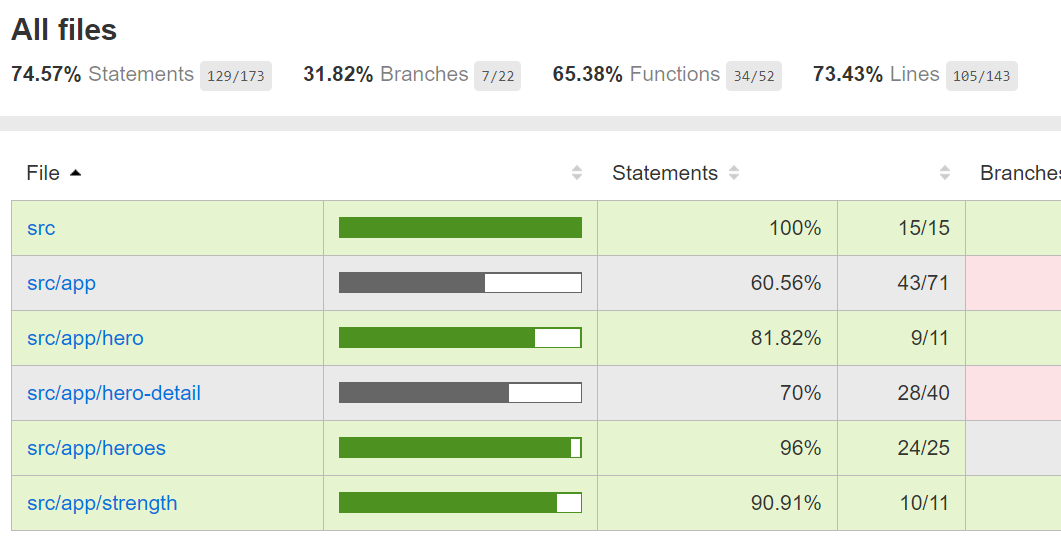
The async is only capable of working with promises.

**Code coverage report:**

"gencode": "ng test --code-coverage"







Limitation being, if we have a component that we have not wrote any test, ie, no spec for it, then that component is not considered for generating reports.