JavaScript - Frameworks - Angular - Testing

# Setup

Angular CLI installs the Jasmine test framework. To run tests, run the karma test runner:

ng test

The karma config is in karma.conf.js

Unit specs are placed in the same folder as the component with the same name, appended with .spec.ts.

Integration tests are placed in a test folder.

<https://angular.io/guide/testing-utility-apis>

### Headless

Add headless chrome for the karma.conf.js to run the tests without opening a new browser a wasting:

browsers: ['ChromeHeadlessCI'],

customLaunchers: {

ChromeHeadlessCI: {

base: 'ChromeHeadless',

flags: ['--no-sandbox']

}

},

Setup a new protractor config to use headless chrome, protractor-headless.conf.js:

const config = require('./protractor.conf').config;

config.capabilities = {

browserName: 'chrome',

chromeOptions: {

args: ['--headless', '--no-sandbox', '--window-size=1600x1000']

}

};

exports.config = config;

Run using:

ng test --browsers=ChromeHeadlessCI

ng e2e --protractor-config=e2e/protractor-ci.conf.js

### Code Coverage

To get code coverage for a test run add the code coverage flag to ng test. For example:

ng test --no-watch --code-coverage

Alternatively, add to test options in angular.json:

"test": {

"options": {

"codeCoverage": true

}

}

# e2e

While unit tests are the first line of defence, it is unrealistic to rely on manual testing to verify the correctness of new features. Angular uses Protractor to run end to end tests using nodejs.

Protractor uses jasmine for test syntax. Sets of tests are called scenarioes. Since tests load the whole application, they are much slower than unit tests, therefore should only test critical system functionality.

## Test Syntax

Test process:

Load page

For most tests, pageloading is simply opening the index, sometime however a script may need to be run to authenticate a user or direct to another section of the application:

browser.get('/');

### Locators

Find elements on the page using:

* class name
* id
* model
* binding

Find elements by using the by.css('<selector>') syntax, for example:

let elem = element(by.css(".nested-class"));

Page objects can be used to encapsulate common locators, for example finding a homepage:

export class HomePage {

nestedClass;

constructor() {

this.nestedClass = element(by.css(".nested-class"));

}

}

Then import and create the homepage in the beforeEach hook.

### Async Features

Asynchronous features can be handled using the wait function:

browser.wait(function() {

return browser.isElementPresent(by.css(".dynamic-text"));

}, 5000);

The browser will then wait a period and then run a callback.

### Actions

There are various actions and events which can be run on elements:

* elem.sendKeys - type something in as an input
* elem.click - click elem
* elem.clear - erase input
* elem.getAttribute('<attr-name>') - get specific attribute
* elem.submit - submit form
* elem.isPresent - test if element is present boolean

Full list: <https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/interactions/Actions.html>

### Expectations

Expectations can then be run on elements and attributes returned.

expect(element.all(by.css('[ng-view] p')).first().getText()).toMatch(/partial for view 2/);

## Mock API Calls

# Services

Test services to ensure they work and provide maintainability.

## Test format

Unit tests follow jasmine format, using describe and it blocks to run tests. Tests are written in typescript.

An example service test would instantiate a service class, then run a method on the service. For example:

describe('ValueService', () => {

let service: ValueService;

beforeEach(() => { service = new ValueService(); });

it('#getValue should return real value', () => {

expect(service.getValue()).toBe('real value');

});

it('#getObservableValue should return value from observable',

(done: DoneFn) => {

service.getObservableValue().subscribe(value => {

expect(value).toBe('observable value');

done();

});

});

});

## Dependencies

Dependencies in Angular are normally injected services, handled by the DI injector. In tests these are normally mocked using jasmine spies, or using Angular TestBed.

TestBed creates a dynamically contructed Angular test module which emulates @NgModule. The TestBed is configured using:

beforeEach(() => {

TestBed.configureTestingModule({ providers: [ValueService] });

service = TestBed.inject(ValueService);

});

Mocks can then be added to the TestBed by creating a spy object and providing with the useValue option:

let masterService: MasterService;

let valueServiceSpy: jasmine.SpyObj<ValueService>;

beforeEach(() => {

const spy = jasmine.createSpyObj('ValueService', ['getValue']);

TestBed.configureTestingModule({

// Provide both the service-to-test and its (spy) dependency

providers: [

MasterService,

{ provide: ValueService, useValue: spy }

]

});

// Inject both the service-to-test and its (spy) dependency

masterService = TestBed.inject(MasterService);

valueServiceSpy = TestBed.inject(ValueService) as jasmine.SpyObj<ValueService>;

});

## Http Services

To test services which rely on HttpClient, inject the HttpClient to the TestBed as normal. Then return asyncData instead of synchronous data:

let httpClientSpy: { get: jasmine.Spy };

let heroService: HeroService;

beforeEach(() => {

// TODO: spy on other methods too

httpClientSpy = jasmine.createSpyObj('HttpClient', ['get']);

heroService = new HeroService(httpClientSpy as any);

});

it('should return expected heroes (HttpClient called once)', () => {

const expectedHeroes: Hero[] =

[{ id: 1, name: 'A' }, { id: 2, name: 'B' }];

httpClientSpy.get.and.returnValue(asyncData(expectedHeroes));

heroService.getHeroes().subscribe(

heroes => expect(heroes).toEqual(expectedHeroes, 'expected heroes'),

fail

);

expect(httpClientSpy.get.calls.count()).toBe(1, 'one call');

});

it('should return an error when the server returns a 404', () => {

const errorResponse = new HttpErrorResponse({

error: 'test 404 error',

status: 404, statusText: 'Not Found'

});

httpClientSpy.get.and.returnValue(asyncError(errorResponse));

heroService.getHeroes().subscribe(

heroes => fail('expected an error, not heroes'),

error => expect(error.message).toContain('test 404 error')

);

});

### HttpClientTestingModule

While simple mock of HttpClient is manageable, use HttpClientTestingModule from the @angular/common/http/testing library to make scenarios more manageable.

The testing library process is:

* service makes requests by subscribing
* expectations are run on what requests are made
* responses are flushed
* expectations are run on output of service responses

To start import the required modules:

// Http testing module and mocking controller

import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';

// Other imports

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

import { HttpClient, HttpErrorResponse } from '@angular/common/http';

Add to TestBed via configureTestingModule and injection method as above.

In the test, make a request via the service. Then expect the request parameters to have been passed using:

* expectOne('<requesturl>') - expect exactly one request to url
* match('<requesturl') - return array of requests to url, expect on length

httpTestingController.expectOne('<requesturl>')

expect(req.request.method).toEqual('GET');

Then flush the request and pass the response data to be returned by the the observer:

req.flush(testData)

Finally verify there are no more requests to validate that all tests have been tested:

httpTestingController.verify()

More than one request can be accounded for using .match instead of .expectOne

Check for errors by passing the correct handlers into the subscribe observer, and expecting an error.

httpClient.get<Data[]>(testUrl).subscribe(

data => fail('should have failed with the 404 error'),

(error: HttpErrorResponse) => {

expect(error.status).toEqual(404, 'status');

expect(error.error).toEqual(emsg, 'message');

}

);

Then flush with an error response:

req.flush(emsg, { status: 404, statusText: 'Not Found' });

Or a network error:

const mockError = new ErrorEvent('Network error', {

message: emsg,

});

req.error(mockError);

# Components

Since a component combines HTML template and a TypeScript class, to test the component, both need to be tested together.

Components can be tested using the DOM or just the class alone without DOM involvement.

## Unit Tests

Unit tests are performed without DOM in a very similar way to testing:

* Create component class instance using new keyword
* Interact with component using API
* Perform expectations on component state

To test Input and Output emitters, subscribe to the observable returned from the output emitter:

it('raises the selected event when clicked', () => {

const comp = new DashboardHeroComponent();

const hero: Hero = {id: 42, name: 'Test'};

comp.hero = hero;

comp.selected.subscribe((selectedHero: Hero) => expect(selectedHero).toBe(hero));

comp.click();

});

### Dependencies

To manage dependencies on services, first mock them using spys or simple objects. Then provide and inject them into the TestBed in beforeEach(). When mocking a class, use the original name with Mock appended. Then use the useClass option on providers to specify the mocked class with the name of the original class:

{ provide: UserService, useClass: MockUserService }

Injecting services into the TestBed can be seen as being similar to providing the service in the root module of the component. Therefore, if the component is provided in the component itself, the service must be injected into the component fixtures injector:

userService = fixture.debugElement.injector.get(UserService);

### Lifecycle Methods

To manage lifecycle methods simply exercise them in a similar way to Angular by calling them manually.

it('should welcome logged in user after Angular calls ngOnInit', () => {

comp.ngOnInit();

expect(comp.welcome).toContain(userService.user.name);

});

### DOM Components

TestBed provides methods to test the creation of components using TestBed.createComponent(). Below the fixture.nativeElement will provide the HTML of the component. For example:

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

import { BannerComponent } from './banner.component';

describe('BannerComponent', () => {

let component: BannerComponent;

let fixture: ComponentFixture<BannerComponent>;

beforeEach(waitForAsync(() => {

TestBed.configureTestingModule({declarations: [BannerComponent]}).compileComponents();

}));

beforeEach(() => {

fixture = TestBed.createComponent(BannerComponent);

component = fixture.componentInstance;

fixture.detectChanges();

});

it('should create', () => {

expect(component).toBeDefined();

});

it('should contain "banner works!"', () => {

const bannerElement: HTMLElement = fixture.nativeElement;

const p = bannerElement.querySelector('p');

expect(p.textContent).toEqual('banner works!');

});

});

fixture.detectChanges() must be run after the component creation to run an Angular change detection cycle. The change detection cycle will interpolate any values from the template into the DOM, allowing them to be tested.

Use dispatchEvent() before detectChanges() to trigger events such as user inputs or button clicks on elements.

### Async Mock Services

<https://angular.io/guide/testing-components-scenarios#async-test-with-fakeasync>

The fakeAsync() test wrapper simulates the passage of time, using tick(<time>) to pass time.

Use asyncData(<data>) to mock async observables, write the helper file:

export function asyncData<T>(data: T) {

return defer(() => Promise.resolve(data));

}

Import and use in test:

getQuoteSpy.and.returnValue(asyncData(testQuote));

# Directives

To test directives it is common to create an artifical test component which will demonstate the full extent of the directive. For example the highlight directive:

import { Directive, ElementRef, Input, OnChanges } from '@angular/core';

@Directive({ selector: '[highlight]' })

export class HighlightDirective implements OnChanges {

defaultColor = 'rgb(211, 211, 211)'; // lightgray

@Input('highlight') bgColor: string;

constructor(private el: ElementRef) {

el.nativeElement.style.customProperty = true;

}

ngOnChanges() {

this.el.nativeElement.style.backgroundColor = this.bgColor || this.defaultColor;

}

}

Create a test component:

@Component({

template: `

<h2 highlight="yellow">Something Yellow</h2>

<h2 highlight>The Default (Gray)</h2>

<h2>No Highlight</h2>

<input #box [highlight]="box.value" value="cyan"/>`

})

class TestComponent { }

Then run tests:

beforeEach(() => {

fixture = TestBed.configureTestingModule({

declarations: [ HighlightDirective, TestComponent ]

})

.createComponent(TestComponent);

fixture.detectChanges(); // initial binding

// all elements with an attached HighlightDirective

des = fixture.debugElement.queryAll(By.directive(HighlightDirective));

// the h2 without the HighlightDirective

bareH2 = fixture.debugElement.query(By.css('h2:not([highlight])'));

});

// color tests

it('should have three highlighted elements', () => {

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

});

it('should color 1st <h2> background "yellow"', () => {

const bgColor = des[0].nativeElement.style.backgroundColor;

expect(bgColor).toBe('yellow');

});

it('should color 2nd <h2> background w/ default color', () => {

const dir = des[1].injector.get(HighlightDirective) as HighlightDirective;

const bgColor = des[1].nativeElement.style.backgroundColor;

expect(bgColor).toBe(dir.defaultColor);

});

it('should bind <input> background to value color', () => {

// easier to work with nativeElement

const input = des[2].nativeElement as HTMLInputElement;

expect(input.style.backgroundColor).toBe('cyan', 'initial backgroundColor');

input.value = 'green';

// Dispatch a DOM event so that Angular responds to the input value change.

// In older browsers, such as IE, you might need a CustomEvent instead. See

// https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent/CustomEvent#Polyfill

input.dispatchEvent(new Event('input'));

fixture.detectChanges();

expect(input.style.backgroundColor).toBe('green', 'changed backgroundColor');

});

it('bare <h2> should not have a customProperty', () => {

expect(bareH2.properties.customProperty).toBeUndefined();

});

# Pipes

Since pipes should be stateless and have one method, they can be tested quite easily. First create the pipe, such as the title-case pipe:

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

@Pipe({name: 'titlecase', pure: true})

export class TitleCasePipe implements PipeTransform {

transform(input: string): string {

return input.length === 0 ? '' :

input.replace(/\w\S\*/g, (txt => txt[0].toUpperCase() + txt.substr(1).toLowerCase() ));

}

}

Then run specs on instance of class:

describe('TitleCasePipe', () => {

const pipe = new TitleCasePipe();

it('transforms "abc" to "Abc"', () => {

expect(pipe.transform('abc')).toBe('Abc');

});

it('transforms "abc def" to "Abc Def"', () => {

expect(pipe.transform('abc def')).toBe('Abc Def');

});

});

Cover the edge cases with the unit test for the pipe, cover implementation standard use in component test.