Angular Test

Why Test?

From personal experience, tests are the best way to prevent software defects. I’ve been on many teams in the past where a small piece of code is updated and the developer manually opens their browser or Postman to verify that it still works. This approach (manual QA) is begging for a disaster.

Tests are the best way to prevent software defects.

As features and codebases grow, manual QA becomes more expensive, time consuming, and error prone. If a feature or function is removed does every developer remember all of its potential side-effects? Are all developers manually testing in the same way? Probably not.

The reason we test our code is to verify that it behaves as we expect it to. As a result of this process you’ll find you have better feature documentation for yourself and other developers as well as a design aid for your APIs.

Why Karma?

Karma is a direct product of the AngularJS team from struggling to test their own framework features with existing tools. As a result of this, they made Karma and have transitioned it to Angular as the default test runner for applications created with the Angular CLI.

In addition to playing nicely with Angular, it also provides flexibility for you to tailor Karma to your workflow. This includes the option to test your code on various browsers and devices such as phones, tablets, and even a [PS3](https://testing.googleblog.com/2012/11/testacular-spectacular-test-runner-for.html) like the YouTube team.

Karma also provides you options to replace Jasmine with other testing frameworks such as [Mocha](https://mochajs.org/) and [QUnit](https://qunitjs.com/) or integrate with various continuous integration services like [Jenkins](https://jenkins.io/), [TravisCI](https://travis-ci.org/), or [CircleCI](https://circleci.com/).

Unless you add some additional configuration your typical interaction with Karma will be to run ng test in a terminal window.

Why Jasmine?

Jasmine is a behavior-driven development framework for testing JavaScript code that plays very well with Karma. Similar to Karma, it’s also the recommended testing framework within the [Angular documentation](https://angular.io/guide/testing#setup) as it’s setup for you with the Angular CLI. Jasmine is also dependency free and doesn’t require a DOM.

As far as features go, I love that Jasmine has almost everything I need for testing built into it. The most notable example would be spies. A spy allows us to “spy” on a function and track attributes about it such as whether or not it was called, how many times it was called, and with which arguments it was called. With a framework like Mocha, spies are not built-in and would require pairing it with a separate library like Sinon.js.

The good news is that the switching costs between testing frameworks is relatively low with differences in syntax as small as Jasmine’s toEqual() and Mocha’s to.equal().

A Simple Test Example

Imagine you had an alien servant named Adder who follows you everywhere you go. Other than being a cute alien companion Adder can really only do one thing, add two numbers together.

To verify Adder’s ability to add two numbers we could generate a set of test cases to see if Adder provides us the correct answer.

Within Jasmine, this would begin with what’s referred to as a [“suite”](https://jasmine.github.io/tutorials/your_first_suite) which groups a related set of tests by calling the function describe.

// A Jasmine suite

describe('Adder', () => {

});

Copy

From here we could provide Adder with a set of test cases such as two positive numbers (2, 4), a positive number and a zero (3, 0), a positive number and a negative number (5, -2), and so on.

Within Jasmine, these are referred to as [“specs”](https://jasmine.github.io/tutorials/your_first_suite) which we create by calling the function it, passing it a string to describe the functionality that’s being tested.

describe('Adder', () => {

// A jasmine spec

it('should be able to add two whole numbers', () => {

expect(Adder.add(2, 2)).toEqual(4);

});

it('should be able to add a whole number and a negative number', () => {

expect(Adder.add(2, -1)).toEqual(1);

});

it('should be able to add a whole number and a zero', () => {

expect(Adder.add(2, 0)).toEqual(2);

});

});

Copy

Within each spec we call expect and provide it what is referred to as an “actual"—the call site of our *actual* code. After the expectation, or expect, is the chained "matcher” function, such as toEqual, which the testing developer provides with the *expected output* of the code being tested.

There are many other matchers available to us other than toEqual. You can see a full list within [Jasmine’s documentation](https://jasmine.github.io/api/edge/matchers.html).

Our tests aren’t concerned with *how* Adder arrives at the answer. We only care about the answer Adder provides us. For all we know, this may be Adder’s implementation of add.

function add(first, second) {

if (true) { // why?

if (true) { // why??

if (1 === 1) { // why?!?1

return first + second;

}

}

}

}

Copy

In other words, we only care that Adder behaves as expected—we have no concern for Adder’s implementation.

This is what makes a practice like test-driven development (TDD) so powerful. You can first write a test for a function and its expected behavior and get it to pass. Then, once it’s passing, you can refactor your function to a different implementation and if the test is still passing, you know your function is still behaving as specified within your tests even with a different implementation. Adder’s add function would be a good example!

Angular setup

We’ll begin by creating our new application using the Angular CLI.

ng new angular-testing --routing

Copy

Since we’ll have multiple views in this application we use the --routing flag so the CLI automatically generates a routing module for us.

From here we can verify everything is working correctly by moving into the new angular-testing directory and running the application.

cd angular-testing

ng serve -o

Copy

You can also verify the application’s tests are currently in a passing state.

ng test

Copy

Adding a home page

Before creating a service to populate our home page with users, we’ll start by creating the home page.

ng g component home

Copy

Now that our component has been created, we can update our routing module’s (app-routing.module.ts) root path to HomeComponent.

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

import { RouterModule, Routes } from '@angular/router';

import { HomeComponent } from './home/home.component';

const routes: Routes = [

{ path: '', component: HomeComponent }

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

Copy

Run the application if it isn’t already and you should now see “home works!” below the default template in app.component.html which was created by the CLI.

Removing AppComponent tests

Since we no longer need the default contents of AppComponent, let’s update it by removing some unnecessary code.

First, remove everything in app.component.html so that only the router-outlet directive remains.

<router-outlet></router-outlet>

Copy

Within app.component.ts, you can also remove the title property.

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent { }

Copy

Finally, you can update the tests in app.component.spec.ts by removing the two tests for some of the contents that were previously in app.component.html.

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

import { RouterTestingModule } from '@angular/router/testing';

import { AppComponent } from './app.component';

describe('AppComponent', () => {

beforeEach(async(() => {

TestBed.configureTestingModule({

imports: [

RouterTestingModule

],

declarations: [

AppComponent

],

}).compileComponents();

}));

it('should create the app', async(() => {

const fixture = TestBed.createComponent(AppComponent);

const app = fixture.debugElement.componentInstance;

expect(app).toBeTruthy();

}));

});

Copy

Testing an Angular service

Now that our home page is set up we can work on creating a service to populate this page with our directory of employees.

ng g service services/users/users

Copy

Here we’ve created our users service within a new services/users directory to keep our services away from the default app directory which can get messy quick.

Now that our service is created, we can make a few small changes to the test file services/users/users.service.spec.ts.

I personally find injecting dependencies within it() to be a bit repetitive and harder to read as it’s done in the default scaffolding for our test file as shown below:

it('should be created', inject([TestService], (service: TestService) => {

expect(service).toBeTruthy();

}));

Copy

With a few minor changes, we can move this into the beforeEach removing the duplication from each it.

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

import { UsersService } from './users.service';

describe('UsersService', () => {

let usersService: UsersService; // Add this

beforeEach(() => {

TestBed.configureTestingModule({

providers: [UsersService]

});

usersService = TestBed.get(UsersService); // Add this

});

it('should be created', () => { // Remove inject()

expect(usersService).toBeTruthy();

});

});

Copy

In the code above, TestBed.configureTestingModule({}) sets up the service we want to test with UsersService set in providers. We then inject the service into our test suite using TestBed.get() with the service we want to test as the argument. We set the return value to our local usersService variable which will allow us to interact with this service within our tests just as we would within a component.

Now that our test setup is restructured, we can add a test for an all method which will return a collection of users.

import { of } from 'rxjs'; // Add import

describe('UsersService', () => {

...

it('should be created', () => {

expect(usersService).toBeTruthy();

});

// Add tests for all() method

describe('all', () => {

it('should return a collection of users', () => {

const userResponse = [

{

id: '1',

name: 'Jane',

role: 'Designer',

pokemon: 'Blastoise'

},

{

id: '2',

name: 'Bob',

role: 'Developer',

pokemon: 'Charizard'

}

];

let response;

spyOn(usersService, 'all').and.returnValue(of(userResponse));

usersService.all().subscribe(res => {

response = res;

});

expect(response).toEqual(userResponse);

});

});

});

Copy

Here we add a test for the expectation that all will return a collection of users. We declare a userResponse variable set to a mocked response of our service method. Then we use the spyOn() method to spy on usersService.all and chain .returnValue() to return our mocked userResponse variable wrapping it with of() to return this value as an observable.

With our spy set, we call our service method as we would within a component, subscribe to the observable, and set its return value to response.

Finally, we add our expectation that response will be set to the return value of the service call, userResponse.

Why mock?

At this point many people ask, “Why are we mocking the response?” Why did we provide our test a return value userResponse that we created ourselves, to manually set what’s being returned from our service? Shouldn’t the service call return the *real* response from the service, whether it’s a hard-coded set of users or a response from an HTTP request?

This is a perfectly reasonable question to ask and one that can be hard to wrap your head around when you first begin testing. I find this concept is easiest to illustrate with a real world example.

Imagine you own a restaurant and it’s the night before opening day. You gather everyone you’ve hired for a “test run” of the restaurant. You invite a few friends to come in and pretend they’re customers who will sit down and order a meal.

No dishes will actually be served in your test run. You’ve already worked with your cooks and are satisfied they can make the dishes correctly. In this test run you want to test the transition from the customer ordering their dish, to the waiter sending that to the kitchen, and then the waiters fulfilling the kitchen’s response to the customer. This response from the kitchen may be one of a few options.

1. The meal is ready.
2. The meal is delayed.
3. The meal cannot be made. We ran out of ingredients for the dish.

If the meal is ready, the waiter delivers the meal to the customer. However, in the event that a meal is late or cannot be made, the waiter will have to go back to the customer, apologize, and potentially ask for a second dish.

In our test run, it wouldn’t make sense to actually create the meals when we want to test the front-end’s (waiter’s) ability to fulfill the requests received from the backend (kitchen). More importantly, if we wanted to test our waiters could actually apologize to customers in the cases where a meal is delayed or cannot be made we would literally be waiting until our cooks were too slow or we ran out of ingredients before our tests for those cases could be confirmed. For this reason, we would “mock” the backend (kitchen) and give the waiters whatever scenario it is that we want to test.

Similarly in code, we don’t actually hit the API when we’re testing various scenarios. We mock the response we may expect to receive and verify that our application can handle that response accordingly. Just like our kitchen example, if we were testing our application’s ability to handle an API call that failed we would literally have to wait for our API to fail to verify it could handle that case—a scenario that hopefully won’t be happening that often!

Adding users

To get this test to pass, we need to implement the service method in users.service.ts.

First, we’ll start by adding our imports and a collection of employees to the service.

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

import { Observable, of } from 'rxjs'; // Add imports

@Injectable({

providedIn: 'root'

})

export class UsersService {

users: Array<object> = [ // Add employee object

{

id: '1',

name: 'Jane',

role: 'Designer',

pokemon: 'Blastoise'

},

{

id: '2',

name: 'Bob',

role: 'Developer',

pokemon: 'Charizard'

},

{

id: '3',

name: 'Jim',

role: 'Developer',

pokemon: 'Venusaur'

},

{

id: '4',

name: 'Adam',

role: 'Designer',

pokemon: 'Yoshi'

}

];

constructor() { }

}

Copy

Then, just below our constructor, we can implement all.

all(): Observable<Array<object>> {

return of(this.users);

}

Copy

Run ng test again and you should now have passing tests including the new tests for our service method.

Add users to the home page

Now that our service method is ready to use, we can work towards populating our home page with these users.

First, we’ll update index.html with [Bulma](https://bulma.io/) to help us with some styling.

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>AngularTesting</title>

<base href="/">

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

<link rel="icon" type="image/x-icon" href="favicon.ico">

<!--Add these-->

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.7.1/css/bulma.min.css">

<script defer src="https://use.fontawesome.com/releases/v5.1.0/js/all.js"></script>

</head>

<body>

<app-root></app-root>

</body>

</html>

Copy

Then within home/home.component.ts we can add a call to our new service.

import { Component, OnInit } from '@angular/core';

import { UsersService } from '../services/users/users.service';

@Component({

selector: 'app-home',

templateUrl: './home.component.html',

styleUrls: ['./home.component.css']

})

export class HomeComponent implements OnInit {

users;

constructor(private usersService: UsersService) { }

ngOnInit() {

this.usersService.all().subscribe(res => {

this.users = res;

});

}

}

Copy

First we import our service and inject it into our component’s constructor. Then we add a call to the service method within ngOnInit and set the return value to our component’s users property.

To display these users to the view, update the template in home/home.component.html.

<section class="section is-small">

<div class="container">

<div class="columns">

<div class="column" \*ngFor="let user of users">

<div class="box">

<div class="content">

<p class="has-text-centered is-size-5">{% raw %}{{user.name}}{% endraw %}</p>

<ul>

<li><strong>Role:</strong> {% raw %}{{user.role}}{% endraw %}</li>

<li><strong>Pokemon:</strong> {% raw %}{{user.pokemon}}{% endraw %}</li>

</ul>

</div>

</div>

</div>

</div>

</div>

</section>

Copy

Now when you run ng serve and view the home page, you should see the users displayed within Bulma boxes.

Finding a single user

Now that our users are being populated into our home page, we’ll add one more service method for finding a single user that will be used for the user profile pages.

First we’ll add the tests for our new service method.

describe('all', () => {

...

});

describe('findOne', () => {

it('should return a single user', () => {

const userResponse = {

id: '2',

name: 'Bob',

role: 'Developer',

pokemon: 'Charizard'

};

let response;

spyOn(usersService, 'findOne').and.returnValue(of(userResponse));

usersService.findOne('2').subscribe(res => {

response = res;

});

expect(response).toEqual(userResponse);

});

});

Copy

Here we add a test for the expectation that findOne will return a single user. We declare a userResponse variable set to a mocked response of our service method, a single object from the collection of users.

We then create a spy for usersService.findOne and return our mocked userResponse variable. With our spy set, we call our service method and set its return value to response.

Finally, we add our assertion that response will be set to the return value of the service call, userResponse.

To get this test to pass, we can add the following implementation to users.service.ts.

all(): Observable<Array<object>> {

return of(this.users);

}

findOne(id: string): Observable<object> {

const user = this.users.find((u: any) => {

return u.id === id;

});

return of(user);

}

Copy

Now when you run ng test you should see all of the tests in a passing state.

Conclusion

At this point I hope testing, both its benefits and the reason for writing them, is starting to become a bit more clear. Personally, I pushed off testing for the longest time and my reasons were primarily because I didn’t understand the why behind them and resources for testing were limited.

What we’ve created in this tutorial isn’t the most visually impressive application but it’s a step in the right direction.

In the next tutorial, we’ll create the user profile page and a service to retrieve a Pokemon image using the [Pokeapi](https://pokeapi.co/). We’ll learn how to test service methods that make HTTP requests and how to test components.

Extra

If you want the tests to display in a more readable format within your terminal, there’s an npm package for this.

First, install the package.

npm install karma-spec-reporter --save-dev

Copy

Once that’s finished, open src/karma.conf.js, add the new package to plugins, and update the progress value within reporters to spec.

module.exports = function (config) {

config.set({

basePath: '',

frameworks: ['jasmine', '@angular-devkit/build-angular'],

plugins: [

require('karma-jasmine'),

require('karma-chrome-launcher'),

require('karma-jasmine-html-reporter'),

require('karma-coverage-istanbul-reporter'),

require('@angular-devkit/build-angular/plugins/karma'),

require('karma-spec-reporter') // Add this

],

client: {

clearContext: false // leave Jasmine Spec Runner output visible in browser

},

coverageIstanbulReporter: {

dir: require('path').join(\_\_dirname, '../coverage'),

reports: ['html', 'lcovonly'],

fixWebpackSourcePaths: true

},

reporters: ['spec', 'kjhtml'], // Update progress to spec

port: 9876,

colors: true,

logLevel: config.LOG\_INFO,

autoWatch: true,

browsers: ['Chrome'],

singleRun: false

});

};

Copy

Now when you run ng test you should have a more visually appealing output for your test suite.