# THE COMPLETE BOOK ON ANGULAR TESTING

The no-nonsense guide that will teach you how to test your Angular applications. Starting today.

BY DANIEL KREIDER

# The complete book on Angular testing

The no-nonsense guide that will teach you how to test your Angular applications. Starting today.

- ## Disclaimer

You need to be warned.

Before we go any further - realize that this book was written by an opinionated developer.

Just like every other code-smearing keyboard-whacker, I’ve got hoards of opinions about software architecture, creation and testing that have been formed by my years of toil and mistake-making.

Some of these opinions might be useful to you. But others of them will do you no good.

So be wise. As you read, make sure you swallow the good pieces and spit out the rest.

- ## Introduction

So you want to learn how to test Angular applications?

I remember the first time I took a lick at writing tests for my Angular application.

That was some years ago. The official Angular docs weren't that great or helpful. And I couldn't find any simple tutorials or blog posts to help me out. I felt like a 5-year-old New Yorker lost in an overgrown jungle not sure where to turn or go.

I got lost in questions like what should I focus on testing? When should I and how do I mock dependencies? And grunted over things like asynchronous functions, etc…

Seemed like my head was constantly spinning with null injection and timeout errors.

I was so frustrated that I gave up, thinking I could get by with manually testing my code before I shipped. To think I didn't need to write tests for my code was a dumb idea that I should've never hatched.

A couple of things happened. First, more bugs began to slip by me and pop their ugly heads in production. Second, my code wasn't as clean and neat as it could've been. And third, I had this mysterious feeling way down in the depths of me somewhere that I wasn't as professional as I could be.

Now, I don't know exactly why you're reading this book but I can make a good guess. Obviously you want to learn how to test Angular applications. But I believe that deep down you're looking for something more. You want to be a better developer. You want to write better code. You want to catch the bugs before they're released into the wild. You want to be proud of what you do.

The fact that you picked up this book, and are reading it right now, is a sign that you care about your profession.

### About the author

But say, who am I anyway?

Well, my name is Daniel Kreider and I'm really bad at a LOT of things. I don't know how to change the oil in my car. I don't have a Twitter, Instagram or Facebook account. And I'm so bad at sales & marketing that I can't even market my way out of a brown paper bag.

But I do know how to create Angular applications. Angular is the thing I excel at.

I've written dozens of articles about Angular that are scattered across the web on places like my own blog, Medium, Dev, JavaScript in Plain English, ITNEXT, Simple Programmer and other random publications. Not too long ago I showed up as an invited guest on the Adventures in Angular podcast.

Oh! Should I mention that I've also contributed to the Angular project? It was a rather small change to the docs… but… when given the chance… I like to crow about my mini contribution. And if you get a chance I highly recommend you contribute to the Angular project.

## Do I have to? Test my code??

Testing your code is not easy to learn or practice. Especially for the young kids on the block that have never written more than a few “Hello World” programs and for loops.

That’s why new developers, who weren’t taught the importance of testing their code, develop the bad habit of never testing it. Except manually of course. And the problem is, that many professional colleges never teach budding programmers to test their code. And the few exceptions make the mistake of never teaching them how to properly test their code.

Add to the problem that the demand for  “professional” software engineers is currently doubling almost every 5 years. Which means that half of the active army of programmers that are hammering their keyboards right now, as you read this book, have 5 years of experience or less.

Our industry is in a constant upheaval. We are part of an industry that is constantly putting up with growing pains.

Combine that with the fact that most of us weren’t taught to test our code. And the few that did get some introductory training weren’t taught how to properly test their code. Is it any wonder that the idea of testing code is scoffed at times?

The fact that you picked up this book, and are reading it right now, is a sign that you care about your profession. And you want to become better at what you do. You want to build applications that you are proud of. And part of that process is making sure the code you write is properly tested.

I recently went out to buy a new freezer for my wife. After a bit of looking we found the one she wanted. Can you guess what I asked the salesman?

The question I asked him was “*Does it work?*”

His answer? “*It’s been **tested** and verified to be working.*”

Call me crazy for asking such an obvious question - I won’t care.

But, if your manager walked into the room as you’re coding, and asked you if the application you’re coding works, what would you say to him?

And if he asked for proof that it works, what would you show him?

The application? Would you click a few buttons here and there to prove that it’s working? Would you log in into it and then back out?

The auto industry tests all the car parts that they make before selling them to the customer. Doctors require all sorts of exams before a patient is permitted to do surgery. Accountants fiddle away for hours to prepare financial records - they're not allowed to miscount, not even by one cent. Why? Because one mistake could put them behind bars.

I'm trying to make a point. The point is that testing is common in other industries. Shouldn't we learn something from that?

## How to become a better developer. Faster.

When you write tests for your production code you'll discover that without fail, sooner or later, you'll come across some code that's hard to test.

Why is it that way?

The reason your code is hard to test is because it is hard to understand and poorly designed. It was slapped together and poorly engineered. Maybe you thought it was a good piece of code, but when you're testing it you find it's not so good after all.

The first thought that might pop into your mind is that you can just skip tests this time. You know it works so why should you test it anyway?

But don't cave in to that kind of low-man thinking. If you've got code that's hard to test that means your code is the fault and not your tests. This is where testing your code really shines.

One of the advantages of testing your code is that it **forces** you to write better code. Giving you the side-benefit of being forced to evolve into a better coder.

It's almost a mystery, yet anyone that's tested their code consistently won't disagree with me.

Writing tests demands that you think differently about the structure and flow of your code. And demand you to become a better developer.

If you don't want to become a better Angular developer then stop reading right here or return this for a refund. Your career and expertise will rot and you'll save yourself some hard work.

But if you want to become a better professional then keep on reading. If you take action on what this book teaches you'll be forced to become a better developer.

## Setting a few expectations

Alright bug hunter! I've promised that you'll be a better developer. And if you're like me then you're hankering to get started.

But before we go any further let's be clear about a few things.

First, I’m going to do my best to make you a bug-squashing expert. I don’t want you up till 2 AM on Saturday morning debugging a production bug. Who in their sound mind would want a life like that anyway?

We want to catch those bugs before they ever go to production by learning how to properly test your Angular applications.

Now, I’m gonna sweat here at my desk to deliver on that promise but you are the one that’s going to have to put in the real effort. I can’t write your tests for you so YOU have the real sweating to do.

You’ll also have the face-smacking realization sometime soon (if you haven’t already) that learning how to write tests won’t solve all your coding problems. As cool as testing is, I refuse to promise that you will effectively catch every bug before it hits production. But I will promise that you can tremendously reduce the amount of production code bugs by applying what I’m about to teach you.

I hate to break it to ya, but that’s the reality of our job. The best solution to never wrestling with another nasty bug is to kiss your profession as a software engineer goodbye.

## ● General Testing Theory (In case you're new to testing)

Listen up friend.

Pay attention please. I want you to get this.

I'm about to tell you something important and I want to make sure you get this. It's something that most developers don't realize - for some reason it's never dawned on them even though it's actually pretty simple.

Here's what I want you to understand and realize.

**Learning how to <u>properly</u> test software is <u>just as hard</u> as learning how to code.**

Yup. You read that right.

We spend a lot of time and energy learning how to write code, bug-eyed at 3 in the morning honing our skills. We find it mesmerizing to put our name in a while loop and watch the console stumble over itself trying to handle our madish demands. And dominating all those algorithms and data structures to do our bidding makes us feel like whizzes.

Until it breaks.

That's where testing comes in. Those bugs jab our pride and we want a way to make sure those lines of code we were proud of never blast us again.

But there's a catch to testing software. It's almost… well… testing software is an art in its own right. Learning how to write good tests is just as hard as learning how to write good code.

It's a common story. We get all into testing our code and pumped up about building a huge suite of tests to make sure our production system is always rock-solid. Slinging a bunch of tests is a job any junior could do, so we eagerly set out and build a boat-load of tests. But soon we burn out and decide that it's not worth the effort. It seems that any small change we make immediately breaks 82 of our 108 tests - not to mention the simple bugs that are still not being caught. The whole testing zeal goes catawampus and we burn out and scream to our other friends that testing isn't worth a hoot.

But all that assumption is wrong. The reason things buckled and went wonky was because we don't know how to test the code we write. We need to invest time learning the art of testing software.

The first step in learning how to test software is to master some general testing lingo and understand the different types of tests that we can write.

### Testing Lingo

When it comes to testing software there are some general terms that get flung around. It doesn't matter what kind of application you're testing. Or what kind of code was written. Or the test suite used. The testing lingo and principles span the programming language, frameworks and applications.

There are 3 types of tests that you can write.

- Unit Test

A unit test takes a unit of code and tests that specific chunk of code. A common example of a unit is a function that takes an input and returns a value.

However, functions aren't the only units that can be tested. You can also test classes, objects, etc… as specific units.

A unit test does not test dependencies on other objects, classes or parts of the application. It takes a specific unit of code and verifies its behavior, regardless of its dependencies.

This is where mocking comes in. Almost every piece of a normal application has some sort of dependency. For example, an Angular service might depend on the HttpService for API calls. Or a component might depend on Angular's Router service to reroute a user.

Regardless of the dependency, when unit testing you mock the dependency which means you simulate it's expected behavior. It's like decoupling the application and testing each piece separately to make sure they've behaving.

But what if you want to test a dependency?

- Integration Testing

Integration tests will test multiple pieces of an application and verify how they are behaving together.

Instead of mocking dependencies, like we do when unit testing, we are now testing the interfaces and couplings of the application.

Say that in our Angular app we have a login component that depends on an authorization service. Instead of mocking the authorization dependency when testing, we actually create a real instance of the authorization service and give it to the login component. This allows us to test the integration and functionality of the login component and authentication service together.

But that's not all.

There's another type of common test called...

- System Testing

What is system testing?

And how is it different from unit or integration testing?

System testing is more popularly known in the Angular world as e2e testing. It’s the testing of the interaction and behavior of the application by simulating a user. It’s testing the complete integration of the application.

Protractor was the default system testing tool for Angular applications until version 12 showed up. Now, Angular supports multiple system testing tools with Cypress being the most popular.

So how does a system test work?

When you write system tests with Protractor or Cypress, it will launch a browser and simulate the user. A common example of a system test is to log in and then log out.

## Conclusion

When learning how to write great tests the first step is to understand the types of tests that you can write.

It’s common for newbies to get tangled in the different testing options and give up because they didn’t learn the general testing theory.

- # Angular Testing Theory

Testing. Angular. Applications.

How do ya do it?

Up till now we've looked at general testing principles and ideas that can be applied to any piece of software, code, framework etc…

But now it's time to take an aggressive bull run and limelight the Angular testing principles.

Where do you start when you have to write tests for your Angular application?

What are the first steps?

What should you test?

Not sure how to mock a dependency? Or if you should just inject it directly?

And what Angular testing tools should you use?

What's Karma anyway?

And how do I use Jasmine?

Or is Mocha better?

Should you test everything? Are unit or integration tests better? What about smock or E2E tests?

Getting started with Angular testing can tie a fellow's mind into pretzel knots - it's hard to know where to focus your limited time to get the best ROI.

And that, my friend, is why I've written this chapter. I'm going to explain the overall theory and principles that you need to unleash your Angular testing skills and become a bang-whiz at doing this.

This will be a quick but thorough introduction to the basic theory and principles you will need to get started with Angular testing. I'll also answer some common questions that developers ask when they first start testing an Angular application.

## Testing lingo you should know before we get started

- **Jasmine**

Jasmine is a testing framework that is included in every Angular project created with the CLI, unless you specifically opt out.

It's got all kinds of features like asynchronous testing, the ability to use 'spies' for implementing test doubles and more.

Remember, this is the default testing framework that Angular uses. It comes with all the needed batteries to write tests.

Assuming we have a simple helloWorld function we could test it with Jasmine like this.

```
describe('Hello world', function() {
  it('says hello', function() {
    expect(helloWorld()).toEqual('Hello world!');
  });
});
```

- **Karma**

Karma is the default test runner. It is not a testing framework.

It takes the Jasmine test files in your Angular project and runs them in the browser, checking and displaying the results of your various tests.

- **Unit**

A unit is the smallest piece of an application that can still be tested.

In most cases, this would be the functions inside your components, services, pipes, etc…

- **Unit Tests**

A unit test is a test for the smallest, testable pieces of an Angular application.

- **Integration Tests**

An integration test is where we combine an object and it's dependencies and test how they cooperate together, instead of mocking the dependencies like we would do in a unit test.

- **E2E Testing**

End-to-end (E2E) testing is when we use Protractor or Cypress to run step tests via browser automation. This is to simulate a real-world user using and clicking around in our application.

When you run an e2e test a browser window will be launched (default’s to Chrome) and you’ll watch as the testing framework simulates clicks, etc… and then print the results of the test.

## Angular testing theory that will let you get started with a BANG!

Consider this.

When we create a unit of code, we usually create it as a function inside our component, service, or another common piece of our Angular application.

But once created, do we leave it untested? Of course not. We test it manually.

We open our browser, let it refresh and manually test to make sure things are working like expected.

But what if we could automate the testing process? While still being confident that our code is bug-free?

That's where unit tests help us and give us **3 significant advantages over manually testing**.

First, after coding a unit, we can create unit tests that make sure our new unit of code is behaving properly without having to manually test.

The second advantage we get from testing is a bug-net for the future in case we ever come back to modify this unit. This will give us the confidence needed when refactoring our code and protect our application from code rot.

Psst.

Listen up please.

What I’m about to tell you in the next paragraph is a secret that many developers never learn for years. When this realization finally dawned on me, it was a tremendous pivotal point that fundamentally changed how I viewed and combatted bugs.

Did you know that most production bugs hide in old features that we coded awhile ago and don't test anymore?

As we add new features we change or modify our Angular code a bit and test the new feature manually but never check an old feature to make sure that it wasn't broken with the changes to a dependency that your old code depends on. If we had written good tests, they would have caught these bugs before our code hit production.

The third advantage we get when writing unit tests is that it forces us to become a better developer and write better code.

And please, remember to be thorough in your testing. Good unit tests will check both the unsuccessful and successful scenarios.

If unit testing is hard then it’s an indication that something is wrong with your code, not the test. For example, if you have too many tests for a single object it indicates that your object is too large and helps you properly split your code into smaller modules.

Which leads us to...

**Isolation**

What is isolation?

Isolation is the act of replacing the dependencies of an object with fake objects like mocks, stubs and spies.

Good unit tests are self-reliant, independent and autonomous. They should not expect a specific order to run in. Or depend on a value from another test.

If your unit tests are tightly coupled then you're trying to build solid tests on sand.

Another advantage of isolation is that it keeps the tests much faster.

So, when you're writing unit tests for your Angular application make sure you mock first level dependencies. Second level dependencies should never be known about.

Clean unit tests are small.

Clean unit tests are understandable.

Clean unit tests know as little as possible about the object's dependencies.

Clean unit tests are reusable.

## So, what should you test?

It's easy to get lost trying to decide which pieces of the Angular application to test.

Err...

What parts must I test and what can I skip?

Do we need 100% code coverage?

And should I be writing tests for that front-end UI stuff that's always evolving? UI tests tend to be brittle so where should an Angular developer set their bug traps?

Most importantly, how do I make sure that I get the best ROI for my time?

That, my friend, is a great question. Let’s peel the covers back a bit and look at some of the main pieces of an Angular application and the priorities they deserve.

### Angular Components

Components are... well... components.

They are the most basic UI building block in an Angular app and many argue that their only function should be to display data.

A component should have no concern as to business logic or where the data came from. It should only focus on taking the data it's meant to receive and presenting it to the user in a meaningful way.

In general, it's a waste of time to test these heavily. Notice I said *heavily*. Especially as they're prone to rapid evolving as your Angular app grows and expands.

To focus lots of developer energy on making sure your components are completely tested is usually a poor investment choice.

At the same time, some general unit tests for your Angular components is a great idea.

### Angular Services

Here's where you'll take an aim and make a great shot!

Assuming that your Angular app has followed general best practices and uses services to handle business logic, data access, HTTP calls, and all the other spicy background stuff then this is where anyone that’s writing tests ought-a-be licking their lips.

**Make sure your Angular services are tested well.** By focusing on this you'll have a greater guarantee of better ROI.

### Pipes

Definitely put those custom pipes to the fire. This is a great place to focus on testing.

Once again, these play an important role in business logic. Making sure they're working properly is a great way to get a better ROI when writing tests.

### Summary

So what should you test?

**Write bug traps for every piece of the application that contains business logic.** This would be pieces like...

- Services
- Custom pipes
- Helper functions

By focusing on the pieces with the business logic and making sure it's tested well you'll be cutting to the heart of the matter and writing Angular tests for the pieces that matter.

And one final tip...

The best ROI you can get with minimum effort will be writing unit tests for your services and other shared pieces of the project.

## Conclusion

Learning how to write good tests is like learning how to write good code.

It just takes practice and time.

It can take years to become a proficient coder and so don't expect to turn into a superhero tester overnight.

But once you’ve got a handle on some of the basic testing theory and principles that I've given you in this chapter, you can expect great results.

- ## Testing Angular Components

Before we throw some Angular tests together and think we're Angular testing experts we should cover some basic concepts that will help us write Angular component tests properly and effectively.

One of the important concepts we need to learn before testing an Angular component is the difference between a DOM test and a unit test.

All Angular components are made up of two pieces - the Typescript code and the HTML code. Usually these are split into two separate files although sometimes they’re hacked together in the same file (oh horrors). But whatever the case, there are two distinct things that we can test.

First, we can test to make sure the component is rendering and displaying information properly. This is considered a DOM test where we actually test the DOM elements of our component.

Second, we can test the component's Typescript file to make sure the "logic" of the component is behaving as we expected. Or, we could create tests that do both DOM and unit testing in one swoop.

We’ll get our fingers wet with basic unit tests and then look at DOM tests.

  - ### How to write simple unit tests

We’ll start by writing your first Angular component test. Then we'll branch out into common component testing scenarios like testing component constructors as well as components with dependencies. Then we'll wrap things up with some pointers on the best practices when testing Angular components.

You've probably noticed the generic test that the Angular CLI generates when we create a new component?

It looks something like this.

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


import { MyComponent } from './my.component';


describe('MyComponent', () => {
  let component: MyComponent;
  let fixture: ComponentFixture<MyComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ MyComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(MyComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

So what's going on in this basic Angular component test file?

Notice the two declared variables component and fixture? Those will be used to reference our component while we test it.

You'll also notice we have two beforeEach functions. One runs an asynchronous method for the asynchronous tests and the other one is not asynchronous.

In the beforeEach functions we're using Angular's TestBed function to set up a testing bed for our tests. TestBed is used to configure and initialize the needed environment for unit testing our components.

And last of all, we have a generic 'should create' test that makes sure our component is truthy. In other words, this simple test just checks to make sure our component is initialized and ready to use.

Let's say we have a basic counter component like this. The idea is that when our add function gets called we'll update the total and then display that back to the user.

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


@Component({
  selector: 'app-counter',
  templateUrl: './counter.component.html',
  styleUrls: ['./counter.component.css']
})
export class CounterComponent implements OnInit {


  total: number = 0;


  constructor() { }


  ngOnInit(): void {
  }


  add(): void {
    this.total = this.total + 1;
  }


}
```

How do we test this component?

It's as simple as this.

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

import { CounterComponent } from './counter.component';

describe('CounterComponent', () => {
  let component: CounterComponent;
  let fixture: ComponentFixture<CounterComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ CounterComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(CounterComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it("total should be one", () => {
    component.total = 0;
    component.add();
    expect(component.total).toEqual(1);
  });
});
```

Now, all you need to do is run the *ng test* in the root directory of your Angular project and watch it run your tests.

Simple?

I know.

So how do we take it a bit further?

How do we test an Angular component that has a service dependency?

Say that we move our counting stuff into a separate Angular service.

There are different ways to do this. We could inject the new CounterService directly into our test component but that's bad practice.

Instead of depending on the service, we should create a mock (also known as a stub). That way we can fake the dependency.

Here's how we do it.

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { CounterService } from '../counter.service';

import { CounterComponent } from './counter.component';

describe('CounterComponent', () => {
  let component: CounterComponent;
  let fixture: ComponentFixture<CounterComponent>;
  let counterServiceStub: Partial<CounterService> = {
    add: () => null,
    get: () => 1,
  }

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ CounterComponent ],
      providers: [
        { provide: CounterService, useValue: counterServiceStub }
      ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(CounterComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it("total should be one", () => {
    component.total = 0;
    component.add();
    expect(component.total).toEqual(1);
  });
});
```

Did you notice how we declared a counterServiceStub and defined how this mock service will behave?

And then we inject that into our TestBed configuration and use our mock service instead of the real thing.

But what about HTTP services?

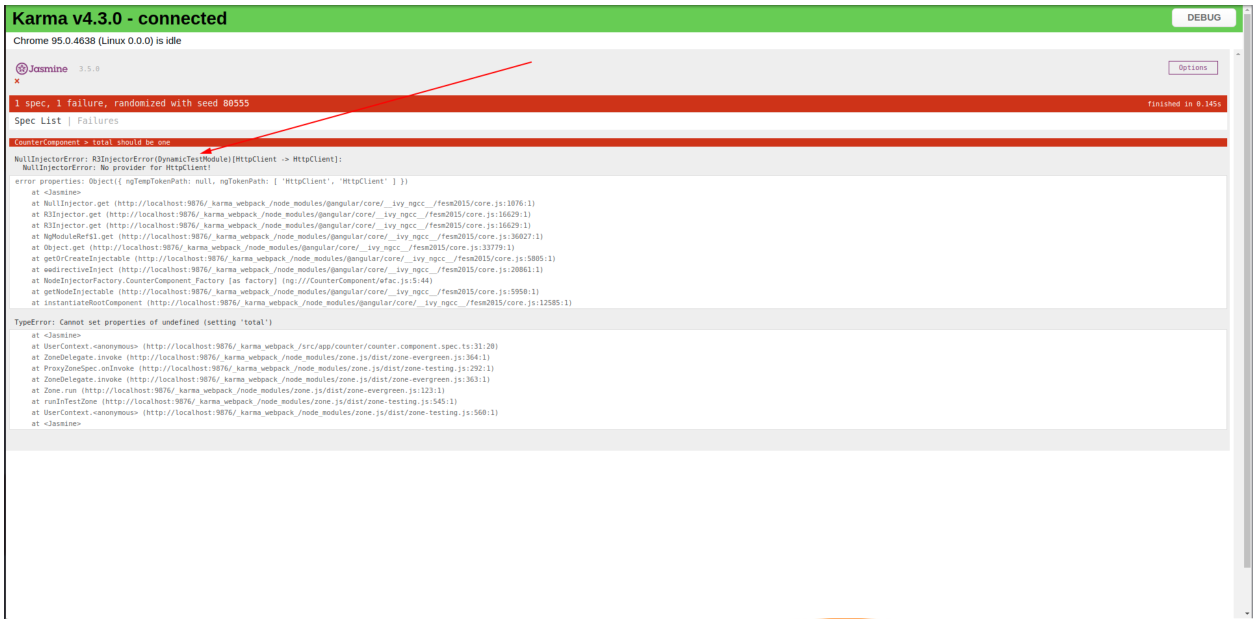
Those are a bit tricker.

So how do you test an Angular component that depends on [Angular's HTTP client]?

Let's say we add the HttpClient to our component to do some HTTP stuff. Like poking a counting service to see what chance we have of hitting the jackpot. 😋

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http'
import { Observable } from 'rxjs';

@Component({
  selector: 'app-counter',
  templateUrl: './counter.component.html',
  styleUrls: ['./counter.component.css']
})
export class CounterComponent {

  constructor(private httpClient: HttpClient) { }

  add(): Observable<number> {
    return this.httpClient.post<number>("http://counting.service/api", { total: this.total });
  }

}
```

You'll notice our tests start giving a weird NullInjectionError - No provider for HttpClient!.



```
Karma v4.3.0 - connected
Chrome 95.0.4638 (Linux 0.0.0) is idle

Jasmine 3.5.0
×
1 spec, 1 failure, randomized with seed 80555                finished in 0.145s
Spec List | Failures

CounterComponent > total should be one

NullInjectorError: R3InjectorError(DynamicTestModule)[HttpClient -> HttpClient]:
  NullInjectorError: No provider for HttpClient!
error properties: Object({ ngTempTokenPath: null, ngTokenPath: [ 'HttpClient', 'HttpClient' ] })
    at <Jasmine>
    at NullInjector.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:1076:1)
    at R3Injector.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:16629:1)
    at R3Injector.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:16629:1)
    at NgModuleRef$1.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:36027:1)
    at Object.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:33779:1)
    at getOrCreateInjectable (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:5805:1)
    at ɵɵdirectiveInject (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:20861:1)
    at NodeInjectorFactory.CounterComponent_Factory [as factory] (ng:///CounterComponent/ɵfac.js:5:44)
    at getNodeInjectable (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:5950:1)
    at instantiateRootComponent (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:12585:1)

TypeError: Cannot set properties of undefined (setting 'total')
    at <Jasmine>
    at UserContext.<anonymous> (http://localhost:9876/_karma_webpack_/src/app/counter/counter.component.spec.ts:31:20)
    at ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-evergreen.js:364:1)
    at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-testing.js:292:1)
    at ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-evergreen.js:363:1)
    at Zone.run (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-evergreen.js:123:1)
    at runInTestZone (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-testing.js:545:1)
    at UserContext.<anonymous> (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-testing.js:560:1)
    at <Jasmine>
```

How do we fix it?

```
import { HttpClient } from '@angular/common/http';
import { async, ComponentFixture, fakeAsync, TestBed, tick } from '@angular/core/testing';
import { of } from 'rxjs';
import { CounterService } from '../counter.service';

import { CounterComponent } from './counter.component';

describe('CounterComponent', () => {
  let component: CounterComponent;
  let fixture: ComponentFixture<CounterComponent>;
  let httpClientSpy: { post: jasmine.Spy }

  beforeEach(async(() => {
    httpClientSpy = jasmine.createSpyObj('HttpClient', ['post']);

    TestBed.configureTestingModule({
      declarations: [ CounterComponent ],
      providers: [
        { provide: HttpClient, useValue: httpClientSpy }
      ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(CounterComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it("add should call remote api", fakeAsync((done: DoneFn) => {
    httpClientSpy.post.and.returnValue(of(1));

    component.add().subscribe(result => {
      expect(result).toEqual(1);
      expect(httpClientSpy.post).toHaveBeenCalled();
      done;
    });
  }));
});
```

Notice what we've just done?

We've created a spy, or in other words, we mocked the HttpClient. More specifically, we simulated the post function.

Then, we created a test to call the add function and made sure that it called our httpClientSpy.

### Angular component testing best practices

When testing Angular components make sure you always mock service dependencies.

Always?

Well... if you're brave enough to evade the tornado storms that integration tests stir up then go ahead and inject the real services into your TestBed.

Otherwise, keep your tests simple and stable by always mocking service dependencies.

You can do this using the built-in Jasmine Spy's that come with the Angular framework. Or you can also check out the jasmin-auto-spies library that makes mocking effortless.

Also, keep in mind that components are a combination of an HTML template and a Typescript class, with the two of these working together to display information to the user.

Some argue that DOM testing is too hard or brittle while others say that it's absolutely necessary. So, make a wise choice and decide what's best for your situation. If unsure, I would recommend leaning toward more unit tests and fewer DOM tests.

## ○ Testing the DOM

How do you write DOM tests for your Angular components?

Maybe you're already writing some basic unit tests for your components but are beginning to realize that testing a component often involves more than just testing a class.

Why?

Because an Angular component interacts with the DOM as well as other components and visual pieces of your Angular applications.

A class test will only tell you how that specific class behaves. But it doesn't actually test if it is properly **displaying**.

That's where Angular component DOM tests come in.

They'll help you make sure that the component is rendering properly and that events like clicks are firing properly, how it responds to user input or how it integrates with child and parent components.

Since Angular is a dynamic front-end framework, it's common for a component to have complicated DOM interactions that dynamically render the page.

That's why we need DOM tests to verify that our component is behaving like it was designed to behave.

So, what do you need to know to write your first DOM test?

And how do you get started when you're lost?

Every Angular test starts with a *describe* function that's used to describe the piece of code that we're testing.

Like this…

```
describe('MyComponent', () => {
    // Set up and test here
});
```

The next step is to create the object we'll be testing, mock its dependencies and so forth.

It will look something like this.

```
describe('MyComponent', () => {
    let component: MyComponent;
    let fixture: ComponentFixture<MyComponent>;

    beforeEach(() => {
        fixture = TestBed.createComponent(LoginComponent);
        component = fixture.componentInstance;
        fixture.detectChanges();
    });
});
```

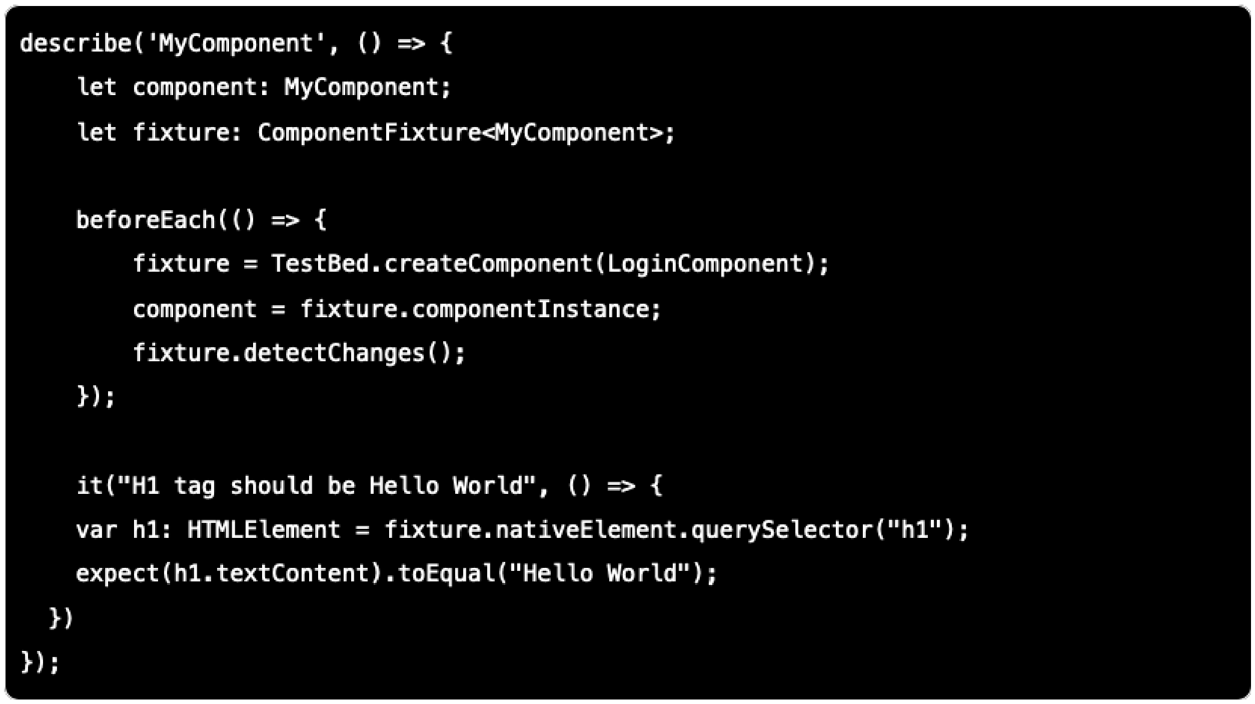
So what have we just done?

First, we declared a variable that will be the component we're testing. And we also declared a second variable that will be our ComponentFixture used to debug and test our component.

Then, we used the *beforeEach* function to do our set-up and initialization. This is a function that will be called every time a test is run. Its purpose is to avoid repetitive set up code.

Now, we've got everything ready to finally write our test!

We'll write an *it* test to describe what we expect our object (in this case an Angular component) to do.



```
describe('MyComponent', () => {
    let component: MyComponent;
    let fixture: ComponentFixture<MyComponent>;

    beforeEach(() => {
        fixture = TestBed.createComponent(LoginComponent);
        component = fixture.componentInstance;
        fixture.detectChanges();
    });

    it("H1 tag should be Hello World", () => {
    var h1: HTMLElement = fixture.nativeElement.querySelector("h1");
    expect(h1.textContent).toEqual("Hello World");
  })
});
```

Tada!

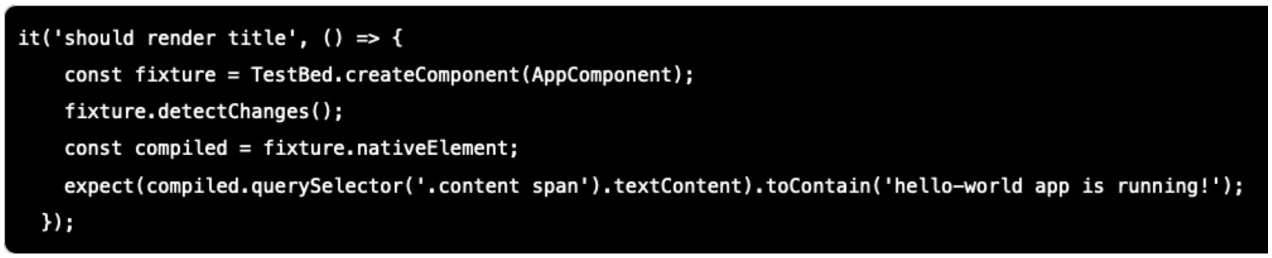
We just wrote our first Angular DOM test by grabbing the H1 element inside our component and verifying that the text we expect is the text that is actually being displayed.

Simple?

Yeah. Angular DOM testing doesn't have to be hard.

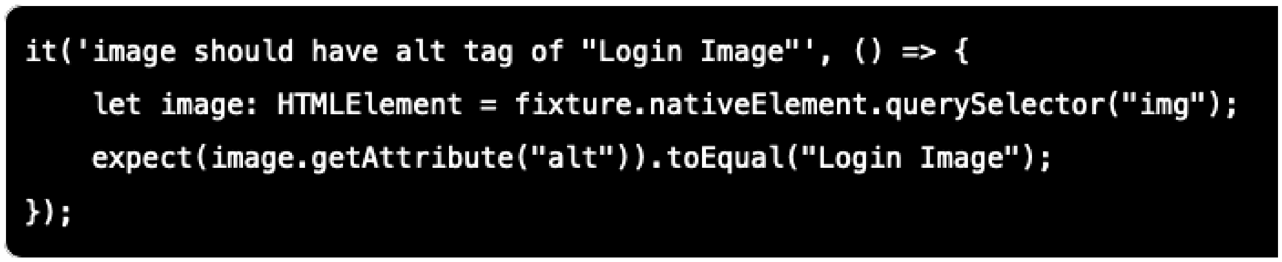
Want to see some more advanced examples?

Here’s how to check for a specific text.



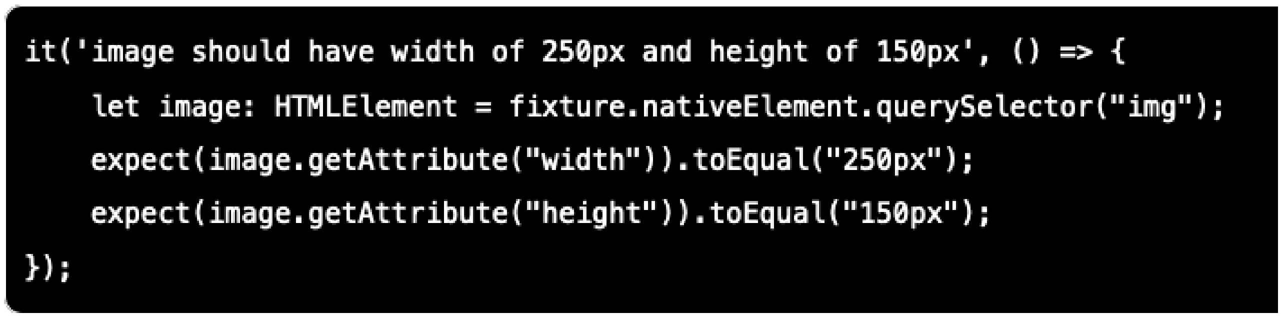
```
it('should render title', () => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.nativeElement;
    expect(compiled.querySelector('.content span').textContent).toContain('hello-world app is running!');
  });
```

How to check for an attribute.



```
it('image should have alt tag of "Login Image"', () => {
    let image: HTMLElement = fixture.nativeElement.querySelector("img");
    expect(image.getAttribute("alt")).toEqual("Login Image");
});
```

How to check image width & height



```
it('image should have width of 250px and height of 150px', () => {
    let image: HTMLElement = fixture.nativeElement.querySelector("img");
    expect(image.getAttribute("width")).toEqual("250px");
    expect(image.getAttribute("height")).toEqual("150px");
});
```

Keep in mind that DOM tests can become brittle if you're not careful. So don't just blindly wield your new testing sword but use it wisely by properly testing the things that matter.

- ## Testing Angular Pipes

Wish you knew how to test your Angular pipes?

Here's how to get started in less than 5 minutes. And even if you have no idea how.

How do you write tests for an Angular pipe? And make sure that it always behaves as intended?

Well...

The good news is that the Angular pipe is probably the easiest piece of our application that we could test.

Why?

First, because we don't have to mock any dependencies.

And second, the logic of a good Angular pipe is very straight-forward. Any Angular pipe worth its salt is simple and logical.

The bad news is... well... there is no bad news.

So, how do we get started?

We'll begin by testing the UpperCase pipe that is already part of the Angular framework.

I know. It doesn't need to be tested. 😊

But for the first example, I want to take a pipe that you're probably familiar with, that works well, and teach you how to test it.

### Testing the UpperCase pipe

So, buster, here's the code to test the upper case pipe.

```
import { UpperCasePipe } from '@angular/common/';

describe('UpperCasePipe', () => {
  it('create an instance', () => {
    const pipe = new UpperCasePipe();
    expect(pipe).toBeTruthy();
  });
  it('pipe should be PIPE', () => {
    const pipe = new UpperCasePipe();
    let value = "pipe";
    let expected = "PIPE";

    expect(pipe.transform(value)).toEqual(expected);
  })
});
```

So how does this work?

We grab a new instance of the UpperCase pipe, call its transform function and check the results.

Simple?

Yup. 😎

How about we try another pipe? The CurrencyPipe.

## Testing the Currency pipe

Angular has a CurrencyPipe that takes an amount and formats it depending on the currency type.

This example is different in that we have to initialize the pipe with a locale so that it knows how to properly transform the input.

```
import { CurrencyPipe } from '@angular/common/';

describe('CurrencyPipe', () => {
  it('create an instance', () => {
    const pipe = new CurrencyPipe("en-US");
    expect(pipe).toBeTruthy();
  });
  it('10.26 should be $10.26', () => {
    const pipe = new CurrencyPipe("en-US");
    let value = "10.26";
    let expected = "$10.26";

    expect(pipe.transform(value)).toEqual(expected);
  })
  it('1 should be $1.00', () => {
    const pipe = new CurrencyPipe("en-US");
    let value = "1";
    let expected = "$1.00";

    expect(pipe.transform(value)).toEqual(expected);
  })
});
```

But what about our own custom pipes?

How do we test those?

## Building our own pipe

For this demo, we'll create a pimple-popping-simple greeting pipe.

It'll have the simple responsibility of taking a string and prepending *Hello* to it.

We'll begin by using the Angular CLI to generate a new pipe.

```
ng generate pipe hello
```

Then we'll edit it to look like this.

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

@Pipe({
  name: 'hello'
})
export class HelloPipe implements PipeTransform {

  transform(value: unknown, ...args: unknown[]): unknown {
    return `Hello ${value}`;
  }

}
```

So how do we test it?

Just like we tested the others.

## Unit testing our new Angular pipe

```
import { HelloPipe } from './hello.pipe';

describe('HelloPipe', () => {
  it('create an instance', () => {
    const pipe = new HelloPipe();
    expect(pipe).toBeTruthy();
  });

  it ("expect John to equal Hello John", () => {
    const pipe = new HelloPipe();
    expect(pipe.transform("John")).toEqual("Hello John");
  })
});
```

Just like before, we begin by creating a new instance of our Angular pipe.

Then, we used it to transform an input and check the output.

Simple?

Sure. But this isn't all I wanted to show you.

## Integration testing our Angular pipe

You're probably smart enough to realize this but in case you haven't caught on yet...

What we've just done is written **a unit test for our Angular pipe.**

But what if we want to write an integration test?

And test it inside of a component to make sure that it's rendering properly?

How do we write an integration test for our Angular pipe?

Inside of the component that uses our Angular pipe, we'll declare a test like this.

```
import { TestBed, async } from '@angular/core/testing';
import { AppComponent } from './app.component';
import { HelloPipe } from './hello.pipe';

describe('AppComponent', () => {
  it('should say Hello Alice', () => {
    const fixture = TestBed.createComponent(AppComponent);
    const hostElement: HTMLElement = fixture.nativeElement;

    var paragraph = hostElement.querySelector("p");
    fixture.detectChanges();

    expect(paragraph.innerText).toBe("Hello Alice");
  });
});
```

So what did we just do?

We created our Angular component and grabbed a reference to the <p> element inside the component that uses our pipe.

Then, we ran the detection changer.

And finally, we made sure that the displayed value is the value that we expected the pipe to return.

We've just learned how to test Angular pipes. And make sure they never behave like monkeys let loose in a jungle.

Testing Angular pipes is one of the easiest pieces of an Angular application to test. Plus, it has one of the best ROI's that I know of.

- ## Testing Angular Services

    - ### General Service Testing

How do you test Angular services?

And suck up those bugs like a vacuum cleaner on steroids. 🤓

How do you do it?

How do you check to make sure that your Angular services are working as intended?

And that they're bug free?

How do you test them to discover any hidden bugs? And fix them before your Angular application is deployed?

In this complete introduction to testing Angular services I'll show you how to do just that. We'll start by testing a very simple service and then advance into testing services that depend on other services, like Angular's HttpClient for example. As well as testing services that return Observables and Promises.

But why do we test Angular services?

Or maybe a better question would be, should we even write any kind of tests for our Angular application?

Testing your code is not easy to learn or practice. Especially for the young kids on the block that have never written more than a few “Hello World” programs and for loops.

That’s why new developers, who weren’t taught the importance of testing their code, develop the bad habit of never testing it. Except manually of course. And the problem is, that many professional colleges never teach budding programmers to test their code. And the few exceptions make the mistake of never teaching them how to properly test their code.

Add to the problem that the demand for “professional” software engineers is currently doubling almost every 5 years. Which means that half of the active army of programmers that are hammering their keyboards right now, as you read this book, have 5 years of experience or less.

Our industry is in a constant upheaval.

It's groaning with growing pains.

The fact that you're reading a book about testing your code (specifically Angular services) is a sign that you care about your profession. You obviously want to become better at what you do. You want to build applications that you are proud of. And part of that process is making sure the code you write is properly tested.

But what does this have to do with testing Angular services?

In most Angular applications the Angular services are the back-bone of the application (depending on how you look at it).

Services are often used to fetch data, manage the application state, handle authentication and authorization and a host of other things. The services should be where your business logic lives and that's **why we test Angular services**.

DOM tests tend to be brittle, and they're useful for some cases, but the business logic is where you want to really aim your guns and if an Angular application is properly designed you'll find the business logic inside of the services.

Let's say we have a basic storage service that is used to store key/value pairs for our application.

It looks like this.

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

@Injectable({
  providedIn: 'root'
})
export class StorageService {

  constructor() { }

  setValue(key: string, value: string) {
    localStorage.setItem(key, value);
  }

  getValue(key:string): string {
    return localStorage.getItem(key);
  }

  clearStorage() {
    localStorage.clear();
  }
}
```

How do we test this?

Since I used the Angular CLI to generate the service I can open the storage.service.spec.ts file and discover that it already created a basic test for me.

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


import { StorageService } from './storage.service';


describe('StorageService', () => {
  let service: StorageService;


  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(StorageService);
  });


  it('should be created', () => {
    expect(service).toBeTruthy();
  });
});
```

One of the bigger challenges when unit testing Angular applications is preparing the object for testing. Once it's prepared then the actual testing is usually very easy.

Fortunately for us Angular services are easy to prepare for testing. In the test above you'll notice that we've created a testing module with the help of TestBed. Then we injected our StorageService into that testing module. And now it's ready for testing.

Here's our tests.

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


import { StorageService } from './storage.service';


describe('StorageService', () => {
  let service: StorageService;


  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(StorageService);
  });


  it('should be created', () => {
    expect(service).toBeTruthy();
  });


  it('should store a value', () => {
    let key = "name";
    let value = "Daniel";


    service.setValue(key, value);


    let result = localStorage.getItem(key);


    expect(result).toEqual(value);
  });
  it('should get value', () => {
    let key = "email";
    let value = "email@email.com";

    localStorage.setItem(key, value);

    let result = service.getValue(key);

    expect(result).toEqual(value);
  });

  it('should clear everything', () => {
    service.clearStorage();
    var result = localStorage.length;
    expect(result).toEqual(0);
  });
});
```

Now if we run *ng test* we'll see a Chrome browser launch, run our tests, and display a green success message.

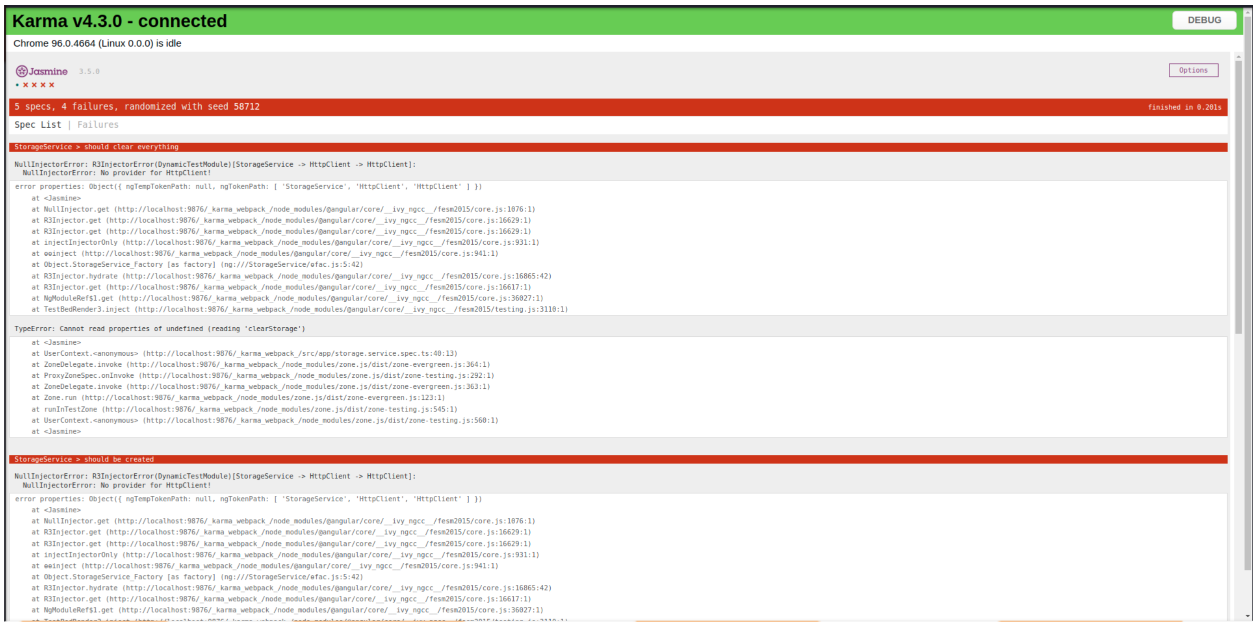
But what about services with dependencies?

Imagine we decide to use a remote storage server instead of local storage.

The first step is to inject the HTTP client like this.

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class StorageService {

  constructor(private httpClient: HttpClient) { }

  setValue(key: string, value: string) {
    localStorage.setItem(key, value);
  }

  getValue(key:string): string {
    return localStorage.getItem(key);
  }

  clearStorage() {
    localStorage.clear();
  }
}
```

But we immediately get an injection error. 😱



```
Chrome 96.0.4664 (Linux 0.0.0) is idle

5 specs, 4 failures, randomized with seed 58712                                finished in 0.201s

StorageService > should clear everything

NullInjectorError: R3InjectorError(DynamicTestModule)[StorageService -> HttpClient -> HttpClient]:
  NullInjectorError: No provider for HttpClient!
error properties: Object({ ngTempTokenPath: null, ngTokenPath: [ 'StorageService', 'HttpClient', 'HttpClient' ] })
    at <Jasmine>
    at NullInjector.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:1076:1)
    at R3Injector.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:16629:1)
    at R3Injector.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:16629:1)
    at injectInjectorOnly (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:931:1)
    at ɵɵinject (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:941:1)
    at Object.StorageService_Factory [as factory] (ng:///StorageService/ɵfac.js:5:42)
    at R3Injector.hydrate (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:16865:42)
    at R3Injector.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:16617:1)
    at NgModuleRef$1.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:36027:1)
    at TestBedRender3.inject (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/testing.js:3110:1)

TypeError: Cannot read properties of undefined (reading 'clearStorage')
    at <Jasmine>
    at UserContext.<anonymous> (http://localhost:9876/_karma_webpack_/src/app/storage.service.spec.ts:40:13)
    at ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-evergreen.js:364:1)
    at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-testing.js:292:1)
    at ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-evergreen.js:363:1)
    at Zone.run (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-evergreen.js:123:1)
    at runInTestZone (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-testing.js:545:1)
    at UserContext.<anonymous> (http://localhost:9876/_karma_webpack_/node_modules/zone.js/dist/zone-testing.js:560:1)
    at <Jasmine>

StorageService > should be created

NullInjectorError: R3InjectorError(DynamicTestModule)[StorageService -> HttpClient -> HttpClient]:
  NullInjectorError: No provider for HttpClient!
error properties: Object({ ngTempTokenPath: null, ngTokenPath: [ 'StorageService', 'HttpClient', 'HttpClient' ] })
    at <Jasmine>
    at NullInjector.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:1076:1)
    at R3Injector.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:16629:1)
    at R3Injector.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:16629:1)
    at injectInjectorOnly (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:931:1)
    at ɵɵinject (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:941:1)
    at Object.StorageService_Factory [as factory] (ng:///StorageService/ɵfac.js:5:42)
    at R3Injector.hydrate (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:16865:42)
    at R3Injector.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:16617:1)
    at NgModuleRef$1.get (http://localhost:9876/_karma_webpack_/node_modules/@angular/core/__ivy_ngcc__/fesm2015/core.js:36027:1)
```

What now?

This is where mocking comes handy. We can use things like mocks and spies to create fake versions of the dependencies and **properly** test our Angular service.

So... back to the test file. Here's how we fix the null injection error. I've excluded the actual tests to keep the example simple.

```
import { HttpClient } from '@angular/common/http';
import { TestBed } from '@angular/core/testing';

import { StorageService } from './storage.service';

describe('StorageService', () => {
  let service: StorageService;
  let httpSpy: jasmine.SpyObj<HttpClient>;

  beforeEach(() => {
    httpSpy = jasmine.createSpyObj('HttpClient', ['get', 'post']);
    TestBed.configureTestingModule({
      providers: [
        {
          provide: HttpClient, useValue: httpSpy
        }
      ]
    });
    service = TestBed.inject(StorageService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });
});
```

So what have we just done?

First, we used jasmine to create a spy object that pretends to be the HTTP client. Then we injected our spy into the testing module.

And what-da-ya-know? Our injection errors vanished!

## Angular how to test service observables.

Now it's time to refactor our storage service to call the remote API instead of using local storage.

Here's what it looks like.

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class StorageService {

  constructor(private httpClient: HttpClient) { }

  setValue(key: string, value: string): Observable<any> {
    return this.httpClient.post("https://storageservice.com", {key: value});
  }

  getValue(key:string): Observable<string> {
    return this.httpClient.get<string>(`https://storageservice.com?key=${key}`);
  }

  clearStorage(): Observable<any> {
    return this.httpClient.delete(`https://storageservice.com`);
  }
}
```

But now all of our tests are broken. 😩

How do we fix them?

Here's our refactored test file.

```
import { HttpClient } from '@angular/common/http';
import { fakeAsync, TestBed } from '@angular/core/testing';
import { of } from 'rxjs';

import { StorageService } from './storage.service';

describe('StorageService', () => {
  let service: StorageService;
  let httpSpy: jasmine.SpyObj<HttpClient>;

  beforeEach(() => {
    httpSpy = jasmine.createSpyObj('HttpClient', ['get', 'post', 'delete']);
    TestBed.configureTestingModule({
      providers: [
        {
          provide: HttpClient, useValue: httpSpy
        }
      ]
    });
    service = TestBed.inject(StorageService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('should store a value', fakeAsync(() => {
    let key = "name";
    let value = "Daniel";

    httpSpy.post.and.returnValue(of(true));
    httpSpy.get.and.returnValue(of(value));

    service.setValue(key, value);

    let result = localStorage.getItem(key);

    expect(result).toEqual(value);
    expect(httpSpy.post).toHaveBeenCalled();
  }));
  it('should get value', fakeAsync((done: DoneFn) => {
    let key = "email";
    let value = "email@email.com";

    httpSpy.post.and.returnValue(of(true));
    httpSpy.get.and.returnValue(of(value));

    service.setValue(key, value);

    service.getValue(key).subscribe((result) => {
      expect(result).toEqual(value);
      expect(httpSpy.get).toHaveBeenCalled();
      done;
    });
  }));

  it('should clear everything', fakeAsync((done: DoneFn) => {
    httpSpy.delete.and.returnValue(of(true));

    service.clearStorage().subscribe((result) => {
      expect(httpSpy.delete).toHaveBeenCalled();
      done;
    });
  }));
});
```

Testing asynchronous code can make your head spin at first but it's actually pretty simple once you get the hang of things.

So, what have we just done?

There are a couple important pieces in the example above that may not go unnoticed.

The first one is the fakeAsync function. That function is used to wrap any test with asynchronous operations. It's a tremendous tool when testing Observables and Promises.

The next is the fact that we set up mock return values for our httpSpy. Notice how I told the httpSpy to return specific values based on how it was called? This is the power of mocking dependencies.

And last of all, notice how we use an expect statement to expect that the httpSpy was called and being used.

And that, my friend, is the complete introduction for beginners on how to start testing Angular services.

- ## Testing HTTP Services

Have you ever noticed...

That...

A common dependency inside an Angular service file is the HttpClient? 😸

Maybe your experience with Angular has been different from mine but I find that the HttpClient is a common dependency in an Angular service.

And rightly so.

It's considered best practice to put your data logic inside a service that can be shared across the application. This leaves the job of displaying or rendering the data to your front-end components. Whereas your service is responsible for retrieving that data from whatever your source happens to be.

But just how do you test a service that depends on the HttpClient?

Once upon a time there was a young Angular developer named Harry.

This developer had learned some basic things about the Angular framework and was given the job of building a shiny, new Angular application for his company.

Harry began coding and doing the best he knew how. He followed all the best practices he read about on the Angular website. He always double-checked to make sure that the HttpClient was never injected in any of his components, leaving the API calls for a service that he then injected into his Angular components.

Things were going well...

Until...

His boss showed up one day and told him that before the new app could hit production he needed to write tests for all the public methods in his services. 🙈 🙈 🙈

"No problem", Harry said, and he went back to his computer and began writing tests.

He had written a dozen tests when he came to a service that depended on the HttpClient module. Confused, he sat down to figure this out and try to decide how to best test this Angular service that depended on the HttpClient module.

First he tried to inject the HttpClient module but that threw a weird injection error.

So he went to Google to try to figure this out and instead of finding the clarification he needed, his head was soon about to throw a stackoverflow exception.

*Should I mock the HTTP Client or have it call a real API server?*

*If I have it call a real API server, then I'll first have to authenticate with it, and how would I do that?*

*How do I mock Angular's HTTP Client without getting an injection error?*

*Maybe I should just skip the tests for this Angular service? I think my boss won't find out.*

So, how should Harry test his Angular service?

Should he use a unit test? Or an integration test?

And that, my friend, is what this chapter about testing Angular HTTP services is for.

I'm going to show you and Harry how to properly test your Angular HTTP services and help shake those nasty bugs out of your code.

Should you mock the Angular HttpClient? Or call a real API service?

When you write tests for your Angular service, should you mock the HTTP calls to your data service (API server)?

Or should you actually call a real server?

Well... it all depends on what you want to accomplish by writing tests for your HTTP service.

Sometimes it's a good idea to call a real API server when writing tests for your HTTP services. But more times than not, those tests are brittle integration that snap, pop and blow up in your face. They break faster and sooner than unit tests.

So instead of making real HTTP calls in your Angular tests, most of the time you should mock the HTTP calls. Or in other words, write unit tests instead of integration tests when testing your data (API) services.

Also, when done properly, unit tests are faster to write and give a higher ROI then integration tests which is why I recommend mocking the Angular HTTP client when testing your data services.

So, now that we've decided to mock the HTTP client, how do we test an Angular service that depends on the HttpClient module?

## Writing tests for a GET request

Here's the quickest way to write a unit test for an Angular service that depends on HttpClient.

First, install jasmine-auto-spies.

```
npm i --include=dev jasmine-auto-spies
```

And then, write the test!

```
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
import { TestBed } from '@angular/core/testing';
import { createSpyFromClass, Spy } from 'jasmine-auto-spies';
import { Customer } from '../models/customer';

import { CustomersService } from './customers.service';

describe('CustomersService', () => {
  let service: CustomersService;
  let httpSpy: Spy<HttpClient>;
  let fakeCustomers: Customer[] = [
    {
      id: "1",
      name: "Fake Customer",
      email: "fake@fake.com"
    },
    {
      id: "2",
      name: "Fake Customer Two",
      email: "fake-two@fake.com"
    }
  ];

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        CustomersService,
        { provide: HttpClient, useValue: createSpyFromClass(HttpClient) }
      ]
    });

    service = TestBed.inject(CustomersService);
    httpSpy = TestBed.inject<any>(HttpClient);
  });
  it('should return an expected list of customers', (done: DoneFn) => {
    httpSpy.get.and.nextWith(fakeCustomers);

    service.getAllCustomers().subscribe(
      customers => {
        expect(customers).toHaveSize(fakeCustomers.length);
        done();
      },
      done.fail
    );
    expect(httpSpy.get.calls.count()).toBe(1);
  });
});
```

So, what did we just do?

The test above is from a simple demo project that attempts to simulate a real-world application.

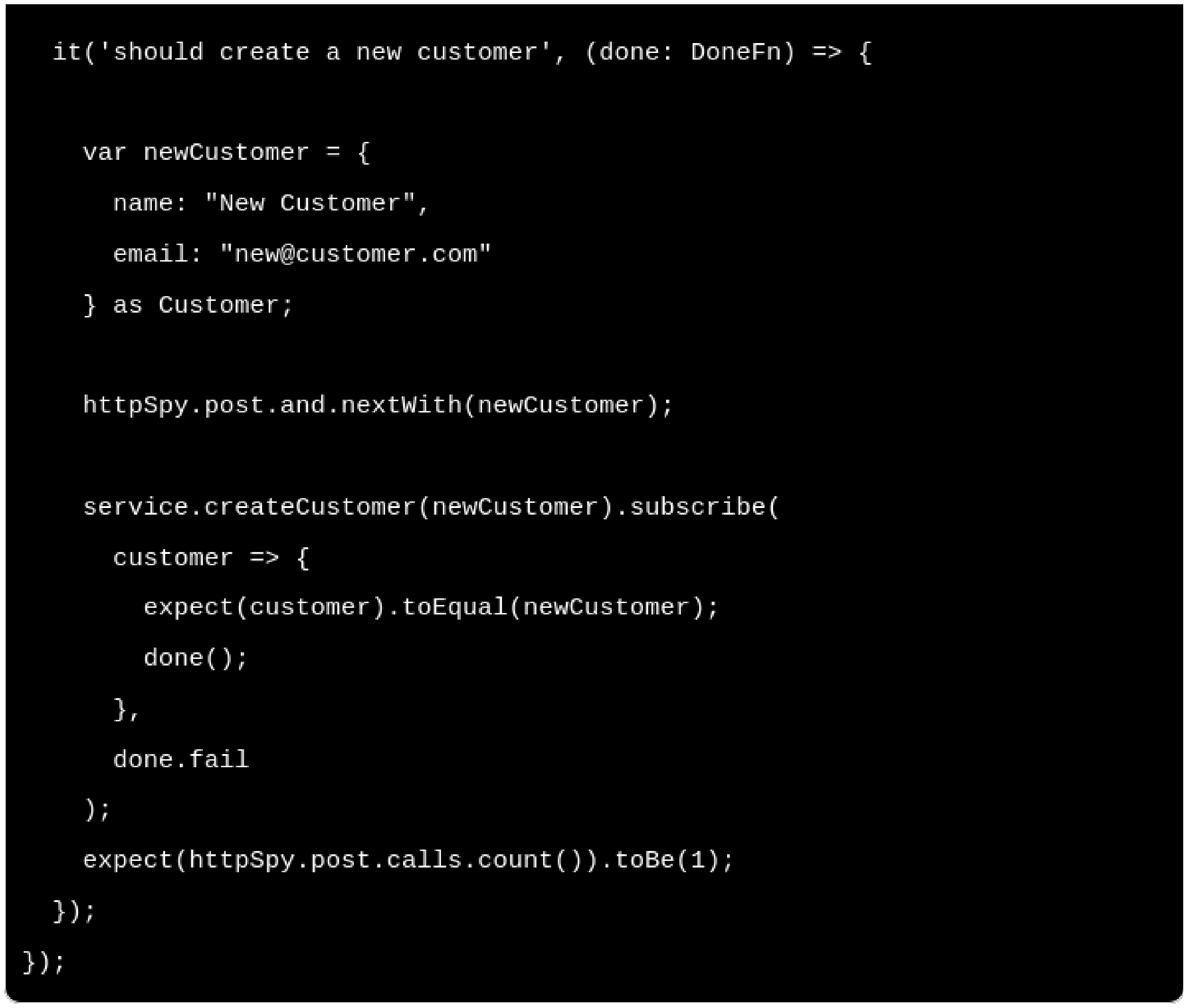
In our beforeEach function we injected the HttpClient as a spy using the handy-dandy jasmine-auto-spies library.

Then, inside the it function, we mocked the HTTP call instead of calling a real server. And then we ran our tests to make sure the getAllCustomers function returns the desired result.

## Writing tests for a POST request

So, what if you have a POST request that you need to write tests for?

Again, use jasmine-auto-spies to mock an HTTP POST request.

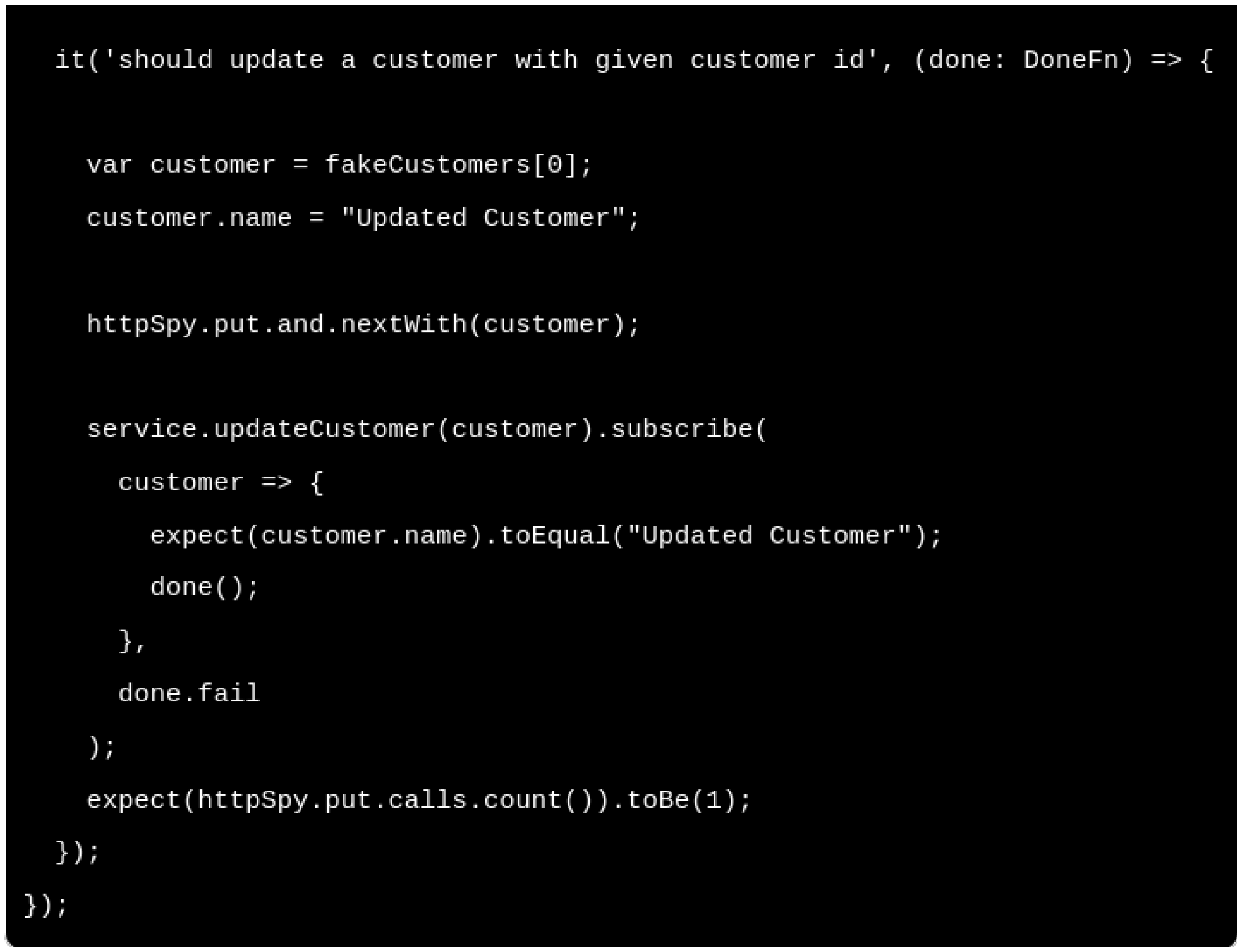


```
  it('should create a new customer', (done: DoneFn) => {

    var newCustomer = {
      name: "New Customer",
      email: "new@customer.com"
    } as Customer;

    httpSpy.post.and.nextWith(newCustomer);

    service.createCustomer(newCustomer).subscribe(
      customer => {
        expect(customer).toEqual(newCustomer);
        done();
      },
      done.fail
    );
    expect(httpSpy.post.calls.count()).toBe(1);
  });
});
```

## Writing tests for a PUT request

What about writing a test for a PUT request?

It's the same as the examples above, except we're swapping POST for PUT.



```
  it('should update a customer with given customer id', (done: DoneFn) => {

    var customer = fakeCustomers[0];
    customer.name = "Updated Customer";

    httpSpy.put.and.nextWith(customer);

    service.updateCustomer(customer).subscribe(
      customer => {
        expect(customer.name).toEqual("Updated Customer");
        done();
      },
      done.fail
    );
    expect(httpSpy.put.calls.count()).toBe(1);
  });
});
```

Writing tests for a DELETE request

```
  it('should update a customer with given customer id', (done: DoneFn) => {

    var customer = fakeCustomers[0];
    customer.name = "Updated Customer";

    httpSpy.put.and.nextWith(customer);

    service.updateCustomer(customer).subscribe(
      customer => {
        expect(customer.name).toEqual("Updated Customer");
        done();
      },
      done.fail
    );
    expect(httpSpy.put.calls.count()).toBe(1);
  });
});
```

## Writing tests for a 404 error

And then, of course, if you're a smart developer you'll want to test for the unexpected errors.

So, how do we test for an HTTP error? Like a 404?

This example was created to mock a 404 response but you can modify it to fit your wants and needs.

```
  it('should return a 404', (done: DoneFn) => {

    var customerId = "89776683";

    httpSpy.get.and.throwWith(new HttpErrorResponse({
          error: "404 - Not Found",
          status: 404
    }));

    service.getCustomer(customerId).subscribe(
      customer => {
        done.fail("Expected a 404");
      },
      error => {
        expect(error.status).toEqual(404);
        done();
      }
    );
    expect(httpSpy.get.calls.count()).toBe(1);
  });
});
```

And that, my friend, completes this section on how to test Angular HTTP services.

By now you and your friend Harry know how to write simple and fast tests for his services. And Harry is already back at it again. His project has a great code coverage score and is almost ready to deploy his new Angular application.

- ## Testing Angular Directives

Angular attribute directives.

How do you test them? Where do you start?

How do you test an Angular attribute directive to make sure that it's acting like you built it to behave?

The Angular attribute directive creates an interesting testing scenario.

Why?

Because most, if not all, of the times you test an attribute directive you will always have to check the DOM to make sure the changes are being reflected.

This is unlike writing unit tests for an Angular component where you can test functionality without verifying the DOM.

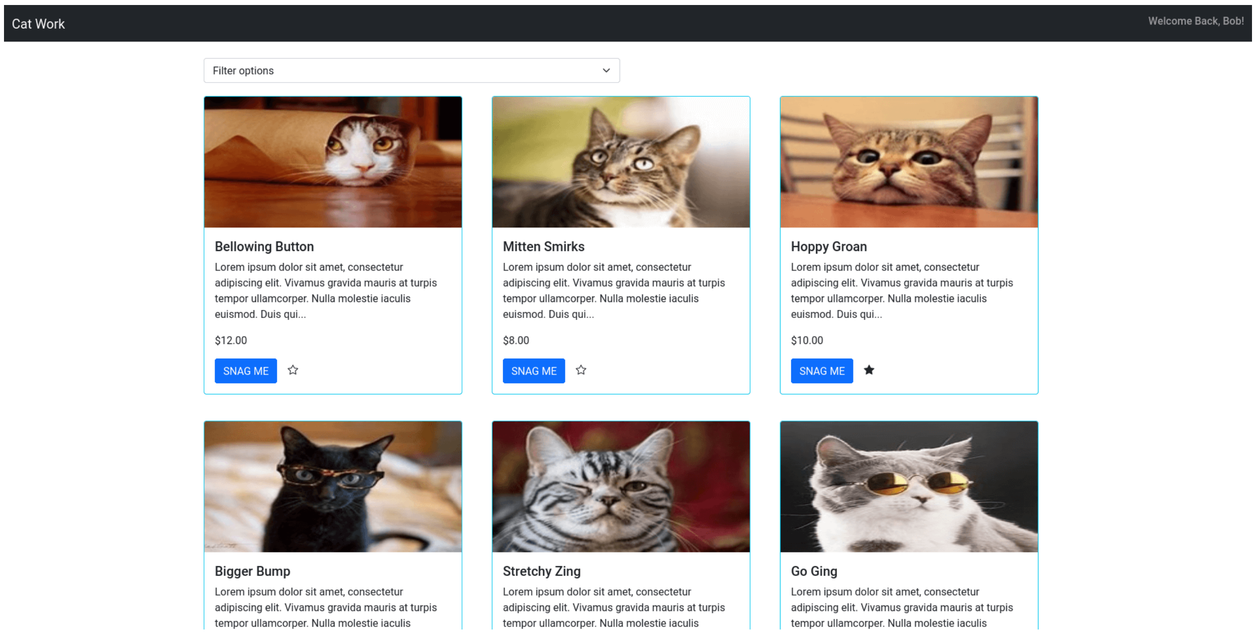
When testing an Angular attribute directive we usually need to check the DOM to make sure the appearance and behavior of our attribute directive is actually being reflected on the page.

Take a look at this example.

Let’s say that in my Angular app, I want to be able to show a star for some of my favorite cats.

So I decided to create an attribute directive to decide if the star icon should be empty or filled, that way I can see my favorite cats.

Here's what it'll look like.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus gravida mauris at turpis tempor ullamcorper. Nulla molestie iaculis euismod. Duis qui...

Using the Angular CLI I ran the following command to generate my new attribute directive.

```
ng generate directive stared
```

And here’s the code.

```
import { Directive, HostBinding, Input } from '@angular/core';

@Directive({
  selector: '[stared]'
})
export class StaredDirective {

  @Input() stared = true;

  constructor() { }

  @Input("class")
  @HostBinding('class')
  get elementClass(): string {
    if (this.stared) {
      return 'bi-star-fill';
    } else {
      return 'bi-star';
    }
  }
}
```

So what are we doing with this new attribute directive?

How does it work?

Based on the input property, we'll add a CSS class to our HTML element that will determine if it's a full or empty star.

Here's how we use it in our component.

```
<i [stared]="true"></i>
```

Cool, eh? 😺

But how do we test it?

Like this.

```
import { Component } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { StaredDirective } from './stared.directive';

@Component({
  template: `<i [stared]="true"></i>`
})
class TestComponent { }

describe('StaredDirective', () => {

  let fixture: ComponentFixture<TestComponent>;

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [TestComponent, StaredDirective]
    })
    .createComponent(TestComponent);

    fixture.detectChanges();
  })

  it('should create an instance', () => {
    const directive = new StaredDirective();
    expect(directive).toBeTruthy();
  });

  it('should be stared', () => {
    var element: HTMLElement = fixture.nativeElement.querySelector("i");
    expect(element.className).toEqual("bi-star-fill");
  });
});
```

And that, my friend, is how to test an attribute in Angular. And verify that the behavior and appearance of our attribute is always working like we expect it to work.

- Testing Reactive Angular Forms

Here's what this chapter on testing reactive Angular forms is all about.

- Discover the best practices and tips for testing reactive Angular forms.
- Create a reactive Angular form and write your first test.
- Learn how to test a contact form.
- Learn how to test a login form.

Say we’ve got a contact form declared like this.

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'app-contact-form',
  templateUrl: './contact-form.component.html',
  styleUrls: ['./contact-form.component.css']
})
export class ContactFormComponent {

  contactForm = new FormGroup({
    name: new FormControl('', Validators.required),
    email: new FormControl('', [Validators.required, Validators.email]),
    message: new FormControl('', [Validators.required])
  });

  constructor() { }

  sendMessage(): void {
    if (this.contactForm.invalid) {
      return;
    }

    var name = this.contactForm.get("name").value;
    var email = this.contactForm.get("email").value;
    var message = this.contactForm.get("message").value;

    // TODO: Send a message to my aunt's nephew's brother's sister-in-law's husband in North Korea.
  }

}
```

If you open the new test file you’ll probably see something like this.

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

import { ContactFormComponent } from './contact-form.component';

describe('ContactFormComponent', () => {
  let component: ContactFormComponent;
  let fixture: ComponentFixture<ContactFormComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ ContactFormComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(ContactFormComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

This was automatically generated by the Angular CLI when we created the component.

How about we begin with a test that makes sure our contact form cannot be sent if the email address given us is not valid?

It's easier than you think.

```
it('should require valid email', () => {
    component.contactForm.setValue({
      "name": "",
      "email": "invalidemail",
      "message": ""
    });

    expect(component.contactForm.valid).toEqual(false);
  });
```

What did we just do?

First, we set the value of our form with the help of the setValue(...) function.

And then we expected the contact form to be invalid.

**What about checking for a valid state?**

Here's the sauce.

```
it('should be valid if form value is valid', () => {
    component.contactForm.setValue({
      "name": "Bobby",
      "email": "bobby@bobby.com",
      "message": "Email me a soda, please."
    });

    expect(component.contactForm.valid).toEqual(true);
  });
```

How about another example? A login form.

Let's say we've got a login form with an HTML template like this.

```
<div class="container">
  <form [formGroup]="loginForm">
    <label for="name">Email</label>
    <input type="email" id="name" formControlName="email" />

    <label for="password">Password</label>
    <input type="password" id="password" formControlName="password" />

    <button
      type="submit"
      value="Submit"
      [disabled]="this.loginForm.invalid"
      (click)="login()"
    >
      Login
    </button>
  </form>
</div>
```

And a Typescript file like this.

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';
import { AuthService } from '../auth-service.service';

@Component({
  selector: 'app-login-form',
  templateUrl: './login-form.component.html',
  styleUrls: ['./login-form.component.css']
})
export class LoginFormComponent {

  loginForm = new FormGroup({
    email: new FormControl('', [Validators.email, Validators.required]),
    password: new FormControl('', Validators.required)
  });

  constructor(private authService: AuthService) { }

  login(): void {
    if (this.loginForm.invalid) {
      return;
    }

    let email = this.loginForm.get('email').value;
    let password = this.loginForm.get('password').value;

    this.authService.login(email, password).subscribe(() => {
      // go to home page
    });
  }

}
```

Notice we're calling an authentication service and then if that succeeds we will redirect the user to the home page.

How do we test this?

We'll have to first create a service spy for our authentication service and configure our spy to return the expected values. Then, inside our test, we'll verify that our authentication service was called with the expected data.

Here's the entire test example.

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { of } from 'rxjs';
import { AuthService } from '../auth-service.service';

import { LoginFormComponent } from './login-form.component';

describe('LoginFormComponent', () => {
  let component: LoginFormComponent;
  let fixture: ComponentFixture<LoginFormComponent>;

  let authServiceSpy = jasmine.createSpyObj('AuthService', ['login']);
  authServiceSpy.login.and.returnValue(of());

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ LoginFormComponent ],
      providers: [
        {
          provide: AuthService, useValue: authServiceSpy
        }
      ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(LoginFormComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
  it('should allow user to log in', () => {
    const formData = {
      "email": "something@somewhere.com",
      "password": "8938ndisn@din"
    };
    component.loginForm.setValue(formData);
    component.login();

    expect(authServiceSpy.login).toHaveBeenCalledWith(formData.email, formData.password);
  })
});
```

What about checking to make sure a user cannot log in if the form has invalid data?

```
it('should not allow user to log in', () => {
    const formData = {
      "email": "invalidemail",
      "password": "8938ndisn@din"
    };
    component.loginForm.setValue(formData);
    component.login();

    expect(component.loginForm.invalid).toEqual(true);
    expect(authServiceSpy.login).toHaveBeenCalledTimes(0);
  });
```

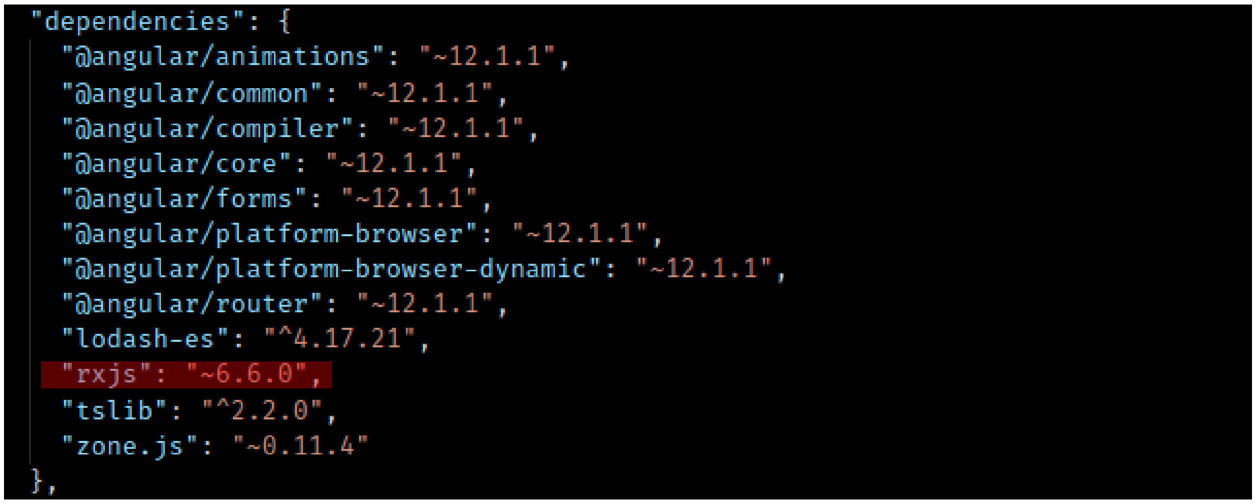
Testing reactive Angular forms is not hard to do.

And man-oh-man is sure gratifying to watch your tests succeed and your screen light up with a bunch of green reports!

- ## How to Test Asynchronous Code

The Angular framework has a gang of dandy-cool features. 😎

One of those is that it comes loaded with the RxJS library - giving you the benefits of reactive programming for browser based applications.



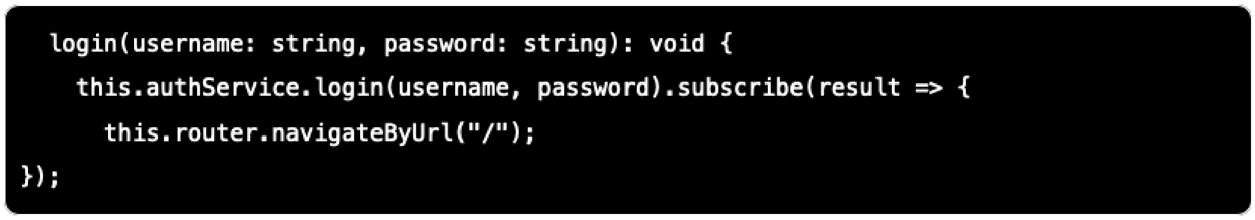
```
"dependencies": {
  "@angular/animations": "~12.1.1",
  "@angular/common": "~12.1.1",
  "@angular/compiler": "~12.1.1",
  "@angular/core": "~12.1.1",
  "@angular/forms": "~12.1.1",
  "@angular/platform-browser": "~12.1.1",
  "@angular/platform-browser-dynamic": "~12.1.1",
  "@angular/router": "~12.1.1",
  "lodash-es": "^4.17.21",
  "rxjs": "~6.6.0",
  "tslib": "^2.2.0",
  "zone.js": "~0.11.4"
},
```

It's common for a new Angular developer to wrestle with RxJS.

But once they've mastered it, they commonly conclude that RxJS is the coolest thing since sliced bread... until they're told to write tests for an Observable or Promise. 😅

For example...

Imagine we have a login function that's part of our login component.

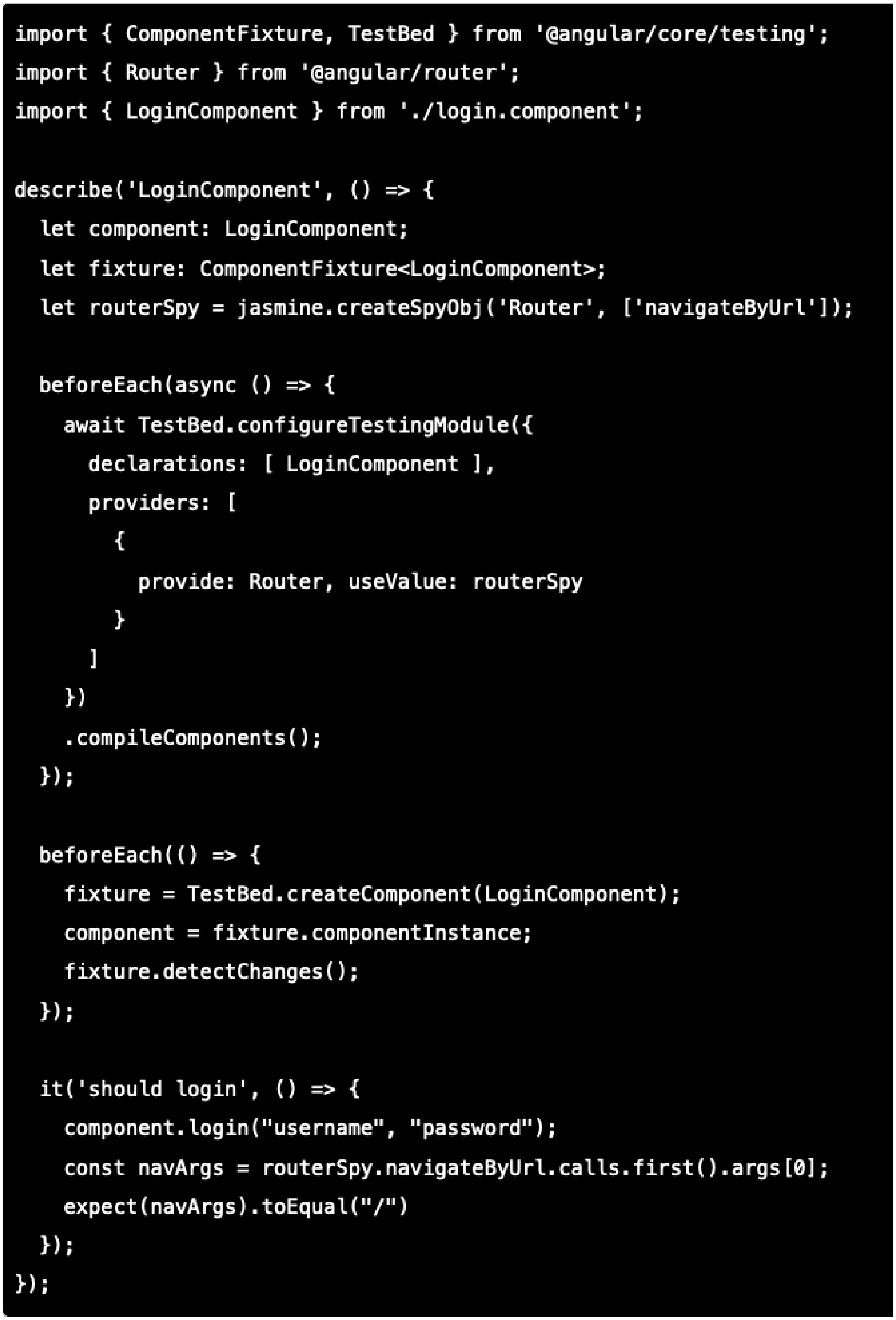


```
  login(username: string, password: string): void {
    this.authService.login(username, password).subscribe(result => {
      this.router.navigateByUrl("/");
});
```

How do you test this function?

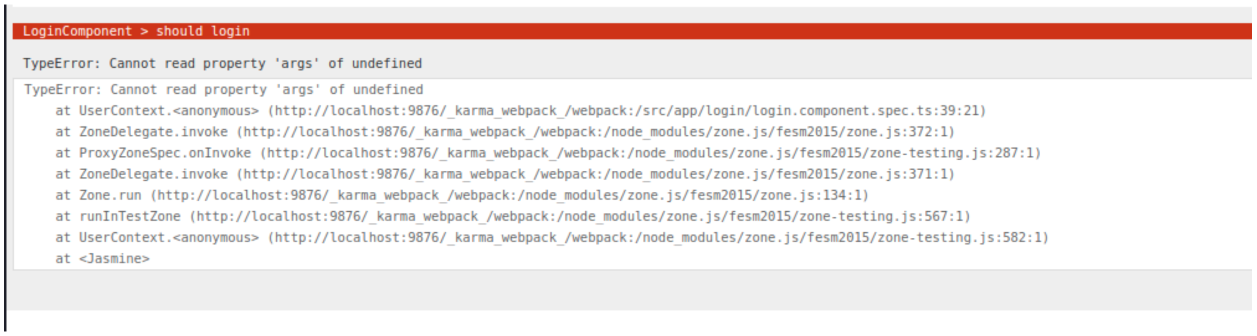
A reasonable test would call the login function with a valid username and password and then verify that we've been routed to the home page.

Like this.



```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { Router } from '@angular/router';
import { LoginComponent } from './login.component';

describe('LoginComponent', () => {
  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let routerSpy = jasmine.createSpyObj('Router', ['navigateByUrl']);

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ LoginComponent ],
      providers: [
        {
          provide: Router, useValue: routerSpy
        }
      ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(LoginComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should login', () => {
    component.login("username", "password");
    const navArgs = routerSpy.navigateByUrl.calls.first().args[0];
    expect(navArgs).toEqual("/")
  });
});
```

But this test give us a null error, saying that our router spy was never called. 😼



```
LoginComponent > should login

TypeError: Cannot read property 'args' of undefined

TypeError: Cannot read property 'args' of undefined
    at UserContext.<anonymous> (http://localhost:9876/_karma_webpack_/webpack:/src/app/login/login.component.spec.ts:39:21)
    at ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:372:1)
    at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone-testing.js:287:1)
    at ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:371:1)
    at Zone.run (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:134:1)
    at runInTestZone (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone-testing.js:567:1)
    at UserContext.<anonymous> (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone-testing.js:582:1)
    at <Jasmine>
```

The reason we get this error is because our expectation was called before the login function was finished.

So how do we fix this weird problem?

Taking the failing test from before, all we have to do is use the fakeAsync and tick methods to fix our asynchronous conflicts.

Here's the new code.

```
  it('should login', fakeAsync(() => {
    component.login("username", "password");
    tick();

    const navArgs = routerSpy.navigateByUrl.calls.first().args[0];
    expect(navArgs).toEqual("/")
}));
```

The new methods can be imported from @angular/core/testing like this.

```
import { fakeAsync, tick } from '@angular/core/testing';
```

And BANG! Our test is now passing! 😎

What did we just do?

To improve our understanding we'll investigate a few of the testing API's that come with Angular.

The Angular testing API comes with a handful of functions that are required when testing asynchronous code that includes things like observables and promises.

Below are the 3 key methods you'll need to know. You'll have to know and understand these to be able to effectively test your Angular application.

- tick

This is to simulate the asynchronous passage of time for any asynchronous code inside a *fakeAsync* zone.

For example, if your asynchronous function takes a second to return a value, you can use the tick function to simulate the passage of a second like this...

```
tick(1000);
```

...and then carry on with your testing.

- fakeAsync

This will wrap a function and execute it in the *fakeAsync* zone.

What does that mean?

It means we're given a zone where we can run asynchronous code, and control time using the *tick* function.

Here's how we use it.

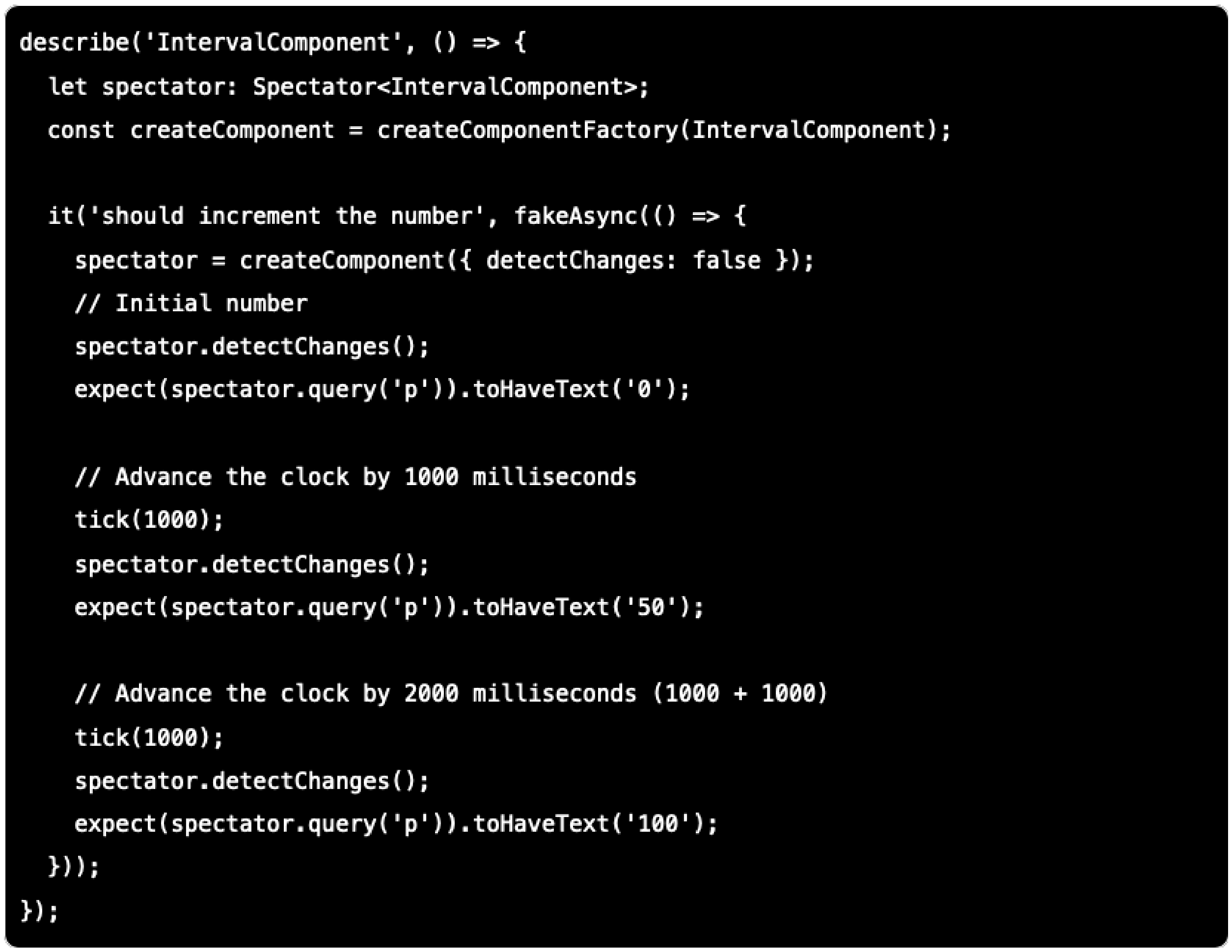
```
describe('this test', () => {
  it('looks async but is synchronous', fakeAsync((): void => {
       let flag = false;
       setTimeout(() => {
         flag = true;
       }, 100);
       expect(flag).toBe(false);
       tick(50);
       expect(flag).toBe(false);
       tick(50);
       expect(flag).toBe(true);
     }));
});
```

- waitForAsync

This function creates an asynchronous test zone that will automatically complete when all asynchronous operations inside its test zone have completed.

## Real-world examples

### Example: DOM testing asynchronous operations



```
describe('IntervalComponent', () => {
  let spectator: Spectator<IntervalComponent>;
  const createComponent = createComponentFactory(IntervalComponent);

  it('should increment the number', fakeAsync(() => {
    spectator = createComponent({ detectChanges: false });
    // Initial number
    spectator.detectChanges();
    expect(spectator.query('p')).toHaveText('0');

    // Advance the clock by 1000 milliseconds
    tick(1000);
    spectator.detectChanges();
    expect(spectator.query('p')).toHaveText('50');

    // Advance the clock by 2000 milliseconds (1000 + 1000)
    tick(1000);
    spectator.detectChanges();
    expect(spectator.query('p')).toHaveText('100');
  }));
});
```

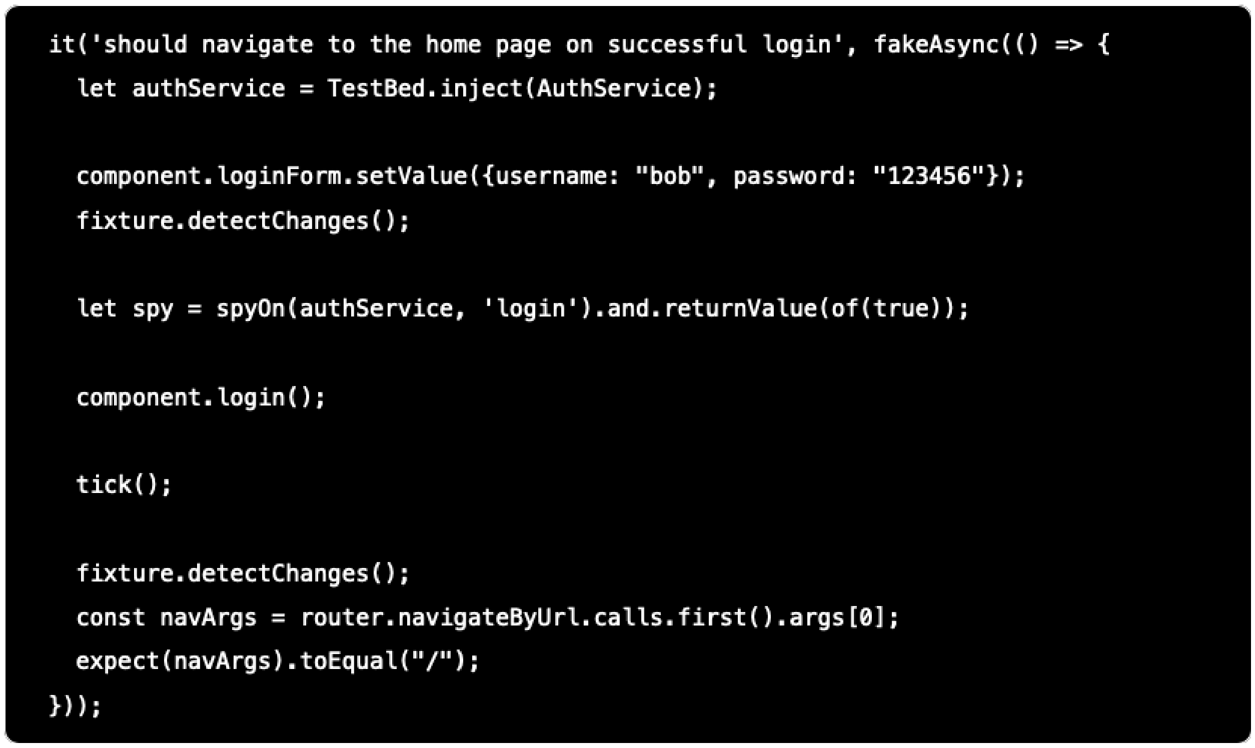
Example: Testing an asynchronous message

```
describe('TestComponent', () => {
  let spectator: Spectator<TestComponent>;
  const createComponent = createComponentFactory(TestComponent);

  beforeEach(() => (spectator = createComponent()));

  it('should show the message on submit and remove it after 2 seconds', fakeAsync(() => {
    expect(spectator.query('p')).not.toExist();
    spectator.click('button');
    expect(spectator.query('p')).toExist();

    // Advance the virtual clock by 2 seconds
    tick(2000);
    spectator.detectChanges();
    expect(spectator.query('p')).not.toExist();
  }));
});
```

Example: Testing an asynchronous service that returns a list of users

```
it('should resolve the promise and show the users list', fakeAsync(() => {
  const usersService = spectator.get(UsersService);
  usersService.getUsers.and.callFake(() => Promise.resolve([{ id: 1 }, { id: 2 }]));

  // Run ngOnInit
  spectator.detectChanges();

  // Resolve all Promises
  flushMicrotasks();

  spectator.detectChanges();
  expect(spectator.queryAll('li').length).toBe(2);
}));
```

Example: Testing an asynchronous login function



```
it('should navigate to the home page on successful login', fakeAsync(() => {
  let authService = TestBed.inject(AuthService);

  component.loginForm.setValue({username: "bob", password: "123456"});
  fixture.detectChanges();

  let spy = spyOn(authService, 'login').and.returnValue(of(true));

  component.login();

  tick();

  fixture.detectChanges();
  const navArgs = router.navigateByUrl.calls.first().args[0];
  expect(navArgs).toEqual("/");
}));
```

## ● E2E Testing

So what is E2E testing?

E2E tests are also known as system tests or smoke tests.

These tests are used to check the function of the entire application by simulating a real user as it interacts with your Angular application.

For example, an E2E test could be simulating a login into an application and then clicking around to make sure that data is properly being displayed.

The most popular tool for E2E testing with Angular used to be Protractor.

Protractor is an end-to-end test framework that was created specifically for Angular and AngularJS applications. It is a Node.js program built on top of Selenium, which is a browser automation framework.

Protractor runs tests against your application running in a real browser, interacting with it as a user would.

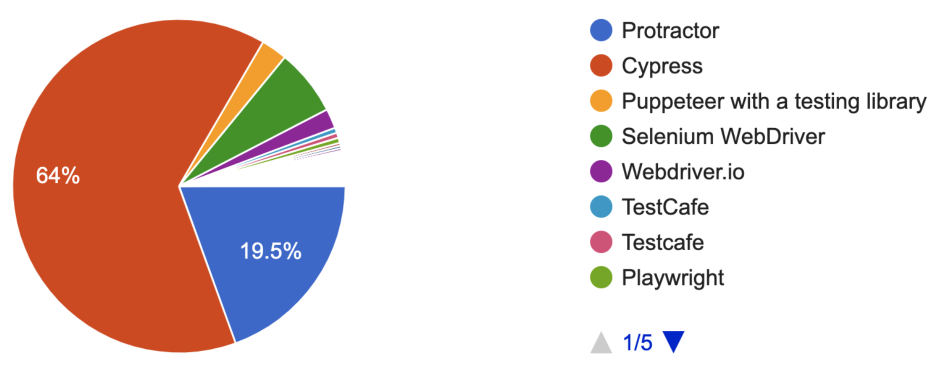
In other words, when you run Protractor your Angular application will launch in a real browser (defaults to Chrome) and interact like a user would.

Protractor tests consist of imitating user interactions like clicking on buttons and filling out input fields. Protractor can also check DOM elements to make sure that your Angular application is behaving like you expect it to behave.

In general testing terminology, Protractor is used to write system tests that test the entire performance and behavior of an application. But in the Angular world these are more commonly known as E2E tests.

However, in a recent survey done by the Angular team, Cypress is by far the most popular tool for testing Angular apps.

What e2e testing tools do you use?



Protractor is losing popularity but does that mean you should ditch it?

The Angular team recently announced plans to end Protractor support. Common complaints were that Protractor didn't test like a real user. That it was hard to trace errors when it fails. And developers often ended up using a lot of console logs to debug problems.

Given these problems and the emergence of tools like Cypress, the Angular team has decided to stop supporting Protractor.

I recommend you use Cypress but, if for some reason you would rather use Protractor then here’s how to do it.

- ### Protractor

So, how do we use Protractor to simulate a real user?

How do we use Protractor to test an Angular application?

When you use the Angular CLI to create an Angular application it automatically configures and prepares Protractor for you.

**Important Update:** As of Angular 12 the Angular team has decided to deprecate Protractor. If you want to use Protractor with Angular 12 or newer then this article will explain how to set it up.

If you open the project in a code editor you will find a folder named e2e.

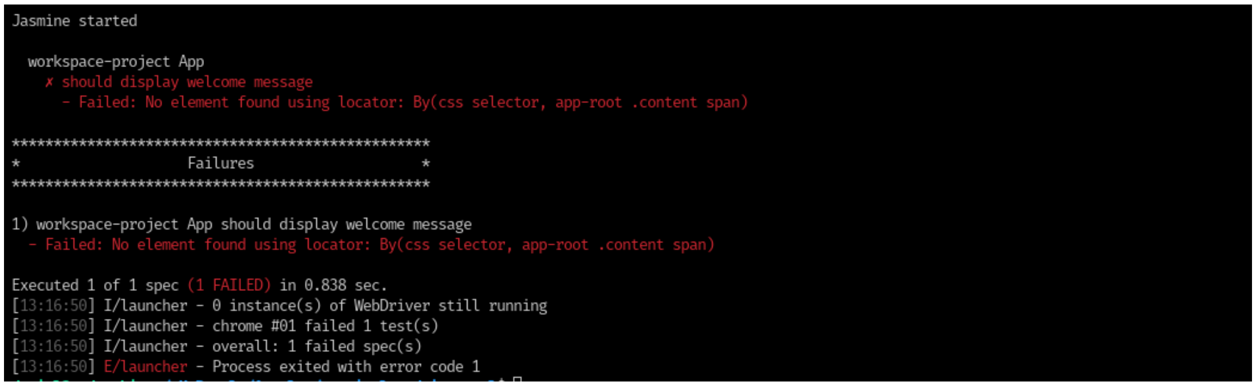
Inside the e2e folder are the configuration files for the Protractor testing framework. And inside this folder you'll find a folder called src that contains the actual tests.

To run the default Protractor tests all you have to do is type the following command.

```
ng e2e
```

And **PRESTO**!

You should see a browser launch, run the test and then print the results in the terminal. If the tests pass you'll see a green success message. Otherwise, a red error.



```
Jasmine started

  workspace-project App
    ✗ should display welcome message
      - Failed: No element found using locator: By(css selector, app-root .content span)

**************************************************
*                    Failures                    *
**************************************************

1) workspace-project App should display welcome message
  - Failed: No element found using locator: By(css selector, app-root .content span)

Executed 1 of 1 spec (1 FAILED) in 0.838 sec.
[13:16:50] I/launcher - 0 instance(s) of WebDriver still running
[13:16:50] I/launcher - chrome #01 failed 1 test(s)
[13:16:50] I/launcher - overall: 1 failed spec(s)
[13:16:50] E/launcher - Process exited with error code 1
```

For the first test, we'll keep things super-duper simple.

We'll write a quick, short test with Protractor that verifies the title of our Angular application.

Open the e2e/src folder and create a new file called app-title.e2e-spec.ts.

Here's the code.

```
import { browser, logging } from 'protractor';


describe('Angular App', function() {
  it('should have a title of App Title', function() {
    browser.get(browser.baseUrl);


    var expected = "App Title";
    var actual = browser.getTitle();


    expect(actual).toEqual(expected);
  });


  afterEach(async () => {
    const logs = await browser.manage().logs().get(logging.Type.BROWSER);
    expect(logs).not.toContain(jasmine.objectContaining({
      level: logging.Level.SEVERE,
    } as logging.Entry));
  });


});
```

What did we just do? And how does this code work?

Inside the test it should have a title of App Title.

We began by telling Protractor to get the home page of our Angular application.

Then we get the title from the home page and compare it to the title we expect it to have to determine if our test failed or passed.

Tada! Tada!! Tada!!!

You've just written your first test using Protractor. 👏 👏 👏

Now that we've learned the basics of writing tests with Protractor you're probably wondering how to advance.

What are some more useful cases?

## Protractor Examples

Here’s how to select an element by ID.

```
import { browser, by, element, logging } from 'protractor';


describe('Angular App', function() {


  it('get login form by id', async () => {
    browser.get(browser.baseUrl);
    var loginForm = element(by.id("login-form"));
    expect(loginForm).toBeDefined();
  });
});
```

And what if you want to check the classes of an HTML element?

```
import { browser, by, element, logging } from 'protractor';

describe('Angular App', function() {

  it('get classes for login button', async () => {
    browser.get(browser.baseUrl);

    var loginButton = element(by.css("button"));
    var actual = loginButton.getAttribute('class');
    var expected = "btn btn-primary btn-lg btn-block";

    expect(actual).toEqual(expected);
  });

  afterEach(async () => {
    const logs = await browser.manage().logs().get(logging.Type.BROWSER);
    expect(logs).not.toContain(jasmine.objectContaining({
      level: logging.Level.SEVERE,
    } as logging.Entry));
  });

});
```

Or get the text in an H1 tag?

```
import { browser, by, element, logging } from 'protractor';


describe('Angular App', function() {


  it('get h1 login tag', async () => {
    browser.get(browser.baseUrl);


    var actual = await element(by.css("h1")).getText();
    var expected = "Loginn";


    expect(actual).toEqual(expected);
  });


  afterEach(async () => {
    const logs = await browser.manage().logs().get(logging.Type.BROWSER);
    expect(logs).not.toContain(jasmine.objectContaining({
      level: logging.Level.SEVERE,
    } as logging.Entry));
  });


});
```

And maybe you want to simulate a click?

```
import { browser, by, element, logging } from 'protractor';


describe('Angular App', function() {


  it('click button', function() {
    browser.get(browser.baseUrl);
    element(by.className("btn")).click();
  });
});
```

Or put text in an input field?

```
import { browser, by, element, logging } from 'protractor';


describe('Angular App', function() {


  it('fill input field', async () => {
    browser.get(browser.baseUrl);


    var emailInput = element(by.id("email"));
    emailInput.sendKeys("email@email.com");
  });
});
```

Or click on a hyperlink?

```
import { browser, by, element, logging } from 'protractor';


describe('Angular App', function() {


  it('fill input field', async () => {

    browser.get(browser.baseUrl);


    var emailInput = element(by.id("email"));

    emailInput.sendKeys("email@email.com");

  });

});
```

And that is how you use Protractor to test your Angular application and make sure the bugs get caught and squashed.

Protractor is a great tool for testing Angular applications.

But keep in mind that e2e tests are limited. And when done wrong they're extremely brittle.

That's why I prefer to depend on unit tests for things like components instead of e2e tests.

- ### Cypress

Want to join the crowd and use Cypress to test your Angular app?

It's probably easier than you think.

Did you know that Cypress is cool?

Cypress is loaded with all kinds of features.

- Supports Firefox, Chrome & Edge
- All in one inclusive testing framework.
- It is for both developers and QA engineers.
- Has native access to everything.
- Works on any frontend framework or website.
- Written only for JavaScript or Typescript frameworks and libraries.
- Is not flaky like Protractor.
- Has a "Time Travel" feature to show what happened at each step of the test.
- Awesome debugging support.

- Supports things like spies, stubs and clocks.
- Automatic waiting feature.
- You can do things like network traffic control.
- Tests like a real user.
- It's reliable and fast.

It has made automated end-to-end testing easier than ever before.

Which leaves us with fewer excuses to create poor quality Angular applications.

Most of the online tutorials I found use the @briebug/cypress-schematic but it should be noted that the package has been deprecated for a while now.

Instead, we'll use the office @cypress/schematic package that's maintained by the Cypress team.

So.

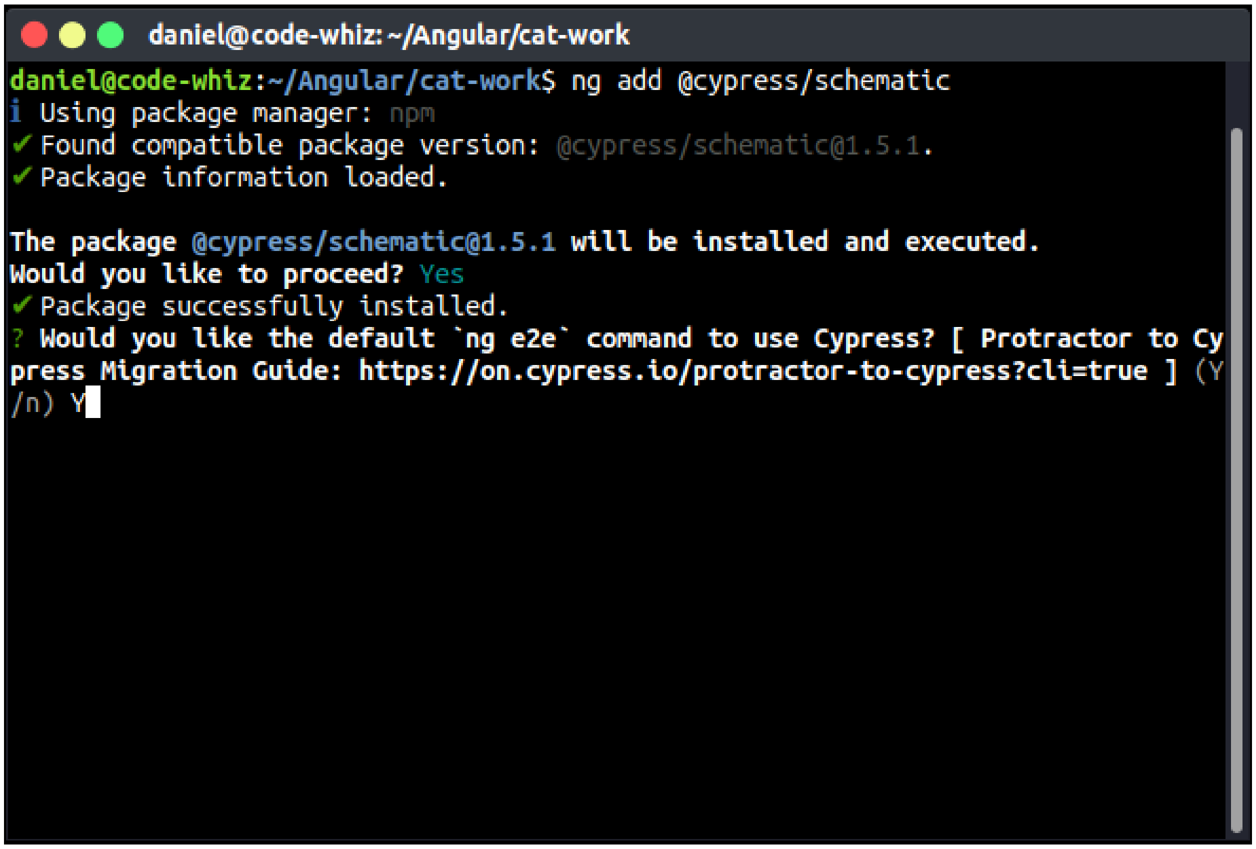
Grab a terminal.

Go to the root directory of your Angular project.

And use the command below to add Cypress to your Angular app.

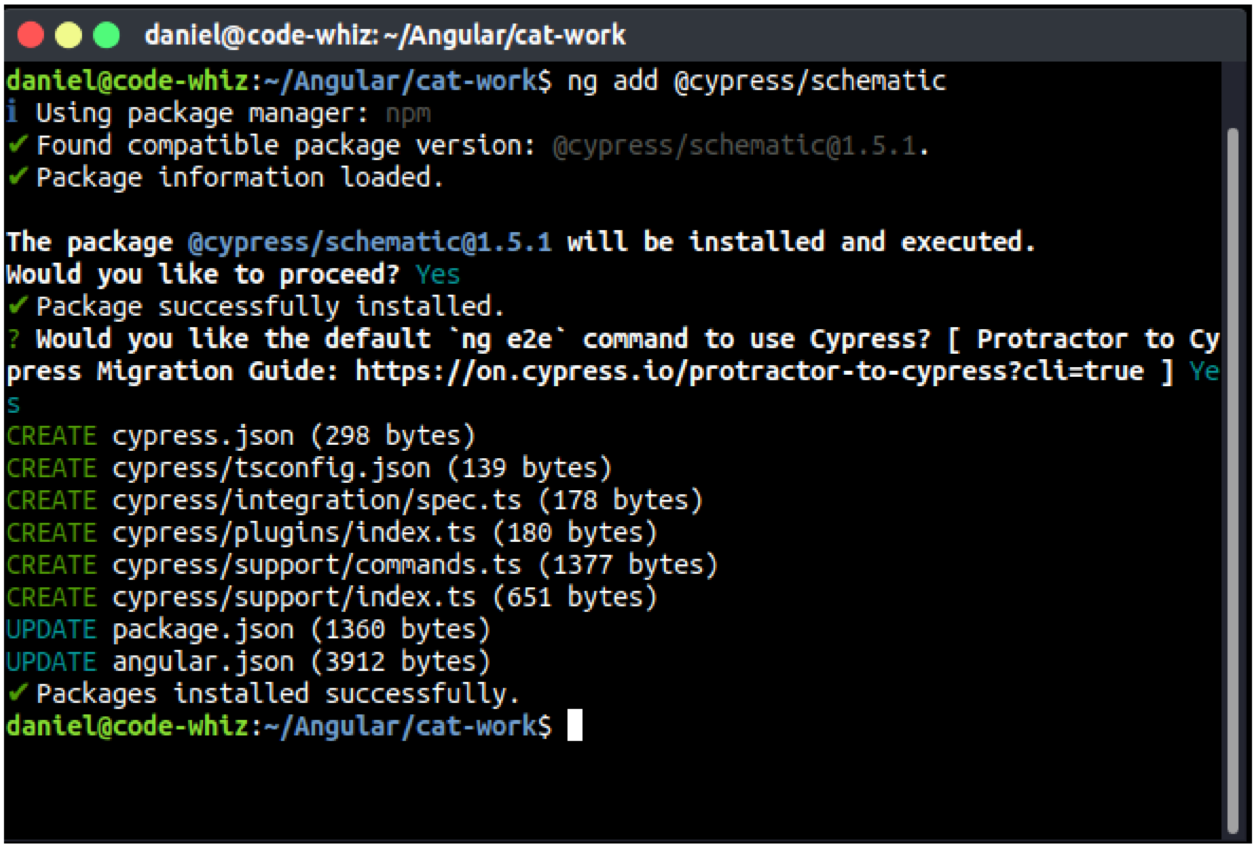
```
ng add @cypress/schematic
```

Next you'll be asked if you want to use the default ng e2e command.



```
daniel@code-whiz:~/Angular/cat-work
daniel@code-whiz:~/Angular/cat-work$ ng add @cypress/schematic
i Using package manager: npm
✔ Found compatible package version: @cypress/schematic@1.5.1.
✔ Package information loaded.

The package @cypress/schematic@1.5.1 will be installed and executed.
Would you like to proceed? Yes
✔ Package successfully installed.
? Would you like the default `ng e2e` command to use Cypress? [ Protractor to Cy
press Migration Guide: https://on.cypress.io/protractor-to-cypress?cli=true ] (Y
/n) Y
```

Once done, you should see a success message like this. 💪



```
daniel@code-whiz:~/Angular/cat-work
daniel@code-whiz:~/Angular/cat-work$ ng add @cypress/schematic
i Using package manager: npm
✔ Found compatible package version: @cypress/schematic@1.5.1.
✔ Package information loaded.

The package @cypress/schematic@1.5.1 will be installed and executed.
Would you like to proceed? Yes
✔ Package successfully installed.
? Would you like the default `ng e2e` command to use Cypress? [ Protractor to Cy
press Migration Guide: https://on.cypress.io/protractor-to-cypress?cli=true ] Ye
s
CREATE cypress.json (298 bytes)
CREATE cypress/tsconfig.json (139 bytes)
CREATE cypress/integration/spec.ts (178 bytes)
CREATE cypress/plugins/index.ts (180 bytes)
CREATE cypress/support/commands.ts (1377 bytes)
CREATE cypress/support/index.ts (651 bytes)
UPDATE package.json (1360 bytes)
UPDATE angular.json (3912 bytes)
✔ Packages installed successfully.
daniel@code-whiz:~/Angular/cat-work$
```

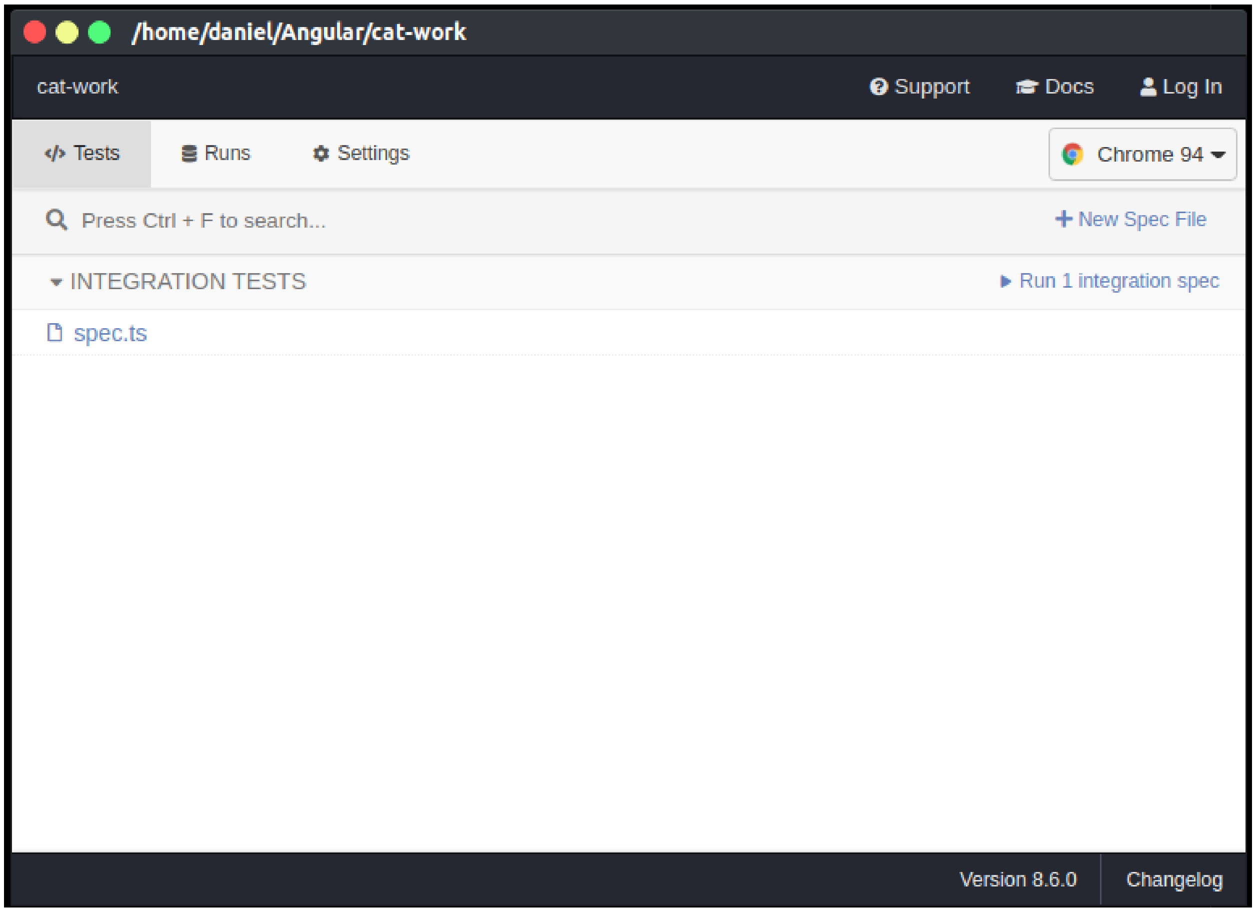
Now, type ng e2e into the command prompt to make sure that Cypress is ready for use.

Let it compile your Angular project...

Watch it start Cypress...

And...

You should see a Cypress window launch. Just like this.



Now that Cypress has been installed you should see a new folder called cypress in the root directory of your Angular project. This is the default location for all Cypress config and test files.

The file we're interested in is cypress\integration\spec.ts.

```
describe('My First Test', () => {
  it('Visits the initial project page', () => {
    cy.visit('/')
    cy.contains('Welcome')
    cy.contains('sandbox app is running!')
  })
});
```

So what do we have going on here?

First, we use the describe block to create a group of tests.

Next, we use the it function to declare an actual test.

Inside the "Visits the initial project page" test we tell Cypress to visit the home page of the Angular application. And then we use the contains function to see if the page contains the text we're looking for.

Simple, eh?

Cypress can do oodles of things.

And their documentation is hands-down awesome.

Best of all, Cypress makes end-to-end testing easy which allows developers like you and I to write better tests and release high-quality software.

We really have no excuses.