**Unit Testing with Jasmine and Karma Theory:**

**Reference URLs:**

<https://codecraft.tv/courses/angular/unit-testing/overview/>

<https://cucumber.io/blog/bdd/bdd-vs-tdd/#:~:text=What%20You're%20Testing,pieces%20of%20functionality%20in%20isolation>.

<https://www.pluralsight.com/blog/software-development/tdd-vs-bdd>

<https://www.positronx.io/angular-unit-testing-application-with-jasmine-karma/>

<https://www.testim.io/blog/angular-component-testing-detailed-guide/>

<https://testing-library.com/docs/angular-testing-library/intro/>

<https://this.isfluent.com/blog/2020/getting-more-value-from-tests-with-angular-testing-library>

<https://www.digitalocean.com/community/tutorials/angular-introduction-unit-testing>

<https://duncanhunter.gitbook.io/testing-angular/testbed-and-fixtures>

<https://livebook.manning.com/book/angular-2-development-with-typescript/chapter-9/62>

<https://angular.io/guide/testing-components-basics>

<https://codecraft.tv/courses/angular/unit-testing/overview/>

<https://medium.com/@abdul_74410/towards-better-testing-in-angular-part-1-mocking-child-components-b51e1fd571da>

<https://jasmine.github.io/pages/getting_started.html> - Standalone Jasmine Distribution

<https://www.freecodecamp.org/news/jasmine-unit-testing-tutorial-4e757c2cbf42/> - Introduction to Jasmine Unit Testing

<https://www.digitalocean.com/community/tutorials/angular-testing-httpclient>

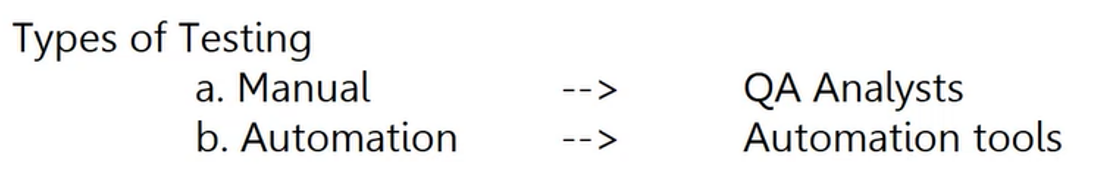
<https://medium.com/netscape/testing-with-the-angular-httpclient-api-648203820712>

[Testing Components – Testing Angular (testing-angular.com)](https://testing-angular.com/testing-components/)

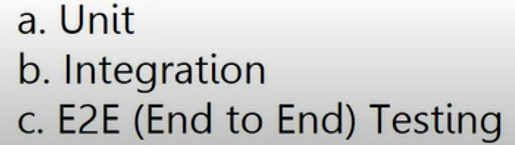
**Why Testing?**



**Types of Testing:**



**Types of Automation Testing:**



**Unit Testing**

This is sometimes also called *Isolated* testing. It’s the practice of testing small isolated pieces of code. If your test uses some external resource, like the network or a database, it’s *not* a unit test.

**Functional Testing**

This is defined as the testing of the complete functionality of an application. In practice with web apps, this means interacting with your application as it’s running in a browser just like a user would interact with it in real life, i.e. via clicks on a page.

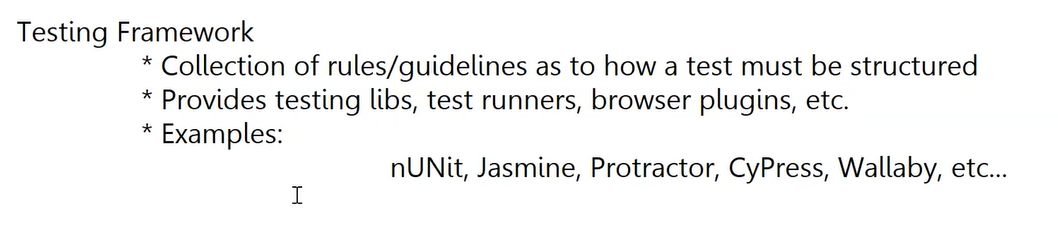
This is also called ***End to End*** or ***E2E*** testing.

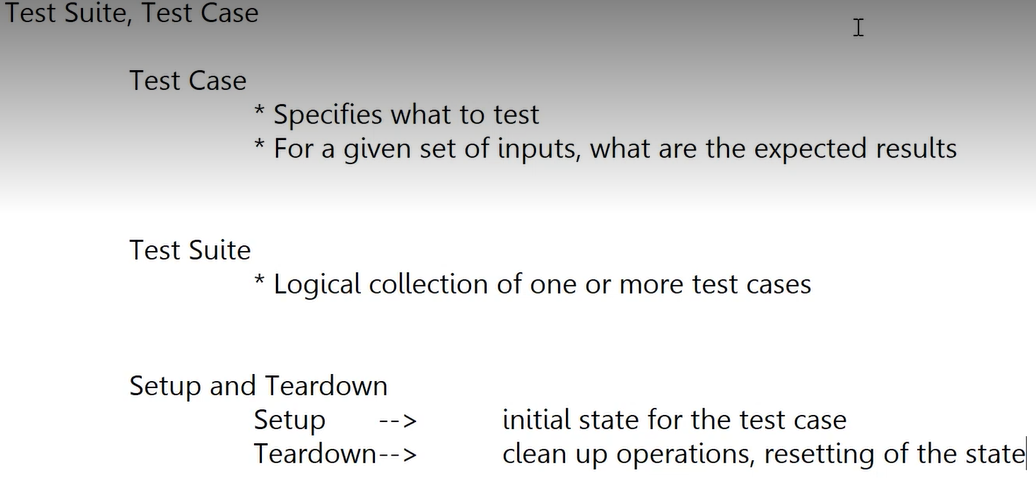
**What is TDD and BDD?**

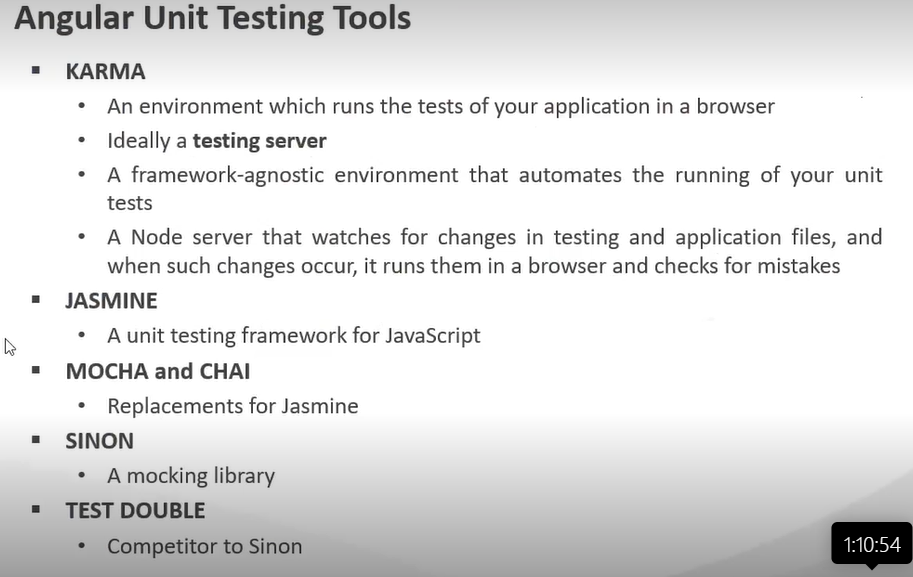
**TDD** is **Test Driven Development.** This means writing a test that fails because the specified functionality doesn't exist, then writing the simplest code that can make the test pass, then refactoring to remove duplication, etc. You repeat this Red-Green-Refactor loop over and over until you have a complete feature.

**BDD** is **Behavior Driven Development.** The key to BDD is to get the specifications from the user. In other words, create tests that are not written by developers. This means tests that are **not** written in a programming language. These tests should be written in a language close to English (or whatever your team speaks.)

**Angular Unit Testing Tools:**







**What is Jasmine and Karma?**

***Jasmine*** *is an open-source behavior-driven testing framework crafted by Pivotal Labs.*

Jasmine provides several valuable functions to write tests. Here are the main Jasmine methods:

**it()**: Declaration of a particular test

**describe()**: It’s a suite of tests

**expect()**: Expect some value in true form

**Installing and Using Jasmine:**

The standalone distribution provides a simple way to run your specs in a web browser. You can download it from the [releases page](https://github.com/jasmine/jasmine/releases).

Included is a sample app and sample specs. Open SpecRunner.html and run the included specs. Both the source files and their respective specs are linked in the <head> of the SpecRunner.html.

To start using Jasmine, replace the source/spec files with your own. Then Load the SpecRunner.html in your favorite browser

/src: contains the source files that you want to test. This may be either deleted if your already have your project's folder setup or can also be used when appropriate for hosting your source code.

/lib: contains the core Jasmine files.

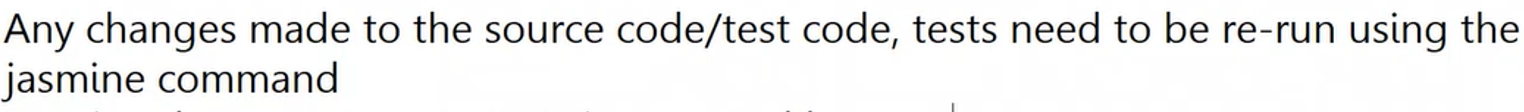
/spec: contains the tests that you are going to write.

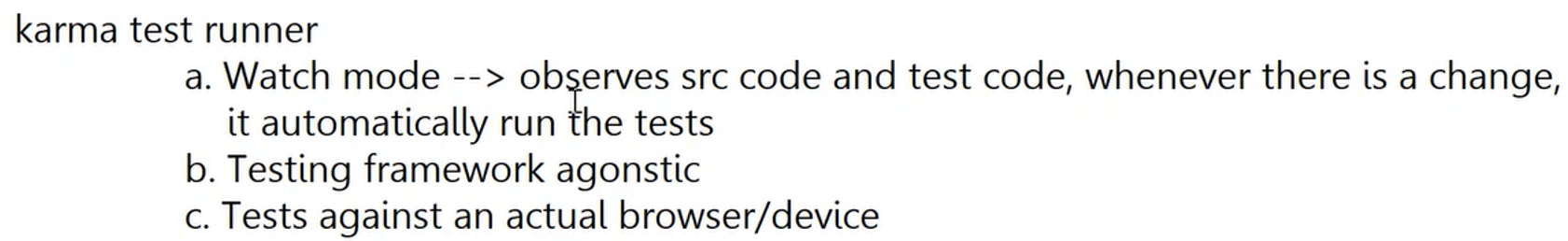
SpecRunner.html: this file is used as a test runner. You run your specs by simply launching this file.

**Karma**

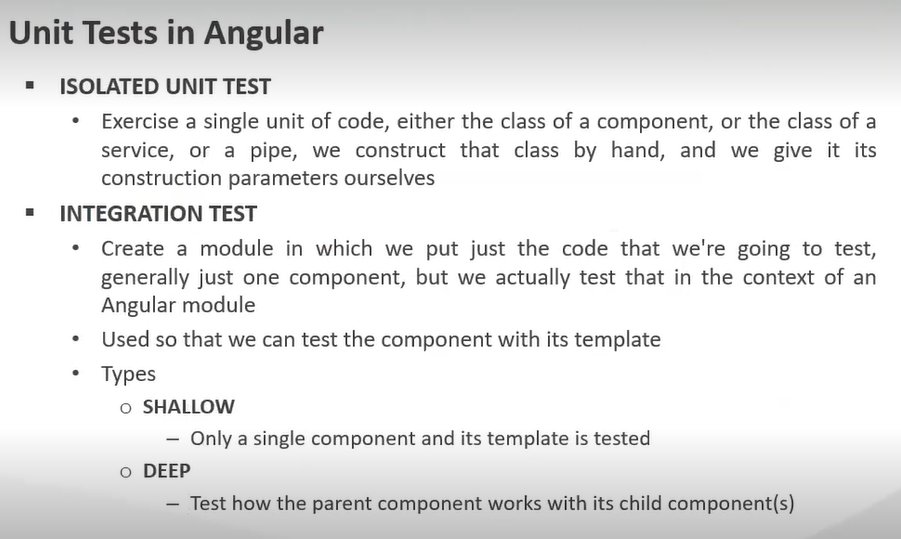
Karma is a test runner tool, it creates a browser instance, run tests to provide the expected results.

The benefit of using Karma is that it can be operated via command line and It refreshes the browser automatically whenever we make even minor changes in our app.

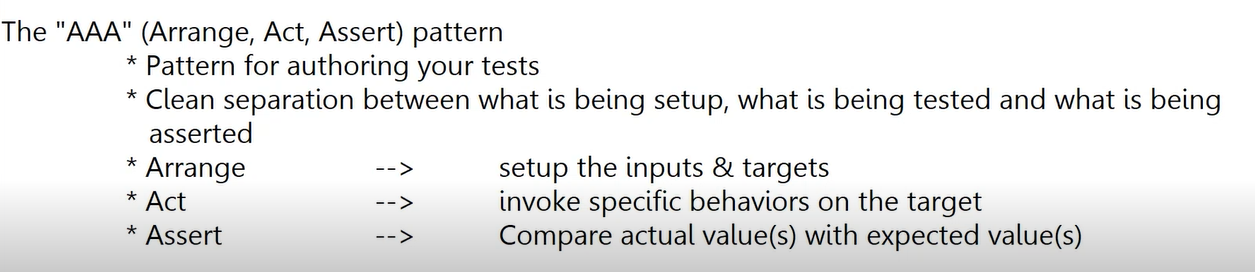




**Types of Unit Testing:**



**Concept of Arrange, Act and Assert:**



**Structure of a Jasmine Unit Test:**

**Jasmine Basic Terminology**

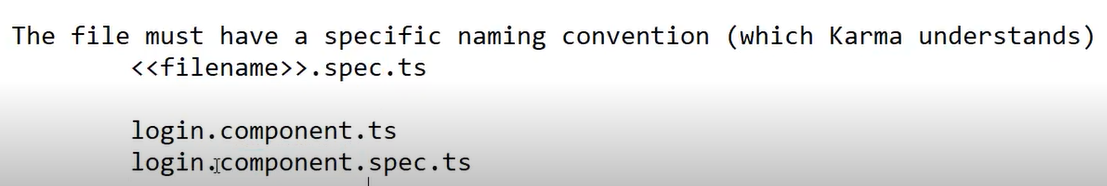
**describe(string, function)** functions take a title and a function containing one or more specs and are also known as a suite or test suite.

**it(string, function)** functions take a title and a function containing one or more expectations and are also known as specs.

**expect(actual)** functions take a value, called an actual. An expect function is typically used alongside a matcher function. Together they return a boolean value that depicts the passing or failing of a spec.

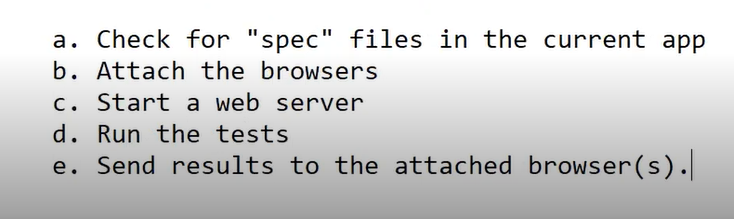
**Matcher functions** take a value that represents the expected value. A matcher function is chained alongside an expect function. Together they return a boolean value that depicts the passing or failing of a spec. Some examples of matchers are toBeTruthy(), toEqual(), and toContain().

**Mandatory Naming Conventions:**



Good practice to keep test files next to artifacts. Test cases should be adjacent to their actual components/pipes/directives.

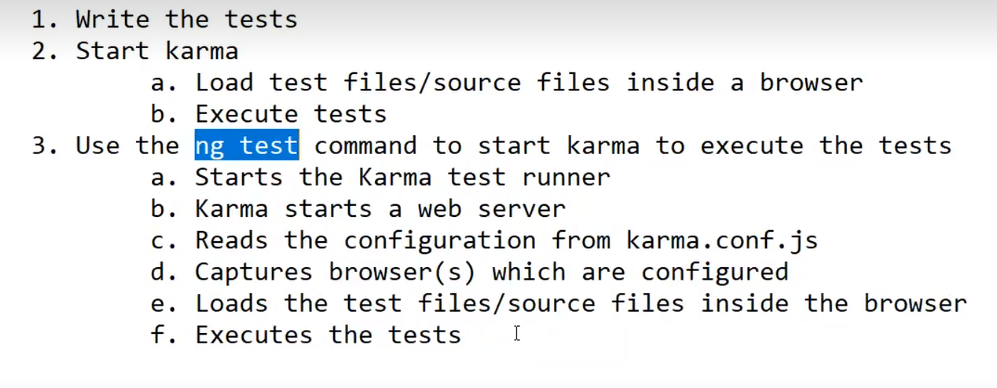
**ng test:** Activates the Karma Test Runner. Use ng test to run the tests.



**spec file:** File that contains one or more unit tests.

**Attach the browsers:** When test cases are run, results are sent to the browsers. Attaches the browsers configured to work with Karma. Starts the web server, attaches the browsers, runs the tests and send the result the attached browsers.

**karma.conf.js: Karma configuration file.** Karma reads configuration using this file. Written using Node.js syntax.



**Jasmine Built-In Matchers:**

Jasmine comes with a few pre-built matchers like so:

expect(array).toContain(member);

expect(fn).toThrow(string);

expect(fn).toThrowError(string);

expect(instance).toBe(instance);

expect(mixed).toBeDefined();

expect(mixed).toBeFalsy();

expect(mixed).toBeNull();

expect(mixed).toBeTruthy();

expect(mixed).toBeUndefined();

expect(mixed).toEqual(mixed);

expect(mixed).toMatch(pattern);

expect(number).toBeCloseTo(number, decimalPlaces);

expect(number).toBeGreaterThan(number);

expect(number).toBeLessThan(number);

expect(number).toBeNaN();

expect(spy).toHaveBeenCalled();

expect(spy).toHaveBeenCalledTimes(number);

expect(spy).toHaveBeenCalledWith(...arguments);

You can see concrete examples of how these matchers are used by looking at the Jasmine docs here: <http://jasmine.github.io/edge/introduction.html#section-Included_Matchers>

**Setup and Teardown:**

Sometimes in order to test a feature we need to perform some setup, perhaps it’s creating some test objects. Also we may need to perform some cleanup activities after we have finished testing, perhaps we need to delete some files from the hard drive.

These activities are called *setup* and *teardown* (for cleaning up) and Jasmine has a few functions we can use to make this easier:

**beforeAll**

This function is called **once**, *before* all the specs in a test suite (describe function) are run.

**afterAll**

This function is called **once** *after* all the specs in a test suite are finished.

**beforeEach**

This function is called *before* **each** test specification (it function) is run.

**afterEach**

This function is called *after* **each** test specification is run.

**Angular Testing Library:**

You want to write maintainable tests for your Angular components. As a part of this goal, you want your tests to avoid including implementation details of your components and rather focus on making your tests give you the confidence for which they are intended. As part of this, you want your testbase to be maintainable in the long run so refactors of your components (changes to implementation but not functionality) don't break your tests and slow you and your team down.

The Angular Testing Library is a very lightweight solution for testing Angular components.

So rather than dealing with instances of rendered Angular components, your tests will work with actual DOM nodes. The utilities this library provides facilitate querying the DOM in the same way the user would. Finding form elements by their label text (just like a user would), finding links and buttons from their text (like a user would).

This library encourages your applications to be more accessible and allows you to get your tests closer to using your components the way a user will, which allows your tests to give you more confidence that your application will work when a real user uses it.

**TestBed Class:**

Angular comes with a testing library that includes the wrappers for Jasmine’s describe(), it(), and xit() functions and also adds such functions as beforeEach(), async(), fakeAsync(), and others.

Because you don’t configure and bootstrap the application during test runs, Angular offers a TestBed helper class that allows you to declare modules, components, providers, and so on. TestBed includes such functions as configureTestingModule(), createComponent(), inject(), and others.

For example, the syntax for configuring a testing module looks similar to configuring @NgModule:

beforeEach(() => {

TestBed.configureTestingModule({

imports: [ ReactiveFormsModule, RouterTestingModule,

RouterTestingModule.withRoutes(routes)],

declarations: [AppComponent, HomeComponent, WeatherComponent],

providers: [{provide: WeatherService, useValue: {} }

]

})

});

The TestBed is the first and largest of the Angular testing utilities. It creates an Angular testing module — a @NgModule class — that you configure with the configureTestingModule method to produce the module environment for the class you want to test.

Configures and initializes environment for unit testing and provides methods for creating components and services in unit tests.

[TestBed](https://angular.io/api/core/testing/TestBed) is the primary api for writing unit tests for Angular applications and libraries.

TestBed is the main utility available for Angular-specific testing. You’ll use TestBed.configureTestingModule in your test suite’s beforeEach block and give it an object with similar values as a regular NgModule for declarations, providers, and imports. You can then chain a call to compileComponents to tell Angular to compile the declared components.

You can create a component fixture with TestBed.createComponent. Fixtures have access to a debugElement, which will give you access to the internals of the component fixture.

Change detection isn’t done automatically, so you’ll call detectChanges on a fixture to tell Angular to run change detection.

Wrapping the callback function of a test or the first argument of beforeEach with async allows Angular to perform asynchronous compilation and wait until the content inside of the async block to be ready before continuing.

### **createComponent()**

After configuring [TestBed](https://angular.io/api/core/testing/TestBed), you call its createComponent() method.

TestBed.createComponent() creates an instance of the Component, adds a corresponding element to the test-runner DOM, and returns a [ComponentFixture](https://angular.io/guide/testing-components-basics" \l "component-fixture).

### **ComponentFixture**

The [ComponentFixture](https://angular.io/api/core/testing/ComponentFixture) is a test harness for interacting with the created component and its corresponding element.

Access the component instance through the fixture and confirm it exists with a Jasmine expectation:

const component = fixture.componentInstance;

expect(component).toBeDefined();

### **beforeEach()**

You will add more tests as this component evolves. Rather than duplicate the [TestBed](https://angular.io/api/core/testing/TestBed) configuration for each test, you refactor to pull the setup into a Jasmine beforeEach() and some supporting variables:

describe('BannerComponent (with beforeEach)', () => {

let component: BannerComponent;

let fixture: ComponentFixture<BannerComponent>;

beforeEach(() => {

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

fixture = TestBed.createComponent(BannerComponent);

component = fixture.componentInstance;

});

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

expect(component).toBeDefined();

});

});

Now add a test that gets the component's element from fixture.nativeElement and looks for the expected text.

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

const bannerElement: HTMLElement = fixture.nativeElement;

expect(bannerElement.textContent).toContain('banner works!');

});

### **nativeElement**

The value of [ComponentFixture.nativeElement](https://angular.io/api/core/testing/ComponentFixture" \l "nativeElement) has the any type. Later you'll encounter the [DebugElement.nativeElement](https://angular.io/api/core/DebugElement" \l "nativeElement) and it too has the any type.

Angular can't know at compile time what kind of HTML element the nativeElement is or if it even is an HTML element. The application might be running on a non-browser platform, such as the server or a [Web Worker](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API), where the element might have a diminished API or not exist at all.

The tests in this guide are designed to run in a browser so a nativeElement value will always be an HTMLElement or one of its derived classes.

Knowing that it is an HTMLElement of some sort, use the standard HTML querySelector to dive deeper into the element tree.

Here's another test that calls HTMLElement.querySelector to get the paragraph element and look for the banner text:

it('should have <p> with "banner works!"', () => {

const bannerElement: HTMLElement = fixture.nativeElement;

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

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

});

### **DebugElement**

The Angular fixture provides the component's element directly through the fixture.nativeElement.

const bannerElement: HTMLElement = fixture.nativeElement;

This is actually a convenience method, implemented as fixture.debugElement.nativeElement.

const bannerDe: DebugElement = fixture.debugElement;

const bannerEl: HTMLElement = bannerDe.nativeElement;

There's a good reason for this circuitous path to the element.

The properties of the nativeElement depend upon the runtime environment. You could be running these tests on a non-browser platform that doesn't have a DOM or whose DOM-emulation doesn't support the full HTMLElement API.

Angular relies on the [DebugElement](https://angular.io/api/core/DebugElement) abstraction to work safely across all supported platforms. Instead of creating an HTML element tree, Angular creates a [DebugElement](https://angular.io/api/core/DebugElement) tree that wraps the native elements for the runtime platform. The nativeElement property unwraps the [DebugElement](https://angular.io/api/core/DebugElement) and returns the platform-specific element object.

Because the sample tests for this guide are designed to run only in a browser, a nativeElement in these tests is always an HTMLElement whose familiar methods and properties you can explore within a test.

Here's the previous test, re-implemented with fixture.debugElement.nativeElement:

it('should find the <p> with fixture.debugElement.nativeElement)', () => {

const bannerDe: DebugElement = fixture.debugElement;

const bannerEl: HTMLElement = bannerDe.nativeElement;

const p = bannerEl.querySelector('p')!;

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

});

The [DebugElement](https://angular.io/api/core/DebugElement) has other methods and properties that are useful in tests, as you'll see elsewhere in this guide.

You import the [DebugElement](https://angular.io/api/core/DebugElement) symbol from the Angular core library.

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

### **By.css()**

Although the tests in this guide all run in the browser, some applications might run on a different platform at least some of the time.

For example, the component might render first on the server as part of a strategy to make the application launch faster on poorly connected devices. The server-side renderer might not support the full HTML element API. If it doesn't support querySelector, the previous test could fail.

The [DebugElement](https://angular.io/api/core/DebugElement) offers query methods that work for all supported platforms. These query methods take a predicate function that returns true when a node in the [DebugElement](https://angular.io/api/core/DebugElement) tree matches the selection criteria.

You create a predicate with the help of a [By](https://angular.io/api/platform-browser/By) class imported from a library for the runtime platform. Here's the [By](https://angular.io/api/platform-browser/By) import for the browser platform:

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

The following example re-implements the previous test with [DebugElement.query()](https://angular.io/api/core/DebugElement" \l "query) and the browser's By.css method.

it('should find the <p> with fixture.debugElement.query(By.css)', () => {

const bannerDe: DebugElement = fixture.debugElement;

const paragraphDe = bannerDe.query(By.css('p'));

const p: HTMLElement = paragraphDe.nativeElement;

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

});

Some noteworthy observations:

* The [By.css()](https://angular.io/api/platform-browser/By#css) static method selects [DebugElement](https://angular.io/api/core/DebugElement) nodes with a [standard CSS selector](https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Getting_started/Selectors).
* The query returns a [DebugElement](https://angular.io/api/core/DebugElement) for the paragraph.
* You must unwrap that result to get the paragraph element.

Deep Integration Testing:

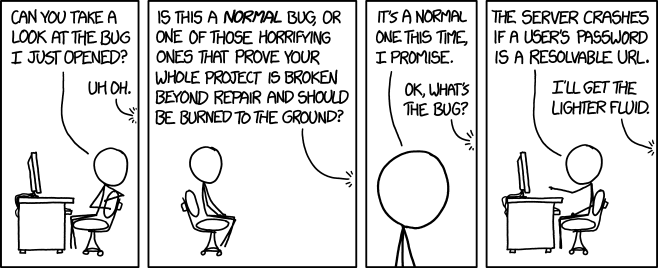
# Towards Better Testing In Angular. Part 1 — Mocking Child Components

## Tips and best practices on how to build better, more robust Angular applications

I remember back to my first job out of university, as a wide-eyed Mechatronics graduate. I managed to score a job as a junior developer, despite having only completed 2 software subjects during my degree.

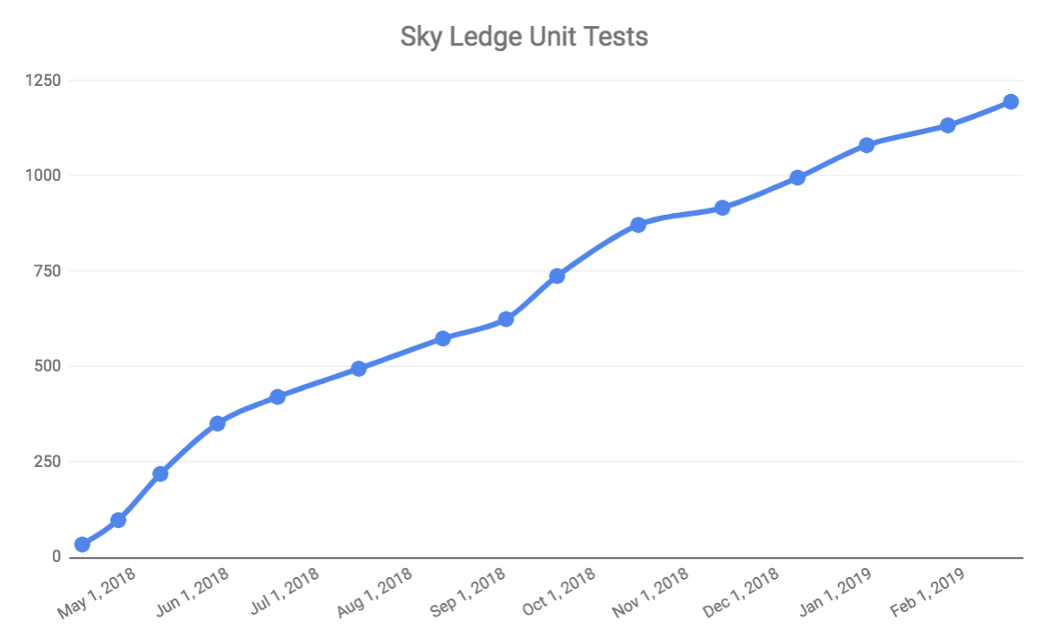
I was hired into a medium sized company to take over development and maintenance of a mature, complex piece of software. The software was used to capture video based on triggers from other systems. It was written in C++ and used ffmpeg as its underlying technology. We had a decent sized QA team, and at least a couple of those spent a bit of time testing this software.

However, there were precisely zero unit tests and non-existent CI. The software was plagued with bugs. Half my time was spent fixing strange and hard-to-reproduce bugs (which I’ve come to realise was an exercise in futility, as the bugs were race conditions caused by a broken threading implementation).



I’ve come a long way since those days. Over the next few years, I was fortunate enough to work with people who were strong advocate of good testing and CI processes, and I’ve internalised those attitudes. No longer do I believe that fixing bugs has to take up a large chunk of a developer’s time.

In the past year at [Sky Ledge](https://www.skyledge.com/), we’ve been actively developing our Angular web app. In that time, we’ve written some 1200 front-end unit tests.



Number of unit tests

In the process of developing the application, we’ve come across a few approaches that help us write more effective unit tests in Angular, specifically to ensure:

1. **More isolated testing**. We want to test the services and components in question, not their dependencies.
2. **Strongly typed template checking.**If a child component definition changes, we want to ensure that dependent unit tests are breaking.
3. **Less coupling between components and services**. Dependency Injection makes it easy to inject services, but if those services are being used all over the shop, refactoring becomes difficult.
4. **Prefer dumb components over smart components.**[This article](https://medium.com/@jtomaszewski/how-to-write-good-composable-and-pure-components-in-angular-2-1756945c0f5b) by

[Jack Tomaszewski](https://medium.com/u/5fcb6183b26?source=post_page-----b51e1fd571da-----------------------------------)

 explains it beautifully. Dumb components (i.e. those that rely on inputs and outputs) are much easier to test than components dependent on multiple services.

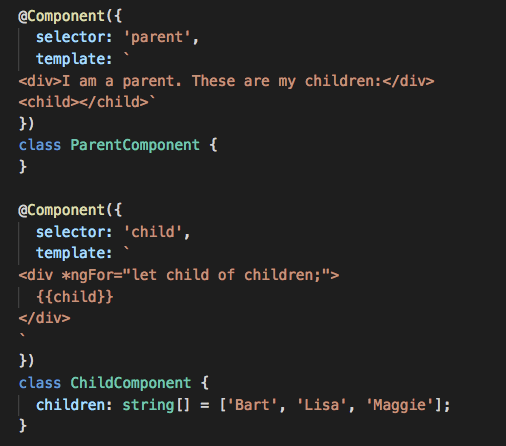
What outcomes should you expect from these articles? I’m hoping it gives you really practical advice on how to build better, more robust Angular applications, to allow you to easily develop and refactor with confidence.

# Mocking Child Components

We’ll consider 4 different approaches to testing components with child components. These are, in order of least recommended to most recommended:

1. Add the child component to declarations array
2. Use NO\_ERRORS\_SCHEMA to ignore the child component
3. Manually mock/stub the child component
4. Use ngMocks to automatically mock the child component

Let’s start with a very simple example. We have an aptly named component, ParentComponent , which includes another component, ChildComponent , in its template:



All ParentComponent is doing is rendering ChildComponent , which in turn renders the hard-coded list of children. Let’s write some code to test this component:



When we run the above test, it fails with the following error:

Failed: Template parse errors:   
'child' is not a known element:  
1. If 'child' is an Angular component, then verify that it is part of this module.   
2. To allow any element add 'NO\_ERRORS\_SCHEMA' to the '@NgModule.schemas' of this component. (" <div>I am a parent. These are my children:</div>

This is Angular’s way of telling us that it doesn’t know about ChildComponent . What are our options here?

# Solution 1. Add ChildComponent to declarations array

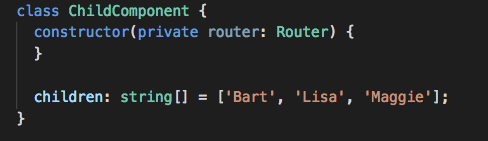
We can always add ChildComponent to the list of declared components:



Adding ChildComponent fixes the test, at the expense of test isolation

Sure enough, if we run the tests again, they succeed. Huzzah!

…not so fast. By doing this, we’ve broken Rule 1: Tests shalt be isolated. By declaring ChildComponent , our tests are now dependent on it to succeed. What happens if we add a dependency to ChildComponent?



Let’s add Router as a dependency to ChildComponent

Given that we haven’t imported the RouterTestingModule into our unit tests, we’re not able to resolve Router , and the test is broken once more:

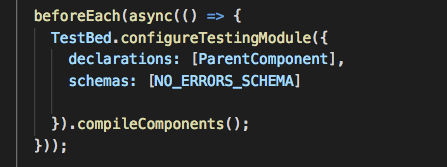
Error: StaticInjectorError(DynamicTestModule)[ChildComponent -> Router]:   
StaticInjectorError(Platform: core)[ChildComponent -> Router]:  
NullInjectorError: No provider for Router!

While we could always import RouterTestingModule , this is clearly an unideal solution. Changes to the internal logic of ChildComponent shouldn’t require us to modify ParentComponent unit tests. It’s annoying enough with 2 components — imagine how much worse it’ll be once we have 20, or 200.

# Solution 2. NO\_ERRORS\_SCHEMA

If you did a Google search for the above error, you’ll probably come across quite a few Stack Overflow posts that recommend using NO\_ERRORS\_SCHEMA. It’s also recommended as an approach in the official [Angular Testing Guide](https://angular.io/guide/testing).

NO\_ERRORS\_SCHEMA tells the compiler to ignore any elements or attributes it isn’t familiar with. Let’s modify our example to use NO\_ERRORS\_SCHEMA :



Adding NO\_ERRORS\_SCHEMA instructs the compiler to ignore ChildComponent

After doing this, our test is working again 🙂. While this is a valid and widely used solution, it’s also one that leaves us vulnerable, as it violates Rule 2: Thou shalt alert us to broken templates. What happens if we misspell the <child> tag?



<child> intentionally misspelled as <chidl>

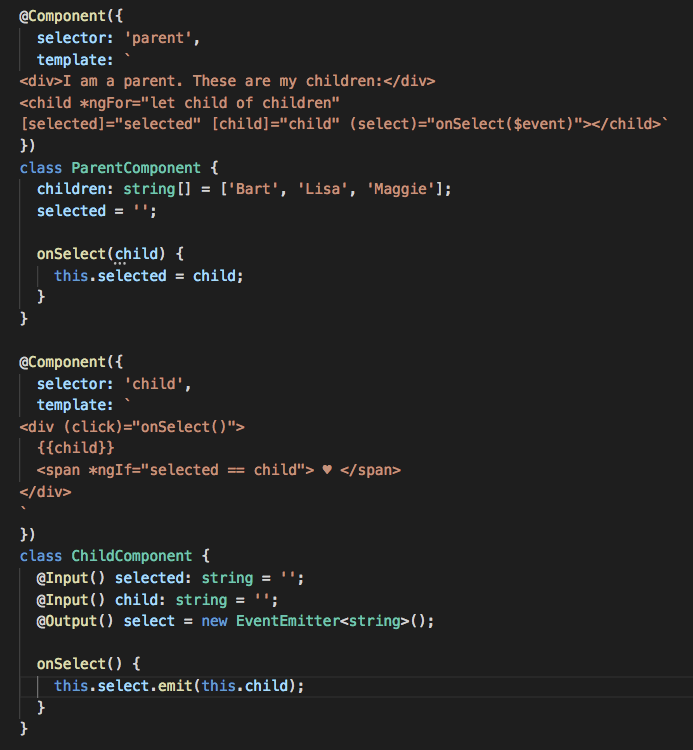
…the test still succeeds. This isn’t what we want. Even if we spelled <child> correctly, the tests will also succeed if we incorrectly spelled inputs: <child [childs]="children"></child> .

This is actually the approach we initially took when starting out with Sky Ledge. However, as the project grew in size and we began refactoring code, we quickly ran into situations where we didn’t realise there was something wrong until we actually ran the app. In one case, a broken component in an infrequently used part of the app remained broken for a couple of weeks until being discovered.

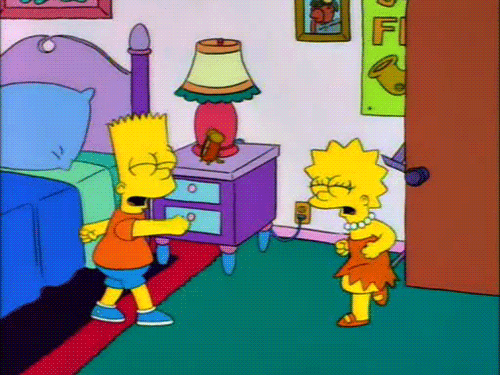
Beyond that, this approach also makes it very difficult to test @Input and @Output on child components. Let’s consider another approach.

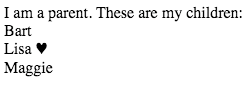
# Solution 3. Manually Mocking Components

We’ll extend our example to add inputs and outputs to ChildComponent:



Instead of ChildComponent having hardcoded names of children, we’ve made the child name an input. We’ve also added selected as an input: if child equals selected , we show a ❤️ next to that child (what better way to promote healthy sibling rivalry than choosing a favourite child?).





Output of the above code

So, how do we test ParentComponent now that ChildComponent is doing a bit more work? Another approach, also recommended in the Angular Testing Guide, is to manually mock (or stub) components. We can create a dummy ChildComponent and use that in place of the real ChildComponent . We just need to ensure that we give it the same selector:



Creating a stub component to use in place of the real component when testing

Now, in our fixture setup, we declare the stub component in place of the real one:



This now isolates testing of the ParentComponent from the internal workings of ChildComponent . For our purposes, we want to treat ChildComponent as a black box — we want to test that we’ve passed it the right inputs and that we handle any outputs it emits correctly. What it does with the inputs, and under what circumstances it emits outputs, is irrelevant to ParentComponent (testing of that logic should happen within the ChildComponent tests).

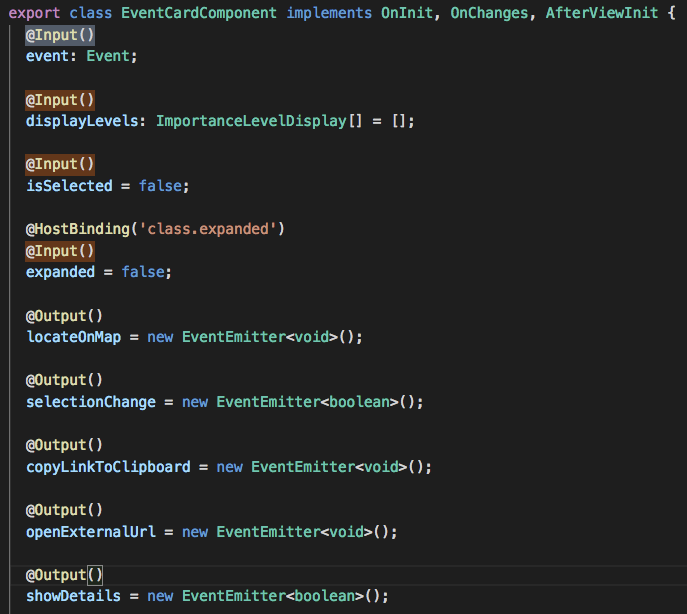
So, let’s write tests to ensure the inputs and outputs are behaving as expected:



Stubbing ChildComponent makes it easy to write detailed tests

Great. We’ve quickly and easily tested the functionality of ParentComponent and its interactions with ChildComponent . So are there any shortcomings with this approach?

The first shortcoming is that its quite verbose. Writing stub components for each component we want to mock can get tiresome very quickly, particularly if the components have lots of inputs and outputs. Here’s an extreme example from Sky Ledge (there’s so many inputs/outputs due to Rule 4 — Thou shalt prefer dumb components over smart components):



An example of a component with many inputs and outputs

Not only is it a pain in the proverbial to write a stub component for something like the above, it’s also prone to code rot (Shortcoming the Second). Let’s go back to ChildComponent. What if we wanted to rename child to something more descriptive, like childName ?



renaming child to childName

…our tests still succeed. We won’t know it’s actually failed until we run the application ([or build it with Ahead-of-Time compilation enabled](https://angular.io/guide/aot-compiler#why-compile-with-aot)). The stub component approach requires us to remember to change the stub each time we change the real component. As you can expect, this is prone to human error and the problem exacerbates as your application grows.

Issues like these can make refactoring applications and maintaining unit tests a chore. Can we do better?

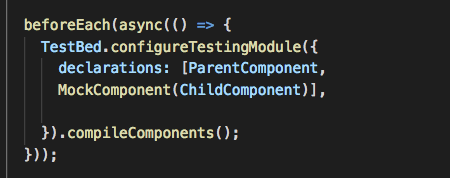
# Solution 4. Mocking Components using NgMock

As it turns out, we can do better. There’s an [excellent library called ngMocks](https://github.com/ike18t/ng-mocks)which makes it trivial to create mock components (as well as directives and pipes). Instead of manually creating components, ngMocks automagically creates type-safe mocks for us, alleviating both shortcomings of Solution 3.

To get started, you’ll first need to install ngMocks:

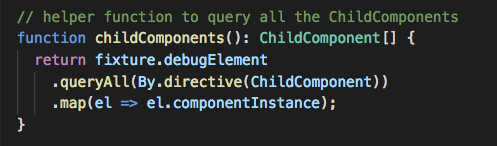
npm install ng-mocks --save-dev

We no longer require ChildComponentStub so feel free to delete it. We’ll rewrite our tests to use ngMocks instead. First, let’s change the test config to automatically mock ChildComponent using ngMock’s MockComponent function:



Mock ChildComponent using MockComponent

We’ll also need to change our helper function to return ChildComponent instead of ChildComponentStub :

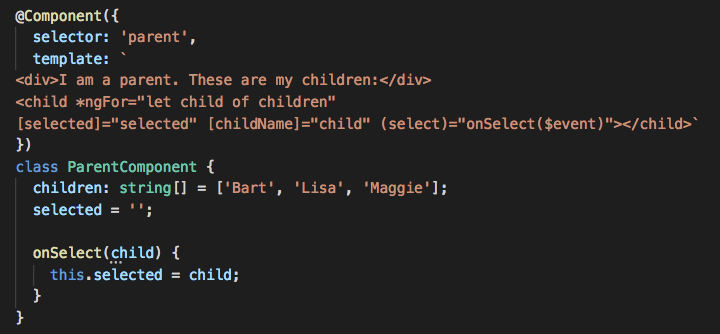


We’ll now get a bunch of compiler errors in our tests complaining that Property ‘child’ does not exist on type ‘ChildComponent’. So rename child to childName and run tests again.

And now…our tests are failing 🎊!

Failed: Template parse errors:   
Can't bind to 'child' since it isn't a known property of 'child'.   
(" a parent. These are my children:</div>   
<child \*ngFor="let child of children" [selected]="selected"   
[ERROR ->][child]="child" (select)="onSelect($event)"></child>")

As MockComponent produces a strongly typed mock, we’re getting template errors that child doesn’t exist as an input on ChildComponent , exactly what we want. Let’s update ParentComponent template to use childName instead:



And just like that, all our tests are passing again.

NO\_ERRORS\_SCHEMA:

This schema allows you to ignore the errors related to any unknown elements or properties in a template. The usage of this schema is generally discouraged because it prevents useful validation and may hide real errors in your template.

**Mocking Dependencies:**

Let’s imagine we have a LoginComponent which works with the AuthService we tested in the previous lecture, like so:

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

import {AuthService} from "./auth.service";

@Component({

selector: 'app-login',

template: `<a [hidden]="needsLogin()">Login</a>`

})

export class LoginComponent {

constructor(private auth: AuthService) {

}

needsLogin() {

return !this.auth.isAuthenticated();

}

}

We inject the AuthService into the LoginComponent and the component shows a *Login* button if the AuthService says the user isn’t *authenticated*.

The AuthService is the same as the previous lecture:

export class AuthService {

isAuthenticated(): boolean {

return !!localStorage.getItem('token');

}

}

We could test the LoginComponent by using a real instance of AuthService but if you remember to trick AuthService into returning true for the isAuthenticated function we needed to setup some data via localStorage.

import {LoginComponent} from './login.component';

import {AuthService} from "./auth.service";

describe('Component: Login', () => {

let component: LoginComponent;

let service: AuthService;

beforeEach(() => { (1)

service = new AuthService();

component = new LoginComponent(service);

});

afterEach(() => { (2)

localStorage.removeItem('token');

service = null;

component = null;

});

it('needsLogin returns true when the user has not been authenticated', () => {

expect(component.needsLogin()).toBeTruthy();

});

it('needsLogin returns false when the user has been authenticated', () => {

localStorage.setItem('token', '12345'); (3)

expect(component.needsLogin()).toBeFalsy();

});

});

We create an instance of AuthService and inject it into out LoginComponent when we create it.

We clean up data and localStorage after each test spec has been run.

We setup some data in localStorage in order to get the behaviour we want from AuthService.

So in order to test LoginComponent we would need to know the *inner workings* of AuthService.

That’s not very *isolated* but also not *too* much to ask for in this scenario. However imagine if LoginComponent required a number of *other* dependencies in order to run, we would need to know the inner workings of a number of other classes just to test our LoginComponent.

This results in *Tight Coupling* and our tests being very *Brittle*, i.e. likely to break easily. For example if the AuthService changed *how* it stored the token, from localStorage to cookies then the LoginComponent test would break since *it* would still be setting the token via localStorage.

This is why we need to test classes in *isolation*, we just want to worry about LoginComponent and not about the myriad of other things LoginComponent depends on.

We achieve this by *Mocking* our dependencies. Mocking is the act of creating something that looks like the dependency but is something *we* control in our test. There are a few methods we can use to create mocks.

**Mocking with Fake Classes:**

We can create a fake AuthService called MockedAuthService which just returns whatever we want for our test.

We can even remove the AuthService import if we want, there really is no dependency on anything else. The LoginComponent is tested in isolation:

import {LoginComponent} from './login.component';

class MockAuthService { (1)

authenticated = false;

isAuthenticated() {

return this.authenticated;

}

}

describe('Component: Login', () => {

let component: LoginComponent;

let service: MockAuthService;

beforeEach(() => { (2)

service = new MockAuthService();

component = new LoginComponent(service);

});

afterEach(() => {

service = null;

component = null;

});

it('needsLogin returns true when the user has not been authenticated', () => {

service.authenticated = false; (3)

expect(component.needsLogin()).toBeTruthy();

});

it('needsLogin returns false when the user has been authenticated', () => {

service.authenticated = true; (3)

expect(component.needsLogin()).toBeFalsy();

});

});

We create a class called MockAuthService which has the same isAuthenticated function as the real AuthService class. The one difference is that we can control what isAuthenticated returns by setting the value of the authenticated property.

We inject into our LoginComponent an instance of the MockAuthService instead of the real AuthService.

In our tests we trigger the behaviour we want from the service by setting the authenticated property.

By using a fake MockAuthService we:

* Don’t depend on the real AuthService, in fact we don’t even need to import it into our specs.
* Make our code less brittle, if the inner workings of the real AuthService ever changes our tests will still be valid and still work.

**Mocking by Overriding Functions:**

Sometimes creating a complete fake copy of a real class can be complicated, time consuming and unnecessary.

We can instead simply extend the class and override one or more specific function in order to get them to return the test responses we need, like so:

class MockAuthService extends AuthService {

authenticated = false;

override isAuthenticated() {

return this.authenticated;

}

}

In the above class MockAuthService extends AuthService. It would have access to all the other functions and properties that exist on AuthService but only override the isAuthenticated function so we can easily control it’s behaviour and isolate our LoginComponent test.

**Note**

The rest of the test suite using mocking via overriding functions is the same as the previous version with fake classes.

**Mocking with Spies:**

A Spy is a feature of Jasmine which lets you take an existing class, function, or object and mock it in such a way that you can control what gets returned from function calls.

Let’s re-write our test to use a Spy on a real instance of AuthService instead, like so:

import {LoginComponent} from './login.component';

import {AuthService} from "./auth.service";

describe('Component: Login', () => {

let component: LoginComponent;

let service: AuthService;

let spy: any;

beforeEach(() => { (1)

service = new AuthService();

component = new LoginComponent(service);

});

afterEach(() => { (2)

service = null;

component = null;

});

it('needsLogin returns true when the user has not been authenticated', () => {

spy = spyOn(service, 'isAuthenticated').and.returnValue(false); (3)

expect(component.needsLogin()).toBeTruthy();

expect(service.isAuthenticated).toHaveBeenCalled(); (4)

});

it('needsLogin returns false when the user has been authenticated', () => {

spy = spyOn(service, 'isAuthenticated').and.returnValue(true);

expect(component.needsLogin()).toBeFalsy();

expect(service.isAuthenticated).toHaveBeenCalled();

});

});

We create a real instance of AuthService and inject it into the LoginComponent.

In our teardown function there is no need to delete the token from localStorage.

We create a *spy* on our service so that if the isAuthenticated function is called it returns false.

We can even check to see if the isAuthenticated function was called.

By using the spy feature of Jasmine we can make any function return anything we want:

spyOn(service, 'isAuthenticated').and.returnValue(false);

In our example above we make the isAuthenticated function return false or true in each test spec according to our needs.

**Summary:**

Testing with real instances of dependencies causes our test code to know about the inner workings of other classes resulting in tight coupling and brittle code.

The goal is to test pieces of code in isolation without needing to know about the inner workings of their dependencies.

We do this by creating Mocks; we can create Mocks using fake classes, extending existing classes or by using real instances of classes but taking control of them with Spies.

**Listing:**

**Login.Component.ts:**

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

import {AuthService} from "./auth.service";

@Component({

selector: 'app-login',

template: `<a [hidden]="needsLogin()">Login</a>`

})

export class LoginComponent {

constructor(private auth: AuthService) {

}

needsLogin() {

return !this.auth.isAuthenticated();

}

}

**Login.Component.spec.ts:**

/\* tslint:disable:no-unused-variable \*/

import { LoginComponent } from './login.component';

import { AuthService } from "./auth.service";

describe('Component: Login', () => {

let component: LoginComponent;

let service: AuthService;

let spy: any;

beforeEach(() => {

service = new AuthService();

component = new LoginComponent(service);

});

afterEach(() => {

service = null;

component = null;

});

it('needsLogin returns true when the user has not been authenticated', () => {

spy = spyOn(service, 'isAuthenticated').and.returnValue(false);

expect(component.needsLogin()).toBeTruthy();

expect(service.isAuthenticated).toHaveBeenCalled();

});

it('needsLogin returns false when the user has been authenticated', () => {

spy = spyOn(service, 'isAuthenticated').and.returnValue(true);

expect(component.needsLogin()).toBeFalsy();

expect(service.isAuthenticated).toHaveBeenCalled();

});

});

**Mocking Http Requests/Response:**

Angular’s [HttpClient](https://www.digitalocean.com/community/tutorials/angular-httpclient-intro) has a testing module, [HttpClientTestingModule](https://angular.io/api/common/http/testing/HttpClientTestingModule), that makes it possible for you to unit test HTTP requests.

The HttpClientTestingModulemakes it easier to mock requests using the HttpTestingControllerservice.

You can use the HttpTestingController to mock requests and the flush method to provide dummy values as responses.

**Step 1 — Setting Up the Project**

For this post, we’ll be working with a service that gets data from an endpoint and a component that calls that service to populate a list of users in the component’s OnInit hook.

You can use @angular/cli to create a new project:

1. ng new angular-httpclienttest-example

Then, navigate to the newly created project directory:

1. cd angular-httpclienttest-example

Create a data.service.ts:

1. ng generate service data

And have it communicate with JSON Placeholder:

src/app/data.service.ts

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

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

@Injectable({ ... })

export class DataService {

url = 'https://jsonplaceholder.typicode.com/users';

constructor(private http: HttpClient) { }

getData() {

const req = new HttpRequest('GET', this.url, {

reportProgress: true

});

return this.http.request(req);

}

}

Then, modify the app.component.ts file:

src/app.component.ts

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

import { HttpEvent, HttpEventType } from '@angular/common/http';

import { DataService } from './data.service';

@Component({ ... })

export class AppComponent implements OnInit {

users: any;

constructor(private dataService: DataService) {}

ngOnInit() {

this.populateUsers();

}

private populateUsers() {

this.dataService.getData().subscribe((event: HttpEvent<any>) => {

switch (event.type) {

case HttpEventType.Sent:

console.log('Request sent!');

break;

case HttpEventType.ResponseHeader:

console.log('Response header received!');

break;

case HttpEventType.DownloadProgress:

const kbLoaded = Math.round(event.loaded / 1024);

console.log(`Download in progress! ${kbLoaded}Kb loaded`);

break;

case HttpEventType.Response:

console.log('Done!', event.body);

this.users = event.body;

}

});

}

}

And add the HttpClientmodule to app.module.ts:

src/app.module.ts

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

import { BrowserModule } from '@angular/platform-browser';

import { HttpClientModule } from '@angular/common/http';

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

HttpClientModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

At this point, you will have an Angular project with a service and client.

**Step 2 — Adding Tests**

Now we’ll setup a spec file for our data service and include the necessary utilities to test out the HttpClient requests. On top of HttpClientTestingModule, we’ll also need HttpTestingController, which makes it easy to mock requests:

data.service.spec.ts

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

import { HttpEvent, HttpEventType } from '@angular/common/http';

import {

HttpClientTestingModule,

HttpTestingController

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

import { DataService } from './data.service';

describe('DataService', () => {

let service: DataService;

beforeEach(() => {

TestBed.configureTestingModule({}

imports: [HttpclientTestingModule],

providers: [DataService]

);

service = TestBed.inject(DataService);

});

});

We use the inject utility to inject the needed services into our test.

With this in place, we can add our test logic:

data.service.spec.ts

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

import { HttpEvent, HttpEventType } from '@angular/common/http';

import {

HttpClientTestingModule,

HttpTestingController

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

import { DataService } from './data.service';

describe('DataService', () => {

beforeEach(() => {

TestBed.configureTestingModule({

imports: [HttpClientTestingModule],

providers: [DataService]

});

});

it(

'should get users',

inject(

[HttpTestingController, DataService],

(httpMock: HttpTestingController, dataService: DataService) => {

const mockUsers = [

{ name: 'Alice', website: 'example.com' },

{ name: 'Bob', website: 'example.org' }

];

dataService.getData().subscribe((event: HttpEvent<any>) => {

switch (event.type) {

case HttpEventType.Response:

expect(event.body).toEqual(mockUsers);

}

});

const mockReq = httpMock.expectOne(dataService.url);

expect(mockReq.cancelled).toBeFalsy();

expect(mockReq.request.responseType).toEqual('json');

mockReq.flush(mockUsers);

httpMock.verify();

}

)

);

});

There’s quite a bit going on, so let’s break it down:

* First we define a couple of mock users that we’ll test against.
* We then call the getData method in the service that we’re testing and subscribe to returned observable.
* If the HttpEventType is of type Response, we assert for the response event to have a body equal to our mock users.
* We then make use of the HttpTestingController (injected in the test as httpMock) to assert that one request was made to the service’s url property. If no request is expected, the expectNone method can also be used.
* We can now make any number of assertions on the mock request. Here we assert that the request hasn’t been canceled and the response is of type json. Additionally, we could assert the request’s method (GET, POST, …)
* Next we call flush on the mock request and pass in our mock users. The flush method completes the request using the data passed to it.
* Finally, we call the verify method on our HttpTestingController instance to ensure that there are no outstanding requests to be made.

For the purposes of this tutorial, you can comment out app.component.spec.ts.

See the result of your testing by running the following command:

1. ng test

Open the test results in the browser:

Output

1 spec, 0 failures, randomized with seed 26321

DataService

should get users

It will display a successful test message.

**Code Coverage:**

The CLI can run unit tests and create code coverage reports. Code coverage reports show you any parts of your code base that might not be properly tested by your unit tests.

To generate a coverage report run the following command in the root of your project.

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

When the tests are complete, the command creates a new /coverage folder in the project. Open the index.html file to see a report with your source code and code coverage values.

If you want to create code-coverage reports every time you test, set the following option in the CLI configuration file, angular.json:

"test": {

"options": {

"codeCoverage": true

}

}

## Code coverage enforcement

The code coverage percentages let you estimate how much of your code is tested. If your team decides on a set minimum amount to be unit tested, enforce this minimum with the Angular CLI.

For example, suppose you want the code base to have a minimum of 80% code coverage. To enable this, open the [Karma](https://karma-runner.github.io/) test platform configuration file, karma.conf.js, and add the check property in the coverageReporter: key.

coverageReporter: {

dir: require('path').join(\_\_dirname, './coverage/<project-name>'),

subdir: '.',

reporters: [

{ type: 'html' },

{ type: 'text-summary' }

],

check: {

global: {

statements: 80,

branches: 80,

functions: 80,

lines: 80

}

}

}

The check property causes the tool to enforce a minimum of 80% code coverage when the unit tests are run in the project.

Find more information about the different coverage configuration options [here](https://github.com/karma-runner/karma-coverage/blob/master/docs/configuration.md)