Angular unit testing 101 (with examples)

Mustapha Aouas Aug 29 *Updated on Sep 12, 2019* · 8 min read

#angular #testing **#javascript** #webdev

The more we add features to our software the more it grows in complexity. And as it grows in complexity, more time is required to manually test it. In fact, as we add new features to our applications, the time required to manually test them increases exponentially!
To prevent that we could take advantage of automated testing as it is the best way to increase the effectiveness, efficiency, and coverage of our applications testing.

In this post, we'll be talking about Angular unit testing, using Karma and Jasmine. By the end of this post, you should feel comfortable writing specs to test your Angular components, directives, pipes, and services as well as learning techniques to test synchronous and asynchronous behaviors.

Before we start

First things first, let's talk about some of the basics and terminologies of testing. It'll help us establish a mental model of how things work so we will be able to understand the later parts a little bit better.

Terminologies

Automated testing

It's the practice of writing code to test our code, and then run those tests. There are 3 types of tests: unit tests, integration tests, and end-to-end (e2e) tests.

Unit test

A unit test or UT is the procedure to check the proper functioning of a specific part of a software or a portion of a program.

Karma

Karma is a test runner. It will automatically create a browser instance, run our tests, then gives us the results. The big advantage is that it allows us to test our code in different browsers without any manual change in our part.

> The pattern that Karma uses to identify test files is `<filename>.spec.ts` . This is a general convention that other languages use. If for some reason you want to change it, you can do so in the `test.ts` file.

I

Jasmine

Jasmine is a popular Javascript testing framework. It comes with test doubles by using spies (we'll define what is a spy later), and assertions built into it out of the box.

Jasmine provides a lot of useful functions to write tests. The three main APIs are:

1. `Describe()` : It's a suite of tests
2. `it()` : Declaration of a single test
3. `expect()` : Expect something to be true for example

Mock

Mock objects are *fake* (simulated) objects that mimic the behavior of real objects in controlled ways.

Fixture

A fixture is a wrapper around an instance of a component. With a fixture, we can have access to a component instance as well as its template.

Spy

Spies are useful for verifying the behavior of our components depending on outside inputs, without having to define those outside inputs. They're most useful when testing components that have services as a dependency.

Basics

The Angular CLI downloads and installs everything you need to test an Angular application with the Jasmine test framework. All you need to do to start testing is to run the following command:

```
ng test
```

This command builds the app in watch mode and launches Karma.

Angular automated testing

Skeleton of a test

Using the three Jasmine APIs mentioned above, a skeleton of a unit test should look like this:

```
describe('TestSuitName', () => {
  // suite of tests here

  it('should do some stuff', () => {
    // this is the body of the test
  });
});
```

When testing, there's a pattern that became almost a standard across the developer community, called AAA (Arrange-Act-Assert). AAA suggests that you should divide your test method

into three sections: arrange, act and assert. Each one of them only responsible for the part in which they are named after.

So the arrange section you only have code required to set up that specific test. Here objects would be created, mocks setup (if you are using one) and potentially expectations would be set. Then there is the Act, which should be the invocation of the method being tested. And on Assert you would simply check whether the expectations were met.

Following this pattern does make the code quite well structured and easy to understand. In general lines, it would look like this:

```
it('should truncate a string if its too long (>20)', () => {
  // Arrange
  const pipe = new TroncaturePipe();

  // Act
  const ret = pipe.transform('123456789012345678901234567890');

  // Assert
  expect(ret.length).toBeLessThanOrEqual(20);
});
```

Configuration & instantiation

In order to access methods of the component we want to test, we first need to instantiate it.
Jasmine comes with an API called `beforeAll()` which is called once before all the tests.
The thing is if we instantiate our component inside this function our tests won't be isolated because the component

properties could be changed by each test, and therefore, a first test could influence the behavior of a second test.
To solve that problem, Jasmine has another API called `beforeEach()`, which is very useful as it lets our tests to be run from the same starting point and thus to be run in isolation.
So, using this API, our test should look something like this:

```
describe('componentName', () => {
  // suite of tests here

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [myComponent],
    });

    fixture = TestBed.createComponent(myComponent);
    component = fixture.componentInstance;
  });

  it('should do some stuff', () => {
    // this is the body of the test

    // test stuff here
    expect(myComponent.methodOfMyComponent()).not.toBe(true);
  });
});
```

All of a sudden we have a lot of new unknown APIs. Let's have a closer look at what we have here.
Angular comes with an API for testing `testBed` that has a method `configureTestingModule()` for configuring a test module where we can import other Angular modules, components, pipes, directives, or services.
Once our testing module configured we can then instantiate for example the component we want to test.

Components

An Angular component combines an HTML template and a TypeScript class.
So, to test a component we need to create the component's host element in the browser DOM.
To do that we use a `TestBed` method called `createComponent()`.
This method will create a fixture containing our component instance and its HTML reference. With this fixture, we can access the raw component by calling its property `componentInstance` and its HTML reference by using `nativeElement`.

With that, an Angular component test should look like this:

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

  // * We use beforeEach so our tests are run in isolation
  beforeEach(() => {
    TestBed.configureTestingModule({
      // * here we configure our testing module with all the declarat
      // * imports, and providers necessary to this component
      imports: [CommonModule],
      providers: [],
      declarations: [HeaderComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(HeaderComponent);
    component = fixture.componentInstance; // The component instantia
    element = fixture.nativeElement; // The HTML reference
  });

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

```
    expect(component).toBeTruthy();
  });

  it('should create', () => {
    // * arrange
    const title = 'Hey there, i hope you are enjoying this article';
    const titleElement = element.querySelector('.header-title');
    // * act
    component.title = title;
    fixture.detectChanges();
    // * assert
    expect(titleElement.textContent).toContain(title);
  });
});
```

> After setting the title in our test, we need to call detectChanges() so the template is updated with the new title we just set (because binding happens when Angular performs change detection).

Pipes

Because a pipe is a class that has one method, transform, (that manipulates the input value into a transformed output value), it's easier to test without any Angular testing utilities.

Bellow an example of what a pipe test should look like:

```
describe('TroncaturePipe', () => {
  it('create an instance', () => {
    const pipe = new TroncaturePipe(); // * pipe instantiation
    expect(pipe).toBeTruthy();
  });

  it('truncate a string if its too long (>20)', () => {
```

```
    // * arrange
    const pipe = new TroncaturePipe();
    // * act
    const ret = pipe.transform('12345678912345678945666123');
    // * asser
    expect(ret.length).toBe(20);
  });
});
```

Directives

An attribute directive modifies the behavior of an element. So you could unit test it like a pipe where you only test its methods, or you could test it with a host component where you can check if it correctly changed its behavior.

Here is an example of testing a directive with a host component:

```
// * Host component:
@Component({
  template: `<div [appPadding]="2">Test</div>`,
})
class HostComponent {}
@NgModule({
  declarations: [HostComponent, PaddingDirective],
  exports: [HostComponent],
})
class HostModule {}

// * Test suite:
describe('PaddingDirective', () => {
  let component: HostComponent;
  let element: HTMLElement;
  let fixture: ComponentFixture<HostComponent>;

  beforeEach(() => {
```

```
    TestBed.configureTestingModule({
      imports: [CommonModule, HostModule], // * we import the host mo
    }).compileComponents();

    fixture = TestBed.createComponent(HostComponent);
    component = fixture.componentInstance;
    element = fixture.nativeElement;

    fixture.detectChanges(); // * so the directive gets appilied
  });

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

  it('should add padding', () => {
    // * arrange
    const el = element.querySelector('div');
    // * assert
    expect(el.style.padding).toBe('2rem'); // * we check if the direc
  });
});
```

Services

Like pipes, services are often easier to test. We could instantiate them with the `new` keyword. That's fine for basic services, but if your service has dependencies, it's better to use the `TestBed.configureTestingModule` API like this:

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

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [LocalService],
    });
```

```
    service = TestBed.get(LocalService); // * inject service instance
  });

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

  it('should set the local', () => {
    // * act
    service.setLocal('fr');
    // * assert
    expect(service.getLocal()).toBe('fr');
  });
});
```

> To get the service instance, we could inject it inside a test by calling `TestBed.get()` (with the service class as the argument).

Well, with that you should be ready to write tests for your Angular applications. That being said, to tackle some of the common difficulties you might encounter while writing Angular tests, I added small cheatsheets you can find in the next section :)

Cheatsheets

Dealing with HTTP requests

To avoid making HTTP requests at each test, one method consists of providing a *fake* service that mocks the real one (the one that communicates via HTTP requests).
Once the fake service is implemented we provide it to the `TestBed.configureTestingModule()` like this:

```
class FakeApiService {
  // Implement the methods you want to overload here
  getData() {
    return of({ items: [] }); // * mocks the return of the real metho
  }
}
//...
TestBed.configureTestingModule({
  imports: [],
  declarations: [myComponent],
  providers: [
    {
      provide: RealApiService,
      useClass: FakeApiService,
    },
  ],
});
//...
```

Dealing with the Angular router

To deal with the Router you could either add the `RouterTestingModule` in the imports of your testing module or you could mock it using the technique we saw in the test above.

Using spies

Spies are an easy way to check if a function was called or to provide a custom return value.
Here is an example of how to use them:

```
it('should do something', () => {
  // arrange
  const service = TestBed.get(dataService);
```

```
  const spyOnMethod = spyOn(service, 'saveData').and.callThrough();
  // act
  component.onSave();
  // assert
  expect(spyOnMethod).toHaveBeenCalled();
});
```

> You could read more about Spies in this great article.

Dealing with asynchronous code

Dealing with promises

```
it('should do something async', async () => {
  //  * arrange
  const ob = { id: 1 };
  component.selected = ob;
  //  * act
  const selected = await component.getSelectedAsync(); // get the pro
  //  * assert
  expect(selected.id).toBe(ob.id);
});
```

Dealing with observables

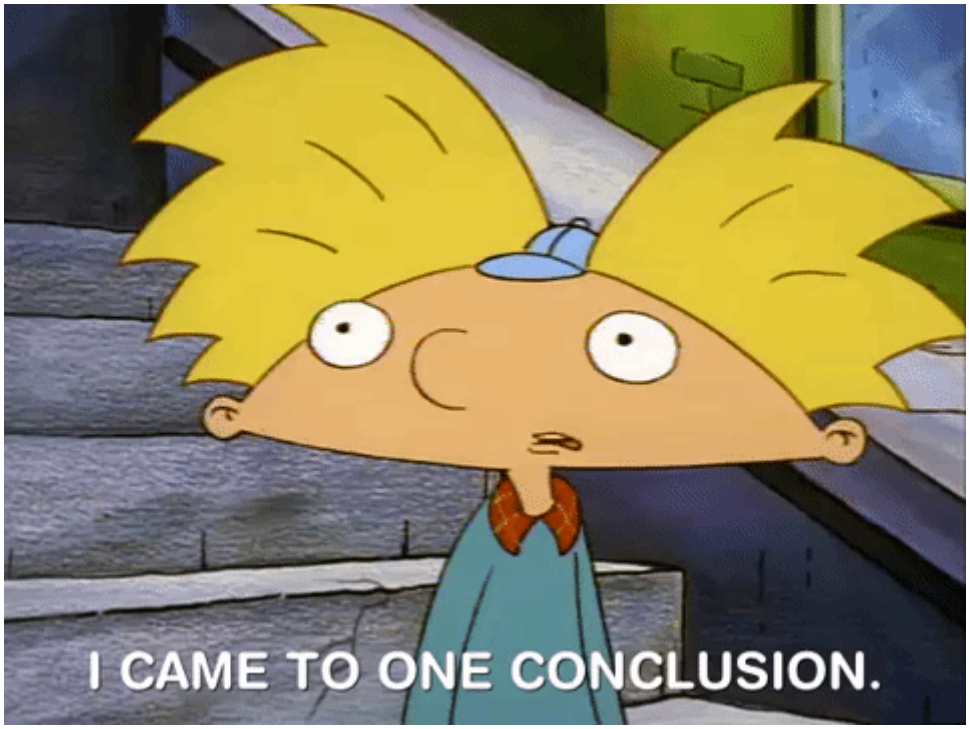
```
it('should do something async', (done) => {
  //  * arrange
  const ob = { id: 1 };
  component.selected = ob;
  //  * act
  const selected$ = component.getSelectedObs(); // get an Observable
  //  * assert
  selected$.subscribe(selected => {
```

```
    expect(selected.id).toBe(ob.id);
    done(); // let Jasmine know that you are done testing
  });
});
```

Dealing with timeouts

```
const TIMEOUT_DELAY = 250;
//...
it('should do something async', (done) => {
  //  * arrange
  const ob = { id: 1 };
  //  * act
  component.setSelectedAfterATimeout(ob);
  // * assert
  setTimeout(() => {
    expect(component.selected.id).toBe(ob.id);
    done(); // let Jasmine know that you are done testing
  }, TIMEOUT_DELAY);
});
```

Conclusion



So, in this article, we saw that the Angular CLI configures everything for us and we only have to run `ng test` to start our tests. Then we saw what is an automated test and how to write one using Jasmine and the Angular testing utilities (for components, pipes, directives, and services). Finally, we saw examples of some of the special cases you might encounter when writing tests.

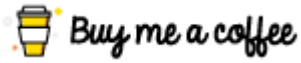
With all this we just scraped the surface of Angular testing, there's so much more to learn about it. That's why this post is the first of an "Angular testing" series. Follow me on twitter @theAngularGuy to get updated when the next posts might come out.

I hope you liked this article, tell me what are your thoughts about automated testing, is it helpful to you or is it more like a struggle?

In the meantime, Happy codding!

What to read next?

- All you need to know about Angular animations
- CSS Grid: illustrated introduction
- 7 Tips to boost your productivity as a web developer □



Mustapha Aouas + FOLLOW

Angular lead developer @Egis — I like sharing what I know, and learning what I don't □□□

@mustapha TheAngularGuy TheAngularGuy

Add to the discussion

PREVIEW SUBMIT

peach777

Nov 13

Great article! one question hope you could help me. All my components are written in app module. When i try to test one component(TestComponent), in the test.component.spec.file, i wrote below code. because i already declare the TestComponent in AppModule and used some third party components, i directly import it. When i run the test case, i didn't see 'fixture' getting printed in the console and the application throws error, TestBed.createComponent() failed and i don't know why...

```
beforeEach(async(() => {
TestBed.configureTestingModule({
declarations: [],
imports: [AppModule]
})
.compileComponents();
fixture = TestBed.createComponent(TestComponent);
console.log('fixture');
console.log(fixture);
}));
```

Error:
TypeError: Cannot read property 'get' of undefined
error properties: Object({ ngDebugContext: DebugContext_({ view: Object({ def: Object({ factory: Function, nodeFlags: 34127873, rootNodeFlags: 33554433, nodeMatchedQueries: 0, flags: 0, nodes: [Object({ nodeIndex: 0, parent: null, renderParent: null, bindingIndex: 0, outputIndex: 0, checkIndex: 0, flags: 33554433, childFlags: 573440, directChildFlags: 573440, childMatchedQueries: 0, matchedQueries: Object({ }), matchedQueryIds: 0, references: Object({ }), ngContentIndex: null, childCount: 1, bindings: [ ], bindingFlags: 0, outputs: [Object({ type: 0, target: 'document', eventName: 'click', propName: null })], element: Object({ ns: '', name: 'app-click-map-page', attrs: [ ], template: null, componentProvider: Object({ nodeIndex: 1, parent: , renderParent: , bindingIndex: 0, outputIndex: 1, checkIndex: 1, flags: 573440, childFlags: 0, directChildFlags: 0, childMatchedQueries: 0, matchedQueries: Object, matchedQueryIds: 0, references: Object, ...
at
at new SharedService (localhost:9876/_karma_webpack_/src...)

1 REPLY

Mustapha Aouas

Nov 13

Hi,

When you import the appModule, you are importing all the exported properties of the appModule.

Do you have your testComponent in the exports array of the appModule ?

1 REPLY

peach777

Nov 14

Yes I export it.. TestComponent extend another class and that class is using Injector to get service without using constructor.. The error happens at: AppModule.Injector.get(TestService) this line, said Cannot read property 'get' of undefined.

I create a question of this on stackoverflow, stackoverflow.com/questions/588577...

Maybe you could help give some advice. Thanks tons. :)

1 REPLY

Anower Jahan Shofol

Nov 11

Nice and clear explanations, Mustapha. I've just started to see Angular testing today, so the thorough explanation really cleared my idea. Thanks :D

2 REPLY

Mustapha Aouas

Nov 12

Thanks for the kind words! Glad i could help :)

1 REPLY

D Chenna Raidu

Aug 30

TheAngularGuy, now I have something to do this weekend. Thank you

2 REPLY

Mustapha Aouas

Aug 30

Glad to hear that, have fun ;)

1 REPLY

code of conduct - report abuse

Classic DEV Post from Mar 29

Code of Conduct generator!

Miloslav Voloskov

68 18

PivotTable.js in .Net Core Razor Pages - Tutorial (With Some More Javascript)

Zoltan Halasz - Dec 6

Blog with Pagination - Guide to Gatsby WordPress Starter Advanced with Previews, i18n and more

Henrik Wirth - Dec 6

How to destructure a class instance without breaking "this" in JavaScript

Shailen Naidoo - Dec 6

Friday Frontend: CSS Subgrids Are Here Edition

Kevin Ball - Dec 6
