## Testing Asynchronous Code

**Learning Objectives**

* Understand the issues faced when testing async code in Jasmine.
* Know how to use the Jasmine done function to handle async code.
* Know how to use the alternative Angular only solutions for testing async code.

**Test setup**

We want to see how we can test *asynchronous* functions.

So we change our AuthService.isAuthenticated() function to an *asynchronous* one that return a promise which resolves into a boolean at a later time.

export class AuthService {

isAuthenticated(): Promise<boolean> {

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

}

}

We also then change our LoginComponent:

export class LoginComponent implements OnInit {

needsLogin: boolean = true;

constructor(private auth: AuthService) {

}

ngOnInit() {

this.auth.isAuthenticated().then((authenticated) => {

this.needsLogin = !authenticated;

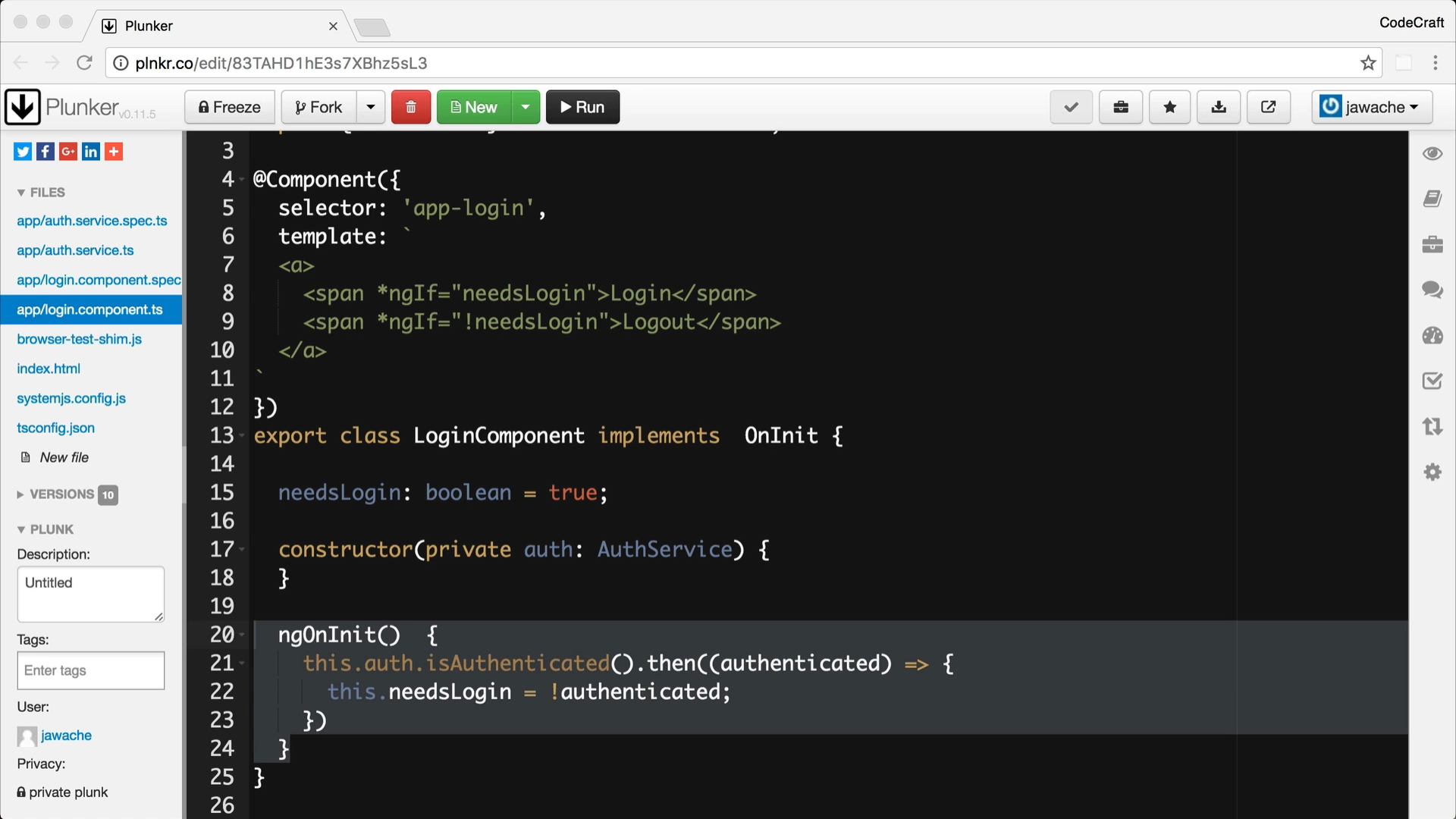
})

}

}

We’ve changed needsLogin from a function into a property and we set the value of this property in the then callback from the promise returned from AuthService.

Importantly we’ve done the above in the ngOnInit() lifecycle function. Probably not the *best* place to put this functionality given that the value might change over time but good for demonstration purposes.



**No asynchronous handling**

Our first attempt might be to try to test our application without taking into account the asynchronous nature of our app, like so:

it('Button label via jasmine.done', () => {

fixture.detectChanges(); ①

expect(el.nativeElement.textContent.trim()).toBe('Login'); ②

spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(true)); ③

component.ngOnInit(); ④

fixture.detectChanges(); ⑤

expect(el.nativeElement.textContent.trim()).toBe('Logout'); ⑥ });

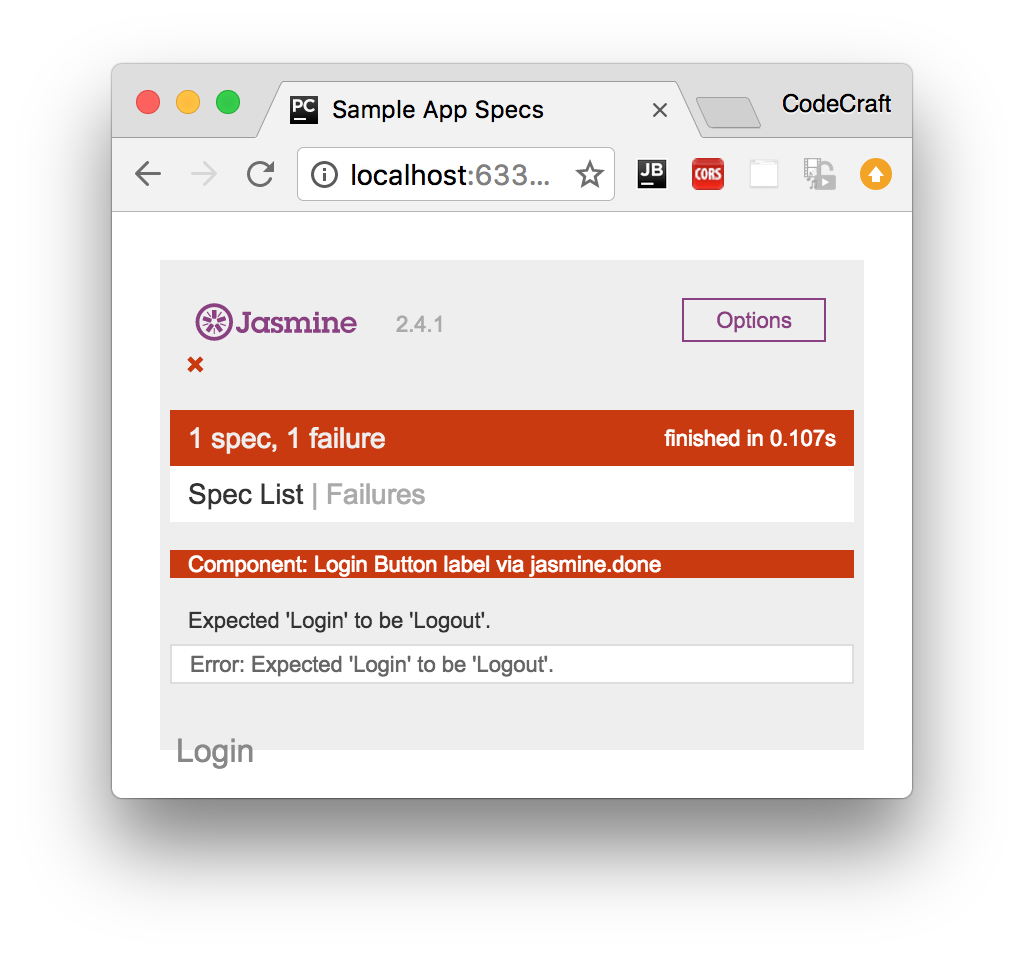
1. We issue our first change detection run so the view does it’s initial update.
2. We expect the button text to display *Login*
3. We change our AuthService so it returns a promise resolved to true.
4. We call component.ngOnInit().

Note: When performing testing we need to call the components life cycle hooks like ngOnInit etc.

1. We issue our second change detection run.
2. We now expect the button text to read *Logout*.

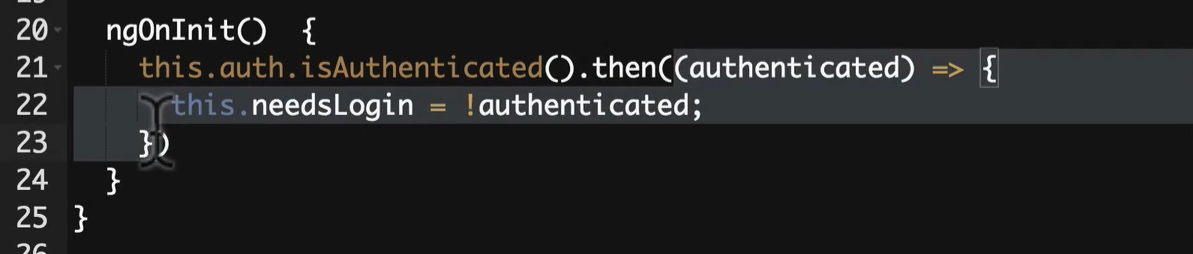
 When performing testing ngOnInit(). Angular won’t do this for us in the test environment.*we* need to call component lifecycle hooks ourselves, like

If we ran the above code we would see it doesn’t pass:



It’s failing on the last expectation. The reason for this is, By the time we run the last expectation the AuthService.isAuthenticated() function hasn’t *yet* resolved to a value.

The following function hasn’t yet been called by the time the last expectation [expect(el.nativeElement.textContent.trim()).toBe('Logout'); ] is called.



Therefore the needsLogin property on the LoginComponent hasn’t been updated. This property(needsLogin) is still true therefore because its true it is showing LOGIN instead of LOGOUT.

There are a few ways we can handle asynchronous code in our tests, **one** is the **Jasmine** way and **two** are **Angular** specific, lets start with the Jasmine way.

**Jasmines** done **function**

Jasmine has a built-in way to handle async code and that’s by the passed in done function in the test specs.

So far we’ve been defining our test specs **without any parameters** but it can take a parameter, but actually jasmine testspec can take parameters and the first one is the DONE function and we call the done function when all our async functions in our testspec are completed. A done function which we call when all the async processing is complete, like so:

it('Button label via jasmine.done', (done) => { ①

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

let spy = spyOn(authService,

'isAuthenticated').and.returnValue(Promise.resolve(true));

component.ngOnInit();

spy.calls.mostRecent().returnValue.then(() => { ②

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Logout');

done(); ③

});

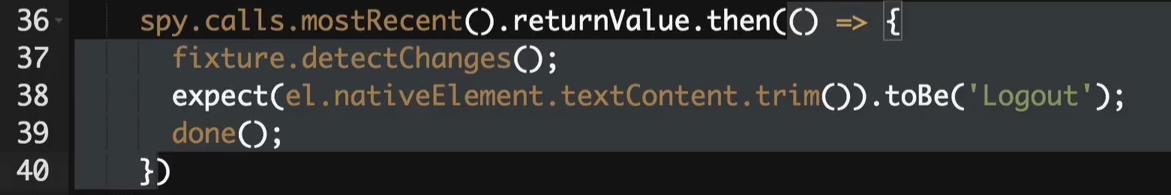
});

1. The jasmine test spec function is passed a function as the first param, we usually call this parameter done.
2. We can add a callback function (using the spy) which is called when the promise returned from isAuthenticated function resolved. In this function we know that the component has the new value of needsLogin and we can add our additional expectation here.
3. When we are done with our asynchronous tasks we tell Jasmine via the done function.

This works as now as we are calling our expectation(LOGOUT) asynchronous in the callback function and this call back function (Point 2 above) is called after Promise is resolved (which is again asynchronous) and when this promise returns a value then this callback(Point 2 above) will be executed and in this callback we are asserting our expectations. So all passes as the code executes in asynchronously.

Jasmine lets us create asynchronous tests by giving us an explict done function which we call when the test is complete.

Although it works trying to understand the code can be difficult as it jumps about and is not executed in the order it’s written in. i.e. Understanding when the following spy function is called requires a deeper understanding of the authService which is not we want to have in our test code. We don’t want to understand the dependent code to write the test 😊



### async and whenStable

Angular has another method for us to test asynchronous code via the async and whenStable functions.

Let’s rewrite the above test to use these and then we will explain the differences.

it('Button label via async() and whenStable()', async(() => { ①

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(true));

fixture.whenStable().then(() => { ②

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Logout');

});

component.ngOnInit();

}));

1. We wrap our test spec function in another function called async.
2. We place the tests we need to run after the isAuthenticated promise resolves inside this

function.

This async function executes the code inside it’s body in a special *async test zone*. This intercepts and *keeps track* of all promises created in it’s body.

It keeps track of Not just the promises created in its body explicitly here but some of these functions calling some other call functions and if those also creates promises then this keeps tracks of those promises also.

i.e. running in async test zone keeps track of all promises inside its function body and when all of those pending function promises are resolved does it then resolve the promise of whenStable.

Only when all of those pending promises have been resolved does it then resolves the promise returned from whenStable.

So by using the async and whenStable functions we now *don’t* need to use the Jasmine spy mechanism of detecting when the isAuthenticated promise has been resolved, like the previous example.

This mechanism is slightly better than using the plain Jasmine solution but there is another version which gives us fine grained control and also allows us to lay out our test code as if it were synchronous.

So now, we don’t need to know the inner workings of the AuthService and no need to use Jasmine Spy mechanism to know when isAuthenticated promise is resolved i.e. we don’t need to isAuthenticated return promise or not and we don’t need to hook it.

Async and WhenStable is generic solution for all the asynchronous solutions to write tests. There is another approach which gives fine grain control and also allows to layout our test code to look like synchronous code.

### fakeAsync and tick

it('Button label via fakeAsync() and tick()', fakeAsync(() => { ①

expect(el.nativeElement.textContent.trim()).toBe('');

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Login');

spyOn(authService, 'isAuthenticated').and.returnValue(Promise.resolve(true));

component.ngOnInit();

tick(); ②

fixture.detectChanges();

expect(el.nativeElement.textContent.trim()).toBe('Logout');

}));

1. Like async we wrap the test spec function in a function called fakeAsync.
2. We call tick() when there are pending asynchronous activities we want to complete.

Like the async function the fakeAsync function executes the code inside it’s body in a special *fake async test zone*. This intercepts and *keeps track* of all promises created in it’s body.

The tick() function *blocks execution* and simulates the passage of time until all pending asynchronous activities complete.

So when we call tick() the application sits and waits for the promise returned from isAuthenticated to be resolved and then lets execution move to the next line.

The code above is now layed our *linearly*, as if we were executing synchronous code, there are no callbacks to confuse the mind and everything is simpler to understand.

 fakeAsync does have some drawbacks, it doesn’t track XHR requests for instance.

ASYNC method have some drawbacks, it doesn’t track XHR requests or HTTP requests that we were making from our test for instance. Then again we should really not making any HTTP request and the test must be isolated really.

**Summary**

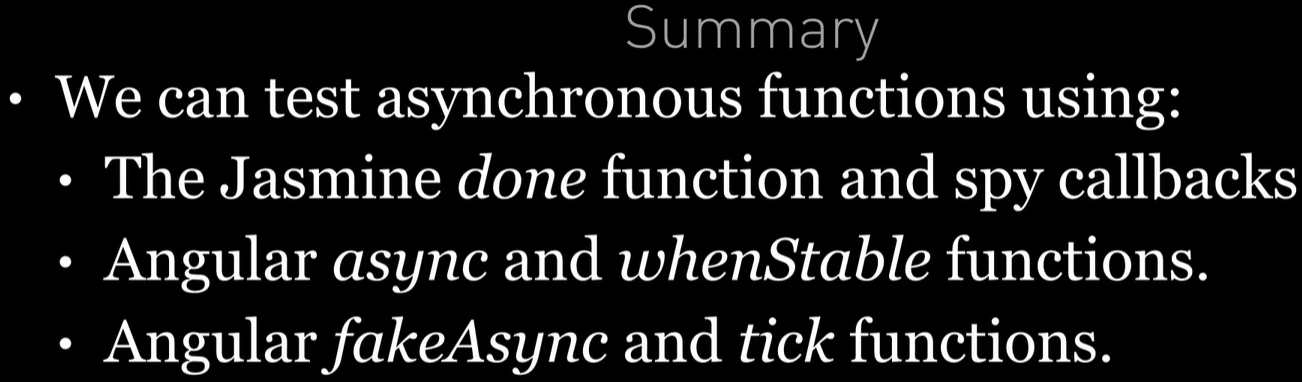
If the code we are testing is asynchronous then we need to take this into account when writing our tests.

There are three mechanisms we can use.

The jasmine done function and spy callbacks. We attach specific callbacks to spies so we know when promises are resolves, we add our test code to those callbacks and then we call the done function. This works but means we need to know about all the promises in our application and be able to hook into them.

Alternatively, We can use the Angular async and whenStable functions, we don’t need to track the promises ourselves but we still need to lay our code out via callback functions which can be hard to read.

Alternatively, We can use the Angular fakeAsync and tick functions, this additionally lets us lay out our async test code as if it were synchronous.



**Listing**

<http://plnkr.co/edit/83TAHD1hE3s7XBhz5sL3?p=preview>