## Testing with Mocks & Spies

In this lecture we are going to discuss how to test classes which have dependencies in isolation by using *Mocks & Spies*.

**Learning Objectives**

* How to *Mock* with fake classes.
* How to *Mock* by extending classes and overriding functions.
* How to *Mock* by using a real instance and a *Spy*.

**Sample code**

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

*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();

}

}

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:

*auth.service.ts*

export class AuthService {

isAuthenticated(): boolean {

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

}

}

**Testing with the real** AuthService

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('canLogin returns false when the user is not authenticated', () => {

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

});

it('canLogin returns false when the user is not authenticated', () => {

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

③

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

});

})

;

1. We create an instance of AuthService and inject it into out LoginComponent when we create it.
2. We clean up data and localStorage after each test spec has been run.
3. 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. (Because we need to know that we need to set token into the localStorage to get our required value. So we need to know inner working 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. And setup all the required data in beforeEach function and clean up in afterEach function.

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('canLogin returns false when the user is not authenticated', () => {

service.authenticated = false;

③

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

});

it('canLogin returns false when the user is not authenticated', () => {

service.authenticated = true;

③

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

});

})

;

1. 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.
2. We inject into our LoginComponent an instance of the MockAuthService instead of the real AuthService.
3. 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;

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.

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

**Mock by using a real instance with Spy**

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

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('canLogin returns false when the user is not authenticated', () => {

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

③

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

expect(service.isAuthenticated).toHaveBeenCalled();

④

});

it('canLogin returns false when the user is not authenticated', () => {

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

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

expect(service.isAuthenticated).toHaveBeenCalled();

});

})

;

1. We create a real instance of AuthService and inject it into the LoginComponent. but we don’t need to care about inner workings of the AuthService i.e. we don’t bother about the fact that we need to set or delete any local storage. Instead we spy on our service so that if IsAuthenticated function is called then Spy makes a return a variable that we control i.e. as directed by us either True/false

SpyOn( *instance that we want to spy on, function we want to spy on*).and.returnValue*(return value);*

1. In our teardown function there is no need to delete the token from localStorage.
2. We create a *spy* on our service so that if the isAuthenticated function is called it returns false.
3. 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:

We don’t bother about the inner workings of the function and simply we can force on the value we want to be returned.

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

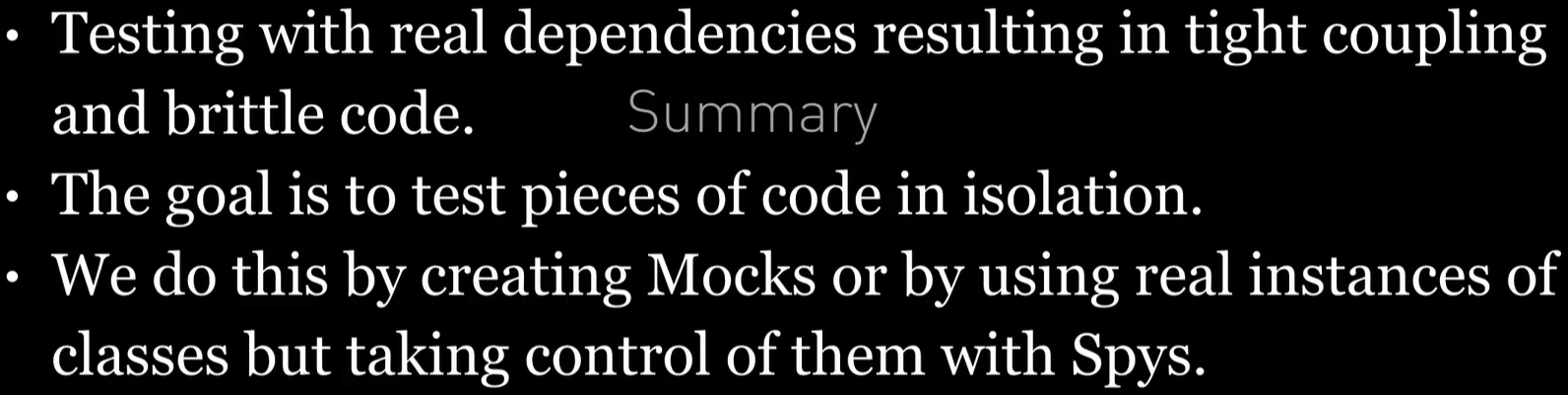
In our example above we make the isAuthenticated function return false or true in each test spec according to our needs.

**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 Spys.



**Listing**

<http://plnkr.co/edit/08ppx8olCnTMpkPdW3eC?p=preview>