## Testing Change Detection

Trying to test whether changes in the state of our application trigger changes in the view without the Angular Test Bed is complicated. However with the ATB it’s much simpler.

In this lecture start interacting with our components template. We add a test to make sure that the bindings in the view updates as we expect when variables in our component change.

**Learning Objectives**

* How to inspect a components view.
* How to trigger change detection so a components view updates based on state changes in our application.

**Setup**

We’ll continue testing our LoginComponent from previous lectures but this time we’ll update the template so we have both a *Login* and *Logout* button like so:

@Component({

selector: 'app-login',

template: `

<a>

<span \*ngIf="needsLogin()">Login</span>

<span \*ngIf="!needsLogin()">Logout</span>

</a>

`

})

export class LoginComponent {

constructor(private auth: AuthService) {

}

needsLogin() {

return !this.auth.isAuthenticated();

}

}

Our test spec file starts close to the version we had in the last lecture like so:

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

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

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

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

import {DebugElement} from "@angular/core"; ①

import {By} from "@angular/platform-browser"; ① describe('Component: Login', () => {

let component: LoginComponent;

let fixture: ComponentFixture<LoginComponent>;

let authService: AuthService;

let el: DebugElement; ②

beforeEach(() => {

// refine the test module by declaring the test component

TestBed.configureTestingModule({

declarations: [LoginComponent],

providers: [AuthService]

});

// create component and test fixture

fixture = TestBed.createComponent(LoginComponent);

// get test component from the fixture

component = fixture.componentInstance;

// UserService provided to the TestBed

authService = TestBed.get(AuthService);

// get the "a" element by CSS selector (e.g., by class name)

el = fixture.debugElement.query(By.css('a')); ③

});

});

1. We’ve imported a few more classes that are needed when interacting with a components view, DebugElement and By class.
2. We have another variable called el which holds reference something called a DebugElement.
3. We store a reference to a DOM element in our el variable.

The fixture as well as holding an instance of the component also holds a reference to something called a DebugElement, this is a wrapper to the low level DOM element that represents the components view, via the debugElement property.

We can get references to other child nodes by querying this debugElement with a By class. The By class lets us query using a a number of methods, one is via a css class like we have in our example another way is to request by a type of directive like By.directive(MyDirective).

We request a reference to the a tag that exists in the components view, this is the button which either says *Login* or *Logout* depending on whether the AuthService says the user is authenticated or not.

We can find out the text content of the tag by calling el.nativeElement.textContent.trim(), we’ll be using that snippet in the test specs later on.

Lets now add a basic test spec like so:

it('login button hidden when the user is authenticated', () => {

// TODO

})

;

**Detect Changes**

The first expectation we place in our test spec might look a bit strange

it('login button hidden when the user is authenticated', () => {

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

});

i.e. above code = we get reference to the el element using the nativeElement(DOM element) and getting the text content and trimming all the whitespaces. We initially *expect* the text inside the a tag to be *blank*.

That’s because when Angular first loads no change detection has been triggered and therefore the view doesn’t show either the Login or Logout text.

fixture is a wrapper for our components environment so we can control things like change detection.

To trigger change detection we call the function fixture.detectChanges(), now we can update our test spec to:

it('login button hidden when the user is authenticated', () => {

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

fixture.detectChanges();

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

});

Once we trigger a change detection run Angular checks property bindings and since the AuthService defaults to not authenticated we show the text Login.

Now lets change the AuthService so it now returns authenticated, like so:

it('login button hidden when the user is authenticated', () => {

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

fixture.detectChanges();

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

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

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

});

But at this point the button content still *isn’t* Logout, that’s because eventhough we changed the function ‘isAuthenticated’ to true and we haven’t changed detection to run and therefore Angular doesn’t know the view needs to be updated. So we need to trigger another change detection run like so:

it('login button hidden when the user is authenticated', () => {

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

fixture.detectChanges();

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

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

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

fixture.detectChanges();

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

});

Now we’ve triggered a second change detection run Angular detected that the AuthService returns true and the button text updated to *Logout* accordingly.

**Summary**

By using the ATB and fixtures we can *inspect* the components view through fixture.debugElement and also trigger a change detection run by calling fixture.detectChanges().

Next up we’ll look at how to can test asynchronous functions in Angular.

**Listing**