## Testing Components

**Learning Objectives**

* How to test a components inputs as well as it’s outputs.
* How to interact with a components view.

**Test setup**

We’ll continue with our example of testing a LoginComponent. We are going to change our component into a more complex version with inputs, outputs, a domain model and a form, like so:

import {Component, EventEmitter, Input, Output} from '@angular/core';

export class User {

①

constructor(public email: string, public password: string) {

}

}

@Component({

selector: 'app-login',

template: `

<

form

>

<label>Email</label>

<input type="email"

#email>

<label>Password</label>

<input type="password"

#password>

<button type="button"

②

(click)="login(email.value, password.value)"

[disabled]="!enabled">Login

</button>

<

/form

>

`

})

export class LoginComponent {

@Output() loggedIn = new EventEmitter<User>();

③

@Input() enabled = true;

④

login(email, password) {

⑤

console.log(`Login ${email} ${password}`);

if (email && password) {

console.log(`Emitting`);

this.loggedIn.emit(new User(email, password));

}

}

}

1. We create a User class which holds the model of a logged in user. Which consists of email and password properties.
2. The button is sometimes disabled depending on the enabled input property value and on clicking the button we call the login function by passing parameters of email and password of the user.
3. The component has an output event called loggedIn.
4. The component has an input property called enabled.
5. In the login function, receives email and password as parameters and we emit a new user model on the loggedIn event(output property).

The component is more complex and uses inputs, outputs and emits a domain model on the output event.

 We are not using the AuthService any more.

We also bootstrap our test suite file like so:

So that we can reuse the TestBed. We are configuring the LoginComponent and grabbing reference to fixture, component and few view elements.

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

let component: LoginComponent;

let fixture: ComponentFixture<LoginComponent>;

let submitEl: DebugElement;

let loginEl: DebugElement;

let passwordEl: DebugElement; beforeEach(() => {

TestBed.configureTestingModule({

declarations: [LoginComponent]

});

// create component and test fixture

fixture = TestBed.createComponent(LoginComponent);

// get test component from the fixture

component = fixture.componentInstance;

submitEl = fixture.debugElement.query(By.css('button'));

loginEl = fixture.debugElement.query(By.css('input[type=email]'));

passwordEl = fixture.debugElement.query(By.css('input[type=password]'));

});

});

We just have a few more debug elements stored on our test suite which we’ll inspect or interact with in our test specs.

**Testing** @Inputs

To test inputs we need to do things:

1. We need to be able to change the input property enabled on our component.
2. We need to check that the button is enabled or disabled depending on the value of our input property.

Solving the first is actually very easy.

Just because it’s an @Input doesn’t change the fact it’s a still just a simple property which we can change like any other property, like so:

it('Setting enabled to false disables the submit button', () => {

component.enabled = false;

})

;

For the second we need to check the disabled property value of the buttons DOM element like so:

it('Setting enabled to false disables the submit button', () => {

component.enabled = false;

fixture.detectChanges();

expect(submitEl.nativeElement.disabled).toBeTruthy();

});

 We also need to call fixture.detectChanges() to trigger change detection and update the view.

**Testing** @Outputs

Testing outputs it somewhat trickier, especially if we want to test from the view.

Firstly lets see how we can track what gets emmitted by the output event and add some expectations for it.

it('Entering email and password emits loggedIn event', () => {

let user: User;

component.loggedIn.subscribe((value) => user = value);

expect(user.email).toBe("test@example.com");

expect(user.password).toBe("123456");

})

;

The key line above is

component.loggedIn.subscribe((value) => user = value);

Since the output event(loggedIn)(EventEmitter) which is actually an *Observable* we can subscribe to it and get a callback for every item emitted.

We store the emitted value to a user object and then add some *expectations* on the user object.

So as soon as something is emitted we store in the local user variable.

How do we actually trigger an event to be fired? We *could* call the component.login(…) function ourselves but for the purposes of this lecture we want to trigger the function from the view.

Firstly lets set some values to our email and password input controls in the view. We’ve already got references to both those fields in our setup function so we just set the values like so:

loginEl.nativeElement.value = "test@example.com";

passwordEl.nativeElement.value = "123456";

Next we trigger a click on the submit button, but we want to do that ***after*** we’ve subscribed to our observable like so:

it('Entering email and password emits loggedIn event', () => {

let user: User;

loginEl.nativeElement.value = "test@example.com"; ①

passwordEl.nativeElement.value = "123456";

component.loggedIn.subscribe((value) => user = value);

submitEl.triggerEventHandler('click', null); ②

expect(user.email).toBe("test@example.com");

expect(user.password).toBe("123456");

});

1. Setup data in our input controls.
2. Trigger a click on our submit button, this synchronously emits the user object in the subscribe callback!.

Trigger events on elements by calling the triggerEventHanlder function and we can pass in which event we want to pass in and the next parameter is the event object we want to pass in. We don’t want to pass in any event object for this and we just want to trigger the event.

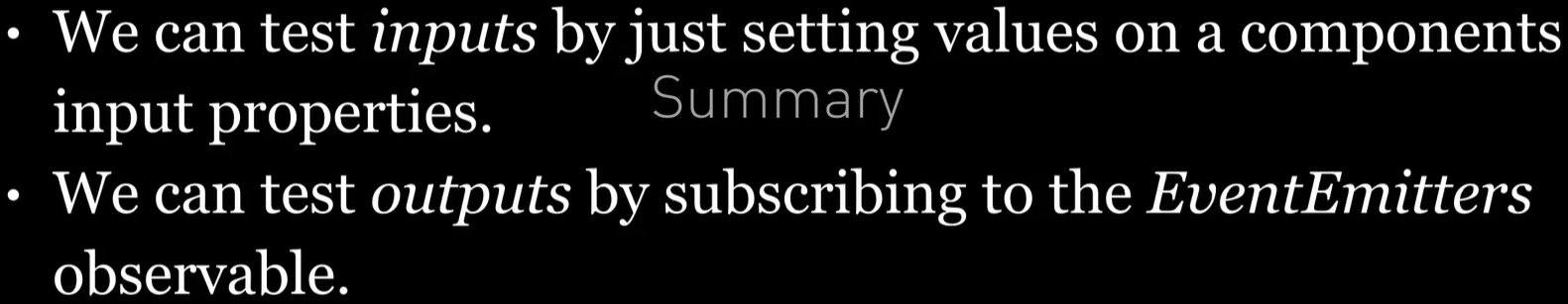
**Summary**

We can test *inputs* by just setting values on a components input properties.

We can test *outputs* by subscribing to an *EventEmitters* observable and storing the emitted values on local variables.

In combination with the previous lectures and the ability to test inputs and outputs we should now have all the information we need to test components in Angular.

In the next lecture we will look at how to test Directives.



**Listing**