1. Create a new project.

ng new AngularTesting

1. app.component.spec.ts

import { TestBed, async } from ‘@angular/core/testing’;

import { AppComponent } from ‘./app.component’;

describe(‘AppComponent’, () => {

beforeEach( async( () => {

TestBed.configureTestingModule({

declarations: [

AppComponent

],

}).compileComponents();

}));

it(‘should create the app’, async(() => {

const fixture = TestBed.createComponent(AppComponent);

const app = fixture.debugElement.componentInstance;

expect(app).toBeTruthy();

}));

it(`should have as title ‘app works!’`, async(() => {

const fixture = TestBed.createComponent(AppComponent);

const app = fixture.debugElement.componentInstance;

expect(app.title).toEqual(‘app works!’);

}));

});

To create a component using Angular CLI use the below command:

ng g c Counter

**counter.component.html**

<div>

<p>Number: {{counter}} </p>

<button (click)=”increment()”>Increment</button>

<button (click)=”decrement()”>Decrement</button>

</div>

**counter.component.ts**

import { Component } from ‘@angular/core’;

@Component({

selector: ‘app-counter’,

templateUrl: ‘./counter.component.html’,

styleUrls: [‘./counter.component.css’]

})

export class CounterComponent {

counter: number;

constructor() {

this.counter = 1;

}

increment() {

this.counter++;

}

decrement() {

this.counter--;

}

};

**counter.component.spec.ts**

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

import { DebugElement } from ‘@angular/core’;

import { CounterComponent } from ‘./counter.component’;

describe(‘CounterComponent’, () => {

let component: CounterComponent;

let fixture: ComponentFixture<CounterComponent>;

let debugElement: DebugElement;

let htmlElement: HTMLElement;

beforeEach(async(() => {

TestBed.configureTestingModule({

declarations : [CounterComponent]

}).compileComponents();

}));

beforeEach(() => {

fixture = TestBed.createComponent(CounterComponent);

component = fixture.componentInstance;

fixture.detechChanges();

debugElement = fixture.debugElement.query(By.css(‘p’));

htmlElement = debugElement.nativeElement;

});

it(‘should increment the counter number by one’, () => {

// Arrange

const initialValue = component.counter;

//Act

component.increment();

fixture.detectChanges();

const newValue = component.counter;

// Assert

expect(newValue).toBeGreaterThan(initialValue);

});

it(‘should display the current number of the counter’, () => {  
 // Assert that the text on screen is of Number: 1

expect(htmlElement.textContent).toEqual(‘Number: 2’);

});

});

# Angular End to End Testing

1. Difference Unit/Component vs. End 2 end Testing:

Unit/Component:

* + Test a single Component, Service, Pipe etc.
  + Test a single specific behavior in a controlled environment.
  + Mock everything you need to test this functionality.
  + Use code coverage analysis to make sure everything is covered.
  + (Usually) Improves code structure and quality.
  + Test edge cases on the most detailed level.
  + Know if a change breaks a specific behavior.
  + Does not test if the whole application or a process works in a real life.

End to end Testing

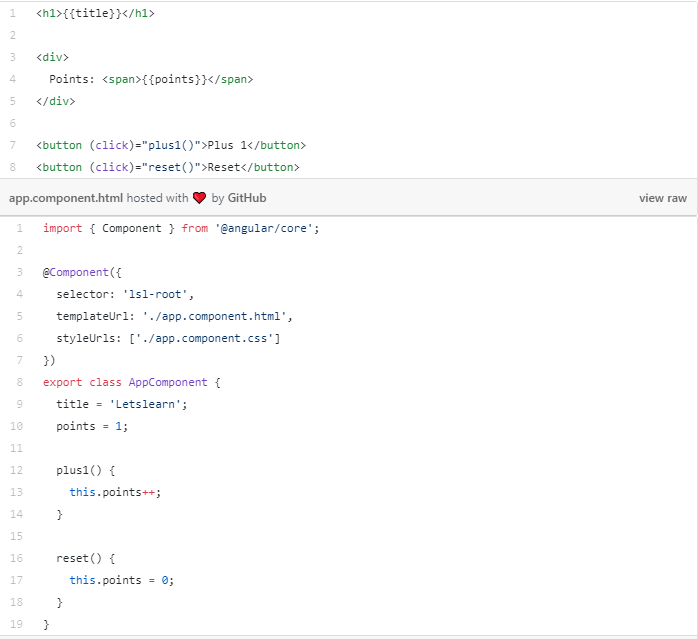
* + Test complete processes “End 2 End”.
  + Test in real environment with the whole application.
  + Test real live situations.
  + Know if most important features work with the complete application.
  + Focussed on the overall functionality and not the specific code structure.
  + Does not test Edge cases.
  + Does not necessarily improve code quality.

1. Mini App example

I took this mini-app as an example:

ng new letslearn-ci

Example – Code for app.component.html/ts:



We have a points variable, a plus 1 button, a reset button and a title. If you click the button, the

Points are increased, if then you click reset, the points go back to 0.

1. Prepare e2e Page Object to Access the Elements

With @angular/cli, you get a e2e folder and structure that suggests to create a page object which helps you to access the elements. The page object for the default page is found in ./e2e/app.po.ts.

**What:** Page Object: A class with functions that return elements like Buttons, Texts or

Whatever you need for the test.



**Why:** If you change your page structure, you only have to change the Page Object, but the

Tests can stay the same.

1. Write e2e tests that walk through a process

Now we can write our e2e tests that use those elements to interact with the page and test its result.

app.e2e-spec.ts

import { LetslearnPage } from './app.po';

describe('letslearn App', () => {

let page: LetslearnPage;

beforeEach(() => {

page = new LetslearnPage();

});

it('Should display Letslearn title', () => {

page.navigateTo();

expect(page.getTitle()).toEqual('Letslearn');

});

it('Should start with 1 point', () => {

page.navigateTo();

expect(page.getPoints()).toEqual('1');

});

it('Should increase points by clicking plus1', () => {

page.navigateTo();

expect(page.getPoints()).toEqual('1');

page.getPlus1Button().click();

expect(page.getPoints()).toEqual('2');

page.getPlus1Button().click();

page.getPlus1Button().click();

page.getPlus1Button().click();

expect(page.getPoints()).toEqual('5');

});

it('Should rest points by clicking plus1', () => {

page.navigateTo();

page.getPlus1Button().click();

page.getPlus1Button().click();

page.getPlus1Button().click();

expect(page.getPoints()).toEqual('4');

page.getResetButton().click();

expect(page.getPoints()).toEqual('0');

});

});

Tests should speak for themselves, meaning that you should be able to read them top down as if it was a story ( with some imaginiation).

1. Run the tests

ng e2e

Top 7 rxjs patterns for angular developers

1. **Static vs Instance Methods**

Problem: You want to use RxJS methods in Angular4.

Solution: You only import the operators you need (tree shaking) from RxJS. There are two types of RxJS operators – Static and Instance. A static method is applied to the Observable class itself. This concept is important because the methods are located in different places.

Instance – rxjs/add/operator/{method}

Static - rxjs/add/observable/{method}

Here’s how we use RxJS instance method:

import { Observable } from ‘rxjs/observable’;

import ‘rxjs/add/operator/combineLatest’;

let hello = Observable.of(‘Hello’)

let world = Observable.of(‘ World’)

hello.combineLatest(world)

And here’s how to use a static method:

import ‘rxjs/add/observable/combineLatest’ ;

Observable.combineLatest(hello, world);

1. **Subscribe to Observable Data**

Problem: You want to extract data from an observable.

Solution: You extract data by subscribing to Observables. You have two main ways of going about this in Angular.

( 1 ). Subscribe with **async** pipe in the HTML or

( 2 ). Subscribe manually in the TypeScript.

In this example, we use method 1 for cats and method 2 for dogs. The result is exactly the same.

cats: FirebaseListObservable<any[]>;

dogs: Array<any[]>;

ngOnInit() {

this.cats = this.db.list('/cats')

this.db.list('/dogs').subscribe(dogs => {

this.dogs = dogs

})

}

In the HTML, we unwrap cats with the async pipe, but dogs are just iterated over like a normal

array.

<div \*ngFor=”let cat of cats | async”></div>

<div \*ngFor=”let dog of dogs”></div>

1. **Unsubscribing from Observable Data**

Problem: You want to prevent memory leaks by unsubscribing from observable data.

Solution: When you use the async pipe this will be handled automatically. If you created a

Subscription in the TypeScript, you will need to unsubscribe manually. The OnDestroy

Lifecycle hook is useful for ending subscriptions.

ngOnDestroy() {

this.subscription.unsubscribe()

})

1. **The SwitchMap (MergeMap ) Pattern**

Problem: You need to extract data from observableA before you can load a related observable.

Solution: This is a common problem with authentication. You might need to load a user, then fetch their associated data from the database. In this example, we load the human, then load the pets that are owned by this human. We use the switchMap operator ( aka mergeMap) to make this possible.

human: FirebaseObjectObservable<any>;

dogs: Observable<any[]>;

ngOnInit() {

this.human = this.db.object('/humans/jeff')

this.dogs = this.human.switchMap(human => {

return this.db.list('/dogs', {

query: {

orderByChild: 'owner',

equalTo: human.name

}

})

})

}

1. **Mapping Observables**

Problem: You want to transform the structure of the data inside an observable.

Solution: Let’s say we want to return the length of a list observable, rather than the data itself. The map operator allows us to change convert the array into a number by returning its length. We can also take an object observable and return one of its properties a string observable.

catCount: Observable<number>;

dogName: Observable<string>;

ngOnInit() {

this.catCount = this.db.list('/cats')

.map(cats => {

return cats.length

})

this.dogName = this.db.object('/dogs/-KpRuD47u8810-BXEn5Q')

.map(dog => {

return dog.name

})

}

1. **Combining Observables**

Problem: You want two object observables as a single array observable.

Solution: Let add a cat and dog into their own observable using the **combineLatest** operator. There are many different **combine operators** in RxJS, so check out the docs to see what works best in your situation.

animals: Observable<any[]>

ngOnInit() {

this.cat = this.db.object('/cats/-KpRuC...Id')

this.dog = this.db.object('/dogs/-KpRuD...Id')

this.animals = Observable

.combineLatest(this.cat, this.dog)

}

1. **The BehaviorSubject Pattern**

Problem: You need to share current data across components or services.

A **BehaviorSubject** is an observable that can receive new data by calling **next(data)**. In this example, make one of the dogs a the **currentDog**, then subscribe to it in the HTML.

This may seem trivial in a single component, but you can create subjects in services, then share the current state of data throughout your application by injecting the service into components.

dogs: FirebaseListObservable<any[]>;

currentDog = new BehaviorSubject(null);

ngOnInit() {

this.dogs = this.db.list('/dogs')

}

changeDog(dog) {

this.currentDog.next(dog)

}

<div \*ngFor = let dog of dogs | async” (click)=”changeDog(dog)”>

{{ dog.name }}

<img [src]=”dog.url”/>

</div>

<h2>Current Dog </h2>

<div>

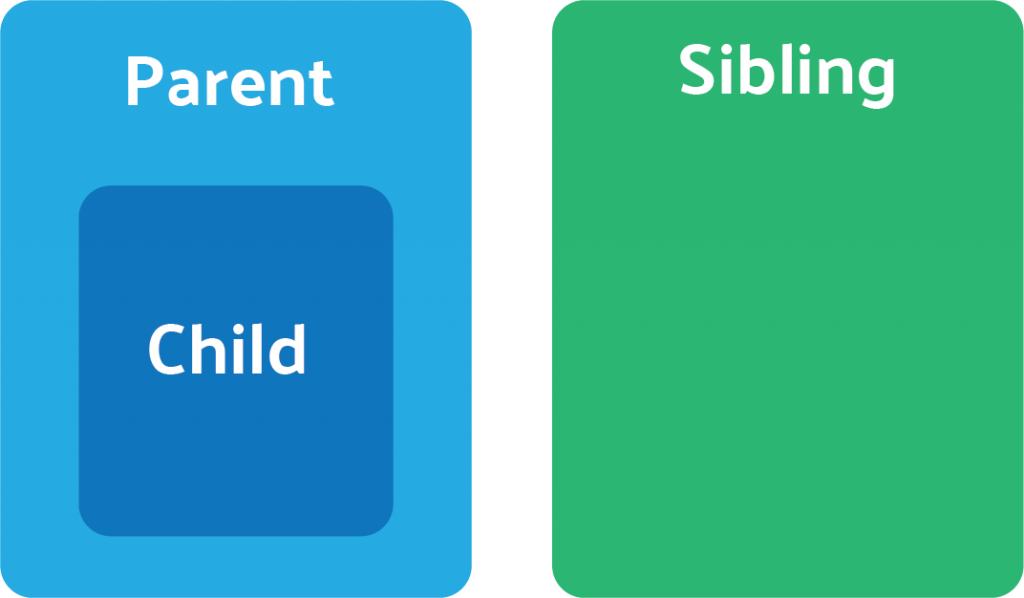
{{ (currentDog | async)?.name }}

<img [src]=”(currentDog | async)?.url”>

</div>

Sharing Data between Angular Components - Four Methods

Data sharing is an essential concept to understand before diving into your first Angular project. In this lesson, I provide four different methods for sharing data between Angular components.



The Parent-Child-Sibling structure of our Angular app.

## Parent to Child: Sharing Data via Input()

This is probably the most common and straightforward method of sharing data. It works by using the [**Input() decorator**](https://angular.io/docs/ts/latest/api/core/index/Input-interface.html) to allow data to be passed via the template.

### parent.component.ts

|  |
| --- |
| import { Component } from '@angular/core';  **@Component**({  selector: 'app-parent',  template: `  <app-child [childMessage]="parentMessage"></app-child>  `,  styleUrls: ['./parent.component.css']  })  export class ParentComponent{  parentMessage = "message from parent"  constructor() { }  } |

### child.component.ts

|  |
| --- |
| import { Component, Input } from '@angular/core';  **@Component**({  selector: 'app-child',  template: `  Say {{message}}  `,  styleUrls: ['./child.component.css']  })  export class ChildComponent {  **@Input**() childMessage: string;  constructor() { }  } |

## Child to Parent: Sharing Data via ViewChild()

ViewChild allows a one component to be injected into another, giving the parent access to its attributes and functions. One caveat, however, is that child won’t be available until after the view has been initialized. This means we need to implement the AfterViewInit lifecycle hook to receive the data from the child.

### parent.component.ts

|  |
| --- |
| import { Component, ViewChild, AfterViewInit } from '@angular/core';  import { ChildComponent } from "../child/child.component";  **@Component**({  selector: 'app-parent',  template: `  Message: {{message}}  <app-child></app-child>  `,  styleUrls: ['./parent.component.css']  })  export class ParentComponent implements AfterViewInit {  **@ViewChild**(ChildComponent) child;  constructor() { }  message:string;  ngAfterViewInit() {  this.message = this.child.message  }  } |

### child.component.ts

|  |
| --- |
| import { Component} from '@angular/core';  **@Component**({  selector: 'app-child',  template: `  `,  styleUrls: ['./child.component.css']  })  export class ChildComponent {  message: string = "Hola Mundo!"  constructor() { }  } |

## Child to Parent: Sharing Data via Output() and EventEmitter

Another way to share data is to emit data from the child, which can be listed to by the parent. This approach is ideal when you want to share data changes that occur on things like button clicks, form entires, and other user events.

In the parent, we create a function to receive the message and set it equal to the message variable.

In the child, we declare a messageEvent variable with the Output decorator and set it equal to a new event emitter. Then we create a function named sendMessage that calls emit on this event with the message we want to send. Lastly, we create a button to trigger this function.

The parent can now subscribe to this messageEvent that’s outputted by the child component, then run the receive message function whenever this event occurs.

### parent.component.ts

|  |
| --- |
| import { Component } from '@angular/core';  **@Component**({  selector: 'app-parent',  template: `  Message: {{message}}  <app-child (messageEvent)="receiveMessage($event)"></app-child>  `,  styleUrls: ['./parent.component.css']  })  export class ParentComponent {  constructor() { }  message:string;  receiveMessage($event) {  this.message = $event  }  } |

### child.component.ts

|  |
| --- |
| import { Component, Output, EventEmitter } from '@angular/core';  **@Component**({  selector: 'app-child',  template: `  <button (click)="sendMessage()">Send Message</button>  `,  styleUrls: ['./child.component.css']  })  export class ChildComponent {  message: string = "Hola Mundo!"  **@Output**() messageEvent = new EventEmitter<string>();  constructor() { }  sendMessage() {  this.messageEvent.emit(this.message)  }  } |

## Unrelated Components: Sharing Data with a Service

When passing data between components that lack a direct connection, such as siblings, grandchildren, etc, you should you a shared service. When you have data that should aways been in sync, I find the **[RxJS BehaviorSubject](https://xgrommx.github.io/rx-book/content/subjects/behavior_subject/index.html" \t "_blank)** very useful in this situation.

You can also use a regular RxJS Subject for sharing data via the service, but here’s why I prefer a BehaviorSubject.

* It will always return the current value on subscription - there is no need to call onnext
* It has a getValue() function to extract the last value as raw data.

It ensures that the component always receives the most recent data.

In the service, we create a private BehaviorSubject that will hold the current value of the message. We define a currentMessage variable handle this data stream as an observable that will be used by the components. Lastly, we create function that calls next on the BehaviorSubject to change its value.

The parent, child, and sibling components all receive the same treatment. We inject the DataService in the constructor, then subscribe to the currentMessage observable and set its value equal to the message variable.

Now if we create a function in any one of these components that changes the value of the message. when this function is executed the new data it’s automatically broadcast to all other components.

### data.service.ts

|  |
| --- |
| import { Injectable } from '@angular/core';  import { BehaviorSubject } from 'rxjs/BehaviorSubject';  **@Injectable**()  export class DataService {  private messageSource = new BehaviorSubject<string>("default message");  currentMessage = this.messageSource.asObservable();  constructor() { }  changeMessage(message: string) {  this.messageSource.next(message)  }  } |

### parent.component.ts

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { DataService } from "../data.service";  **@Component**({  selector: 'app-parent',  template: `  {{message}}  `,  styleUrls: ['./sibling.component.css']  })  export class ParentComponent implements OnInit {  message:string;  constructor(private data: DataService) { }  ngOnInit() {  this.data.currentMessage.subscribe(message => this.message = message)  }  } |

### sibling.component.ts

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { DataService } from "../data.service";  **@Component**({  selector: 'app-sibling',  template: `  {{message}}  <button (click)="newMessage()">New Message</button>  `,  styleUrls: ['./sibling.component.css']  })  export class SiblingComponent implements OnInit {  message:string;  constructor(private data: DataService) { }  ngOnInit() {  this.data.currentMessage.subscribe(message => this.message = message)  }  newMessage() {  this.data.changeMessage("Hello from Sibling")  }  } |

## Angular Testing In Depth : Components

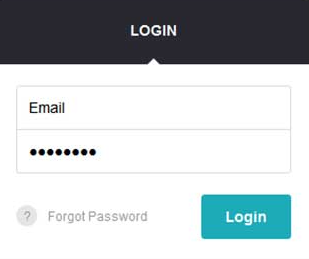
Learn how to test Components in Angular. We will start with writing isolated tests for a component and finish with integration tests for the rendered component.

Components enable us to display content on our website. The Angular compiler parses component templates and displays them according to the given state. The business logic inside the components can be tested with traditional unit tests just like services. But what happens with the displayed HTML from the template and the interaction with other components? In this tutorial I’ll show you how to test the rendered components, starting from isolate unit tests to fully integrated ones.

This article is the third part of a series in which I share my experience testing different building blocks of an Angular application. It relies heavily on Dependency Injection based testing and it is recommended that you read the first part if you are not familiar with the concepts.

**The component we’ll test**

The component we will test is a login form. It doesn’t directly access the Authentication service, instead it just informs the parent component about the submission through an ***Output*** property. The form consists of two fields: an email and a password field. The email address can be preloaded through an ***Input*** property if it is passed down from the wrapping parent component.



The two fields are handled by a reactive form created in the component. With reactive forms, we can add validations to the fields and access their values without touching the DOM. I’ve chosen reactive forms because they are more flexible and easier to test than template-driven forms. Template-driven forms can only be tested through the DOM, making test writing harder.

The [FormBuilder](https://angular.io/docs/ts/latest/api/forms/index/FormBuilder-class.html" \t "_blank) is created inside the constructor and adds validation to the input fields. When the email input changes, the [ngOnChanges](https://angular.io/docs/ts/latest/api/core/index/OnChanges-class.html" \t "_blank) lifecycle hook passes it down to the field. Setting the value on one of the forms control also updates its value in the HTML input element.

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

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({

selector: 'login-form',

templateUrl: './login-form.template.html'

})

export class LoginFormComponent {

private loginForm: FormGroup;

constructor(private formBuilder: FormBuilder) {

this.loginForm = this.formBuilder.group({

email: ['', [Validators.required, Validators.email]],

password: ['', Validators.required]

});

}

@Input() email: string;

@Output() submitted = new EventEmitter();

ngOnChanges(change) {

if (change.email) {

this.loginForm.controls['email'].setValue(change.email.currentValue);

}

}

onSubmit({ email, password }) {

this.submitted.emit({ email, password });

}

}

The template only includes tags that are necessary for the form to function. The value of the formControlName property will be the reference name when we access our inputs inside the form controls. With the [formGroup] property, we fire up the form handling with the components FormGroup and listen to form submissions with (ngSubmit). When we click on the submit button, Angular will catch the event and pass it to the onSubmit handler function.

<form (ngSubmit)="onSubmit(loginForm.value)" [formGroup]="loginForm">

<input type="text" formControlName="email" id="login-email">

<input type="password" formControlName="password" id="login-password">

<button type="submit">Log In</button>

</form>

### Isolated tests

If we just want to focus on the business logic, we can treat and test the component as a service. The Component decorator extends the Injectable decorator, which means it can be created as a service. We just have to pass the component to the providers array in the module dependencies. When testing in isolation, the template never gets compiled. It only gets the required dependencies through the constructor.

import { LoginFormComponent } from './login-form.component';

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

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

import { Component } from '@angular/core';

describe('Isolated', () => {

let subject: LoginFormComponent;

beforeEach(() => {

TestBed.configureTestingModule({

providers: [LoginFormComponent],

imports: [FormsModule, ReactiveFormsModule]

});

});

beforeEach(inject([LoginFormComponent], (loginForm: LoginFormComponent) => {

subject = loginForm;

}));

it('should send credentials on submit', () => {

subject.submitted.subscribe(({ email, password }) => {

expect(email).toEqual(expectedEmail);

expect(password).toEqual(expectedPassword);

});

subject.onSubmit({ email: expectedEmail, password: expectedPassword });

});

});

The test focuses on the method that is called when the form is submitted. It only passes the given email and password to the EventEmitter after destructuring the input object. We don't have to pass Jasmine's asynchronous done callback to the testcase because the EventEmitter acts synchronously.

Isolated tests are good when you want to focus on the component's logic. These tests are also much faster than any other solution. The only downsides are that it won't detect errors in the template nor check the interactions with other components.

### Shallow tests

If we also want to detect errors inside the template, but still focus on a single component, shallow tests are the way to go. The key difference compared to isolated tests is that here the component is compiled.

Inside the beforeEach block, the component class moves to the declarations property instead of providers. Before we can create an instance of the component, it has to be compiled. The compileComponents method does the task asynchronously. It can't be synchronous, because templates and styles can be referenced with relative urls and the fetching of these resources is asynchronous by nature. We have to wait for these tasks to complete. We can do the waiting with the async helper function. In the background, async creates a new zone and waits until every asynchronous operation is finished within that zone. This way, we don't have to fiddle with Jasmine's done callback.

describe('Shallow', () => {

beforeEach(async(() => {

TestBed.configureTestingModule({

declarations: [LoginFormComponent],

imports: [FormsModule, ReactiveFormsModule]

});

TestBed.compileComponents();

}));

it('should send credentials on submit', () => {

let fixture = TestBed.createComponent(LoginFormComponent);

let component: LoginFormComponent = fixture.componentInstance;

let element = fixture.nativeElement;

fixture.detectChanges();

element.querySelector('#login-email').value = expectedEmail;

element.querySelector('#login-email').dispatchEvent(new Event('input'));

element.querySelector('#login-password').value = expectedPassword;

element.querySelector('#login-password').dispatchEvent(new Event('input'));

fixture.detectChanges();

component.submitted.subscribe(({ email, password }) => {

expect(email).toEqual(expectedEmail);

expect(password).toEqual(expectedPassword);

});

element.querySelector('button[type="submit"]').click();

});

});

With the createComponent method, we will have access to the component instance (componentInstance) and the generated HTML fragment (nativeElement). We test the same thing as before: what is emitted on the output at form submission. The big difference is that we fill the inputs and click the submit button. Filling the inputs with valid data is necessary because the form validation leaves the submit button disabled as long as the inputs contain invalid data.

To make the component work, we have to call the detectChanges method after every change. The detectChanges method does the synchronization of the component instance and the generated HTML. Otherwise the component won't know that the input's value has changed. When we modify an input, triggering the input event manually is also necessary because this is the event that Angular listens for.

Also before doing anything inside the nativeElement, we have to call detectChanges first. Calling the method does the first round of property checks on the component and fills out the template based on it.

Finally, we can use the native DOM methods and selectors on the nativeElement property.

With shallow tests, we gain the ability to test the templates, but it comes with a price. These tests run much slower by including the compilation step.

**Integration tests**

The next step is to test the component through its interactions with other components. With integration tests, not only the template, but inputs and outputs will also be tested.

The setup is very similar to shallow tests. We have to set up and compile components. The difference is that we have one more component that uses the login form component inside its template. The wrapper component passes down the predefined email address and listens for the submit event.

describe('Integration', () => {

@Component({

selector: 'site',

template: `<login-form [email]="email" (submitted)="onFormSubmit($event)"></login-form>`

})

class SiteComponent {

email = expectedEmail;

storedEmail: string;

storedPassword: string;

onFormSubmit({ email, password }) {

this.storedEmail = email;

this.storedPassword = password;

}

}

beforeEach(async(() => {

TestBed.configureTestingModule({

declarations: [LoginFormComponent, SiteComponent],

imports: [FormsModule, ReactiveFormsModule]

});

TestBed.compileComponents();

}));

it('should send credentials on submit', () => {

let fixture = TestBed.createComponent(SiteComponent);

let component: SiteComponent = fixture.componentInstance;

let element = fixture.nativeElement;

fixture.detectChanges();

expect(element.querySelector('#login-email').value).toEqual(expectedEmail);

element.querySelector('#login-password').value = expectedPassword;

element.querySelector('#login-password').dispatchEvent(new Event('input'));

fixture.detectChanges();

element.querySelector('button[type="submit"]').click();

expect(component.storedEmail).toEqual(expectedEmail);

expect(component.storedPassword).toEqual(expectedPassword);

});

});

The modification of the input fields is the same, but the assertions are different. This time we don't write assertions for the login form, but write them for the wrapper component instead. This way, we ensure the bindings are correct.

There is no considerable slowdown compared to shallow tests. Integration tests need a bit more setup upfront, but we can test the interactions between the components.

Angular reactive forms:

**app.component.ts:**

import { Component } from ‘@angular/core’;

import { FormBuilder, FormGroup, Validator } from ‘@angular/forms’;

@Component({

selector: ‘app-root’,

templateUrl: ‘./app.component.html’,

styleUrls: [‘./app.component.css’]

})

export class AppComponent {

rForm : FormGroup;

post : any;

description: string = ‘’;

name : string = ‘’;

titleAlert: string = ‘’;

constructor( private fb: FormBuilder ) {

this.rForm = fb.group({

‘name’ : [null, Validators.required],

‘description’: [null, Validators.compose([

Validators.required,

Validators.minLength(30),

Validators.maxLength(500)

])],

‘validate’ : ‘’

});

});

}

ngOnInit() {

this.rForm.get(‘validate’).valueChanges.subscribe( (validate) => {

if ( validate == ‘1’ ) {

this.rForm.get(‘name’).setValidators([

Validators.required,

Validators.minLength(3)]);

this.titleAlert = ‘ You need to specify atleast 3 charecters’;

}

else {

this.rForm.get(‘name’).setValidators(Validators.required);

}

});

}

}

**app.component.html:**

<div \*ngIf=”!name ; else forminfo”>

<form [formGroup] = “rForm” (ngSubmit)=”addPost(rForm.value)”>

<div class=”form-container”>

<div class=”row columns”>

<h1>My Reactive Form </h1>

<label> Name

<input type=”text” **formControlName** = “name”>

</label>

<div class=”alert” \*ngIf**=”!rForm.controls[‘name’].valid &&**

**rForm.controls[‘name’].touched**”> {{titleAlert}} </div>

<label> Description

<textarea **formControlName** = “description”></textarea>

</label>

<label for=”validate”>Minimum of 3 charecters.</label>

<input type=”checkbox” name=”validate” **formControlName** =

“validate” value=”1”> On </input>

<input type=”submit” class=”button expanded” value=”submitform”

[disabled] = “!rForm.valid”></input>

</div>

</div>

</form>

</div>

<ng-template #forminfo>

<div class=”form-container”>

<div class=”row columns”>

<h1> {{name}} </h1>

<p> {{description}} </p>

</div>

</div>

</ng-template>

Angular Reactive Forms

Reactive Forms

* Must use directives from ReactiveFormsModule

Two step process:

* + Create model programmatically
  + Link HTML template to that model using directives

**Model** – use classes in @angular/forms to construct model to be used as a data structure to store the form’s data.

FormControl - atomic form input (e.g. input field)

FormGroup - collection of form controls ( password & confirm )

FormArray - variable length Form type.

Reactive Forms Model

**FormConrol** – can represent a HTML element or can be a more complex fragment. It stores the current value of HTML element it corresponds to, the elements validity status and whether its been modified ( via dirty, pristine, valid, invalid etc)

let petname = new FormControl(‘timber’)

**FormGroup** – collection of form controls. It aggregates the values and status of each Control. If one control is invalid, the entire Group becomes invalid.

let formModel = new FormGroup({

from: new FormControl(),

to: new FormControl()

});

**FormArray** – variable length Form type. It’s a collection of fields capable of growing arbitrarily. Controls associated with indices.

let formModel = new FormGroup({

phoneNumbers: new FormArray({

new FormControl(),

new FormControl()

})

});

Reactive Forms Directives (ReactiveFormsModule)

Form directives:

formGroup – binds a DOM element to an instance of FormGroup class.

formGroupName - links DOM element in template to variable name defined in the parent

FormGroup.

formControl - binds a single DOM element to an instance of a FormControl class.

formControlName - links DOM element in template to name defined in a FormControl

formArrayName - links DOM element to an instance of a FormArray class

**formGroup**

Binds an instance of FormGroup to the HTML <form> element. Inside its scope, a formControlName can be used to link one of its child FormControl instances to a DOM element.

@Component({ … })

class UserComponent{

userModel : FormGroup = new FormGroup({

username: FormControl = new FormControl()

});

}

<form [formGroup]=”userModel”>

<input type = ”text” formControlName = “username”>

</form>

**formGroupName**

links a child FormGroup

class UserComponent {

userModel: FormGroup = new FormGroup({

username : new FormControl(),

credentials : new FormGroup({

password: new FormControl(),

confirm: new FormControl()

})

})

}

<form [formGroup]=”userModel”>

Links <div> Dom elements to the nested FormGroup called credentials in the Parent FormGroup

<div> Username : <input type=”text” formControlName=”username”></div>

<div formGroupName=”credentials”>

<div> Password: <input type=”text” formControlName=”password”></div>

<div> Confirm: <input type=”text” formControlName=”confirm”></div>

</div>

</form>

Links <input> DOM element to credentials.confirm

**formControl**

binds a single field to an instance of FormControl.

Simplest way to use Forms API without a FormGroup and still get reactive behavior as well as validation

class StockPriceComponent {

stockSymbol : FormControl = new FormControl();

constructor() {

this.stockSymbol.valueChanges

.debounceTime(700)

.switchMap(symbol => this.getStockPrice(symbol))

.subscribe(price=>console.log(price));

}

}

<input type=”text” [formControl]=”stockSymbol”>

**FormArrayName**

Links a child FormArray to a DOM element.

class UserComponent {

phoneModel: FormGroup = new FormGroup({

**phoneNumbers** : new FormArray([

new FormControl()

});

addPhoneNumber() {

this.phoneModel.get(‘phoneNumbers’).push(new FormControl());

console.log(this.phoneModel.value);

}

}

<form **[formGroup]**=”phoneModel”>

<ul **formArrayName**=”**phoneNumbers**”>

<li \*ngFor=”let e of phoneModel.get(‘phoneNumbers’).controls; let i=index”>

<input [formControlName]=”I”>

</li>

<button type=”button” (click) =”addPhoneNumber()”>Add Phone Number</button>

</ul>

</form>

**FormBuilder**

Provides a simplified implementation of reactive forms by providing a structure that allows you to list variables and corresponding validators.

constructor(formb: FormBuilder) {

this.formModel = formb.group({

‘name’: [‘nameValidator’], 🡺 2nd argument validator, 3rd argument async validator

‘phone’: [‘phoneValidator’],

‘address’: [‘’] 🡪 Initial value of the FormControl

});

}

**Predefined Validators (basic types)**

Validator class – static predefined methods that correspond to standard HTML5 attributes:

required

minLength,

maxLength,

pattern

import { FormControl, Validators } from ‘@angular/forms’;

let name = new FormControl(‘’, Validators.required);

let phone = new FormControl(‘’, [Validators.required, Validators.minLength(10)]);

let isPhoneValid = phone.valid;

isPhoneValid

{

required: true

}

phoneErrors

{

minLength: {

requiredLength: 10

actualLength: 7

}

}

let phoneErrors: {[key: string]: any} = phone.errors;

Angular Routing:

import { RouterModule, Routes } from ‘@angular/router’;

const appRoutes : Routes = [

{ path: ‘home’, component: HomeComponent},

{ path: ‘about’, component: AboutComponent},

{ path: ‘user’, children: [

{ path: ‘list’, component: UserListComponent , children: [

{path: ‘detail/:name’, component: UserComponent}

]

}

]

{ path: ‘’, redirectTo: ‘/home’, pathMatch: ‘full’},

{ path: ‘\*\*’, redirectTo: ‘home’, pathMatch: ‘full’}

];

**user-list.component.html**

<div class=”container”>

<h2> UserList Page </h2>

<ul>

<li \*ngFor=”let user of users” [routerLink]=”[‘detail’, user.name]”> {{user.name}} </li>

</ul>

<router-outlet></router-outlet>

</div>

**user.component.html**

import { Component, OnInit } from ‘@angular/core’;

import { ActivatedRoute } from ‘@angular/router’;

@Component({

selector: ‘app-user’,

templateUrl: ‘./user.component.html’,

styleUrls: [‘./user.component.css’]

})

export class UserComponent implements OnInit {

name: any;

sub: any;

constructor(private route: ActivatedRoute) {}

ngOnInit() {

this.sub = this.route.params.subscribe( params => {

this.name = params[‘name’]

})

}

}