Angular Notes :

**Installation :::::**

1. There are 2 package manager for Angular **npm** and **yarn**. Mostly used is npm.
2. Need Node/NPM installed –> Install by downloading the application **node-v10.16.2-x64.exe** (https://nodejs.org/en/download)
3. Run command to install Latest Angular –> **npm install –g @angular/cli**
4. To install any specific Angular version -> **npm install –g @angular/cli@6.1.2**
5. To upgrade the existing version -> **npm install –g @angular/cli@latest**
6. To uninstall angular -> **npm uninstall –g @angular/cli**

Then -> **npm cache clean** OR **npm cache verify**

1. We can also manually change the version of Angular in **“package.json”** file.
2. **> ng update <package>**

Analyses your package.json

Recommend updates to your application

3rd parties can provide update scripts using schematics.

Automatically update your code

ng update @angular/core --force // we are forcing it to do so

1. **> ng add**

Add new capabilities to your application

ng add @angular/material -- to install material support in application

1. CLI + Material starter templates
   1. Material Sidenav

ng generate @angular/material:material-nav --name=my-nav

this will automatically create a html template with side nav bar.

* 1. Dashboard

ng generate @angular/material:material-dashboard --name=my-dashboard

this will automatically create a html template with empty dashboard.

* 1. Datatable

ng generate @angular/material:material-table --name=my-table

this will automatically create a html template with table view, with sorting and pagination.

1. To update your application from a version to another version of angular Angular team has provided a link :-

**https://update.angular.io**

1. **rxjs 6 new changes –**

**import{ Observable, Subject, asapScheduler, pipe, of, from, interval, merge, fromEvent } from ‘rxjs’;**

// creation methods, types, schedulers and utilities from rxjs

**import{ map, filter, scan }from ‘rxjs/operators’;**  // All pipeable operators from rxjs/operators

**Operators were renamed :-**

catch() => **catchError()**

do() => **tap()**

finally() => **finalize()**

switch() => **switchAll()**

throw() => **throwError()**

fromPromise() => **from()**  // this automatically detects the type

1. **ng build --prod**

This is to build the Angular application for production

**Project Creation ::::::::**

1. Create New Project -> **ng new MyProjectName -d**

Here the “**-d**” will just show complete list of files which will be created inside this new project. IT WONT CREATE THE ACTUAL PROJECT.

1. The file name ends with “**\*.spec.ts**” are Testing purpose files.
2. Create New Project, without SPEC files -> **ng new MyProjectName --skip-tests**
3. To Run the project -> **ng serve** (go inside new project directory)

Check similar commands -> **ng s** OR **ng serve --open** OR **ng s --o**

1. The project will run at -> **localhost:4200**
2. Use **Visual Studio Code** editor.
3. Angular CLI configuration File -> **angular.json** (earlier it was **angular-cli.json**)
4. Application source code resides in – **src** folder
5. The files outside of this ‘src’ folder are used for **building, testing, maintaining & deploying**

**Adding new Packages ::::::::**

1. To install Bootstrap3 & jQuery -> **npm install bootstrap@3 jquery --save**
2. We can check the entries of Bootstrap & Jquery in ‘**package.json**’ under ‘**dependencies**’ section
3. Also we can see the related folders in ‘node\_modules’
4. In ‘**angular.json**’ we can see the included paths of bootstrap & jquery files inside **styles** & **scripts** block.

Module ::::::::::

1. A module is a thing that consolidates all the components. It’s like a container.
2. Command to create new module :- > ng generate module myModule

This will create a new folder inside our ‘app’ folder with name ‘myModule’

Inside this ‘myModule’ there will be a file – ‘myModule.module.ts’

1. In myModule.module.ts :-

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

import { CommonModule } from ‘@angular/common’;

@NgModule({

imports: [CommonModule],

declarations: []

})

export class MyModule{}

1. Create a component inside myModule -- > ng generate component myModule/view-component
2. Now myModule.module.ts will have entry for this component :-

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

import { CommonModule } from ‘@angular/common’;

import { ViewComponentComponent } from ‘./view-component/ view-component. component’;

@NgModule({

imports: [CommonModule],

declarations: [ViewComponentComponent]

})

export class MyModule{}

1. If we try to use our ‘view-component’ selector at ‘app.component.html’, it won’t work.

To make it work – we have to specify it in ‘app.module.ts’

1. In app.module.ts –

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

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

import { myModule } from ‘./myModule/myModule.module’;

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule, myModule],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule{}

1. But still the ‘view-component’ selector at ‘app.component.html’, won’t work.

Because, in ‘myModule’ we will have to export the components as well, and specify that the components are ok to use outside myModule.

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

import { CommonModule } from ‘@angular/common’;

import { ViewComponentComponent } from ‘./view-component/ view-component. component’;

@NgModule({

imports: [CommonModule],

declarations: [ViewComponentComponent],

exports: [ViewComponentComponent]

})

export class MyModule{}

**Creating Components ::::::::**

1. **App** Component is the default component.
2. To generate new Component other than the default component -> **ng g c employee/create-employee --spec=false --flat=true**

‘g’ means generate

‘c’ means component

‘employee’ is a new folder

‘create-employee’ is the component

‘--spec=false’ to avoid testing files

‘--flat=true’ as we don’t want any dedicated folder for this component

1. Another command - > **ng g c employee-list –it –is**

-it means inline template

-is means inline style

1. We can create as many components as we need. Each component get imported in **app.module.ts** file at declarations array.
2. So, we can have multiple components in a single module.
3. In ‘index.html’ set the base path by **<base href=”/”>** within head tags. (Angular does it automatically)

**Component :::::::**

1. In **app.component.ts** file we can see the **AppComponent** class having **@Component** as decorator
2. We can see this same structure in every component.
3. @Component decorator is the **Meta-Data** for AppComponent class.
4. The @Component decorator is a function which is attached to the class right below it.
5. Due to @Component decorator the class becomes a Component class.
6. As part of the Meta-Data we have –

**selector** : It is basically a custom HTML tag, that used to represent this component class

**templateUrl** : It is the template for the <app-root> component

**styleUrls** : styles that apply only to our component

1. The selector name is used as a tag in HTML page to render the Component.

selector : ‘app-root’, // In **app.component.ts** file

<app-root></app-root> // In **index.html** under body tag

1. The templateUrl points to the HTML file that represents the view for our component.

templateUrl : ‘./app.component.html’, // now the app.component.html code will be replaced with the <app-root>

1. The styleUrls points the css file

styleUrls : [‘./app.component.css’] // this app.component.css will apply to app.component.html only

1. To create new component –

**ng g c test**

It creates 4 new files, and update app.module.ts file

We can see the import of new component in app.module.ts file

**import { TestComponent } from ‘./test/test.component’;**

It will be added to the **declarations** array their

**declarations: [AppComponent,TestComponent],**

1. We can place the selector wherever we need the output of component to be rendered.

For newly created component we can specify it’s selector tag at parent **app.component.html** file

1. To run the angular application – **ng serve** OR **npm start**
2. There are 3 ways to specify the selector
   1. selector: ‘app-test’, // use it as <app-test></app-test>
   2. selector: ‘.app-test’, // selector as a class, use it as <div class=”app-test”></div>
   3. selector: ‘[app-test]’, // selector as an attribute, use it as <div app-test></div>
3. The templateUrl property points to the HTML template file, But we can specify the template inline as well,
   1. templateUrl : ‘./test.component.html’, // pointing to the html template file
   2. template : ‘<div>…………..</div>’ // inline template
   3. template : `<div>. // multiple line template using `back-tik`

.

.

</div>`,

1. Same with styleUrls property, we can have inline styles
   1. styleUrls : [‘./test.component.css’]
   2. styles : [` // multiple line css with `back-tik`

div{

color : red;

}

`]

1. Component Interaction :-

When the parent component want to send some data to child component and vice versa.

We use @Input() & @Output() decorators

**@Output()** 🡪 Child Component to Parent Component, Child will send out events to indicate something.

**@Input()** 🡪 Parent Component to Child Component, It can accept input from the parent.

Let’s take an example, App component is the parent and Test Component is the child.

Now we will send a name “Hello Vinay” from App component to Test Component.

And a message “Hey Coder” from Test Component to the app component.

In app.component.ts class 🡪

public name = ”Hello Vinay”;

In app.component.html 🡪

<div style=”text-align:center”>

<h1>Welcome to {{title}}!</h1>

</div>

<app-test **[parentData]=”name”**></app-test> <!—app-test is the selector of Test Component -->

In test.component.ts 🡪

import { Component, OnInit, Input } from ‘@angular/core’; // need to import Input decorator

template : `<h2>{{“Hello ” + parentData}}</h2>` // printing the data coming from parent

@Input() public parentData; // here we use @Input(), to indicate that the value is coming from parent

We can set the aliases as well for parentData name.

@Input(‘parentData’) public name; // Input is still parentData but we are going to refer it as ‘name’

So we need to modify the template as well –

template : `<h2>{{“Hello ” + name}}</h2>`

**Now we can see the output, for First condition\**

We can also use getters and setters to add logic at **test.component.ts.**

private \_parentData:string;

get parentData():string{

return this.\_parentData;

}

@Input() // we are placing Input decorator over setter method

set parentData(value:string){

this.\_parentData = value;

// we can have our own logic over ‘value’ here.

}

Instead of getters & setters we can use ngOnChanges life cycle hook as well

At test.component.ts

export class TestComponent implements OnChanges{ // import OnChanges from angular/core

@Input() public parentData:string;

constructor(){}

// ngOnChanges get called whenever there is a change in input property of the component

// SimpleChanges takes 3 parameters – previousValue,currentValue,firstChange

ngOnChanges(changes: SimpleChanges){ // import SimpleChanges from angular/core

console.log(changes);

const pData = changes[‘parentData’];

if(pData.currentValue == “Hello Vinay”){

// do something

}else{ // do something }

}

}

Note : ngOnChanges only works with child components

// Use template reference variables to access child properties and methods in Parent Component.

In test.component.ts inside class –

name = “Vinay”;

message:string; // we can use this to load our output and display it over our template

greetVin(){alert(“Hello Vinay”);}

in app.component.html –

<app-test **#child** [parentData]=”name”></app-test> // with this reference variable ‘child’ we can access all the properties and methods from test component

So, now we can use interpolation directly as –

{{child.name}}

<button (click)=”child.greetVin()”></button>

// But if we want to use child properties and methods in parent component class we have to use ViewChild decorator

In app.component.ts –

export class AppComponent implements AfterViewInit{ // import AfterViewInit from angular/core

@ViewChild(TestComponent) childComponentRef:TestComponent; // make imports

ngAfterViewInit(){

this.childComponentRef.message = ‘Message from parent component’;

// we are accessing the ‘message’ property declared in TestComponent here

}

}

**// Output decorator :-**

In same test.component.ts 🡪

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

template : `<h2>{{“Hello ” + parentData}}</h2>

<button (click)=”fireEvent()”>Send Event</button>`

In Test Component class

@Output() public childEvent = new EventEmitter(); // creating an event that we want to send to the parent, with Output decorator

Now create a method fireEvent() -

fireEvent(){

this.childEvent.emit(‘Hey Coder’);

}

In app.component.ts 🡪

public message = ””; // add this property in component class

greet(value:string){alert(“Hello ”+value);} // method

In app.component.html 🡪

<div style=”text-align:center”>

<h1>{{message}}</h1>

</div>

<app-test (childEvent)=”message=$event” [parentData]=”name”></app-test> <!—app-test is the selector of Test Component --> <!—message is variable from app.component.ts --> <!-- $event is the “Hey Coder” -->

// if we have some method like ‘greet()’ in app.component.ts accepting parameter

<app-test (childEvent)=”greet($event)” [parentData]=”name”></app-test>

**Now we can see the output, for our Second condition\**

**// Component Interactions with Service**

Lets say we have 2 components Teacher & Student they may or may not be related.

But they both share a same service.

Now suppose the Teacher want to send a message to Student and Student also want to respond back.

For this situation we will use **Subject** Observable from rxjs

**A Subject is an Observer and Observable.**

**Subject involves taking the notifications from a single source and forwarding them to one or more destinations.**

We use Subject in Service.

So, Teacher can send the notification to the Subject and Subject pushes the notification to all the components that have subscribe to it.

AppComponent is our teacher & ChildComponent is our Student

Now create a service by name ‘interaction.service.ts’ in src folder

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

import {Subject} from ‘rxjs’;

@Injectable({providedIn:’root’})

export class InteractionService{

private \_teacherMessageSource = new Subject<string>();

teacherMessage$ = this.\_teacherMessageSource.asObservable();

constructor(){}

sendMessage(message:string){ // getting the message

this.\_teacherMessageSource.next(message); // pushing the message

}

}

In app.component.html –

<h1>Teacher</h1>

<button (click)=”greetStudent()”>Greet Student</button>

<button (click)=”appreciateStudent()”>Appreciate Student</button>

<app-child></app-child>

In child.component.html –

<h1>Student</h1>

In app.component.ts –

constructor(private \_interactionService:InteractionService){} // import InteractionService

greetStudent(){

this.\_interactionService.sendMessage(‘Good Morning Students’);

}

appreciateStudent(){

this.\_interactionService.sendMessage(‘Well Done’);

}

In child.component.ts –

constructor(private \_interactionService:InteractionService){} // import InteractionService

ngOnInit(){

this.\_interactionService.teacherMessage$.subscribe( // subscribing the observable

message => {

if(message==’Good Morning Students’){

alert(“Good Morning Teacher”);

}else if(message==’Well Done’){

alert(“Thank you Teacher”);

}

}

);

}

**Interpolation ::::::::**

1. How to display the dynamic outputs on html page,

Suppose we want to greet the user as – Welcome **Vinay**

And the name of the user might be coming from some web-service or a web-api

So, the value may change at any point of time.

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

@Component({

selector: ‘app-test’,

template: `

<h2>Welcome {{name}} </h2>

`,

styles: []

})

export class TestComponent implements OnInit{

public name = “Vinay”;

constructor(){}

ngOnInit(){}

greatUser(){ // custom method

return “Hello ” + this.name;

}

}

1. As we can see in above code that the **name** is being used to display the dynamic value over HTML template with **interpolation brackets {{…..}}**. ‘name’ can get it’s value from variety of sources.
2. The interpolation can be used for various purposes –
   1. {{2+2}} // this will evaluate and display the result – 4
   2. {{“Welcome ” + name}} // concatenate the strings – Welcome Vinay
   3. {{name.length}} // will display the length of the string - 5
   4. {{name.toUpperCase()}} // calling methods, name will be in uppercase – VINAY
   5. {{greatUser()}} // we can also call the custom methods defined by us in our Component class.
   6. {{a = 2+2}} // ERROR, we can not do this – template parse error
   7. {{window.location.href}} // ERROR, can not read property location,, we can only do it in the class then bind it to the template.

**Property Binding :::::**

1. Attributes and properties are not the same

Attributes are defined by the HTML, they does not change once initialize

Properties are defined by the DOM, they can change

<input type=”text” value=”Vinay”> // here the value is an Attribute,, and it will display inside the textbox

If we go in browser console –

$0.getAttribute(‘value’) // will print -- Vinay

$0.value // will also print – Vinay

Now change the text in textbox on browser as “NewValue”, so the value should be changed.

$0.getAttribute(‘value’) // will print -- Vinay

$0.value // will also print – NewValue

1. Attribute value is the initial value, property value is the current value.
2. public myId = “testid”; // declare inside component class

<input [id]=”myId” type=”text”> // at our html template, binding to the property

This will appear in console as - <input **id=”testid”** type=”text”>

We are binding to the **id** property of textbox.

This can also be done with interpolation – <input id=”{{myId}}” type=”text”>

But, interpolation has the limitation that it can work **only with string values**.

And there can be Boolean properties that we may also need. That’s why use property binding.

Ex. – **disabled** attribute of input box, which takes only true/false

<input disabled=”false” type=”text”> // no any effect of false

<input disabled=”{{false}}” type=”text”> // no any effect of false

<input [disabled]=”false” type=”text”> // property binding, it works and enable the text box

<input [disabled]=”isDisabled” type=”text”> // now we can set true/false at isDisabled variable from class.

<img [src]=”imgUrl” alt=”Image”> // to bind the image url

There is an alternative to property binding syntax,

Instead of using [….] this , we can use ‘bind-‘

<input **bind-disabled=”isDisabled”** type=”text”>

**Class Binding ::::::**

1. If we have a class in css that can be used with our html tag.

.text-success{ color: green; } // css class

<h2 class=”text-success”>Vinay</h2> // using the class in html tag

1. For binding we need to create a new Property, and bind it with HTML tag

public successClass = “text-success”; // the text-success class is defined in “styles : [``]”

<h2 [class]=”successClass”>Vinay</h2>

1. If we provide both class binding and class attribute to the same tag, the class attribute doesn’t work

<h2 [class]=”successClass” class=”text-special”>Vinay</h2> // text-special is another css class, but it won’t affect.

1. Let suppose we want to display text in red if Error is there –

public hasError = true; // a property in component class

.text-danger{color:red;} // a css class in ‘styles : [``]’

<h2 [class. text-danger]=”hasError”>Vinay</h2> // right now the value is true. So it will display red color.

// modify the hasError to false and red color will be removed

1. If we want to make it conditionally apply to multiple classes use ngClass directive.

A directive is nothing but the custom HTML attribute.

We will need some properties in our Component class.

public isSpecial = true;

public hasError = false;

public messageClasses = { // an object

“text-success” : !this.hasError, // text-success is a css class with some formatting code

“text-danger” : this.hasError, // text-danger is a css class with some formatting code

“text-special” : !this.isSpecial // text-special is a css class with some formatting code

}

Now in our template, bind ngClass to an object ‘messageClasses’ –

<h2 [ngClass]=”messageClasses”>Vinay</h2>

Here angular identifies that the messageClasses has to be applied to the Vinay.

It checks within the object what all the properties are set to true.

Check the output by changing the hasError and isSpecial values either true/false.

**Style Binding ::::::**

1. Inline styles to HTML elements under template.
2. template : `<h2 [style.color]=”’orange’”>Style Binding</h2>`
3. We can have conditional styles as well –

template : `<h2 [style.color]=”hasError ? ‘red’ : ‘green’”>Style Binding</h2>`

if hasError is true color would be red, if it’s false color would be green.

1. We can also apply property value to the style binding.

Define a variable in our component class

public highlightcolor = “orange”;

Now in template –

template : `<h2 [style.color]=”highlightcolor”>Style Binding</h2>`

1. Style object binding using ngStyle –

In our component class create an object having multiple style properties –

public titleStyles ={

color : “blue”,

fontStyle : “italic”

}

Now in template -

template : `<h2 [style.color]=”titleStyles”>Style Binding</h2>`

The element will be displayed with both the properties.

**Event Binding ::::::**

1. For the events like mouseclick, keyboard events we need the data flow from HTML Template to Class.

Class to Template --- Data Binding

Template to Class --- Event Binding

1. Suppose we have a button at our template and once we click on it, the Text should be displayed.

template : `<button (click)=”onClick()”></button>` -- click is the event and onClick() is the method in class.

Now create a method onClick() in our component class

onClick(){

console.log(“Welcome to my Portal”);

}

Now try to click on the button on the browser and check console for output.

1. To print on browser, on button click –

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

@Component({

selector: ‘app-test’,

template: `

<h2>Welcome {{name}} </h2>

<button (click)=”onClick()”>Greet</button>

{{greeting}}

`,

styles: []

})

export class TestComponent implements OnInit{

public name = “Vinay”;

public greeting = “”;

constructor(){}

ngOnInit(){}

onClick(){ // custom method

console.log(“Inside the console”);

this.greeting = ‘Welcome to my portal’;

}

}

As soon as we will click on the button “Greet” the message “Welcome to my portal” will be displayed on browser.

1. $event -> this can give us the information about the DOM event which was raised. For example the information about the click event.

template: `

<h2>Welcome {{name}} </h2>

<button (click)=”onClick($event)”>Greet</button> <!-- $event is the special variable -->

{{greeting}}

`,

Now we will need to modify onClick() method in component class as well

onClick(event){

console.log(event);

this.greeting = event.type; <!-- This will print the property value, we can use any property likewise -->

}

Now in browser console after button click we can see the MouseEvent along with all the information.

1. We can also have inline event handling, without handler –

<button (click)=”greeting=’Welcome Vinay’”>Greet</button>

{{greeting}}

This will print “Welcome Vinay” to ‘greeting’ variable.

**Template Reference Variable ::::**

1. These are used to quickly validate input boxes.

<input type=”text” #myInput>

<button (click)=”logMessage(myInput.value)”></button>

Now in our component class create the method –

logMessage(value){

console.log(value);

}

The reference variable can be used to refer an HTML element and all of it’s DOM property.

<button (click)=”logMessage(myInput)”></button> <!-- This will print <input type=”text”> in console -->

**Two Way Binding :::::**

1. To make our model and view in sync.
2. Two Way Binding allows us to update a property and at the same time display the value of that property.
3. It’s also known as a Banana in a Box [(ngModel)]

<input [(ngModel)]=”name” type=”text”>

{{name}}

In our Component class

public name = “”;

In app.module.ts file import Forms module.

import { FormsModule } from ‘@angular/forms’;

then add it to the imports array –

imports : [

BrowserModule,

**FormsModule**

]

Now in Browser we can type anything inside input box which will get displayed on browser immediately.

1. **Split Two way binding –**

It is used to execute some custom logic over two way binding

The split can be done with ‘ngModel’ directive and the ‘name’ property

<input [ngModel]=”name” (ngModelChange)=”name=$event” type=”text”>

{{name}}

$event – refers to the updated value of the input element.

The above code also works same as [(ngModel)]

Lets modify it and add method name

<input [ngModel]=”name” (ngModelChange)=”greetUser($event)” type=”text”>

{{name}}

Now in the component class lets define the greetUser() method.

export class AppComponent{

public name:string;

public userName:string;

greetUser(updatedValue){

this.userName = updateValue;

// Here we can add our logic

if(updatedValue == ‘Vinay’){

alert(‘Hello Vinay’);

}

}

}

1. **With Getters and Setters –**

<input [(ngModel)]=”customerName” type=”text”>

{{ customerName }}

In the component class

export class AppComponent{

private \_customerName:string;

get customerName():string{

return this.\_customerName;

}

set customerName(value:string){

this.\_customerName = value;

if(value == ‘Vinay’){

alert(‘Hello Vinay’);

}

}

}

**ViewChild ::::::**

1. **Access a DOM element in a Component class –**

**To set the focus on particular field we use viewChild decorator.**

<input **#nameRef** [(ngModel)]=”name” type=”text”>

{{ name }}

In the component class

export class AppComponent implements AfterViewInit{ // import AfterViewInit from angular/core

@ViewChild(‘**nameRef**’)

nameElementRef:ElementRef; // place imports for ViewChild and ElementRef in angular/core

// use AfterViewInit life cycle hook to write component initialization code over references

ngAfterViewInit(){

this.nameElementRef.nativeElement.focus(); // to set focus on name field by default

// to access native element property focus we use nativeElement

console.log(this.nameElementRef); // by this we can see at browser console all the DOM element properties and methods available with nativeElement which we can use.

}

}

**Structural Directives ::::::**

1. **Structural Directives are the directives that let you add or remove HTML elements from the DOM.**
2. There are 3 common Structural Directives –
   1. ngIf
   2. ngSwitch
   3. ngFor
3. **ngIf 🡪**

<h2 \*ngIf=”**true**”>Welcome Vinay</h2>

If we inspect above line in browser console, we can see the <h2> tag is surrounded by <app-test></app-test> tag.

<h2 \*ngIf=”**false**”>Welcome Vinay</h2>

And now we can check again in browser console that the <h2> tag is not available at all, but we can see <app-test></app-test> tag.

We can use the variable as well to set truthy or falsy value.

public displayName = true; // at our component class, we can make it false as well

<h2 \*ngIf = “displayName”>Welcome Vinay</h2> // at – template

If – Else :

<h2 \*ngIf=”displayName; else **elseBlock**”>Welcome Vinay</h2>

<ng-template **#elseBlock**>

<h2>Name is Hidden</h2>

</ng-template>

In above code the #elseBlock represents the block of code which will work if condition doesn’t match.

If—Then—Else :

<div \*ngIf=”displayName; then **thenBlock**; else **elseBlock**”></div>

<ng-template **#thenBlock**>

<h2>Welcome Vinay</h2>

</ng-template>

<ng-template **#elseBlock**>

<h2>Name is Hidden</h2>

</ng-template>

1. **ngSwitch 🡪**

In our component class -

public **color** = “red”; // blue, green, orange

In template –

template : `

<div [ngSwitch]=”**color**”>

<div \*ngSwitchCase=”’red’”>You picked Red color</div>

<div \*ngSwitchCase=”’blue’”>You picked Blue color</div>

<div \*ngSwitchCase=”’green’”>You picked Green color</div>

<div \*ngSwitchDefault>Pick again</div>

</div>

`

1. **ngFor 🡪**

Let create an array in our component class

Public colors = [“red”,”blue”,”green”,”yellow”];

In template –

<div \*ngFor=”let color of colors”>

<h2>{{color}}</h2>

</div>

The above loop will print the items in list one by one.

We can print the index as well -

<div \*ngFor=”let color of colors; index as i”>

<h2>{{i}} {{color}}</h2>

</div>

Another use -

<div \*ngFor=”let color of colors; first as f”>

<h2>{{f}} {{color}}</h2>

</div>

The Output will be :-

true red

false blue

false green

false yellow

Another use -

<div \*ngFor=”let color of colors; last as l”>

<h2>{{l}} {{color}}</h2>

</div>

The Output will be :-

false red

false blue

false green

true yellow

Another use -

<div \*ngFor=”let color of colors; odd as o”>

<h2>{{o}} {{color}}</h2>

</div>

The Output will be :-

false red

true blue

false green

true yellow

Another use -

<div \*ngFor=”let color of colors; even as e”>

<h2>{{e}} {{color}}</h2>

</div>

The Output will be :-

true red

false blue

true green

false yellow

**Pipes ::::**

1. Pipes allow us to transform data before displaying on the view.
2. Built in ->

public name = “Keyboard”; // create this in our component class

public message = “my name is vinay”; // create this in our component class

Then in our template :-

{{name | lowercase}} -- this will display the value of name in lowercase

{{name | uppercase}} -- this will display the value of name in uppercase

{{message | titlecase}} -- this will display the value of message like “my name is vinay” in titlecase as “My Name Is Vinay”

{{name | slice:3}} -- this will display the value of name starts from the index 3.

{{name | slice:3:5}} -- this will display the value of name starts from the index 3 upto 5(not including 5th letter).

Create below object at our component class

public person = {

“firstName” : “John”,

“lastName” : “Doe”

}

Then in our template :-

{{ person | json }} -- this will display json format representation output for our person object in browser.

For Numbers :---

{{5.678 | number:’1.2-3’}} -- 1 : minimum number of integer digit, 2: minimum number of decimal digit, 3: maximum number of decimal digit. Output = 5.678

{{5.678 | number:’3.4-5’}} -- Output = 005.6780 , minimum 4 digits in decimal & maximum 5

{{5.678 | number:’3.1-2’}} -- Output = 005.68 , maximum 2 digits, so ‘78’ rounded off to 8.

{{0.25 | percent}} -- Output = 25%

{{0.25 | currency}} -- Output = $0.25 ,, by default the currency is US dollor

{{0.25 | currency:’GBP’}} -- Output = £0.25 ,, custom, Great Britain Pound

{{0.25 | currency:’GBP’:’code’}} -- Output = GBP0.25 ,, custom, to display the code GBP

Create below property for Date at our Component class

Public date = new Date();

then in our Template :-

{{date}} -- Output = Sun Dec 03 2017 21:48:52 GMT+0530 (India Standard Time)

{{date | date:’short’}} -- Output = 12/3/17, 9:49PM

{{date | date:’shortDate’}} -- Output = 12/3/17

{{date | date:’shortTime’}} -- Output = 9:49PM

{{date | date:’medium’}}

{{date | date:’mediumDate’}}

{{date | date:’mediumTime’}}

{{date | date:’long’}}

{{date | date:’longDate’}}

{{date | date:’longTime’}}

Pipe operator transforms the data only for the view, It does not change the value of the property in the class.

**Services :::::**

1. Service is a class with specific purpose.
2. Use Service when need to –
   1. Share Data among various components
   2. Implement application logic (logic should be independent of any individual component, so that any component can use it if required)
   3. External Interactions (Connecting to a Database)
3. Naming convention for service file is - \*.service.ts (employee.service.ts & the class name EmployeeService)
4. To avoid the violation of

DRY (Do Not Repeat Yourself),

Single Responsibility Principle (a component class should have only 1 responsibility),

we use Services.

1. Suppose we need the same data to be shared in more than 1 component then we should go for the services which will be shared by all the components wherever that data is needed.
2. **How do we go about using the service? Answer is – Dependency Injection (DI).**
3. **Code without DI – drawbacks**

Class Engine{

Constructor(){}

}

Class Tires{

Constructor(){}

}

Class Car{

engine;

tires;

constructor(){

this.engine = new Engine();

this.tires = new Tires();

}

}

Now lets say our Engine class changed, now it needs an parameter –

Class Engine{

Constructor(para){} // petrol, diesel

}

So our Car class will also broke. And we will need to fix it.

Means our code is not flexible.

This code is not suitable for testing as well, anytime we instantiate a new Car we get same Engine and same Tires.

1. **DI as a Design Pattern**

DI is a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.

Class Engine{

Constructor(){}

}

Class Tires{

Constructor(){}

}

Class Car{

engine;

tires;

constructor(engine,tires){ // Parameterize Constructor for DI

this.engine = engine;

this.tires = tires;

}

}

Now here the Car class doesn’t create the dependencies anymore it just consumes them. Creation of dependencies is external to Car class.

So, both the drawbacks are solved.

var myEngine = new Engine();

var myTires = new Tires();

var myCar = new Car(myEngine,myTires);

var myEngine = new Engine(para); // changing Engine class, doesn’t affect others

var myTires = new Tires();

var myCar = new Car(myEngine,myTires);

Now we can easily Mock the data for Testing as well. So, this code is suitable for testing as well.

But, this is ok for limited dependencies, What if the Car class has multiple dependencies, and also those dependencies might be dependent on other dependencies.

var myCar = new Car(myEngine,myTires,depA,depB,………..,depZ);

So this becomes extremely difficult for us, here the DI framework comes into picture.

1. **DI as a Framework**

The DI Framework has **Injector** where we register all our dependencies.

So, the Injector is a container for all the dependencies - Engine,Tires,DepA,DepB,………..,DepZ…

So if you want a Car, so you just ask for a Car and the Injector will provide it. The Framework will handle all the dependencies.

**So the Steps for us are –**

Create a service EmployeeService class.

Register the service with the Angular built-in Injector.

Declare the service as a dependency in the class that needs it.

EmployeeList & EmployeeDetails need EmployeeService class.

1. To create a service, go to our project path and run the command :-

**>ng g s employee**

g – generate

s - service

1. This will create a new service file by name – **employee.service.ts**  in the **src** Folder

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

@Injectable()

export class EmployeeService{

constructor(){}

}

1. Now lets create an method in Service to return the data. At employee.service.ts file in EmployeeService class.

getEmployees(){

return[

{“id”: 1, “name”: “Vinay”, “age”: 29},

{“id”: 2, “name”: “John”, “age”: 25},

{“id”: 3, “name”: “Rocky”, “age”: 26},

{“id”: 4, “name”: “Jery”, “age”: 32},

];

}

1. Now let suppose we need to use this service at 2 components EmployeeList & EmployeeDetails

In EmployeeListComponent class –

Initialize the employee array as an empty array.

public employees =[];

In EmployeeDetailsComponent class as well –

Initialize the employee array as an empty array.

public employees =[];

1. We can register the service at any Component or any Module,

If we register the service in a component than the service will be accessible to that component as well it’s child components.

If we register the service in a Module then the service will be accessible to all it’s child Components.

Let’s register our employee.service.ts in app.module.ts

For registering a service we use Providers Meta-Data.

So, in app.module.ts –

Import…….. List of All the Imports

import{ EmployeeService } from ‘./employee.service’;

@NgModule({

declarations : [

AppComponent,

EmployeeListComponent,

EmployeeDetailComponent

],

imports : [

BrowserModule

],

**providers : [EmployeeSerive],**

bootstrap : [AppComponent]

})

export class AppModule{}

Also from Angular6 onwards we can make the service available to all components from inside service itself –

So, no need to specify it in app.module.ts file explicitly.

@Injectable({

// we declare that the service should be created by the root application injector.

**providedIn: ‘root’,**

**// providedIn tells angular that the root injector is responsible for creating an instance of the service.**

**//Services that are provided this way are automatically made available to the entire application and don’t need to be listed in any particular module.**

})

export class EmployeeService{}

1. Declare our EmployeeSevice as dependency in EmpList and EmpDetail

Dependencies should be defined in constructor parameter.

In employee-list.component.ts at EmployeeListComponent class –

@Component({

selector : ‘employee-list’,

template : `

<h2>Employee List</h2>

<ul \*ngFor=”let employee of employees”>

<li>{{employee.name}}</li>

</ul>

`,

styles: []

})

export class EmployeeListComponent implements OnInit{

public employees = [];

constructor(private \_employeeService:EmployeeService){ } // import EmployeeService

ngOnInit() {

this.employees = this.\_employeeService.getEmployees(); // fetching employees array

}

}

Here, the \_employeeService is the instance of EmployeeService class.

The ngOnInit get called once component has initialized. So we are fetching the data after initialization of component.

In employee-detail.component.ts at EmployeeDetailComponent class –

@Component({

selector : ‘employee-detail’,

template : `

<h2>Employee Detail</h2>

<ul \*ngFor=”let employee of employees”>

<li>{{employee.id}}. {{employee.name}} – {{employee.age}}</li>

</ul>

`,

styles: []

})

export class EmployeeDetailComponent implements OnInit{

public employees = [];

constructor(private \_employeeService:EmployeeService){ } // import EmployeeService

ngOnInit() {

this.employees = this.\_employeeService.getEmployees(); // fetching employees array

}

}

In app.component.html file –

<div style=”text-align:center”><h1>Welcome to {{title}}!</h1></div>

<employee-list></employee-list>

<employee-detail></employee-detail>

Check output…….

1. @Injectable decorator tells angular that this service might itself have injected dependencies,

So if we want to inject a service into another service @Injectable decorator is a must otherwise it’s not necessary.

Means as per our above example we can have the EmployeeService class without @Injectable decorator. But it’s a best practice to have it.

@Injectable is required only for the service and not for the component, Because the Component has @Component decorator which itself tells angular that it might have external dependencies.

**HTTP & Observables :::::::**

1. Suppose we want to fetch data from another web server in our service EmployeeService.

Then we will need to have an **HTTP call** from our getEmployees() method

EmployeeService makes a **Get** call to HTTP to fetch the data & HTTP returns an **Observable** back to EmployeeService, Then EmployeeService need to cast the Observable to get the data.

1. Observables is a HTTP response that arrive asynchronously over time.
2. Once we get the data in our Service we do not provide that information to each & every component, we only provide that information to the component which have subscribed to that Service.
3. As per our earlier example EmployeeListComponent & EmployeeDetailsComponent need the data from EmployeeService, and if the EmployeeService is fetching the data from some external web service then it will get observable in return, so EmployeeListComponent & EmployeeDetailsComponent need to subscribe that observable.
4. Once EmployeeListComponent & EmployeeDetailsComponent get the data they can do anything with it.

EmployeeListComponent can display only name,

EmployeeDetailsComponent can display complete data.

1. **RxJS (Reactive extensions for JavaScript) is a library that enables us to work with Observables in Angular.**
2. In app.module.ts file, let’s import the HttpClientModule

import { HttpClientModule } from ’@angular/common/http’;

And mention it in imports section

imports:[

BrowserModule,

HttpClientModule

]

Now in the employee.service.ts file –

import { Injectable } from ’@angular/core’;

import { HttpClient } from ’@angular/common/http’;

@Injectable()

export class EmployeeService{

private \_url:string = “assets/data/employees.json”;

constructor(private http:HttpClient){}

getEmployees(){

return this.http.get(this.\_url); // we can replace the path to external web server.

// this return is returning Observables by default.

}

}

Now as a dummy JSON web service provider, we will create a employees.json file at path src/assets/data/employees.json

[

{“id”: 1, “name”: “Vinay”, “age”: 29},

{“id”: 2, “name”: “John”, “age”: 25},

{“id”: 3, “name”: “Rocky”, “age”: 26},

{“id”: 4, “name”: “Jery”, “age”: 32},

{“id”: 5, “name”: “Frank”, “age”: 42}

]

//But we want the array of Employees to provide to EmployeeList & EmployeeDetails Components

Let’s create a new file ‘employee.ts’ inside our app folder.

export interface IEmployee{

id: number,

name: string,

age: number

}

Now we have an IEmployee in which the observable can be cast into.

Modify the getEmployees() method –

getEmployees():Observable<IEmployee[]>{

return this.http.get<IEmployee[]>(this.\_url); // also import IEmployee

}

Now we will get an observable having Employee array.

Here, do not forget to import Observable as well.

import { Observable } from ’rxjs/Observable’;

import { IEmployee } from ’./employee’;

In employee-list.component.ts file at ngOnInit() method –

ngOnInit(){

this.\_employeeService.getEmployees().subscribe(data => this.employees = data); // we are subscribing now

}

In employee-detail.component.ts file at ngOnInit() method –

ngOnInit(){

this.\_employeeService.getEmployees().subscribe(data => this.employees = data); // we are subscribing now

}

NOW check the output….

1. To handle HTTP Errors/Exceptions on an observable we make use of the catch operator

Inside employee.service.ts, import the catch operator

import ’rxjs/add/operator/catch’;

Then we add the catch operator on our observable

getEmployees():Observable<IEmployee[]>{

return this.http.get<IEmployee[]>(this.\_url).catch(this.errorHandler);

}

So, “catch(this.errorHandler)” is the method will get called whenever there is an exception.

Now let’s define this errorHandler() method in employee.service.ts

errorHandler(error: HttpErrorResponse){ // import HttpErrorResponse from ‘@angular/common/http’

return Observable.throw(error.message || “Server Error”); // If error.message is NULL throw “Server Error”

// import throw as well - import ‘rxjs/add/observable/throw’;

}

With ‘throw’ we throw the exceptions to the components which have subscribed to this observable.

So, let’s display the error message in the components that have subscribed to this Service.

In employee-list.component.ts –

@Component({

selector: ‘employee-list’,

template: `

<h2>Employee List</h2>

**<h3>{{errorMsg}}</h3>**

<ul \*ngFor=”let employee of employees”>

<li>{{employee.name}}</li>

</ul>

`,

styles:[]

})

export class EmployeeListComponent implements OnInit{

public employees = [];

public errorMsg;

constructor(private \_employeeService: EmployeeService){}

ngOnInit(){

this.\_employeeService.getEmployees().subscribe(data => this.employees = data,

**error => this.errorMsg = error // We receive the error here if any**

);

}

}

In employee-detail.component.ts –

@Component({

selector: ‘employee-detail’,

template: `

<h2>Employee Detail</h2>

**<h3>{{errorMsg}}</h3>**

<ul \*ngFor=”let employee of employees”>

<li>{{employee.id}}. {{employee.name}} – {{employee.age}}</li>

</ul>

`,

styles:[]

})

export class EmployeeDetailComponent implements OnInit{

public employees = [];

public errorMsg;

constructor(private \_employeeService: EmployeeService){}

ngOnInit(){

this.\_employeeService.getEmployees().subscribe(data => this.employees = data,

**error => this.errorMsg = error // We receive the error here if any**

);

}

}

Check OUTPUT..….

To create an error situation :- change the file name at employee.service.ts in \_url from employees.json to employees1.json

You can see the error message.

**Interceptors ::::**

1. Interceptors are used to check http requests in & out.

Create a folder ‘http-interceptors’ inside ‘app’ folder

Inside ‘http-interceptors’ folder we can create multiple interceptors.

Let’s create one – ‘auth-header-interceptor.ts’

import { Injectable } from ’@angular/core’;

import { HttpInterceptor } from ’@angular/common/http’;

@Injectable()

export class AuthHeaderInterceptor implements HttpInterceptor{

intercept(){}

}

1. **Every interceptor must have a method intercept()**

**Routing ::::::::**

1. Create routing module -> **ng g m app-routing --flat=true --module=app**

‘g’ means generate

‘m’ means module

‘--flat=true’ as we don’t want any dedicated folder

‘--module=app’ we want this routing module to be imported into root application module ‘**app.module.ts**’

In **app.module.ts** –

import { **AppRoutingModule** } from ‘./app-routing.module’;

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

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule,**AppRoutingModule**],

providers: [],

bootstrap: [AppComponet]

})

export class AppModule{}

1. In **app-routing.module.ts** The class name will be ‘**AppRoutingModule**’ as per routing module naming convention
2. Now we can setup the routes in **app-routing.module.ts** file
3. Import -> **import {RouterModule, Routes} from ‘@angular/router’;**

Also import all components like - CreateEmployeeComponent, ListEmployeesComponent

As there may be many such components require so instead of importing them one by one, we can make the array of all the components then export that array and import wherever require.

Create a new constant outside the class AppRoutingModule in **app-routing.module.ts** file

**export const routingComponents = [CreateEmployeeComponent, ListEmployeesComponent, Page-Not-Found]**

then import it in **app.module.ts** file –

import { **AppRoutingModule, routingComponents** } from ‘./app-routing.module’;

under @NgModule in -- declarations: [AppComponent**, routingComponents**],

So now we do not need to make separate imports for each component in app.module.ts file.

We just have to import them in app-routing.module.ts file.

1. Now in app-routing.module.ts file -
2. **const appRoutes: Routes = [**

**{path: ‘’, redirectTo: ‘/list’, pathMatch: ‘full’},**

**{path: ‘create’, component: CreateEmployeeComponent},**

**{path: ‘list’, component: ListEmployeesComponent},**

**{path: ‘\*\*’, component: Page-Not-Found}**

**] ;**

1. Now Modify @NgModule ->

**@NgModule({**

**Imports: [**

**RouterModule.forRoot(appRoutes)**

**],**

**exports: [RouterModule]**

**})**

1. **exports: [RouterModule]** -> to make Router Module available to root module i.e. **app.module.ts**
2. Now we can place the links /**list** & /**create** in **app.component.html** file

**<a routerLinkActive=”active” routerLink=”list”>Click to List</a>**

**<a routerLinkActive=”active” routerLink=”create”>Click to Create</a>**

Here,

**routerLinkActive=”active”** -> will make the Link **active** once we click on it, and others will be inactive.

We can modify the style for **active** in **styles.css**

**nav a.active { color: #039be5; }**

1. We also need to add **<router-outlet></router-outlet>** tag in **app.component.html** file where the components will get loaded once we clicked on **Click to List** OR **Click to Create** hyperlinks.
2. Now run –

**localhost:4200/list**

**localhost:4200/create**

**localhost:4200**

1. To handle invalid url routing we use wild card routes.

localhost:4200/test -- is not available and it will throw an error.

In such situation we can navigate the user to a new Page-Not-Found component.

**{path: ‘\*\*’, component: Page-Not-Found}**

This route must be the very last route in configuration, because of top-down approach. Otherwise we will get this Page-Not-Found component for each one.

1. **{path: ‘’, redirectTo: ‘/list’, pathMatch: ‘full’},**

Is the default route.

Here pathMatch tell the router how to match the URL with the configured route.

There are 2 options : **full / prefix**

Prefix is not much of use

1. How to pass parameters with route!!!

In Department-List component we will have list of departments, each department will have a unique id.

If you click on the department name you will get navigate to Department-Details component passing the id of the department as a route parameter.

And in Department-Details component we will read that parameter and display the id.

So, in department-list.component.ts –

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

**import { Router } from ‘@angular/router’;**

@Component({

selector: ‘app-department-list’,

template: `

<h3>Department List </h3>

<ul class=”items”>

<li **(click)=”onSelect(department)”** \*ngFor=”let department of departments”>

<span class=”badge”>{{department.id}}</span> {{department.name}}

</li>

</ul>

`,

styles : []

})

export class DepartmentListComponent implements OnInit{

departments = [

{“id”:1,”name”:”Angular”},

{“id”:2,”name”:”Node”},

{“id”:3,”name”:”MongoDB”},

{“id”:4,”name”:”Ruby”},

{“id”:5,”name”:”Bootstrap”},

]

constructor(**private router: Router**){}

ngOnInit(){}

onSelect(department){

// to rout from here we need Router service

this.router.navigate([‘/departments’,department.id]);

}

}

Now in app-routing.module.ts –

const appRoutes: Routes = [

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

{path: ‘departments’, component: DepartmentListComponent},

**{path: ‘departments/:id’, component: DepartmentDetailComponent},**

{path: ‘employees’, component: EmployeeListComponent},

{path: ‘\*\*’, component: Page-Not-FoundComponent}

] ;

Here the **“:id”** is the placeholder for a route parameter.

Now we will read the parameter and display it on browser –

For that we will use Activated Route Service

Inside **department-detail.component.ts** file –

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

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

@Component({

selector: ‘app-department-detail’,

template: `

<h3>You selected Department with id = {{**departmentId**}} </h3>

`,

styles : []

})

export class DepartmentDetailComponent implements OnInit{

public **departmentId**;

constructor(**private route: ActivatedRoute**){}

ngOnInit(){

// here we will read the route parameter

l**et id = parseInt(this.route.snapshot.paramMap.get(‘id’));**

// now this id will get the id which comes with URL

this.departmentId = id;

}

}

1. In the above example we used the **snapshot** way to get an id from parameter.

But it has drawback – When we are navigating from 1 component back to the same component the snapshot approach will not work. Because the component is already initialized and next time the ngOnInit will not be called.

So to overcome this we use paramMap observable.

Let’s modify the example and add PREV and NEXT button to navigate back and forth between different departments.

Inside **department-detail.component.ts** file –

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

import { ActivatedRoute, **Router, ParamMap** } from ‘@angular/router’;

@Component({

selector: ‘app-department-detail’,

template: `

<h3>You selected Department with id = {{departmentId}} </h3>

**<a (click)=”goPrev()”>PREV</a>**

**<a (click)=”goNext()”>NEXT</a>**

`,

styles : []

})

export class DepartmentDetailComponent implements OnInit{

public departmentId;

constructor(private route: ActivatedRoute, **private router: Router**){}

ngOnInit(){

// remove old snapshot approch

**this.route.paramMap.subscribe((params: ParamMap) => let id = parseInt(params.get(‘id’)));**

// this will return observable. Now it can detect and read the change

this.departmentId = id;

}

**goPrev(){**

**let prevId = this.departmentId – 1;**

**this.router.navigate([‘/departments’,prevId])**

**}**

**goNext(){**

**let nextId = this.departmentId + 1;**

**this.router.navigate([‘/departments’,nextId])**

**}**

}

**Check OutPut now….**

1. **Optional Route Parameters –**

As per our above example we want to add a back button in department-details

If we click on Back button we should be navigated back to department-list and previously clicked departments must be highlighted.

Inside **department-detail.component.ts** file –

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

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

@Component({

selector: ‘app-department-detail’,

template: `

<h3>You selected Department with id = {{departmentId}} </h3>

<a (click)=”goPrev()”>PREV</a>

<a (click)=”goNext()”>NEXT</a>

**<div>**

**<button click=”goToDepartments()”>Back</button>**

**</div>**

`,

styles : []

})

export class DepartmentDetailComponent implements OnInit{

public departmentId;

constructor(private route: ActivatedRoute, private router: Router){}

ngOnInit(){

// remove old snapshot approch

this.route.paramMap.subscribe((params: ParamMap) => let id = parseInt(params.get(‘id’)));

// this will return observable. Now it can detect and read the change

this.departmentId = id;

}

goPrev(){

let prevId = this.departmentId – 1;

this.router.navigate([‘/departments’,prevId])

}

goNext(){

let nextId = this.departmentId + 1;

this.router.navigate([‘/departments’,nextId])

}

**goToDepartments(){**

**let selectedId = this.departmentId ? this.departmentId : null;**

**this.router.navigate([‘/departments’, {id: selectedId, test:’testvalue’}]);**

// […] here it is known as the link parameters array

// test doesn’t exist, but our route still works because all the parameters are optional

// it does not require **‘departments/:id’** in app-routing.module.ts

**}**

}

Inside **department-list.component.ts** –

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

**import { Router,ActiviatedRoute, ParamMap } from ‘@angular/router’;**

@Component({

selector: ‘app-department-list’,

template: `

<h3>Department List </h3>

<ul class=”items”>

<li **(click)=”onSelect(department)” [class.selected]=”isSelected(department)”** \*ngFor=”let department of departments”>

<span class=”badge”>{{department.id}}</span> {{department.name}}

</li>

</ul>

`,

styles : []

})

export class DepartmentListComponent implements OnInit{

public selectedId;

departments = [

{“id”:1,”name”:”Angular”},

{“id”:2,”name”:”Node”},

{“id”:3,”name”:”MongoDB”},

{“id”:4,”name”:”Ruby”},

{“id”:5,”name”:”Bootstrap”},

]

constructor(private router: Router, **private route:ActivatedRoute**){}

ngOnInit(){

// remove old snapshot approch

this.route.paramMap.subscribe((params: ParamMap) => let id = parseInt(params.get(‘id’)));

// this will return observable. Now it can detect and read the change

**this.selectedId = id;**

}

onSelect(department){

// to rout from here we need Router service

this.router.navigate([‘/departments’,department.id]);

}

isSelected(department){

**return department.id === this.selectedId;**

}

}

At **styles.css** –

.items li.selected{

background-color : #cfd8dc;

color: white;

}

1. Relative Navigation –

Till now in above examples we have used absolute paths like “/departments”.

Due to absolute path there is a lack of flexibility in routes.

Let’s suppose we got a requirement after development of application that instead of “/departments” they want “/department-list” as a name. For this we will need to change in our Routing Module each and every occurrence of “/department” in various components.

We can solve this with Relative Navigation –

Inside **department-list.component.ts** –

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

import { Router,ActiviatedRoute, ParamMap } from ‘@angular/router’;

@Component({

selector: ‘app-department-list’,

template: `

<h3>Department List </h3>

<ul class=”items”>

<li (click)=”onSelect(department)” [class.selected]=”isSelected(department)” \*ngFor=”let department of departments”>

<span class=”badge”>{{department.id}}</span> {{department.name}}

</li>

</ul>

`,

styles : []

})

export class DepartmentListComponent implements OnInit{

public selectedId;

departments = [

{“id”:1,”name”:”Angular”},

{“id”:2,”name”:”Node”},

{“id”:3,”name”:”MongoDB”},

{“id”:4,”name”:”Ruby”},

{“id”:5,”name”:”Bootstrap”},

]

constructor(private router: Router, **private route:ActivatedRoute**){}

ngOnInit(){

// remove old snapshot approch

this.route.paramMap.subscribe((params: ParamMap) => let id = parseInt(params.get(‘id’)));

// this will return observable. Now it can detect and read the change

this.selectedId = id;

}

onSelect(department){

// this doesn’t care about the URL, just append the departmentId to the current route

**this.router.navigate([department.id], {relativeTo: this.route});**

}

isSelected(department){

return department.id === this.selectedId;

}

}

Let’s do same in **department-detail.component.ts** file –

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

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

@Component({

selector: ‘app-department-detail’,

template: `

<h3>You selected Department with id = {{departmentId}} </h3>

<a (click)=”goPrev()”>PREV</a>

<a (click)=”goNext()”>NEXT</a>

<div>

<button click=”goToDepartments()”>Back</button>

</div>

`,

styles : []

})

export class DepartmentDetailComponent implements OnInit{

public departmentId;

constructor(private route: ActivatedRoute, private router: Router){}

ngOnInit(){

// remove old snapshot approch

this.route.paramMap.subscribe((params: ParamMap) => let id = parseInt(params.get(‘id’)));

// this will return observable. Now it can detect and read the change

this.departmentId = id;

}

goPrev(){

let prevId = this.departmentId – 1;

this.router.navigate([‘/departments’,prevId])

}

goNext(){

let nextId = this.departmentId + 1;

this.router.navigate([‘/departments’,nextId])

}

goToDepartments(){

let selectedId = this.departmentId ? this.departmentId : null;

**this.router.navigate([‘../’, {id: selectedId, test:’testvalue’}], {relativeTo.this.route});**

// [../] here it is known as the back url

// test doesn’t exist, but our route still works because all the parameters are optional

// it does not require ‘departments/:id’ in app-routing.module.ts

}

}

1. Child Routes –

Some routes may only be viewed within other routes, in such a situation we create child routes.

Let’s create 2 components department-overview & department-contact

We want to add both of these new components to add to our departmentDetails Component.

Inside app-routing.module.ts –

const appRoutes: Routes = [

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

{path: ‘departments’, component: DepartmentListComponent},

{

path: ‘departments/:id’,

component: DepartmentDetailComponent,

**children: [**

**{path:’overview’, component: DepartmentOverviewComponent},**

**{path: ‘contact’, component: DepartmentContactComponent}**

**]**

},

{path: ‘employees’, component: EmployeeListComponent},

{path: ‘\*\*’, component: Page-Not-Found}

] ;

Here we are using **children** property to introduce child routes, having an array of routes.

Also import **DepartmentOverviewComponent** & **DepartmentContactComponent.**

And also add them to **routingComponent** array.

So, we have mentioned that these 2 components are to be displayed only inside the DepartmentDetailComponent.

Now we have to use <router-outlet> to specify where to display both of them

In the **department-detail.component.ts** –

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

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

@Component({

selector: ‘app-department-detail’,

template: `

<h3>You selected Department with id = {{departmentId}} </h3>

**<p>**

**<button (click)=”showOverview()”>Overview</button>**

**<button (click)=”showContact()”>Contact</button>**

**</p>**

**<router-outlet></router-outlet>**

<p>

<button (click)=”goPrev()”>PREV</button>

<button (click)=”goNext()”>NEXT</button>

</p>

<div>

<button click=”goToDepartments()”>Back</button>

</div>

`,

styles : []

})

export class DepartmentDetailComponent implements OnInit{

public departmentId;

constructor(private route: ActivatedRoute, private router: Router){}

ngOnInit(){

// remove old snapshot approch

this.route.paramMap.subscribe((params: ParamMap) => let id = parseInt(params.get(‘id’)));

// this will return observable. Now it can detect and read the change

this.departmentId = id;

}

goPrev(){

let prevId = this.departmentId – 1;

this.router.navigate([‘/departments’,prevId])

}

goNext(){

let nextId = this.departmentId + 1;

this.router.navigate([‘/departments’,nextId])

}

goToDepartments(){

let selectedId = this.departmentId ? this.departmentId : null;

this.router.navigate([‘../’, {id: selectedId, test:’testvalue’}], {relativeTo.this.route});

// [../] here it is known as the back url

// test doesn’t exist, but our route still works because all the parameters are optional

// it does not require ‘departments/:id’ in app-routing.module.ts

}

**showOverview(){**

**this.router.navigate([‘overview’],{relativeTo: this.route});**

**}**

**showContact(){**

**this.router.navigate([‘contact’],{relativeTo: this.route});**

**}**

}

We can have default and invalid child routes as well.

**Differential Loading in Angular-8 :::::**

1. It Automatically makes your angular apps more performant
2. When we build an application in Angular-8 for production, it creates 2 bundles
   1. For Modern browsers that support ES6+
   2. For Older browsers that only support ES5

So, when the app is opened in the browser the correct bundle is automatically loaded.

**Dynamic imports for Lazy routes ::::::**

1. In a big angular application we may have lots of Routes with lots of feature modules, Loading the bundle for that entire application when the user is visiting the site can effect on performance.

The better way is to initially load the code that is necessary only for the initial routes, and then load the code for the other routes, only when the user navigates to those routes for the first time, This will improve the performance.

This concept is lazy loading.

Lazy loaded route config before Angular8 – {path:’/user’, loadChildren:’./user/user.module#UserModule’}

From Angular 8 we use dynamic import –

{path:`/user`, loadChildren: () => import(`./user/user.module`).then(m => m.UserModule) }

**Ivy ::::::::::**

1. Ivy is the rendering engine on which Angular team is working

It is used to translate the templates and components into regular HTML and JavaScript that the browser can understand.

But it’s still not stable.

So, in Angular8 we have the option to switch to obtain Ivy.

ng new demo --enable-ivy

Ivy has the power to generate considerably smaller bundles, makes incremental compilation easier and is going to be basis for future innovations.

**Bazel ::::::**

1. A build tool, under experiment till date.

**Forms :::::::::::**

1. The concept behind Angular forms –

Component Template (HTML Forms) => Component Class (Data Binding) => Service (Data Processing) => Server

1. There are 2 ways to Create Forms

Template Driven Forms

Reactive Forms OR Model Driven Forms

1. **Template Driven Forms :-**

Heavy on Templets (most of the code in .html file)

Form creation in HTML

Easy to build and understand

Great for Simple forms, But for Complex forms it’s not recommended, as the HTML can get messy.

Not easy to unit test. (Need to run end to end test with browser)

1. **Reactive Forms** OR **Model Driven Forms :-**

Form building in code (most of the code in our .ts file)

More flexible

Easy to add dynamic input elements

Can adjust validations at run time

Easy for Unit tests as most of the logic in component class

1. **In Template Driven Forms –**
2. At app.component.html –

<div class=”container-fluid”>

<h1>Bootcamp Enrollment Form</h1>

<div class=”alert alert-danger” \*ngIf=”errorMsg”>{{errorMsg}}</div>

<form #userForm=”ngForm” \*ngIf=”!submitted” novalidate (ngSubmit)=”onSubmit()”>

<!—we have defined onSubmit() event handler at app.component.ts -->

{{ userForm.form.valid }} <!—the form is valid/not valid i.e. true/false -->

<hr/>

{{ userForm.values | json }}

<hr/>

{{ userModel | json }}

<div ngModelGroup=”address”> <!—This is a group -->

<div class=”form-group”>

<label>Street</label>

<input type=”text” class=”form-control” name=”street” ngModel>

</div>

<div class=”form-group”>

<label>City</label>

<input type=”text” class=”form-control” name=”city” ngModel>

</div>

<div class=”form-group”>

<label>State</label>

<input type=”text” class=”form-control” name=”state” ngModel>

</div>

<div class=”form-group”>

<label>Postal Code</label>

<input type=”text” class=”form-control” name=”postalcode” ngModel>

</div>

</div>

<div class=”form-group”>

<label>Name</label>

<input type=”text” class=”form-control” name=”userName” #name required [(ngModel)]=”userModel.name” [class.is-invalid]=”name.invalid && name.touched” >

</div>

{{ name.className }}

<small class=”text-danger” [class.d-none]=”name.valid || name.untouched”>Name is required</small>

<div class=”form-group”>

<label>Email</label>

<input type=”email” class=”form-control” name=”email” [(ngModel)]=”userModel.email”>

</div>

<div class=”form-group”>

<label>Phone</label>

<input type=”tel” required class=”form-control” name=”phone” [(ngModel)]=”userModel.phone” #phone=”ngModel” pattern=”^\d{10}$” [class.is-invalid]=”phone.invalid && phone.touched”>

</div>

<!-- <small class=”text-danger” [class.d-none]=”phone.valid || phone.untouched”>Phone number is required and must be 10 digits</small> -->

<div \*ngIf=”phone.errors && (phone.invalid || phone.touched)”>

<small class=”text-danger” \*ngIf=”phone.errors.required”>Phone number is required</small>

<small class=”text-danger” \*ngIf=”phone.errors.pattern”>Phone number must be 10 digits</small>

</div>

<div class=”form-group”>

<select required #topic=”ngModel” [class.is-invalid]=”topic.invalid && topic.touched” class=”custom-select” name=”topic” [(ngModel)]=”userModel.topic” >

<option value=””>I am interested in</option>

<option \*ngFor=”let topic of topics”>{{topic}}</option>

</select>

<small class=”text-danger” [class.d-none]=”topic.valid || topic.untouched”>Please choose a topic</small>

</div>

<div class=”mb-3”>

<label>Time Preference</label>

<div class=”form-check”>

<input class=”form-check-input” [(ngModel)]=”userModel.timePreference” type=”radio” name=”timePreference” value=”morning”>

<label class=”form-check-label”>Morning (9am – 12pm)</label>

</div>

<div class=”form-check”>

<input class=”form-check-input” [(ngModel)]=”userModel.timePreference” type=”radio” name=”timePreference” value=”evening”>

<label class=”form-check-label”>Evening (5pm – 8pm)</label>

</div>

</div>

<div class=”form-check mb-3”>

<input class=”form-check-input” type=”checkbox” name=”subscribe” [(ngModel)]=”userModel.subscribe” >

<label class=”form-check-label”>Send me promotional offers</label>

</div>

<button [disabled]=”userForm.form.invalid || topicHasError” class=”btn btn-primary” type=”submit”>Submit Form</button>

</form>

</div>

// #userForm - is the template reference variable [At - <form #userForm=”ngForm”>]

// ngModel - is to tell Angular please keep track of this element. [<input type=”text” ngModel>]

// Once we gave name to each and every element and ngModel,,, {{ userForm.values | json }} – this will show the entered data in fields on screen. It is used just for check.

// like ngModel there is another directive available – ngModelGroup

// this is used to group some fields together. Or to create a subgroup within a form

// when its come to data binding with template driven forms we have 3 directives ngForm,ngModel, ngModelGroup

// create a class – ng generate class User

// this command will create a new file user.ts.

export class User{

constructor(

public name: string,

public email: string,

public phone: number,

public topic: string,

public timePreference: string,

public subscribe : boolean

){}

}

// {{ userModel | json }} -- to display all data in object

// to bind properties of the model to the ngModel directives, for which we use

[ngModel]=”userModel.name” // do this for all fields

It was just ngModel earlier

// <option selected>I am interested in</option> change it as

<option value=””>I am interested in</option>

// Now we can see that all the fields on browser are filled with userModel object values.

// At this point if you try to change the name from ‘Rob’ to something else, only Angular object get the updated value, and the userModel is just the same way. Because we are doing only one way binding with [ngModel]

// while working with forms we need two way binding, to achieve this change [ngModel] to [(ngModel)] everywhere.

// For form validity, there are classes for Form Control based on its state and validity.

|  |  |  |
| --- | --- | --- |
| **State** | **Class if True** | **Class if False** |
| The control has been visited | ng-touched | ng-untouched |
| The control’s value has changed | ng-dirty | ng-pristine |
| The control’s value is valid | ng-valid | ng-invalid |

// not visited Form Control has - ng-untouched

// if visited by click or tab, and navigate away - ng-touched

// when the form loads its value is not yet changed - ng-pristine

// If the form control value changed – ng-dirty

// if the form control value is valid – ng-valid (if we have used required attribute and value is there)

// if form control value is invalid – ng-invalid (if we have used required attribute and no value)

// To check the current class applied to userName form control use - #name

Then - {{ name.className }} ,, this will print the class name.

// There are similar properties available in ngModel

|  |  |
| --- | --- |
| **Class** | **Property** |
| ng-untouched | Untouched |
| ng-touched | Touched |
| ng-pristine | Pristine |
| ng-dirty | Dirty |
| ng-valid | Valid |
| ng-invalid | Invalid |

// To access these ngModel properties modify #name as #name = “ngModel”

This is a reference to ngModel directives

Just ‘#name’ (template reference variable) points to the input element in the DOM.

// To see the changes in class modify {{ name.className }} to {{ name.untouched }} OR {{ name.pristine }}

OR {{ name.valid }}

// For conditional validation and to apply css classes to appear the error we can use class binding

[class.is-invalid]=”name.invalid && name.touched”

// meaning of above line - we want to apply is-invalid css class when name is visited but invalid

// So, if the name field is empty it will turn to red border as it’s a required field\

// Further in validation, let’s suppose we want to do pattern matching with phone field, use below .

#phone=”ngModel” pattern=”^\d{10}$” [class.is-invalid]=”phone.invalid && phone.touched”

// But, this is not enough, We also need to display proper error message statement below each field.

For which let’s use <small></small> tag after our ‘name’ field

<small [class.d-none]=”name.valid || name.untouched”>Name is required</small>

// The ‘d-none’ is the boothstrap4 class for display none.

// We can use the “errors” object we can display specific error messages, as we have used it with our phone field above.

\*ngIf=”phone.errors.pattern”

// For select field validation we need to think about custom validation as well on ‘blur’ & ‘change’ events.

validateTopic() -- to custom validate method. Specify it in ‘app.component.ts’

// The modified <select> control, with custom validation with ‘default’ will be –

<select (blur)=”validateTopic(topic.value)” (change)=”validateTopic(topic.value)” required #topic=”ngModel” [class.is-invalid]=”topicHasError && topic.touched” class=”custom-select” name=”topic” [(ngModel)]=”userModel.topic” >

<option value=”default”>I am interested in</option>

<option \*ngFor=”let topic of topics”>{{topic}}</option>

</select>

<small class=”text-danger” [class.d-none]=”!topicHasError || topic.untouched”>Please choose a topic</small>

// Form level validation

<button [disabled]=”userForm.form.invalid || topicHasError”

If the form is invalid, disable the submit button

// Submitting form data

novalidate -- use this with <form > tag. This will prevent browser validation when we do submit.

(ngSubmit) -- An event to submit the form, use at <form> tag

define ‘onSubmit()’ event handler at <form> tag

// first we hit on submit button which calls onSubmit() of app.component.ts, onSubmit() method makes a call to the enroll(this.userModel) method, with userModel which is our User object.

In enroll() method we make the actual HTTP request and send the data to the server.

Server can be any……

// Once the form is submitted, it’s advisable that to disable the submit button or atleast hide entire form as well.

// we have used “submitted” property in app.component.ts.

// now in <form> tag. <form \*ngIf=”!submitted”>

// now after submit form will be disappeared.

// right before the beginning of <form> tag add a <div> tag

<div class=”alert alert-danger” \*ngIf=”errorMsg”>{{errorMsg}}</div>

At app.component.ts –

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

import {User} from ‘./user’;

import {EnrollmentService} from ‘./enrollment.service’;

@Component({

selector: ‘app-root’,

templateUrl: ‘./app.component.html’,

styleUrls: [‘./app.component.css’]

})

export class AppComponent{

topics = [‘Angular’, ‘React’, ‘Vue’]; // for select dropdown in template

topicHasError = true;

submitted = false;

errorMsg = “”;

userModel = new User(‘Rob’,’rob@test.com’,5555666677,’’,’morning’,true);

// we have created custom object, which we can display on screen

// (‘Rob’,’rob@test.com’,5555666677,’default’,’morning’,true); -- for Select field add default.

constructor(private \_enrollmentService: EnrollmentService){}

validateTopic(value){

if(value === ‘default’){ // if in userModel we pass ‘default’ as a value for select

this.topicHasError = true;

}else{

this.topicHasError = false;

}

}

onSubmit(){

this.submitted = true;

console.log(this.userModel); // we can see the output at console after hitting submit button

// create a service ‘enrollment’, as enrollment.service.ts file

this.\_enrollmentService.enroll(this.userModel).subscribe(

data => console.log(‘Success!’,data),

//error => console.error(‘Error!’,error) // simple logging error

error => this.errorMsg = error.statusText // Modified error, assigning error statusText to the error property

)

}

}

At app.module.ts –

import {BrowserModule} from ‘@angular/platform-browser’;

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

import {FormsModule} from ‘@angular/forms’; // importing FormsModule

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

import {HttpClientModule} from ‘@angular/common/http’;

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule, FormsModule, HttpClientModule], // adding FormsModule

providers: [],

bootstrap: [AppComponent]

})

export class AppModule{ }

At enrollment.service.ts –

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

import {HttpClient, HttpErrorResponse} from ‘@angular/common/http’;

import {User} from ‘./user’;

import {catchError} from ‘rxjs/operators’; // Here we will be catching the errors from the server

import {throwError} from ‘rxjs’; // then throwing it to the subscribed component

@Injectable({

providedIn: ‘root’

})

export class EnrollmentService{

\_url = “”; // here we will post our form data after submission, create the ‘express’ server with ‘server.js’ and provide the value to \_url as - “http://localhost:3000/enroll”

constructor(private \_http: HttpClient){}

enroll(user:User){

return this.\_http.post<any>(this.\_url, user)

.pipe(catchError(this.errorHandler)) // the post request will return the response as observable, so we need to subscribe it at app.component.ts,, catchError is to handle errors

}

errorHandler(error: HttpErroResponse){

return throwError(error); // throwing back the error

}

}

// Setting up express server

// create a new folder outside somewhere with name ‘server’

// initialize a new pacakage.json file command - > npm init --yes

// let’s install the dependencies - > npm install --save express body-parser corse

// express -> web server

// body-parser -> is a middle ware to handle form data

// corse -> is a package to make requests across different imports

// After this all the dependencies are installed in the server folder, which we can see at package.json

// Now create a new file inside ‘server’ folder with name ‘server.js’

// in ‘server.js’ file –

const express = require(‘express’);

const bodyParser = require(‘body-parser’);

const cors = require(‘cors’);

const PORT = 3000; // port no. on which our server will run on

const app = express(); // an instance of express

app.use(bodyParser.json()); // specify body-parser to handle json data

app.use(cors()); // using cors package

app.get(‘/’, function(req, res){

res.send(‘Hello from server’);

})

// in app.post our angular will post it’s data.

app.post(‘/enroll’, function(req, res){

console.log(req.body); // the req.body contains user data which was submitted.

res.status(200).send({“message”: “Data received”}); // try by changing 200 to 401

// 401 is for Unauthorized.

})

app.listen(PORT, function(req, res){

console.log(“Server running on localhost:”+PORT);

});

// Now our basic server is ready, so run the command to launch it - > node server

// we can check from browser as well – just go for localhost:3000 and you will see “Hello from server”

// Now we can update the ‘\_url’ in enroll.service.ts file. \_url = <http://localhost:3000/enroll>;

// now when we will submit the form in browser console we can see – Data Received.

// in server terminal we will get user object

// we are also catching the error in the service then throw the error to the component that has subscribed to the service. In the Component assign the error status message to a property then bind it to the Html. So we will have an error message displayed.

1. **In Reactive form –**
2. We create the entire form control tree in the component class code.
3. There are 2 classes where we create the form control tree

FormGroup

FormControl

1. Basically, **FormGroup** is a collection of **FormControls**. But it can also have another FormGroup nested within it.
2. By definition FormGroup constructor has got **3 parameters**, First is required and other 2 are optional.
3. The First parameter is the collection of child controls.
4. Each child control is represented by **Key & Value** pairs.

Key : String (it accepts String type values)

Value : AbstractControl (it accepts AbstractControl type values)

1. In **create-employee.component.ts** file ->

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

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

**@Component({**

**selector: ‘app-create-employee’,**

**templateUrl: ‘./create-employee.component.html’,**

**styleUrls: [‘./create-employee.component.css’]**

**})**

**export class CreateEmployeeComponent implements OnInit{**

**employeeForm:FormGroup;**

**constructor(){}**

**ngOnInit(){**

**this.employeeForm = new FormGroup({**

**fullName: new FormControl(),**

**email: new FormControl()**

**});**

**}**

**}**

1. Both FormGroup & FormControl class are derived from AbstractControl class. So the inheritance relationship allow us to pass FomControl() as parameter to FormGroup constructor. -> ‘**fullName: new FormControl()**’
2. employeeForm is the property of FormGroup type. (FormGroup becomes datatype here)
3. ngOnInit() is the lifecycle hook
4. We have used 2 FormControls above for fullName & email
5. Key is fullName and Value is instance of FormControl()
6. Key is email and Value is instance of FormControl()
7. As we are using ReactiveformsModule we need to specify it in **app.module.ts** file

**import { ReactiveFormsModule } from ‘@angular/forms’;**

1. And add it inside imports array

**imports:[**

**BrowserModule,**

**AppRoutingModule,**

**ReactiveFormsModule**

**]**

1. Now at **create-employee.component.html** file

**<form [formGroup]=”employeeForm” (ngSubmit)=”onSubmit()”>** <!—formGroup is a directive, and employeeForm is a property --> <!—onSubmit() will be called once form get submitted -->

**<label for=”fullName”>Full Name</label>**

**<input id=”fullName” formControlName=”fullName” type=”text”>** <!—formControlName is a directive, and fullName is a string, So no need of square brackets -->

**<label for=”email”>Email</label>**

**<input id=”email” formControlName=”email” type=”text”>**

**<button type=”submit”>Save</button>**

**</form>**

1. By clicking on “Save” button a submit event is raised and this form get submitted.
2. Create ‘**onSubmit()**’ method in ‘**create-employee.components.ts**’ file

onSubmit(): void{

console.log(this.employee.value);

}

1. We are returning nothing so void, and by console.log we can see the form data in console.

**Using Properties in Form :::::::::::**

1. Both FormControl & FormGroup are derived from AbstractControl base class

**export declare class FormControl extends AbstractControl{……}**

**export declare class FormGroup extends AbstractControl{……}**

**export declare abstract class AbstractControl{……}**

1. AbstractControl has several properties

**value**

**errors**

**valid**

**invalid**

**dirty**

**pristine**

**touched**

**untouched**

**ngModelChange from Angular 6 :::::::::::**

1. Previously, below code works well :-

<input [(ngModel)]=”name” (ngModelChange)=”onChange($event)”>

onChange(value){

console.log(value); // would log updated value

}

1. Previously But with Handler, it was not correct :-

<input #modelDir=”ngModel” [(ngModel)]=”name” (ngModelChange)=”onChange(modelDir)”>

onChange(ngModel: NgModel){

console.log(ngModel.value); // would log old value, not updated value

}

1. Angular 6 onwards New change :-

onChange(ngModel: NgModel){

console.log(ngModel.value); // will log updated value

}

**ViewChild and ContentChild :::::**

1. Earlier ---- @ViewChild(‘foo’)foo:ElementRef;
2. From Angular8 ---- @ViewChild(‘foo’,{static:false}) foo: ElementRef;

**Unit Testing ::::**

1. Types of Tests :-
   1. Unit Tests (Job of Developers)
      1. Test a component in isolation, without external resources like database, file etc.
      2. Easy to Write
      3. Fast in execution
      4. Don’t test the functionality (gives less confidence)
   2. Integration Tests (Job of Developers)
      1. Test a component with external resources like - component & service together
      2. Gives better confidence than unit tests
   3. End-to-end Tests (Job of the QA team)
      1. Test the entire app as whole
      2. Test the complete functionality (give more confidence)
      3. Slow and fragile
   4. Ideal scenario (suggested way)
      1. Spend your more time to write Unit and Integration tests
      2. Write only few End-to-end tests for the key functionality.
2. Angular Testing Tools/frameworks :-
   1. Jasmine
      1. It is a behavior-driven development framework for testing JavaScript code.
      2. It doesn’t has any external dependency, And it doesn’t require the DOM.
   2. Karma
      1. It is a test runner, for writing and running unit tests.
      2. So whenever we write any test code it picks it up and execute it.
      3. Finally it will give us outcome of the test whether pass/fail.
   3. Protractor
      1. It is used to write and run End-to-end tests. (QA teams can make use of it).
   4. Angular Testing Utilities (Like :- TestBed)
      1. These utilities are provided by angular team itself, they help us create the test environment.
      2. Helps to test the various interactions, The interaction between component with service
3. How to write the unit tests :-
   1. Test file should have \*.spec.ts extension (Karma looks for these files only and runs them)
   2. Command To run test -- > ng test
   3. In order to write tests in angular we have to make use of 2 functions :- (both are part of Jasmine)
      1. describe()
      2. it()
   4. describe() -- > to define a suite of tests (means a group of related tests)

describe(‘suite-name’,function);

suite-name -- will always be the function name to be tested

function -- Here we use => arrow function to write ‘specs’ i.e. it()

So under one describe we can have all the tests written for that component together.

* 1. it() -- > for individual tests, Within 1 describe we can have 1 or more it() functions.

Each it() statement is a test case, which is known as ‘spec’.

it(‘spec-name’,function);

spec-name -- some description about the test, should explain the work of test

function -- Here we use => arrow function, implementation of actual test case

* 1. expect() -- > to check the result with toBe() function, toBe() is a matcher.

expect() takes actual result as parameter.

toBe() takes expected result as parameter.

Example-1 for numeric value :-

// Suppose we have a class file – compute.ts

export function compute(number){

if(number < 0){

return 0;

}

return number + 1;

}

// And it’s spec file would be – compute.spec.ts

import { compute } from ‘./compute’;

// test suite – group of related tests

describe(‘compute’,() => {

// spec or a test

it(‘should return 0 if input is negative’,() => {

const result = compute(-1);

expect(result).toBe(0);

});

it(‘should increment the input if is positive’,() => {

const result = compute(1);

expect(result).toBe(2);

});

});

Example-2 for text value :-

// The class file -- greet.ts

export function greet(name){

return ‘Hello, ’ + name;

}

// the spec file - greet.spec.ts

import { greet } from ‘./greet’;

describe(‘greet’,() => {

it(‘should include the name in the message’, () => {

const name = ‘hari’;

expect(greet(name)).toContain(name);

});

});

Example-3 for testing a class/service :-

// The class file -- city.service.ts

export class CityService{

cities = [`Bengaluru`, `Mumbai`, `Chennai`, `new delhi`];

getCities(){ return this.cities; }

addCity(city: string){ this.cities.push(city); }

}

// the spec file - city.service.spec.ts

import { CityService } from ‘./city.service’;

xdescribe(‘greet’,() => { // PUTTING x with describe, WILL NOT EXECUTE THIS describe by Karma.

it(‘should include the name in the message’, () => { // WE CAN PUT x WITH it AS WELL

const name = ‘hari’;

expect(greet(name)).toContain(name);

});

});

describe(‘CityService’,() => {

let cityService: CityService;

// runs before every test

beforeEach(() => { // runs before every it() execution

// set up

cityService = new CityService(); // As this instantiation is required for all specs, common code

});

// runs after every test

afterEach(() => {

//tear down

// perform clean up activity

});

it(‘should check if specified cities are present in the array’, () => {

const cities = cityService.getCities();

expect(cities).toContain(‘bengaluru’);

expect(cities).toContain(‘mumbai’);

});

it(‘should check if new city is added to the array ’, () => {

// Arrange

const newCity = ‘pune’;

// Act

cityService.addCity(newCity);

// Assert

expect(cityService.cities).toContain(newCity);

});

});

Example-4 for testing a component having dependency on a service :-

// in - products.component.ts

@Component({

Selector: ‘app-products’,

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

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

})

export class ProductComponent implements OnInit{

products: Product[] = [];

error: AppError;

constructor(private service: ProductsService){} // Product Component has dependency on this service.

ngOnInit(){

this.service.getProducts().subscribe( // here we are getting all the products list from service

// But while testing it, we have to put dummy data. We use spyOn() method

)

}

}

// in - products.component.spec.ts

describe(‘ProductsComponent’, () => {

let component: ProductsComponent;

let service: ProductsService;

beforeEach(() => {

service = new ProductsService(null);

component = new ProductsComponent(service);

});

it(‘should set products property with the items returend’, () => {

// Arrange

const products: Product[] = [ // Fake Products array

{

id: 1,

name: ‘p1’,

description: ‘p1 description’,

price: 10,

isAvailable: true

},

{

id: 2,

name: ‘p2’,

description: ‘p2 description’,

price: 20,

isAvailable: false

},

{

id: 3,

name: ‘p3’,

description: ‘p3 description’,

price: 30,

isAvailable: true

}

];

spyOn(service, ‘getProducts’).and.callFake(() => {

return Observable.from([products]);

});

// spyOn is looking whether the service has been called, and also if the particular method is called like – getProducts() from service class.

// If yes, then provide fake implementation for it with Observables.\

// This will avoid the interaction of the Component and service.

// Instead of above Fake method we can have plain Observable as below –

// spyOn(service,’getProducts’).and.returnValue(Observable.from([products]));

// Whenever the returnValue is called, an Observable of Products will be return.

component.ngOnInit(); // Act, Make the actual call

expect(component.products).toEqual(products); // Assert

});

it(‘should set the error property if server returns an error when getting products’, () => {

const error = new AppError(‘server error’); // for AppError there is a ‘app-error.ts’ file

spyOn(service, ‘getProducts’).and.returnValue(Observable.throw(error)); // Throwing an error

expect(component.error).not.toBeDefined();

component.ngOnInit();

expect(component.error).toBeDefined();

expect(component.error.originalError).toEqual(‘server error’);

});

})

// in app-error.ts file –

export class AppError{

constructor(public originalError? : any){}

}

// The testing like above is known as Arrange-Act-Assert pattern.

// There are other methods also available like – beforeAll() & afterAll()

// Spies –

spyOn(object,’methodName’).and.callFake(function)

* The function should match ‘methodName’ signature

spyOn(object,’methodName’).and.returnValue(observable)

// Observables we can use -

Observable.from([array]) -- to return the Object Array

Observable.empty() -- just to make sure the function is called, and not worried about what it returns.

Observable.throw(error)

// different expects we can use -

expect(spy).toHaveBeenCalled()

expect(spy).toHaveBeenCalledWith(id)

1. ng test --code-coverage -> Command to create Code-Coverage report in folder : apps/store-app/coverage

Open the index.html file in there to see the report details.

1. Angular testing Utilities – TestBed

Used to test interaction between a component and its template OR

1 component with another Component

It’s from package - @angular/core/testing

1. D
2. d