**Angular (additional)**

1. **Component Selector**

We can use component as attribute by just changing selector of component

@**Component**({

selector: '[app-servers]',

template: '<div app-servers></div>’, // component as attribute

styleUrls: ['./servers.component.css']

})

We can use component as class by just changing selector of component

@**Component**({

selector: '.app-servers',

templateUrl: './servers.component.html',

styleUrls: ['./servers.component.css']

})

Html:

<div *class*="container">

<div *class*="row">

<div *class*="col-xs-12">

<h3>In App component</h3>

<hr>

*<!-- <app-servers></app-servers> -->*

<div *class*="app-servers"></div>

</div>

</div>

</div>

1. **Data binding**
2. Output Data ( Typescript => Html )

String Interpolation => {{data}}

Property Binding => [property] = “data”

1. React to Events (Typescript <= Html)

Event Binding => (event) = “expression”

1. Tow way Binding

[(event)] = ”data”

1. **ngIf**

ngIf use to manipulate DOM so and its Structural directive so we need have to use \* along with it.

<p \**ngIf*="boolean\_value">Text to display</p>

ngIf with else:

<p \**ngIf*="isTrue; else other\_content">If Text to display</p>

<ng-template #*other*\_content>Else text to display</ng-template>

1. **ngStyle**

For styling component dynamically with property binding we use this directive. Unlike structural directive it won’t add or remove element. It will only add styling elements to the components.

Html:

<p [*ngStyle*] = "{backgroundColor : getColor()}" > server is {{serverState}} </p>

Component:

**getColor**(){

return this.serverState == 'online' ? 'green':'red';

}

1. **ngFor with index**

<p \**ngFor* = "let server of serverCounts; let i = index">

{{i+1}} ) {{server}}

</p>

1. **Custom data binding. (Input to child component):**

**app component:**

<app-server-element \*ngFor="let serverElement of serverElements" [srvElement]="serverElement"> </app-server-element>

**Server-element.ts:**

export class ServerElementComponent implements OnInit {

@Input('srvElement') element:{type:string, name:string, content:string}

constructor() { }

ngOnInit() {}

}

**Server-element-html:**

<div class="panel panel-default">

<div class="panel-heading">{{ element.name }}</div>

<div class="panel-body">

<p>

<strong \*ngIf="element.type === 'server'" style="color: red">

{{ element.content }}</strong>

<em \*ngIf="element.type === 'blueprint'">{{ element.content }}</em>

</p>

</div>

</div>

1. **Custom data binding. (Output to Parent component):**

For sending output to the parent component we need to use event and @Output decorator

**App.component.ts:**

<app-cockpit

(serverCreated)="onServerAdded($event)"

(bluePrintCreated)="onBlueprintAdded($event)"></app-cockpit>

Here **serverCreated,** **bluePrintCreated** are events created using **EventEmmiter** in **Cockpit component** which are binded with “**onServerAdded($event)**" and "**onBlueprintAdded($event)**". Here, **$event** is data output received from child component passed to parent component as data.

**Cockpit.component.ts:**

export class CockpitComponent implements OnInit {

@Output() serverCreated=new EventEmitter<{serverName:string, serverContent:string, serverType:string}>();

@Output() bluePrintCreated=new EventEmitter<{serverName:string, serverContent:string, serverType:string}>();

newServerName = '';

newServerContent = '';

onAddServer(){

this.serverCreated.emit(

{serverName:this.newServerName,

serverContent:this.newServerContent,

serverType:'server'}

)

}

onAddBlueprint(){

this.bluePrintCreated.emit(

{serverName:this.newServerName,

serverContent:this.newServerContent,

serverType:'blueprint'}

)

}

}

**Cockpit.component.html:**

<div class="row">

<div class="col-xs-12">

<p>Add new Servers or blueprints!</p>

<label>Server Name</label>

<input type="text" class="form-control" [(ngModel)]="newServerName">

<label>Server Content</label>

<input type="text" class="form-control" [(ngModel)]="newServerContent">

<br>

<button

class="btn btn-primary"

(click)="onAddServer()">Add Server</button>

<button

class="btn btn-primary"

(click)="onAddBlueprint()">Add Server Blueprint</button>

</div>

</div>

**App.component.ts:**

export class AppComponent {

serverElements = [{type:'server',name:'testServer', content:'Just a test server'}];

onServerAdded(ServerData:{serverName:string, serverContent:string, serverType:string}) {

this.serverElements.push({

type: ServerData.serverType,

name: ServerData.serverName,

content: ServerData.serverContent

});

}

onBlueprintAdded(bluePrintData:{serverName:string,serverContent:string,serverType:string}) {

this.serverElements.push({

type: bluePrintData.serverType,

name: bluePrintData.serverName,

content: bluePrintData.serverContent

});

}

}

1. **View Encapsulation**

In default behaviour of angular css belong to any component is applicable for that component only. Not to the other component its style are applicable. For enabling global apply of that css. We need to add ViewEncapsulation in decorator of that component (google it).

1. **Projecting content into component**

We can inject html content in component and use it as modular component. Use to create dynamic widget which will change as injected. Like tab or anything.

**App.component.html:**

<app-server-element \*ngFor="let serverElement of serverElements"

[srvElement]="serverElement">

<strong \*ngIf="serverElement.type === 'server'"

style="color: red">{{ serverElement.content }}</strong>

<em \*ngIf="serverElement.type === 'blueprint'">{{ serverElement.content }}</em>

</app-server-element>

**Server-element.component.ts:**

<div class="panel-body">

<p>

<ng-content></ng-content>

</p>

</div>

1. **Local referannce in templet (without ngModel):**

We can use local reference which can then used in component as data.

**Html:**

<input type="text" class="form-control" #serverName>

<button class="btn btn-primary" (click)="onAddServer(serverName)">Add Server</button>

**Component:**

onAddServer(serverName){ // we can decleare serverName as HtmlInputElement as type

console.log(serverName.value);

}

1. **Attribute vs Structural Directive**

**Attribute**: used like normal attribute of html element. (Property binding or event binding eg. [ngClass], [ngStyle]).

**Structural (\*)**: used to manipulate DOM adding or removal of html element. (\*ngIf, \*ngFor)

1. **Custom Directive**

**Directive.ts:**

@Directive({

selector: '[appBasicHighlight]',

})

export class BasicHighlightDirective implements OnInit {

constructor(private elementRef: ElementRef) {

}

ngOnInit(): void {

this.elementRef.nativeElement.style.backgroundColor = 'green';

}

}

**At consuming Html:**

<p appBasicHighlight>Styled with my basichightlight directive</p>

Also we need to register this directive in **app.module.ts.**

@NgModule({

declarations: [

BasicHighlightDirective

]

For generating directive we can use ng generate directive [name] or ng g d [name] instead of manual method defined above.

Different types of directive implementation method :

**1) Using Rendered2 service.**

**2) @HostListener to listen host events (like mouse over and other to change something)**

**3) @HostBinding to bind to Host Properties. (bind specific property to data member and simply change it)**

**4) Binding to directive properties (inject property from html element using @input to datamember):**

<p appBetterHighlight [backgroundColor]="'red'" [defaultColor]="'yellow'">Styled with my betterhightlight directive</p>

1. **Working of structural directive with asterisk(\*) :**

Structural directives are base on the <template> tag.

The \* before the attribute selector indicates that a structural directive should be applied instead of a normal attribute directive or property binding. Angular2 internally expands the syntax to an explicit <template> tag.

<!-- Examples (A) and (B) are the same -->

<!-- (A) \*ngIf paragraph -->

<p \*ngIf="condition">

Our heroes are true!

</p>

<!-- (B) [ngIf] with template -->

<template [ngIf]="condition">

<p>

Our heroes are true!

</p>

</template>

1. **Services (Dependency Injection):**

Services are injected in component or service using constructor.

Services are simple .ts class without having any decorator over it. (yes, its correct no need of @Injectable).

export class CounterService{

}

**@Injectable** – is used when we want to use service into service. Consumer service should be defined as @Injectable and consumable service will be simple .ts class

1. **Inter component communication (Event emmiter):**

**Recipe-list.html:**

<div class="row">

<div class="col-md-12 ">

<app-recipe-item \*ngFor="let recipeEl of recipes" [recipe]="recipeEl" (click)="recipeDetail(recipeEl)"></app-recipe-item>

</div>

</div>

**Recipelist.component.ts:**

export class RecipeListComponent implements OnInit {

recipes : recipe[];

constructor(private rs:RecipeService) { }

ngOnInit() {

this.recipes = this.rs.getRecipesList();

}

recipeDetail(recipeEl : recipe){

this.rs.selectedRecipe.emit(recipeEl);

}

}

**Recipe.component:**

@Component({

selector: 'app-recipes',

templateUrl: './recipes.component.html',

styleUrls: ['./recipes.component.css'],

providers:[RecipeService]

})

export class RecipesComponent implements OnInit {

recipeEl:recipe;

constructor(private rec :RecipeService) { }

ngOnInit() {

// subscribe will keep watch on changing content return by service.

this.rec.selectedRecipe.subscribe((recipe:recipe)=>{

this.recipeEl=recipe;

});

}

}

**Recipe.html**

<div class="col-md-5">

<app-recipe-list ></app-recipe-list>

</div>

<div class="col-md-7">

<app-recipe-details \*ngIf="recipeEl; else noRecipeSelected" [recipeElement]="recipeEl"></app-recipe-details>

<ng-template #noRecipeSelected>Select Recipe First</ng-template>

</div>

**Recipe.service:**

import { recipe } from "./recipe.model";

import { EventEmitter } from "@angular/core";

export class RecipeService {

private recipes : recipe[] = [

new recipe('test recipe 1','simple rec 1',

'https://i.ytimg.com/vi/GjwxuQqT\_yg/maxresdefault.jpg'),

new recipe('test recipe 2','simple rec 2',

'https://i.ytimg.com/vi/GjwxuQqT\_yg/maxresdefault.jpg')

];

selectedRecipe:EventEmitter<recipe> = new EventEmitter<recipe>();

getRecipesList(){

return this.recipes.slice(); // to pass duplicate copy not original

}

}

1. **Spread Syntax (ecmascript feature)**  
   allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

function myFunction(v, w, x, y, z) { }

var args = [0, 1];

myFunction(-1, ...args, 2, ...[3]);

we can push element set as it is in array without iteration by this operator :

addIngredients(ings:Ingredient[]){

this.ingredients.push(...ings);

}

1. **Router navigation absolute path:**

If we are on path localhost:4200/servers

For navigation in routerLink of we use any route. like, servers as **routerLink = ‘servers’** it will append to current url with /server eg . localhost:4200/servers/servers

If we user **routerLink = ‘/servers’** then it will append route path to localhost eg. localhost:4200/servers

1. **Styling active directive:**

For making nav bar or tab active on router add routerLinkActive=”<class\_name>”

<li role="presentation" routerLinkActive="active">

<a routerLink="/servers">Servers</a>

</li>

<li role="presentation" routerLinkActive="active">

<a routerLink="/users">Users</a>

</li>

1. **Route Parameter:**

For using route parameter we need to define route with :<anything> followed by URI.

{ path: 'user/:id', component: UserComponent }

For using this in component we need to add service in constructor

constructor(private route: ActivatedRoute) {} //import it from angular @Angular/router

for using it :

console.log(this.route.snapshot.params['id']))

1. **Outsource route configuration:**

In this method we will be having separate routing file **app-routing.module.ts** which will have all routs of application.

**app-routing.module.ts:**

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

// imports of all other components defined in appRoutes variable.

const appRoutes: Routes = []

@NgModule({

imports: [

RouterModule.forRoot(appRoutes)

// RouterModule.forRoot(appRoutes, { useHash: true})

],

exports: [

RouterModule

]

})

export class AppRoutingModule {

}

For using this globally we need to use **RouterModule** class from **@angular/router** inside @NgModule .

Also for using this module we need to import it into main module for this we need to use property of @ngModule ie. **exports** array. And import it in **app.module.ts**

**app.module.ts :**

import { AppRoutingModule } from './app-routing.module';

@NgModule({

declarations: [],

imports: [ AppRoutingModule ],

providers: [],

bootstrap: []

})

1. **Redirecting route :**

Inject router service in constructor:

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

for using it

this.router.navigate(['new'], {relativeTo:this.route});

1. **Guard ( canActivate, canActivateChild ) :**

Guard is code statement execute before routing happen. **CanActivate** interface help us to achieve this.

**Auth-guard.service.ts :**

export class AuthGuardService implements CanActivate {

canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean | Observable<boolean> | Promise<boolean> {

//logic for validating condition.

return true; } //canActive() only validates parent route.

canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean | Observable<boolean> | Promise<boolean> {

//logic for validating condition.

return true; } //canActiveChild() only validates child routes.

}

Add providers in app.module.ts:

providers: [AuthGuardService]

apply this guard to parent as well as child by this way:

**app-routing.module.ts:**

const appRoutes: Routes =

{

path: 'servers',

// canActivate: [AuthGuardService],

canActivateChild: [AuthGuardService],

component: ServersComponent,

children: [{ path: ':id', component: ServerComponent, resolve: { server: ServerResolver}]

}

1. **Observable :**

This is an interface to handle a variety of common asynchronous operations like :

* The EventEmitter class extends Observable.
* The HTTP module uses observables to handle AJAX requests and responses.
* The Router and Forms modules use observables to listen for and respond to user-input events.

**Note : Angular uses RxJs for using Observable.**

1. **Simple observable:**

import { Observable,interval } from 'rxjs';

export class HomeComponent implements OnInit {

ngOnInit() {

const myNumber = interval(1000); //created observable

myNumber.subscribe((data:number)=>{ // observer subscription

console.log(data);

});

}}

1. **Binding and using custom observable :**

import { Observable,interval, Observer } from 'rxjs';

ngOnInit() {

const myObservable = Observable.create((observer:Observer<string>)=>{

setTimeout(()=>{observer.next('first call');},1000);

setTimeout(()=>{observer.next('second call');},2000);

setTimeout(()=>{observer.error('Error Occured');},5000);

});

myObservable.subscribe(

(data:string)=>{console.log(data);},

(error:string)=>{console.log(error)}

);

}

1. **Unsubscribe observable :**

It is necessary to unsubscribe the subscribed observable as it can be run asynchronously and can create serious memory leak issue. And by using unsubscribe scope of observable is limited to that component only.

import { Observable,interval, Observer, Subscription } from 'rxjs';

export class HomeComponent implements OnInit, OnDestroy {

custMyObservable : Subscription;

ngOnInit() {

const myNumber = interval(1000); //created observable

this.custMyObservable = myNumber.subscribe((data:number)=>{ // observer subscription

console.log(data);

});

}

ngOnDestroy() {

this.custMyObservable.unsubscribe();

}

}

1. **Subject in angular (same like observable):**

It can be used as replacement of event emitter in angular.

Service.ts :

import { Subject } from "rxjs";

export class userActivatedService{

userActivated = new Subject();

}

Initializing :

onActivate(){

this.useractivatedservice.userActivated.next(this.id);

}

Consuming :

ngOnInit() {

this. useractivatedservice.userActivated.subscribe((id:number)=>{

console.log(id);

});

}

1. **Operators with observable:**

These are functions that build on the observables foundation that enables sophisticated manipulation of collections. For example, RxJS defines operators such as **map() –** convert into array**, filter() –** gives boolean value by condition**, concat(),** and **flatMap().**

const squareValues = map((val: number) => val \* val);

You can use pipes to link operators together. Pipes let you combine multiple functions into a single function.

const squareOddVals = pipe(

filter((n: number) => n % 2 !== 0),

map(n => n \* n) ); // equivalent to (n:number) => return n\*n

**Here in following example of interval() we have to use Pipe() intentionally even it is having only one operator :**

const myNumber = interval(1000)

.pipe(map((n : number) => n\*2)); // equivalent to (n => n \* n)

With pipe

const myNumber = interval(1000)

.pipe(

map(n => n),

filter(n => n%3===0)

);

1. **Angular Form : Template Driven**

Here angular helps to create object from form elements.  
For using this we need to have **‘FormModule’** imported from **‘@Angular/forms’** in **‘app.module.ts’**.

**ngModel** directive is helps to register a control in form.

Html:

<form (ngSubmit)="onSubmit(f)" #f="ngForm">

<input type="text" id="username" class="form-control" ngModel name="username">

</form>

Component.ts

import { NgForm } from '@angular/forms';

export class AppComponent {

onSubmit(form : NgForm){

console.log(form);

}

}

**Accessing form using @ViewChild:** This is used to access the form before submitting in some scenario.

Html:

<form (ngSubmit)="onSubmit()" #f="ngForm">

<input type="text" id="username" class="form-control" ngModel name="username">

</form>

Component.ts

import { NgForm } from '@angular/forms';

export class AppComponent {

@ViewChild('f')

applicationform : NgForm;

onSubmit(){

console.log(this.applicationform);

}

}

1. **Validation by angular (Template Driven):**

Angular helps to validate form. If its validated **ngForm objects ‘valid’ property will be true or false.**

Eg.

**required**, **email** these are sample validators others are at [**https://angular.io/api?type=directive**](https://angular.io/api?type=directive)

**form state :**

we can decide to perform any operation on forms state as follows :

html:

<form (ngSubmit)="onSubmit()" #f="ngForm">

<input type="email" id="email" class="form-control" ngModel

name="email" required email #email="ngModel">

<span class="help-block" \*ngIf="!email.valid && email.touched">Invalid email</span>

<button class="btn btn-primary" type="submit" [disabled] = "!f.valid">Submit</button>

</form>

Css:

input.ng-invalid.ng-touched{

border : 1px solid red;

}

1. **Reactive forms :**

More control over html components then TD forms.

We have to use **ReactiveFormModule from @angular/forms** instead of **FormsModule** (it is used for template driven forms).

We need to map html component to the **formcontrol** **from @angular/forms**

For validators we need to use **validators from @angular/forms**

**Html:**

<form [formGroup]="signupForm">

<input type="text" id="username"class="form-control" formControlName="username">

<span \*ngIf="!signupForm.get('username').valid &&

signupForm.get('username').touched" class="help-block">Enter Username </span>

<input type="text" id="email" class="form-control" formControlName="email">

<span \*ngIf="!signupForm.get('email').valid &&

signupForm.get('email').touched" class="help-block">Enter valid Email </span>

<div class="radio" \*ngFor="let gender of genders">

<label>

<input type="radio" name="gender" [value]="gender"

formControlName="gender">{{ gender }}

</label>

</div>

<button class="btn btn-primary" type="submit"

(click)="formSubmit()">Submit</button>

</form>

**formControlName** directive use to register html control in **FormGroup.**

**Css:**

input.ng-touched.ng-invalid{

border: 2px solid red;

}

**Component.ts:**

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

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

export class AppComponent implements OnInit {

genders = ['male', 'female'];

signupForm : FormGroup;

ngOnInit() {

this.signupForm = new FormGroup({

'username': new FormControl(null, Validators.required),

'email' : new FormControl(null, [Validators.required, Validators.email]),

'gender' : new FormControl('male'),

});

}

formSubmit(){

console.log(this.signupForm);

}

}

Here **FormGroup** use to create reactive form.

**FormControl() constructor** has initial state (value) and validator or validators array as an argument.

1. **Custom Validator :**

**Component.ts :**

ngOnInit() {

this.signupForm = new FormGroup({

'username': new FormControl(null, [Validators.required,

**this.forbiddenNames.bind(this)**]) //bind() helps angular to find method from outside

});

}

forbiddenNames(control : FormControl):{[s:string]:boolean}{

console.log(this.forbiddenUserNames.indexOf(control.value));

if(this.forbiddenUserNames.indexOf(control.value) !== -1){

return {'nameIsForbidden':true}

}

return null; // we cannot return {'nameIsForbidden':false}

} // if validation fails only 'null' for fail validation

**For using this error code (optional):**

We can use specific ngIf statement for particular error as follows:

Html:

<span \*ngIf="!signupForm.get('username').errors['nameIsForbidden']" class="help-block">

Invalid User Name

</span>

1. **Pipes :**

Pipe use to transform the output rendered on view eg. **uppercase** , **date**

<strong>{{ server.name }}</strong>

<strong>{{ server.name | uppercase}}</strong>

Here , **uppercase** is pipe userd with ‘**|**’ symbol pipe as separator.

**Parameterized pipes:**

We can assign parameter to pipes

We can separate multiple parameter If pipe support that :

**{{ property | pipename : ’parameter1’ : ’parameter2’ }}**

Eg.

{{ server.started | date:'fullDate'}}

We can explore more angular pipes on **angular.io>>docs and search for pipe.** With parameters examples and shortcuts.

**Using multiple pipes (chaining) :**

{{ server.started | date:'fullDate' | uppercase }}

Here, pipe order is very important. Pipe execution sequence is left 🡪 right.

If we use uppercase on date datatype (as server.started is date type) directly we will end up getting error as follows

{{ server.started | uppercase | date:'fullDate' }} *//Error*

1. **Custom Pipe:**

shortenpipe.pipe.ts :

import { PipeTransform, Pipe } from "@angular/core";

@Pipe({

name:'shorten'

})

export class ShortenPipe implements PipeTransform {

transform(value: String) {

if(value.length>10){

return value.substr(0,10) + " ...";

}

return value;

}

}

Now we need to register this in app.module.ts :

@NgModule({

declarations: [

AppComponent,

**ShortenPipe**

],

……. })

On view side use it as :

<strong>{{ server.name | uppercase | shorten }}</strong>

**\* Custom parameterized pipe :**

shortenpipe.pipe.ts :

import { PipeTransform, Pipe } from "@angular/core";

@Pipe({

name:'shorten'

})

export class ShortenPipe implements PipeTransform {

transform(value: String, limit:number) {

if(value.length>limit){

return value.substr(0,limit) + " ...";

}

return value;

}

}

On view side use it as :

<strong>{{ server.name | uppercase | shorten:10 }}</strong>

We can create pipe using command also **ng g p filter**

1. Asd
2. Asd
3. Asd
4. Fdg
5. Hjbk
6. Jnk