ANGULAR

Build Angular applications with components.

Components define areas of responsibility in the UI that let you reuse sets of UI functionality.

A component class 🡺 Handles data and functionality

An HTML template 🡺 Determines the UI

Component-specific styles 🡺 Define the look and feel

ng new prjname –no-strict

no routing & css

examples

npm install --save bootstrap@4

angular.json 🡺 architect -> styles -> "node\_modules/bootstrap/dist/css/bootstrap.min.css"

===============================

**COMPONENTS**

App component 🡺 binds together everything 🡺 root / starting point

New project 🡺 server details display

new folder under app 🡺 server

create new **server** component 🡺 gets registered in app module

overwrite app component.html

<h2> App Component</h2>

<hr>

<app-server></app-server>

create new **servers** component 🡺 gets registered in app module

change the server app component.html

<h2> App Component</h2>

<hr>

<app-servers></app-servers>

change the server app component.html

<p>servers works!</p>

<app-server></app-server>

<app-server></app-server>

<app-server></app-server>

<app-server></app-server>

using bootstrap in designs 🡺 change app comp.html

<div class="container">

<div class="row">

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

<app-servers></app-servers>

</div>

</div>

</div>

SELECTING A COMPONENT

By TAG

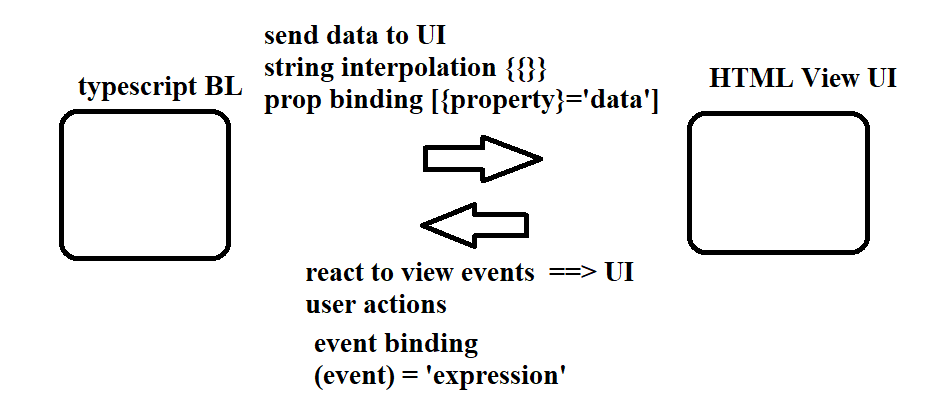
By ATTRIBUTE []

By CLASS .

By id or any other option won’t work

Assignment

**DATA BINDING**



**ADDING BL & DISPLAYING IN UI / VIEW 🡺 STRING INTERPOLATION {{ }}**

Server comp class

serverId:number = 1001; serverStatus:string = 'Offline';

getServerId()

{

return this.serverId;

}

getServerStatus()

{

return this.serverStatus;

}

Server comp html

<p>{{ 'server with number'}} {{ serverId }} is currently {{ serverStatus }} </p>

<p>{{ 'server with number'}} {{ getServerId() }} is currently {{ getServerStatus() }} </p>

**ADDING BL AND DISPLAYING IN UI / VIEW 🡺 PROPERTY BINDING [ ]**

Servers html

<button class="btn btn-primary" disabled> Add New Server</button>

Servers comp class

addNewServer:boolean = false;

constructor()

{

setTimeout(() => {

this.addNewServer = true;

}, 5000);

}

Servers html

<button class="btn btn-primary" [disabled] = "!addNewServer"> Add New Server</button>

<p> {{ addNewServer }} with string interpolation </p>

<p [innerText]="addNewServer" > with property binding</p>

Serving will enable the button after 5 seconds

**REACT TO USER EVENTS FROM UI 🡺 EVENT BINDING ( )**

Servers comp class

serverCreated : string = "Not yet";

new method

onServerCreation()

{

this.serverCreated = "Created";

}

Servers html

<button class="btn btn-primary" [disabled] = "!addNewServer" click)="onServerCreation()">

Add New Server</button>

<p> {{ serverCreated }} </p>

**EVENT MODULE IN EVENT BINDING**

Servers html

<label>Server Name : </label>

<input type="text" class="form-control" (input)="onServerNameEntry($event)"> <br> <br>

<p> {{ serverName }} </p>

Servers comp

serverName : string = '';

onServerNameEntry(event:Event)

{

//console.log(event);

this.serverName = (<HTMLInputElement>event.target).value;

}

**Two-Way-Binding 🡺 Combine property and event binding**

Add the FormsModule to the imports[] array in the AppModule.

Add the import from @angular/forms in the app.module.ts file:

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

FormsModule in import section

[( ngModel )]

<input type="text" [(ngModel)]="serverName" > <br> <br>

**DIRECTIVES**

Directives are classes that add additional behaviour to elements in Angular applications. Define custom directives to attach custom behaviour to elements in the DOM.

To Display the server created or not status message

Servers html

<p \*ngIf="serverCreated=='Created'"> {{ serverCreated }} </p>

**STRUCTURAL DIRECTIVE** TO CHANGE DOM DYNAMICALLY 🡺 \*

1. Ngif with else 🡺 uses ng-template

<p \*ngIf="serverCreated=='Created'; else beforeServer"> {{ serverCreated }} </p>

<p> {{ serverName }} </p>

<ng-template #beforeServer>

<p>{{ serverCreated }}</p>

</ng-template>

1. ngFor 🡺 Dynamically change component lists thro loop

servers comp

serversList = ['S1', 'S2'];

onServerCreation()

{

this.serverCreated = "Created";

*this.serversList.push(this.serverName);*

}

Servers html

<app-server \*ngFor="let server of serversList"></app-server>

Accessing index position in ngFor

Server comp

serverNumbers = [101, 102, 103];

Server html

<div \*ngFor="let num of serverNumbers; let i = index"

[ngStyle]="{backgroundColor : i == 0 ? 'lightblue' :''}" >

<p > server with {{ num }} {{ getServerId() }} is currently {{ getServerStatus() }} </p>

</div>

1. ngSwitch

<container\_element [ngSwitch]="switch\_expression">

<inner\_element \*ngSwitchCase="match\_expresson\_1">...</inner\_element>

<inner\_element \*ngSwitchCase="match\_expresson\_2">...</inner\_element>

<inner\_element \*ngSwitchCase="match\_expresson\_3">...</inner\_element>

<inner\_element \*ngSwitchDefault>...</element>

</container\_element>

**ATTRIBUTE DIRECTIVES CHANGE THE ELEMENT WHERE THEY ARE USED 🡺 [ ]**

1. ngStyle can be changed into an attribute 🡺 [ngStyle]

server comp

constructor()

{

this.serverStatus = Math.random() > 0.5 ? 'Online' : 'Offline';

}

server html

<p [ngStyle]="{'background-color' :'red'}" >server with number {{ getServerId() }} is currently {{ getServerStatus() }} </p>

server comp

getColor()

{

return this.serverStatus == 'Online' ? 'green' : 'red';

}

server html

<p [ngStyle]="{'background-color' :getColor()}" >server with number {{ getServerId() }} is currently {{ getServerStatus() }} </p>

OR

<p [ngStyle]="{backgroundColor :getColor()}" >server with number {{ getServerId() }} is currently {{ getServerStatus() }} </p>

1. ngClass 🡺 dynamically adds or removes CSS classes to HTML elements checked in elements of Dev tools

server css

.online

{

color: white;

}

Server html

<p [ngStyle]="{'background-color' :getColor()}"

[ngClass]="{ online : serverStatus =='Online' }" >

server with number {{ getServerId() }} is currently {{ getServerStatus() }} </p>

PIPES

Pipes are used to transform data on a template, without writing a boilerplate code in a component.

To transform data, we write the code in the component, For example, we want to transform any date into a format like '16 Apr 2018' or '16-04-2018', We need to write separate code in the component.

So instead of writing separate boilerplate code, we can use the built-in pipe called DatePipe which will take input and transform it into the desired date format.

{{today **|** date : ‘fullDate’}}

today is the component variable, which specifies the current date.

date represent DataPipe

fullDate is an optional parameter or argument which specifies the date format.

O/P Monday, April 16, 2018

**Built-In Pipes**

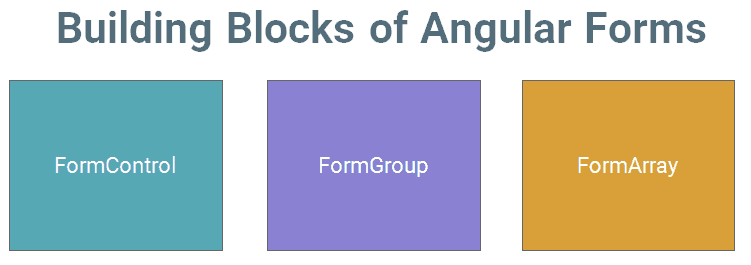
Angular comes with a collection of built-in pipes such as DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, DecimalPipe, and PercentPipe.

**Angular Forms**

The data entry forms can be very simple to very complex. It can contain large no of input fields, Spanning multiple tabs. Forms may also contain complex validation logic interdependent on multiple fields.

Some things forms are expected to do

* Initialize the forms fields and present it to the user
* Capture the data from the user
* Track changes made to the fields
* Validate the inputs
* Display helpful errors to the user



FormControl

A FormControl represents a single input field in an Angular form. The FormControl is just a class.

First Name : <input type="text" name="firstname" />

FormControl 🡺 object 🡺 encapsulates all information related to the single input element. It Tracks the value and validation status of each of these control

A FormControl object is created for each form field. Refer 🡺 component class and inspect its properties and methods (valid/invalid, pristine/dirty, touched/untouched ) etc & add validation rules to it.

let firstname= new FormControl(); //Creating a FormControl in a Reactive forms

firstname.value //Returns the value of the first name field

firstname.errors // returns the list of errors

firstname.dirty // true if the value has changed (dirty)

firstname.touched // true if input field is touched

firstname.valid // true if the input value has passed all the validation

FormGroup

FormGroup 🡺 collection of FormControls .

Each FormControl 🡺 property in a FormGroup. with the control name 🡺 key.

Often forms have more than one field. Simple way to manage the Form controls together.

address

city : <input type="text" name="city" >

Street : <input type="text" name="street" >

PinCode : <input type="text" name="pincode" >

FormGroup 🡺 wrapper interface around a collection of FormControls.

FormGroup tracks the status of each child FormControl and aggregates the values into one object. with each control name as the key

let address= new FormGroup (

{

street : new FormControl(""),

city : new FormControl(""),

pinCode : new FormControl("")

}

)

address 🡺 FormGroup, consisting of 3 Form Controls 🡺 single value of FG 🡺 address.value

Return value

address {

street :"",

city:"",

Pincode:""

}

can access child control as

address.get("street")

address.errors // returns the list of errors

address.dirty // true if the value of one of the child control has changed (dirty)

address.touched // true if one of the child control is touched

address.valid // true if all the child controls passed the validation

The Angular form is itself a FormGroup

FormArray

FormArray 🡺 array of form controls.

Difference.

FormGroup 🡺 each FormControl is a property with the control name as the key.

FormArray 🡺 array of form controls.

cities : new FormArray

(

[

new FormControl('Mumbai'),

new FormControl('Delhi')

]

);

cities() : FormArray {

return this.contactForm.get("cities") as FormArray

}

cities.errors // returns the list of errors

cities.dirty // true if the value of one of the child control has changed (dirty)

cities.touched // true if one of the child control is touched

cities.valid // true if all the child controls passed the validation

**Template-driven form & Model-driven forms(Reactive)**

Template Driven Forms 🡺 specify behaviours/validations using directives and attributes in our template. All things happen in Templates hence very little code is required in the component class. This is different from the reactive forms, where we define the logic and controls in the component class.

The Template-driven forms

The form is set up using ngForm directive

controls are set up using the ngModel directive

ngModel also provides the two-way data binding

The Validations are configured in the template via directives

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

add html code and class in component

ngForm

Form 🡺 form elements 🡺 angular automatically converts it into a Template-driven form. This is done by the ngForm directive. 🡺 Angular adds it automatically

Include FormsModule 🡺Angular 🡺look out for any <form> tag 🡺HTML template 🡺 ngForm directive 🡺 automatically detects the <form> tag and automatically binds to it.

Binds itself to the <Form> directive, Creates a top-level FormGroup instance,

CreatesFormControl instance for each of child control, which has ngModel directive.