Angular 4.? CLI Notes

# CLI Notes

## Install Node JS

From Node JS command window make sure you have latest Angular CLI installed

npm install – g @angular/cli

for version ng –version (or ng –v may work)

Creating Project --- ng new hello-world

This will create a folder called “hello-world” and within that folder create the project

to start app - ng serve

this runs things on localhost:4200

## webpack is a compile deployment tool

need details

## using cls to create stuff

need to open a new terminal window – in VSCode use ctrl + ` or from the menu toggle panel

to create a new

* component – ng g c course – this will create a new folder called course and create all the files needed and create references in app module
* service – ng g s email – this will create a service named email in the current folder

# Angular Notes

## Decorators

* Component – used to create a component
* Injectable () – used when dependencies are in the constructor for dependency injection (Component decorator internally includes the Injectable decorator)

## Component Notes

Selector –

* for <div class=”courses”> use “.courses” as selector
* for element <courses> use “courses” as selector,
* for <div id=”courses” use “#courses” as selector

## Dependency Injection / Providers in Modules

This will create a new instance. It will work but it is tightly bound

constructor (){

let service = new CoursesService();

this.courses = service.getCourses();

}

This uses dependency injection:

constructor (service: CoursesService){

//let service = new CoursesService();

this.courses = service.getCourses();

}

And requires a provider to be added to the module

providers: [

CoursesService

],

This will create a single instance that is used throughout the module

## Data Binding

String Interpolation - {{some value}} (one way only)

Property Binding - <img [src]=’somevar’> (one way only) – only works on html attributes that map directly to dom element. [attr.colspan] works vs [colspan] which is not in the dom

Class binding - if true target class will be added (variation of property binding)

[class.active]="isActive"

<button class="btn btn-primary" [class.active]="isActive" >Save</button>

Style binding – variation of property binding

<button [style.backgroundColor]="isActive ? 'blue' : 'white'" >button 2</button>

Event binding – binds dom raised events – (onClick)=’doSomething($event)’

doSomething($event){ //now evaluate the event passed…

$event.stopPropagation(); //stops event from bubbling up

}

Event filtering –

Old - <input (keyup)=’doSometing($event)’>

Java code if($event.keyCode === 13) //for enter

Angular - <input(keyup.enter) = ‘doSomething($event)’

Template Vars - <input #email (keyup.event)=’doSomething(#email.value)’/>

Two Way Binding - - [(ngModel)] = ‘asdf’ - banana in a box - must import angular forms

<input (keyup.enter)='doSomething()' [(ngModel)]="email"/>

When use string interpolation vs property binding – interpolation work well for adding dynamic values between headings or divs to render TEXT. Property works best for setting properties.

## Adding Bootstrap

From terminal - npm install bootstrap – save (downloads bootstrap and adds to node module folder - - save adds as a dependency in package json)

In main css file (in src dir) add the following line to import bootstrap

@import "~bootstrap/dist/css/bootstrap.css"

## Package.json

“bootstrap”: “^3.3.7 - ^ use most recent major version. Will not install major ver 4

Adding things in package.json will load everything for node module vs adding everything to source control.

Use npm install to load everything in package.json.

## Pipes

Format Data

* |uppercase – converts to uppercase
* | uppercase | lowercase – you can chain pipes
* | number - (for decimal)
* | number:’1.2-4’ - 1 position before . and 2 to 4 after will add leading 0s
* | currency – by default to usd
* |currency:’AUD’ – aus dollars
* |currency:’AUD”:true – aust dollars and $
* |currency:’AUD’:true;’3.2-2’
* |date:’shortDate’ – formats as short date

Custom Pipe

‘text | summary ‘ – summary is custom pipe that will format to just show a summary

Use @Pipe to create

## Reusable Components

To create an input – method 1 (preferred)

For Input need to import Input

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

Making a var an input var

@Input() isFavorite: boolean;

using an alias (recommended)

@Input('is-fav') isFavorite: boolean;

To create an input – method 2

@Component({

selector: 'favorite',

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

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

inputs: ['isFavorite']

})

To create an Output property

Import Output and EventEmitter

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

Declare

@Output() change = new EventEmitter();

Attach to event

onClick(){

this.isFavorite = ! this.isFavorite;

this.change.emit();

}

To pass data

this.change.emit(this.isFavorite);

To process data

<favorite [isFavorite]="post.isFavorite" (change)="onFavoriteChanged($event)"></favorite>

onFavoriteChanged(isFav){

alert(isFav);

}

Objects can be passed

Using an alias

@Output('somealais') change = new EventEmitter();

## Applying Styles

1. Using StyleUrl in Component Decorator – this keeps it in shadow dom and stuff doesn’t leak out
2. Using Style property in Component Decorator
3. Inline in HTML

## ngContent

allows custom content…

## Directives

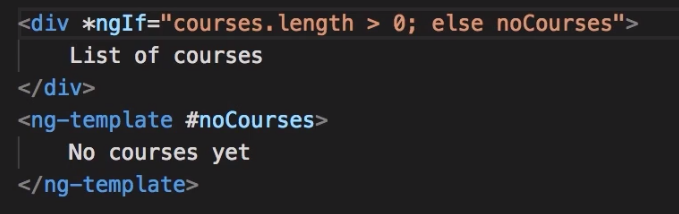
Directives are used to modify the DOM

* Structural – modify the structure of the DOM. Prefix with \*.
* Attribute – modify the attributes of DOM elements

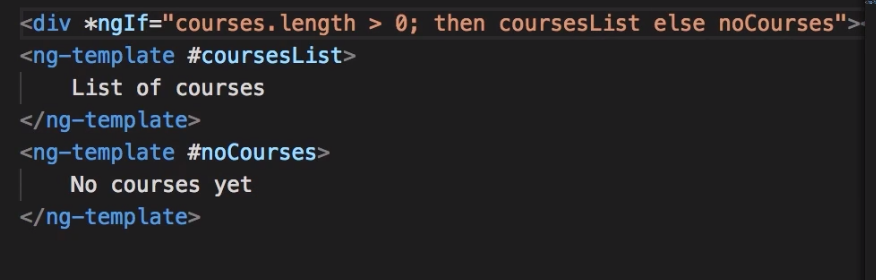
\*ngIf

\*ngIf=”someMethodToReturnTrueOfFalse()” -- elements added to or removed from dom

Using if/else to hide and display



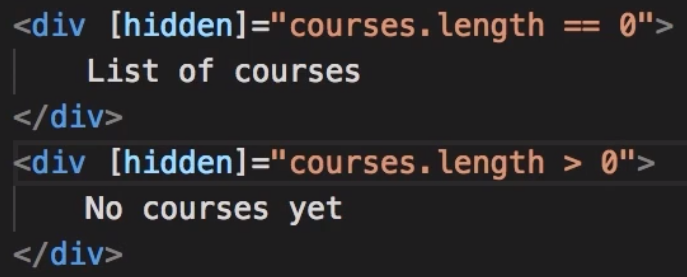
Or (preferred)



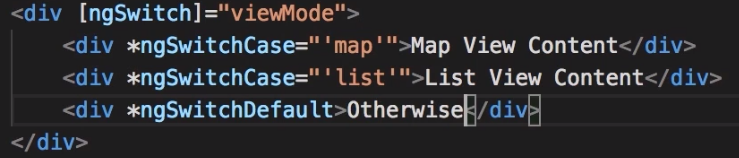
### Hidden Property

This will still exist in the DOM vs \*ngIf (

small tree of objects doesn’t matter unless costly to build then use hidden. Else us \*ngIf



### \*ngSwitchCase

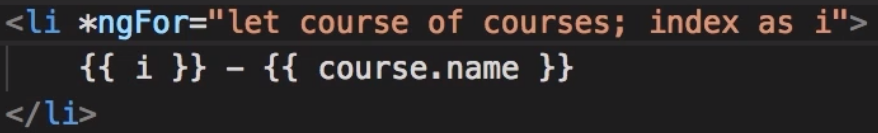


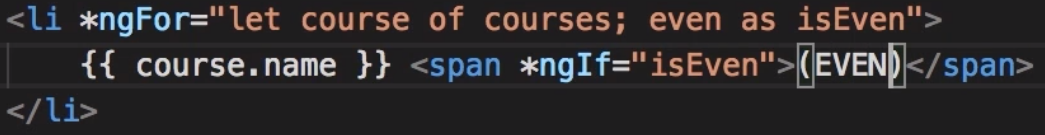
### \*ngFor

Render a list of objects (actuall NgForOf)

Exported values include: (go to angular docs for all)

* Index
* First
* Last
* Even
* Odd

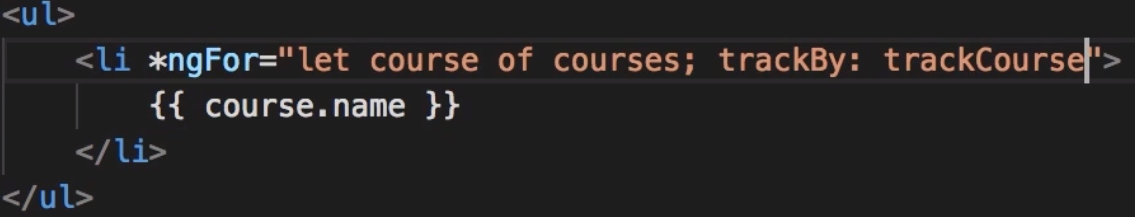


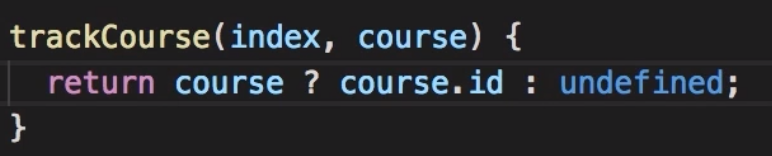


### ngFor and Change Detection

ajax request triggers change detection

More efficient





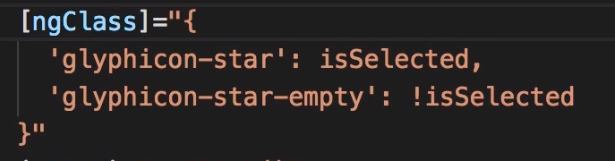
For simple list no problem, large list with lots of markup try this

### “\*”

Tells angular to use ng template

### ngClass

To set class dynamically



### ngStyle

Safe Traversal Operator – “Elvis Operator” obj?.value

## CUSTOM Directives

Cls ng g d input-format

Use @Directive tag

Applies code to element

import { Directive, HostListener, ElementRef } from '@angular/core';

@Directive({

selector: '[appInputFormat]'

})

export class InputFormatDirective {

constructor(private el: ElementRef) { }

@HostListener('focus') onFocus(){

console.log('in on focus');

}

@HostListener('blur') onBlur(){

let value: string = this.el.nativeElement.value;

this.el.nativeElement.value = value.toLowerCase();

console.log('in on blur');

}

}

To call

test input dir: <input type="text" appInputFormat/>

HostListener allows us to subscribe to events raised from DOM element that has this attribute.

ElementRef – give us access to DOM objects

To pass values

<br>

<!-- test input dir: <input type="text" appInputFormat [format]="'uppercase'"/> // for multiple attributes -->

test input dir: <input type="text" [appInputFormat]="'uppercase'"/> <!-- for one attribute -->

<br><br>

import { Directive, HostListener, ElementRef, Input } from '@angular/core';

@Directive({

selector: '[appInputFormat]'

})

export class InputFormatDirective {

//@Input('format') format: string ; -- to set multi properties must have name

@Input('appInputFormat') format: string ; // to set 1 properties can use selector name

constructor(private el: ElementRef) { }

@HostListener('focus') onFocus(){

console.log('in on focus');

}

@HostListener('blur') onBlur(){

let value: string = this.el.nativeElement.value;

if(this.format == 'lowercase'){

this.el.nativeElement.value = value.toLowerCase();

} else {

this.el.nativeElement.value = value.toUpperCase();

}

console.log('in on blur');

}

}

## Template Building Forms

FormControl class allows dev to see the following

* Value
* Touched
* Untouched
* Dirty
* Pristine
* Valid
* Validation errors

For each input value you need a form control object

Form Group – consists of form controls

Create via Template-driven or explicitly code (Reactive formerly Model Driven Forms)

Diff

Reactive

* Allow more control over validation logic
* Good for complex forms
* Unit test

Template-driven

* Simple forms only
* Simple validation
* Easier to code
* Less code
* Less control

### Template Driven Forms

ngModel creates object for you. Must have name attribute.

All needed to set

ngModel name="firstName"

and to app.module.ts add

import {FormsModule} from '@angular/forms'

and

imports: [

BrowserModule,

FormsModule

Built in validators:

* Required
* Min length
* Max length
* Pattern

<input required minlength="3" maxlength="10" pattern="bananna"

ngModel name="firstName" id="firstName" type="text" class="form-control"

#firstName="ngModel" (change)="log(firstName)" >

<div class="alert alert-danger" \*ngIf="firstName.touched && !firstName.valid">

<div \*ngIf="firstName.errors.required">First Name is Required.</div>

<div \*ngIf="firstName.errors.minlength">First Name is too shoret.</div>

<div \*ngIf="firstName.errors.pattern">First Name is incorrect pattern.</div>

</div>

</div>

ngForm automatically applied to form tag.

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

Use ngModelGroup to split things up if desired

Two classes that control form objects

* FormControl – one input field
* FormGroup – group of input fields

Disable form submit button

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

#### To see what is behind the form

<p>{{f.value | json}}</p>

### Checkbox

<select ngModel name="contactMethod" id="contactMethod" class="form-control">

<option value=""></option>

<option \*ngFor="let method of contactMethods" [ngValue]="method" >{{method.name}}</option>

</select>

Use ngValue for complex objects

For multi select

<select multiple ngModel name="contactMethod" id="contactMethod" class="form-control">

<option value=""></option>

<option \*ngFor="let method of contactMethods" [ngValue]="method" >{{method.name}}</option>

</select>

## Reactive Forms

To app.module add

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

imports: [

BrowserModule,

FormsModule,

ReactiveFormsModule

Must manually create form in code

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

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

@Component({

selector: 'signup-form',

templateUrl: './signup-form.component.html',

styleUrls: ['./signup-form.component.css']

})

export class SignupFormComponent {

form = new FormGroup({

'username': new FormControl(), //could be FormGroup

'password': new FormControl()

});

}

In Template add

<form [formGroup]="form">

<input

formControlName="username"

### Validation

Do not use html validation

Add validators to imports

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

For Required

form = new FormGroup({

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

'password': new FormControl('', Validators.required) //could be FormGroup

});

To expose property

get username(){

return this.form.get("username");

}

In html

<div \*ngIf="username.touched && username.invalid" class="alert alert-danger">Required</div>

For multiple validations

form = new FormGroup({

'username': new FormControl('', [

Validators.required,

Validators.minLength(3),

Validators.maxLength(10)

]),

Html

<div \*ngIf="username.touched && username.invalid" class="alert alert-danger">

<div \*ngIf="username.errors.required">Required</div>

<div \*ngIf="username.errors.minlength">Too Short</div>

<div \*ngIf="username.errors.maxlength">Too Long</div>

</div>

### Custom Validator Functions

import {AbstractControl, ValidationErrors } from '@angular/forms';

export class UsernameValidators{

static cannotContainSpace(control: AbstractControl ) : ValidationErrors | null {

if((control.value as string ).indexOf(' ') != -1){

return {cannotContainSpace : true };

}

return null;

}

}

### For Async Validators

Code in custom class

static shouldBeUnique(control: AbstractControl ) : Promise< ValidationErrors | null> {

return new Promise((resolve, reject)=>{

setTimeout( ()=>{

if(control.value === 'Shit'){

resolve({shouldBeUnique : true });

} else {

resolve(null);

}

}, 2000 );

});

}

In Component add

form = new FormGroup({

'username': new FormControl('', [

Validators.required,

Validators.minLength(3),

Validators.maxLength(10),

UsernameValidators.cannotContainSpace

], [UsernameValidators.shouldBeUnique]),

Checking for asyn complete and displaying a pending message while waiting

<div \*ngIf="username.pending">Checking Validations</div>

## Validations on Submit

In html wire up submit

<form [formGroup]="form" (ngSubmit)="login()">

<div \*ngIf="form.errors" class="alert alert-danger">

failed login

</div>

In code

this.form.setErrors({

inValidLogin: true

});

## Form Array

When dealing with an array of objects use the FormArray instead of FormControl

Form Builder vs manual build

form1 : FormGroup;

form = new FormGroup({

name: new FormControl('', Validators.required),

contact: new FormGroup({

email: new FormControl(),

phone: new FormControl()

}),

topics : new FormArray([])

});

constructor(fb: FormBuilder) {

this.form1 = fb.group({

name: ['', Validators.required],

contact: fb.group({

email: [],

phone: []

}),

topic : fb.array([])

});

}

## Typescript notes

Use let vs var for creating a var. Scopes better.