# Angular

Instalación

Tenemos que tener disponible la última versión de Node.js, que nos permitirá descargar Angular-cli. Node.js viene con npm. Angluar-cli es una herramienta en línea de comandos bastante útil para trabajar con proyectos angular2 o angular4. Angular-cli nos creará un proyecto básico o esqueleto, incluye el compilador typescript y más.

npm install –g @angular/cli@latest

Si ya lo tenemos instalado:

npm uninstall –g @angular/cli

npm cache clean

npm install –g @angular/cli@latest

Con ng – version muestra las versiones:

Crear proyecto con angular-cli

ng new proyecto1

cd proyecto1

En tsconfig.app.json tenemos:

{

"extends": **"../tsconfig.json**",

"compilerOptions": {

"outDir": "../out-tsc/app",

"baseUrl": "./",

"module": "es2015",

"types": []

},

"exclude": [

"test.ts",

"\*\*/\*.spec.ts"

]

}

En el fichero tsconfig.json de la carpeta podemos configurar opciones del compilador, ejemplo:

{

"compileOnSave": false,

"compilerOptions": {

"outDir": "./dist/out-tsc",

"sourceMap": true,

"declaration": false,

"moduleResolution": "node",

"emitDecoratorMetadata": true,

"experimentalDecorators": true,

"target": "es5",

// Para pruebas

"**noImplicitThis**": true,

"typeRoots": [

"node\_modules/@types"

],

"lib": [

"es2017",

"dom"

]

}

}

ls, para ver las carpetas y ficheros generados pro angular-cli para el proyecto. 

Fichero package.json

En este fichero podemos ver la versión de Node.js, del compilador /transpilador typescript.

Este fichero se encuentra ubicado en la carpeta raíz del proyecto, en nuestro caso en proyecto1.

Fichero main.ts

Ubicado en /src.

En este fichero es donde se indicará el módulo que será el punto de entrada de la aplicación.

/src/main.ts:

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

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

import { environment } from './environments/environment';

platformBrowserDynamic().bootstrapModule(AppModule)

.catch(err => console.log(err));

Fichero index.html

Ubicado en /src.

Como cualquier aplicación o sitio web, los proyectos Angular disponen de un archivo index.html, ubicado en la carpeta src. Además del código habitual de un archivo html, este archivo se caracteriza por incluir la etiqueta <app-root></app-root>, que será la etiqueta del web component donde se ‘renderice’ todo el código de la aplicación. Dentro de la etiqueta raíz anterior, podemos añadir un texto que indique que la aplicación está cargando o bien un spinner, que se mostrará durante la carga de la aplicación si esta se demora. Por ejemplo: src/index.html … <app-root><p>Cargando aplicación…</p></app-root> … Si la aplicación es muy ligera, cargará tan rápido el componente raíz, que ese texto de carga, directamente no se mostrará. En la etiqueta <head></head> de este archivo, podemos añadir como en cualquier otra aplicación, CDN de librerías de fuentes e iconos así como frameworks de estilo como por ejemplo Bootstrap 4. Por ejemplo, nosotros vamos a emplear las fuentes Google y Bootstrap 4, para lo cual añadimos, dentro del <head></head>:

Fichero styles.css

Ubicado en /src.

Aquí podemos incluir todos los estilos globales, es decir para toda la aplicación. No hay que indicarlo en index.html.

Directorio assets

Ubicado en /src.

Aquí ponemos todos los fichero estáticos de la aplicación como imágnes, videos, etc..



En todas las aplicaciones Angular, existe al menos un módulo raíz, que se encuentra ubicado en el archivo app.module.ts generado por Angular CLI en el directorio src/app.

Módulos y componentes.

Módulos

Un módulo en angular es el conjunto de código dedicado a un ámbito concreto de la aplicación o funcionalidad.

En Angular, los módulos se definen mediante una clase decorada con @NgModule. Toda aplicación de Angular tendrá al menos un módulo, el llamado módulo principal o raíz (root module), que, para el caso de aplicaciones pequeñas será único. Los módulos, se definen en archivos TypeScript y podemos decir que están compuestos de tres bloques o apartados de código.

Apartado para importación

Aquí se importan librerías y paquetes de angular así como elementos externos que se emplearán en el módulo.

Apartado para el decorador

Se emplea el decorador @NgModule, que recibe un objeto mediante el cual se configuran los metadatos:

Declarations

Las declaraciones son las llamadas vistas de un módulo. Hay 3 tipos de vistas o declaraciones, los componentes, las directivas y los pipes.

Imports

En este apartado se indican las dependencias o paquetes que empleará este módulo, cuyo origen se define en las importaciones al inicio del archivo.

Providers

Son los servicios utilizados por el módulo, disponibles para todos los componentes, y que centralizan la gestión de datos o funciones para inyectarlos en los componentes.

Bootstrap

Este metadato define la vista raíz de la aplicación y es utilizado solo por el módulo raíz. No confundir con el popular framework de estilos del mismo nombre.

Apartado para la exportación

Para dar nombre al módulo y ser exportable.

Ejemplo fichero modulo

Este ejemplo ha sido generado por angular-cli para proyecto1 (/src/app/app.module.ts):

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

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

import { AppComponent } from './app.component';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Componentes

Un componente controla una zona de espacio de la pantalla.

Un componente es una clase estándar de ES6 decorada con @Component.

Es equivalente en angularjs a un controlador.

En los componentes se controla la lógica de la aplicación, su vista HTML y el enlace con otros componentes.

Apartado para importación

Aquí se importan requeridos por el componente.

Apartado para el decorador

Se emplea el decorador @Component, que recibe un objeto mediante el cual se configuran los metadatos:

Selector

Define la etiqueta HTML donde se renderizará el componente.

Template

Archivo html con la vista del componente que no tiene la estructura de un archivo html, se incrustará, en este caso en el archivo index.html.

Style

Define el archivo css con los estilos del componente.

Apartado para la exportación

Para dar nombre al componente y ser exportable.

Ejemplo fichero componente

Fichero generado por angular-cli para proyecto1 /src/app/app.component.ts:

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

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

title = 'app';

}

Arranque del proyecto generado

npm start o ng server

Que arrancará en el puerto 4200

Para arrancar en puerto 2700: ng server –port 2700

El primer fichero typescript en ejecutarse es /src/main.ts:

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

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

import { environment } from './environments/environment';

if (environment.production) {

enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule)

.catch(err => console.log(err));

Creación de components con angular-cli

Desde la carpeta raíz del proyecto:

ng generate component <nombrecomponente>

o de forma abreviada:

ng g c <nombrecomponente>

Este comando actualiza el fichero app.module.ts, incluyendo el componente en la sección import y en el decorador @NgModule y en concreto en el metadato ‘declaration’.

app.module.ts, después de añadir el componente por ejemplo copyright quedaría así:

ng generate component copyright

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

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

import { AppComponent } from './app.component';

import { CopyrightComponent } from './copyright/copyright.component';

@NgModule({

declarations: [

AppComponent,

CopyrightComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Data Binding

Es la comunicación o enlace de datos entre el componente y la vista html (su template).

Este modo de comunicación también tiene el nombre de “String Interpolation”.

Para nuestros ejemplos, vamos a declarar un archivo de clase, y los ts crearan los objetos para el paso a la vista y viceversa

Creamos el directorio “modelo” y la clase la definimos en /modelo/desarrolador.modelo.ts:

export class Desarrollador { public id: number; public nombre: string;

public apellidos: string;

public ciudad: string;

constructor (id: number, nombre: string, apellidos: string, ciudad: string){

this.id = id;

this.nombre = nombre;

this.apellidos = apellidos;

this.ciudad = ciudad;

}

}

Interpolación

Comunicación de la fuente (ts) hacia la vista (html)

Su sintaxis, en la que se puede incluir una propiedad, un objeto o una expresión JavaScritp entre dobles llaves, conocidad como sintaxis moustache:

{{propiedad/objeto/expresión}}

Ejemplo 1:

creamos el componente viewmodelo

ng g c viewmodelo -spec false

En el ficheor viewmodelo.component.ts, importamos la clase desarrollador, y se crea el objeto desarrollador. Se emplea el constructor para asignar los atributos:

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

import { Desarrollador } from '../modelo/desarrollador.modelo';

@Component({

selector: 'app-viewmodelo',

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

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

})

export class ViewmodeloComponent implements OnInit{

desarrollador = new Desarrollador(1, 'Juan', 'Gutiérrez', 'Madrid');

constructor() { }

ngOnInit() {

}

}

En el fichero viewmodelo.component.html, es donde utilizamos la expresión mustage para mostrar el contenido del objeto desarrollador:

<div class="container"> <h4>Información del Desarrollador</h4>

<hr>

<h5>id: {{desarrollador.id }}</h5>

<h5>Nombre: {{desarrollador.nombre }}</h5>

<h5>Apellidos: {{desarrollador.apellidos }}</h5>

<h5>Ciudad: {{desarrollador.ciudad }}</h5>

</div>

Property Binding

Comunicación de la fuente (ts) hacia la vista (html)

En este caso, se trata de un enlace que relaciona un atributo con una expresión, con la siguiente sintaxis: [atributodelelementoHTML] = “ expresión “

Creamos el componente ejpropertybinding

ng g c ejpropertybinding --spec false

y en ejpropertybindind.component.ts:

texto = 'Escribe algo';

constructor() { setTimeout(() => { this.texto = 'por favor'; }, 3000);

}

y en fichero template:

<input type="text" [placeholder]="texto">

y en fichero app.component.html renderizamos con:

<div class="container" >

<h3>Ejemplo de Property Binding</h3>

<hr>

<app-ejpropertybinding></app-ejpropertybinding>

</div>

Event Binding

Comunicacion de la vista (html) hacia la fuente (ts)

Se trata de un movimiento de datos dela vista hacia el modelo y se desencadena por un evento en el cliente web.

La sintaxis tiene la siguiente forma:

evento=”nombreMetodo()”;

Creamos el component ejeventbuilding

Y en el fichero ts:

texto = 'Originalmente el texto se carga así';

modTexto() {

if (this.texto === 'arriba')

this.texto = 'abajo';

else

this.texto = 'arriba';

}

Y en la vista:

<button class="btn btn-success"

(click)="modTexto()">Modificar Texto</button>

<h3> {{ texto }} </h3>

Two-way Binding

Es posible la comunicación de la vista a la fuente y viceversa.

Se enlaza el valor de un elemento HTML de la vista con la propiedad de un componente:

[(directiva)] = “nombredelapropiedad”

Creamos el componente ej2waybinding

ng g c ej2waybinding - - spec false

En ts:

export class Ej2waybindingComponent implements OnInit {

texto = 'Texto original a cargar';

constructor() { }

ngOnInit() {

}

En la vista:

<label>Introduce un valor</label>

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

<h3>{{texto}}</h3>

Directivas

Las directivas son clases angular para crear, configurar e interactuar con elementos html del DOM.

Las directivas llevan el decorador @Directive en las clases angular.

Hay tres tipos de directivas:

Componentes

Se puede decir que los componentes con su directiva @Component, es una aplicación de las directivas.

Directivas de atributo

Modifican el aspecto o comportamiento de los elemento del DOM. Tienen el prefijo ng.

Directiva ngStyle

Sirve para establecer de manera dinámicas los atributos de un elemento del DOM.

[ngStyle]=”{ expresión/propiedad/método }”

Creamos componente ejdirectivangstyle

ng g c ejdirectivangstyle --spec false

En .html:

<h4>Introduzca la puntuación del Alumno</h4>

<input type="number" class="form-control" [(ngModel)]="puntuacion">

<hr>

<h4 style="display: inline-block;">Puntuación obtenida:&nbsp; </h4>

<h4 style="display: inline-block;" [ngStyle]="{color:setColor()}">{{puntuacion}}</h4>

En .ts:

puntuacion:number;

setColor() {

return this.puntuacion >= 5? 'green': 'red';

}

Directiva ngClass

Esta directiva es similar a ngStyle y sirve para establecer la clase (css) de un elemento del HTML.

Creamos el componente para demo:

ng g c ejdirectivangclass --spec false

en .html:

<h4>Introduzca la puntuación del Alumno</h4>

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

<div \*ngIf="puntuacion">

<h4 \*ngIf="puntuacion >= 0 && puntuacion <= 10; else aviso"

[ngClass]="{ aprobado: puntuacion >= 5, suspenso: puntuacion <5 }"> Puntuación obtenida: {{puntuacion }}

</h4>

<ng-template #aviso>

<h4 \*ngIf="puntuacion > 10" class="advertencia"> Introduzca una puntuación menor a 10</h4>

<h4 \*ngIf="puntuacion < 0" class="advertencia"> Introduzca una puntuación mayor o igual a 0</h4>

</ng-template>

En .ts:

puntuación:number;

En .css:

.advertencia { color: white; background-color: orange; padding: 10px; }

.aprobado { color: white; background-color: green; padding: 10px; }

.suspenso { color: white; background-color: red; padding: 10px; }

Directivas estructurales

Añaden, reemplazan o eliminan elementos del DOM.

Estas directivas van precedidas por el carácter “\*”.

Directiva \*ngif

Con esta directiva podemos controlar que un elemento HTML se muestre o no dependiendo de una condición definida mediante una propiedad o método.

\*ngIf=”expresión/propiedad/metodo”

lo veremos con un ejemplo, para ello creamos :

ng g c ejdirectivangif --spec false

En ejdirectivangif.component.ts:

definimos dos propiedades tipo string nombre1 y nombre2

nombre1:string;

nombre2:string;

setResultado(){

return this.capital === "Madrid"? true: false;

}

y en ejdirectivangif.component.html:

<label>Nombre y Apellidos</label>

<input type="text" class="form-control" [(ngModel)]="nombre1" placeholder="Complete su nombre y apellidos">

<button type="submit" class="btn btn-primary" \*ngIf="nombre1">Enviar</button

Con las línea de arriba, si el atributo nombre1 tiene valor, se muestra el boton “Enviar”.

<label>Nombre y Apellidos</label>

<input type="text" class="form-control" [(ngModel)]="nombre2" placeholder="Complete su nombre y apellidos">

<button type="submit" class="btn btn-primary" \*ngIf="nombre2; else desactivado">Enviar</button>

<ng-template #desactivado>

<button type="submit" class="btn btn-primary" **disabled**>Enviar</button>

</ng-template>

Con las líneas de arriba, si el atributo nombre2 tiene valor se muestra el botón “Enviar” activado en caso contrario se muestra desactivado

<h3>¿Cuál es la capital de España?</h3>

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

<p \*ngIf="!capital; else tienecontenido">Complete la pregunta</p>

<ng-template #tienecontenido>

<h4 \*ngIf="setResultado(); else incorrecto">¡Correcto!</h4>

</ng-template>

<ng-template #incorrecto>

<h4>Lo siento, inténtelo de nuevo</h4>

</ng-template>

En las líneas de arriba, si la propiedad capital no tiene valor muestra “Complete la pregunta”, en caso contrario comprueba si el contenido es Madrid mediante el método setResultado() y en ese caso muestra “¡Correcto!”, en caso contrario muestra “Lo siento, inténtelo de nuevo”.

Directiva \*ngFor

Para realizar iteraciones y presentar listados.

\*ngFor=”let objeto/propiedad of objetos/propiedades”

Crea dentro del elemento html una variable local con let que recorrerá el array definido por of y proveniente del componente.

ng g c ejdirectivangfor -spec

En ts:

cursos:string[];

constructor() {

this.cursos = ['java', 'cobol', 'angular'];

}

En .html:

<h3>Cursos Disponibles</h3>

<ul>

<li \*ngFor="let curso of cursos">

<h4>{{curso}}</h4>

</li>

</ul>

Creación de directivas

Podemos crear una plantilla de directive con Angular CLI:

ng generate directive directivamenu

El fichero .ts correspondiente (directivamenu.directive.ts), se crea en la carpeta app, igual que app.component.ts. No se crean ficheros .css ni .html como en el caso de component.

marcamos las modificaciones en verde:

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

@Directive({

selector: '[appDirectivamenu]'

})

export class DirectivamenuDirective {

@HostBinding ('class.item-orange') private mostrar: boolean = false;

@HostListener('mouseover') onOver() {

this.mostrar = true;

}

@HostListener('mouseout') onOut() {

this.mostrar = false;

}

}

La clase “.item-orange” se define en stype.css ubicado en la carpeta src.

Ahora definimos un componente para probar esta directiva:

ng g c testdirecitvamenu -- spec false

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

ts:

@Component({

selector: 'app-testdirectivamenu',

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

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

})

export class TestdirectivamenuComponent implements OnInit {

constructor() { }

ngOnInit() {

}

}

html:

<div class="container">

<h1 appDirectivamenu>Título</h1>

<h1 appDirectivamenu>Descripción</h1>

<h1 appDirectivamenu>Precio</h1>

</div>

Luego incorporamos componente (testeador) y derectiva en app.module.ts:

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

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

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

import { AppComponent } from './app.component';

import { CopyrightComponent } from './copyright/copyright.component';

import { FechaactualComponent } from './fechaactual/fechaactual.component';

import { ViewmodeloComponent } from './viewmodelo/viewmodelo.component';

import { EjpropertybindingComponent } from './ejpropertybinding/ejpropertybinding.component';

import { EjeventbindingComponent } from './ejeventbinding/ejeventbinding.component';

import { Ej2waybindingComponent } from './ej2waybinding/ej2waybinding.component';

import { EjdirectivangifComponent } from './ejdirectivangif/ejdirectivangif.component';

import { EjdirectivangstyleComponent } from './ejdirectivangstyle/ejdirectivangstyle.component';

import { EjdirectivangclassComponent } from './ejdirectivangclass/ejdirectivangclass.component';

import { EjdirectivangforComponent } from './ejdirectivangfor/ejdirectivangfor.component';

import { DirectivamenuDirective } from './directivamenu.directive';

import { C1Component } from './c1/c1.component';

import { TestdirectivamenuComponent } from './testdirectivamenu/testdirectivamenu.component';

@NgModule({

declarations: [

AppComponent,

CopyrightComponent,

FechaactualComponent,

ViewmodeloComponent,

EjpropertybindingComponent,

EjeventbindingComponent,

Ej2waybindingComponent,

EjdirectivangifComponent,

EjdirectivangstyleComponent,

EjdirectivangclassComponent,

EjdirectivangforComponent,

DirectivamenuDirective,

C1Component,

TestdirectivamenuComponent

],

imports: [

BrowserModule,

FormsModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Directiva @Input



Se utiliza para pasar valores a propiedades de componentes hijos. El componente que recibe los valores (hijo) es el que utilizar la directiva @Input.

ng g c usodirectivainputpadre -- spec false

gn g c usodirecitvainputhijo --spec false

En el .ts del hijo declaramos las variables que queremos mostrar en su .html:

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

@Input() titulo: string;

@Input() aviso: string;

En .html utilizamos la expresión moustage para renderizar dichas variables o propiedades:

<h1>{{titulo}}</h1>

<h4 style="display: inline-block;">{{aviso}}</h4>

En .ts del padre definimos los valores que queremos aplicar:

tituloImpuestoPorPadre: string = 'Avisos';

avisos: string[] = ['Aviso 1', 'Aviso 2', 'Aviso 3'];

En .html del padre, indicamos el componente hijo para renderizarlo:

<app-usodirectivainputhijo [titulo] = "tituloImpuestoPorPadre"></app-usodirectivainputhijo>

<br>

<app-usodirectivainputhijo \*ngFor="let avisop of avisos" [aviso]="avisop"> </app-usodirectivainputhijo>

En el .html del app.modulo.html:

<app-usodirectivainputpadre></app-usodirectivainputpadre> </div>

Y en .ts del app.modulo.ts:

import { UsodirectivainputpadreComponent } from './usodirectivainputpadre/usodirectivainputpadre.component';

import { UsodirectivainputhijoComponent } from './usodirectivainputhijo/usodirectivainputhijo.component';

…

@NgModule({

declarations: [

…

UsodirectivainputpadreComponent,

UsodirectivainputhijoComponent

]

Directiva @Output

Pipes

Un pipe es un código angular que nos permite transformar la salida o la vista.

La implementación de un pipe debe seguir la siguiente sintaxis:

{{dato | nombredelpipe:’opciones’}}

Pipe Date

en .ts:

hoy : date = new Date();

en .html:

{{hoy | date:’y/M/d’}}

Pipe Uppercase Lowercase

En .ts:

ciudad:string = ‘hola”;

<p> {{ciudad | uppercase}}, {{hoy | date:'d-M-y' | uppercase}} a las {{hoy | date:'H:m Z'}}</p>

Pipe Decimal

<p> El resultado es {{resultado | number:’2.2-2’}}</p>

Donde el primer dígito representa el mínimo número de enteros, el segundo dígito, el número mínimo de decimales y el tercer dígito el máximo número de decimales.

Con el valor 1,148, presentará:

01.15

Pipe Currency

<p>La cotización actual del dólar es de {{dolareuro | currency:'EUR':true}}</p>

En .ts:

dolareuro: number = 0.94;

Mostrará: €0.94

Pipe i18nSelect

En .ts:

…

nombre: string = 'Laura';

sexo: string = 'mujer';

encabezamiento: any = {'hombre':'Estimado', 'mujer':'Estimada'}

…

En .html:

<p> {{sexo | i18nSelect: encabezamiento}} {{nombre}} </p>

Creación de pipes

ng g pipe pipeeuro

Este comando creat /app/pipeeuro.pipe.es

Si quisiéramos crear una carpeta para nuestros pipes:

1. creamos un carpeta .app/mispipes/
2. entramos en esa carpeta que acabamos de crear.
3. ng g pipe pipeeuro2

Equivalente a los tres pasos de arriba sería:

ng g pipe mispipes/pipeeuro2

En .ts:

transform(value: any, args?: any): any {

const transformado = value + '€';

return transformado;

}

Y en .html:

<p> {{propiedad | pipeeuro2}}</p>

Servicios e inyección de dependencias

Los servicios permiten centralizar en uso común en muchos componentes a través de la inyección de dependencias.

Normalmente abastecen a los componentes de datos.

También se les llama providers y dentro de este apartado del app.module.ts se incluirán.

Vamos a crear el servicio proveedores en la carpeta servicios

ng generate service servicios/proveedores.

Angular CLI, no importará el servicio en app.module.ts como hace con componentes, directivas o pipe. Habrá que sopesar si la inclusión en provider sea en el componente consumidor o cliente o en app.module.ts (¿será un singleton?). Lo tendremos que incorporar nosotros y lo lógico sería hacerlo en el componente que lo va a utilizar:

import { ProveedoresService } from './servicios/proveedores.service';

Y luego se añade en el apartado providers del decorador @Component en el caso de componente cliente o en @ngModule:

providers: [ProveedoresService],

Los servicios se inyectan en los componentes a través de métodos. En nuestro ejemplo diseñamos un método que devuelve la versión del servicio:

getVersion () {

return ‘Version 1.0.0’;

}

Creamos un cliente de este servicio con angular cli:

ng generate component testproveedores

En .app/testproveedores/testproveedores.component.ts:

Deberemos importar nuevamente el servicio:

import {ProveedoresService} from '../servicios/proveedores.service';

Y añadimos ProveedoresService en el apartado provider, del decorador @Component.

Angular CLI, no importará el servicio en app.module.ts como hace con componentes, directivas o pipe. Lo tendremos que incorporar nosotros y lo lógico es hacerlo en el componente que lo va a utilizar.

***El servicio ProveedoresService es importado por su cliente mientras que PresupuestosService es importado en app.moduele.ts y estará disponibre para cualquier componente***

El constructor de este componente queda:

constructor( private proveedoresService : ProveedoresService) { }

con esto se implementa en el constructor un parámetro para igualar a la clase del servicio.

Y en el método ngOnInit(), se invoca al método definido en el servicio:

this.version = this.proveedoresService.getVersion(

Y dentro de ngOnInit, es decir cuando Angular cargue el componente, se iguala la propiedad mensaje a una propiedad que llama al método getProveedores del servicio.

Luego en .html podemos mostrar la versión con {{versión}}

Routing

Configuración del routing de una aplicación

Para habilitar esta funcionalidad de angular, tendremos que hacer los siguiente en el app.module.ts:

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

Establecemos las rutas en el array routes:

const routes: Routes = [

{path:'', component: InicioComponent},

{path: 'testproveedores',component: TestproveedoresComponent},

{path: 'directivainput',component: UsodirectivainputpadreComponent},

{path:'\*\*', component: InicioComponent}

];

Y dentro de su decorador @NgModule:

imports: [ BrowserModule, RouterModule.forRoot(routes) ],

Luego en app.module.html:

<div class="container"> <router-outlet></router-outlet> </div>

Para las pruebas lo que se ha hecho es añadir el component inicio, y en su .html, hemos pasado lo que tenía app.component.html, quedando app.component.html, como hemos puesto unas líneas más arriba.

Por tanto ahora, podemos poner como url: localhost:4200/testproveedores y nos mostrará únicamente ese componente.

Navegación mediante links

Mediante el routing podemos establecer un sistema de navegación mediante links que proporciona dos ventajas fundamentales:

• Menús. Este sistema, permite establecer links en menús de navegación que permite a los usuarios familiarizarse rápidamente con los componentes de la aplicación.

• SPA. Los links llaman a los componentes sin refrescar la web, lo que permite aumentar la velocidad de ejecución de la aplicación y conservar los estados de la misma, sobre todo en la gestión de datos.

Con la configuración de rutas establecida en app.module.ts, podemos crear un componente header con el siguiente .html:

<nav class="navbar navbar-light bg-faded rounded navbar-toggleable-md">

<button class="navbar-toggler navbar-toggler-right"

type="button" data-toggle="collapse" data-target="#containerNavbar"

aria-controls="containerNavbar" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<a class="navbar-brand" href="#">#</a>

<div class="collapse navbar-collapse" id="containerNavbar">

<ul class="navbar-nav mr-auto w-100 justify-content-end">

<li class="nav-item" routerLinkActive="active"

[routerLinkActiveOptions]="{exact:true}">

<a class="nav-link" routerLink="**/**">Inicio </a>

</li>

<li class="nav-item" routerLinkActive="active">

<a class="nav-link" routerLink="**/testproveedores**">Proveedores </a>

</li>

<li class="nav-item" routerLinkActive="active">

<a class="nav-link" routerLink="**/directivainput**">Test directiva @Input</a>

</li>

</ul>

</div>

</nav>

Formularios

Podemos diseñar formularios gestionados por HTML (Template-Driven) o gestinoados desde un componente (Reactive).

Formularios Template-drive

src/app/app.module.ts:

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

Y en los imports de la clase añadimos:

imports: [ BrowserModule, RouterModule.forRoot(routes), FormsModule ],

creamos el component que gestionará este formulario:

ng g c formproveedores/addprovee --spec false

Y en addprovee.Component.ts:

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

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

@Component({

selector: 'app-addprovee',

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

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

})

export class AddproveeComponent implements OnInit {

**@ViewChild('formprovee') formpro: NgForm;**

proveedor: any;

constructor() {

this.proveedor = {

nombre: '',

cif: '',

direccion: '',

cp: '',

localidad: '',

provincia: '',

telefono: null,

email: '',

contacto: ''

}

}

ngOnInit() {

}

**onSubmit()** {

this.proveedor.nombre = this.formpro.value.nombre;

this.proveedor.cif = this.formpro.value.cif;

this.proveedor.direccion = this.formpro.value.direccion;

this.proveedor.cp = this.formpro.value.cp;

this.proveedor.localidad = this.formpro.value.localidad;

this.proveedor.provincia = this.formpro.value.provincia;

this.proveedor.telefono = this.formpro.value.telefono;

this.proveedor.email = this.formpro.value.email;

this.proveedor.contacto = this.formpro.value.contacto;

this.formpro.reset();

}

En .html:

<form (ngSubmit)="**onSubmit()**" #formprovee="ngForm">

<div class="form-group">

<label>Nombre</label>

<input type="text"

class="form-control"

id="nombre"

name="nombre"

ngModel>

</div>

<div class="form-group">

<label>C.I.F.</label>

<input type="text"

class="form-control"

id="cif"

name="cif"

ngModel>

</div>

……………..

<button type="submit"

class="btn btn-success">

Añadir Proveedor</button>

</form>

Carga de datos con campos select

En html:

<div class="form-group">

<label>Provincia</label>

<select class="form-control"

id="provincia"

ngModel

name="provincia">

<option value="">Seleccione la provincia</option>

<option \*ngFor="let provincia of provincias" [value]="provincia">{{provincia}}

</option>

</select>

</div>

Validación de campos mediante HTML (Template-Driven)

En caso de utilizar formulario con la técnica Template Driven, se tendrá que utilizar la validación mediante HTML.

Gracias a los estado de los campos, angular permite implementar clases CSS y elemento HTML dinámicos de ayuda a la introducción de datos por parte del usuario.

Vamos a indicar el campo email como requerido, hay que tener en cuenta que aunque no se haya rellenado el campo emaill, angular dejará habilitado el botón submit.

Para habilitar el botón submit cuando todos los campos sean validos (valid):

<button type="submit"

class="btn btn-success"

[disabled]="!formpro.valid">Añadir Proveedor

</button>

<p class="alert alert-danger "

\*ngIf="!formpro.valid">Por favor complete todos los campos

</p>

Con [disabled]="!formpro.valid, establecemos el atributo disabled a true si no todos los campos son válidos.

Formularios Reactive

La generación y gestión del formulario se realiza del lado del componente typescript.

ng g c formreactivopresupuesto/addpres --spec false

Lo primero que tenemos que hacer es en app.module.ts:

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

Para configurar router y tener acceso directo.

Y

{path: 'addprovee', component: AddpresComponent},

..

imports:{

…

ReactiveFormsModule

…

Opcional: en src/app/header/header.component.html

…

<li class="nav-item" routerLinkActive="active">

<a routerLink="/**addpres**">Añadir Presupuesto

</a>

</li>

…

En formreactivopresupuesto/addpres/addpres.component.html:

Hay que tener en cuenta que en el formulario habría que:

<form [formGroup]="presupuestoForm" (ngSubmit)="onSubmit()">

Y que para cada campo del formulario habría que especificar el nombre del campo, por ejemplo para concepto:

formControlName="concepto"

En formreactivopresupuesto/addpres/addpres.component.ts:

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

Y creamos dentro de la clase de este componente:

presupuestoForm: FormGroup;

presupuesto: any;

Seguidamente en el constructor, para generar el formulario, creamos un objeto de nombre, por ejemplo, pf de la clase FormBuilder de Angular:

…

constructor(private pf: FormBuilder) {}

…

También habrá que implementar el método onSubmit():

onSubmit (){

this.presupuesto = this.savePresupuesto();

}

savePresupuesto (){

const savePresupuesto = {

proveedor: this.presupuestoForm.get('proveedor').value,

fecha: this. presupuestoForm.get('fecha').value,

concepto: this.presupuestoForm.get('concepto').value,

base: this.presupuestoForm.get('base').value,

tipo: this.presupuestoForm.get('tipo').value,

iva: this.presupuestoForm.get('iva').value,

total: this.presupuestoForm.get('total').value

};

return savePresupuesto;

}

Validación de campos programática (formularios Reactive)

En el ts, añadimos Validators:

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

En html:

<button type="submit"

class="btn btn-primary" [disabled]="!presupuestoForm.valid" >Añadir Presupuesto

</button>

<p class="alert alert-danger" \*ngIf="!presupuestoForm.valid"> Por favor complete todos los campos </p>

Con lo que solo se habilita el botón cuando todos los campos sean válidos.

Para indicar que los campos del formulario que hay que validad, tenemos que hacer lo siguiente en .ts:

constructor(public **pf : FormBuilder**) { }

ngOnInit() {

this.presupuestoForm = this.pf.group({

proveedor: [''**, Validators.required**],

fecha: ['', **Validators.required]**,

….

Vamos a validar también que el campo concepto tenga al menos 12 bytes:

concepto: ['', [Validators.required, Validators.minLength(12)]],

Para más información sobre validación: <https://angular.io/api/forms/Validators>

Para dar información al usuario sobre las validaciones en los campos del formulario, vamos a tomar como ejemplo el campo proveedor:

<div class="form-group">

<label for="proveedor">Proveedor</label>

**<i class="fa fa-check-circle check"**

**\*ngIf="presupuestoForm.controls.proveedor.valid"></i>**

**<i class="fa fa-exclamation-circle uncheck"**

**\*ngIf="presupuestoForm.controls.proveedor.invalid &&**

**presupuestoForm.controls.proveedor.touched"></i>**

<input type="text"

class="form-control"

id="proveedor"

formControlName="proveedor">

**<p class="alert alert-danger" \*ngIf="presupuestoForm.controls.proveedor.invalid &&**

**presupuestoForm.controls.proveedor.touched">**

El campo Proveedor es obligatorio.

</p>

</div>

Captura de cambios de valor (formularios Reactive)

En estos tipos de formularios podemos hacer que se ejecute el método onChange() cuando alguno de los campos cambie de valor. Para ello en nuestro ejemplo de alta de presupuesto, vamos a calcular el tipo y el total, que son dependientes de la base y del tipo.

Para ello en .ts:

añadimos en la clase:

base: any;

tipo: any;

iva: any = 0;

total: any = 0;

Y en el método onInit:

…

tipo: ['', Validators.required],

**iva: this.iva,**

**total: this.total**

…

Definimos el método onChange():

onChange(): void {

this.presupuestoForm.valueChanges.subscribe(valor => {

this.base = valor.base;

this.tipo = valor.tipo;

this.presupuestoForm.value.iva = this.base \* this.tipo;

this.presupuestoForm.value.total = this.base + (this.base \* this.tipo); }); }

}

En el método onChange(), utilizamos el observable valueChanges y nos sucribimos a él para obtener el objeto valor, que se actualizará cada vez que se produzca un cambio en algún campo del formulario.

En html:

Hay que cambiar los campos calculados a partir de tipo y base:

…

<div

class="form-group">

<label for="iva">Importe IVA</label>

<input type="number" class="form-control" id="iva" formControlName="iva" **[(ngModel)]="presupuestoForm.value.iva"**

**disabled**>

</div>

<div

class="form-group">

<label for="total">Total Factura IVA Incluido</label>

<input type="number" class="form-control" id="total" formControlName="total" **[(ngModel)]="presupuestoForm.value.total"**

**disabled**>

</div>

…

Conexión con el servidor

Angular utiliza los métodos http para realizar conexiones con servidores de bases de datos que nos permitan almacenar de manera persistente los mismos.

Base de datos en FireBase

https://firebase.google.com

Nos permitirá tener un servidor CRUD de manera sencilla.

Servicio HTTP: POST

Vamos a generar el servicio presupuestos:

ng generate service servicios/presupuestos

***El servicio ProveedoresService es importado por su componente-cliente mientras que PresupuestosService es importado en app.moduele.ts y estará disponibre para cualquier componente.***

En app.module.ts:

…

import {HttpModule} from '@angular/http';

import {PresupuestosService} from 'app/servicios/presupuestos.service';

…

Y

..

providers: [PresupuestosService],

…

En presupuestos.service.ts:

…

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

import {Headers, Http, Response} from '@angular/http';

import 'rxjs/Rx';

…

También:

presURL = 'https://comprasapp-fdc30.firebaseio.com/presupuestos.json';

constructor(private http:Http) { }

Y creamos el método que hará el POST:

postPresupuesto( presupuesto: any) {

const newpres = JSON.stringify(presupuesto);

const headers = new Headers({ 'Content-Type': 'application/json' });

return this.http.post( this.presURL, newpres, {headers})

.map( res => { console.log(res.json());

return res.json(); })

En el componente cliente de este servicio addpres.componet.ts:

…

import {PresupuestosService} from '../../servicios/presupuestos.service';

…

Y añadimos el servicio mediante el constructor:

…

constructor(private pf: FormBuilder,

private presupuestoService: PresupuestosService) { }

…

Y En el submit(), se llama al servicio:

onSubmit (){

this.presupuesto = this.savePresupuesto();

this.presupuestoService.postPresupuesto(this.presupuesto)

.subscribe(newpres => { })

}

Servicio HTTP: GET

Vamos a mostrar la lista de presupuesto que hemos incorporado en firebaseio.com.

En src/app/servicios/presupuestos.service.ts:

…

getPresupuestos () {

return this.http.get( this.presURL) .map( res => res.json());

}

…

Vamos a crear un componente para mostrar la lista de presupuesto obtenida con GET:

ng g c formreactivopresupuesto/listapres --spec false

Añadimos ruta en app.module.ts:

…

{path: 'presupuestos', component: PresupuestosComponent},

…

En src/app/header/header.component.html:

<li class="nav-item" routerLinkActive="active">

<a class="nav-link" routerLink="**/listapresupuestos**">Presupuestos </a>

</li>

En src/app/formreactivopresupuesto/listapres/listapres.component.ts creamos la tabla para mostrar los presupuestos:

<h3>Listado de Presupuestos</h3>

<a class="btn btn-primary float-md-right" routerLink="/addpres">Añadir nuevo presupuesto</a>

<br>

<table class="table table-bordered table-striped tabla" style="margin-top: 40px;">

<thead>

<tr class="filters">

<th>Proveedor</th>

<th>Fecha</th>

<th>Concepto</th>

<th class="text-right">Base</th>

<th class="text-right">IVA</th>

<th class="text-right">Total</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let presupuesto of **listaDePresupuestos** ">

<td>{{ presupuesto.proveedor }}</td>

<td>{{ presupuesto.fecha | date:'dd-MM-yyyy' }}</td>

<td>{{ presupuesto.concepto }}</td>

<td class="text-right">{{ presupuesto.base | number:'1.2-2' }}</td>

<td class="text-right">{{ presupuesto.iva | number:'1.2-2'}}</td>

<td class="text-right">{{ presupuesto.total | number:'1.2-2'}}</td>

</tr>

</tbody>

</table>

En listapres.component.ts:

import {PresupuestosService} from '../../servicios/presupuestos.service';

…

**listaDePresupuestos**: any[] = [];

…

En el constructor, declaramos el servicio y llamamos a su método getPresupuestos. Como este método nos devuelve un objeto con todos los presupuestos en el formato de Firebase, lo iteramos con un for para convertirlo en objetos individuales con una id. Esos objetos, los añadimos al array presupuestos del componente con el método JavaScript push.

constructor(private presupuestosService: PresupuestosService) {

this.presupuestosService.getPresupuestos()

.subscribe(presupuestos => {

for ( const id$ in presupuestos) {

const p = presupuestos[id$];

p.id$ = id$;

this. **listaDePresupuestos**.push(presupuestos[id$]);

}

})

}

Servicio HTTP: PUT

Vamos a sustituir un presupuesto. Para ello necesitamos una nueva url y dos métodos en el servicio dedicado a presupuestos (app/servicios/presupuestos.service.ts):

presupuestoURL = 'https://comprasapp-fdc30.firebaseio.com/presupuestos';

…

getPresupuesto($id:string){

const url = `${ this.presupuestoURL }/${ id$ }.json`;

return this.http.get( url)

.map( res => res.json());

}

putPresupuesto( presupuesto: any, id$: string) {

const newpre = JSON.stringify(presupuesto);

const headers = new Headers({

'Content-Type': 'application/json'

});

const url = `${ this.presupuestoURL }/${ id$ }.json`;

return this.http.put( url, newpre, {headers})

.map( res => {

console.log(res.json());

return res.json();

})

}

Para utilizar estos servicios para modificaciones creamos un nuevo componente:

ng g c formreactivepresupuesto/editpres -- spec false

En app.module.ts:

{path: 'editpres/:id', component: EditpresComponent},

…

En editpres.component.ts:

constructor(private pf: FormBuilder,

private presupuestoService: PresupuestosService,

private router: Router,

private activatedRouter: ActivatedRoute) {

this.activatedRouter.params

.subscribe( parametros => {

this.id = parametros['id'];

this.presupuestoService.getPresupuesto( this.id)

.subscribe( unPresupuesto => this.presupuesto = unPresupuesto)

});

}

onSubmit() {

this.presupuesto = this.savePresupuesto();

this.presupuestoService.putPresupuesto( this.presupuesto, this.id )

.subscribe(newpre => {

this.router.navigate(['/listapresupuestos'])

})

}

Servicio HTTP: DELETE

En servicios/presupuestos.service.ts:

**delPresupuesto** ( id$: string ) {

const url = `${ this.preURL }/${ id$ }.json`;

return this.http.delete( url )

.map( res => res.json());

}

**eliminarPresupuesto**(id$) {

this.presupuestosService.**delPresupuesto**(id$)

.subscribe( res => {

this.listaDePresupuestos = [];

this.presupuestosService.getPresupuestos()

.subscribe(presupuestos => {

for ( const id$ in presupuestos) {

const p = presupuestos[id$];

p.id$ = id$;

this.listaDePresupuestos.push(presupuestos[id$]);

}

})

});

}

Añadimos columna con botón delete en listapres.component.html:

<td class="text-right">

<button class="btn btn-danger" (click)="**eliminarPresupuesto**(presupuesto.id$)">Eliminar</button>

</td>