# Angular



ES5 o ECMAScript 5 es el javascript más extendido y primero, capaz de ser interpretado por prácticamente todos los navegadores. Después salió el ES6, que incorpora clases y módulos, y finalmente TypeScript que no es interpretado por ningún navegador y que tiene que ser transcompilado a javascript para su despliegue.

traceur y babel son transpilers de ES6 A ES5.

angular-cli

Framework que facilita el desarrollo de proyectos 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 angular. Angular-cli nos creará un proyecto básico o esqueleto, incluye el compilador typescript y más.

Con Node Version Manager (nvm) podemos seleccionar entre las diferentes versiones de nodejs que tengamos instaladas.

Nvm list -> para mostrar las versiones que tenemos instaladas

Nmv use <versión con la que deseamos trabajar>

Para instalar angular/cli global (debemos tener permisos de administrador):

npm install –g @angular/cli@latest

Para crear una aplicación sin angular-cli global, podemos utilizar

npx -p @angular/cli ng new my-app

npx -p @angular/cli[version] ng new my-app

npx también se puede utilizar para ejecutar los binarios locales (en node\_modules)

para ver la versión instalada:

ng - -version (global)

npm run – ng –version (con angular-cli local)

Actualización

Si ya lo tenemos instalado (global):

npm uninstall –g @angular-cli

npm cache clean [–force]

npm install –g @angular/cli@latest

Para actualizar en local:

rm -rf node\_modules

npm uninstall --save-dev angular-cli

npm install --save-dev @angular/cli@latest

npm install

Con ng – version muestra las versiones.

Si no utilizamos angular-cli, podemos instalar typescript:

npm intall –g typescript

Y luego los typings, que son librerías generales que se utilizan desde typescript

npm install –g typings

Crear proyecto con angular-cli

ng new proyecto1

Con angular-CLI podemos iniciar un proyecto que utilice SASS así:

ng new My\_New\_Project --style=sass

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. 

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.



Fichero package.json

En este fichero deben estar las dependencias del proyecto, tanto las de producción como las de desarrollo (devDependencies). 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 .angular-cli.json

En este fichero, entre otras cosas podemos establecer:

- El punto o clase inicial de ejecución.

- Carpeta con los fuentes.

- Carpeta para colocar los compilados

- etc

"$schema": "./node\_modules/@angular/cli/lib/config/schema.json",

"project": {

"name": "proyecto1"

},

"apps": [

{

"root": "src",

"outDir": "dist",

"assets": [

"assets",

"favicon.ico"

],

"index": "index.html",

"main": "main.ts",

"polyfills": "polyfills.ts",

"test": "test.ts",

"tsconfig": "tsconfig.app.json",

"testTsconfig": "tsconfig.spec.json",

"prefix": "app",

"styles": [

"styles.css"

],

"scripts": [],

"environmentSource": "environments/environment.ts",

"environments": {

"dev": "environments/environment.ts",

"prod": "environments/environment.prod.ts"

}

}

],

"e2e": {

"protractor": {

"config": "./protractor.conf.js"

}

},

"lint": [

{

"project": "src/tsconfig.app.json",

"exclude": "\*\*/node\_modules/\*\*"

},

{

"project": "src/tsconfig.spec.json",

"exclude": "\*\*/node\_modules/\*\*"

},

{

"project": "e2e/tsconfig.e2e.json",

"exclude": "\*\*/node\_modules/\*\*"

}

],

"test": {

"karma": {

"config": "./karma.conf.js"

}

},

"defaults": {

"styleExt": "css",

"class": {

"spec": false

},

"component": {}

}

}

Fichero main.ts

Ubicado en /src.

Al crear un proyecto con angular-cli, este será el fichero o la clase que establece el punto de entrada cuando se arranca con ng server.

/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 imágenes, videos, etc..

Integración sass en angular-CLI

Podemos generar un proyecto teniendo en cuenta desde el principio que vamos a trabajar con SASS.

ng new proyecto1–style=sass

También podemos indicar –style=less y –style=scss

Si tenemos el proyecto ya generado, podemos indicar a angular-CLI que pasamos a sass con:

ng set defaults.styleExt scss

En fichero .angular-cli.json nos va a quedar:

"defaults": {

"styleExt": "sass",

"component": {

}

}

Si establecemos la siguiente estructura de directorios:

|- src/

|- sass/

|- \_variables.sass

|- \_mixins. sass

|- styles.sass

y en styles. sass importamos varialbes y mixins con:

// src/sass/styles.sass

@import './variables';

@import './mixins';

Entonces para aplicar todo, los establecemos en .angular-CLI.json con.

"styles": [

"sass/styles. sass ",

\*\*// aquí podemos incluir ficheros css

],

Si construimos el proyecto con ng build - - prod, vemos como genera un css con los resultados de la compilación.

Compilar scss o sass

sass --watch input.scss output.css

You can watch and output to directories by using folder paths as your input and output, and separating them with a colon. In this example:

sass --watch app/sass:public/stylesheets

Sass would watch all files in the app/sass folder for changes, and compile CSS to the public/stylesheets folder.

Actualizar angular-cli y otros paquetes de un proyecto (local)

Podemos actualizar un proyecto (local) a esta nueva versión de angular-cli:

Global package:

npm uninstall -g @angular/cli

npm cache verify

# if npm version is < 5 then use `npm cache clean`

npm install -g @angular/cli@latest

Local project package:

rm -rf node\_modules dist # use rmdir /S/Q node\_modules dist in Windows Command Prompt; use rm -r -fo node\_modules,dist in Windows PowerShell

npm install --save-dev @angular/cli@latest

npm install

Podemos actualizar los paquetes de un proyecto con ng update.

npm update [-g] [<pkg>...]

aliases: up, upgrade

It will also install missing packages. As with all commands that install packages, the --dev flag will cause devDependencies to be processed as well.

If the -g flag is specified, this command will update globally installed packages.

If no package name is specified, all packages in the specified location (global or local) will be updated.

As of npm@2.6.1, the npm update will only inspect top-level packages. Prior versions of npm would also recursively inspect all dependencies. To get the old behavior, use npm --depth 9999 update.

EXAMPLES

IMPORTANT VERSION NOTE: these examples assume npm@2.6.1 or later. For older versions of npm, you must specify --depth 0 to get the behavior described below.

For the examples below, assume that the current package is app and it depends on dependencies, dep1 (dep2, .. etc.). The published versions of dep1 are:

{

"dist-tags": { "latest": "1.2.2" },

"versions": [

"1.2.2",

"1.2.1",

"1.2.0",

"1.1.2",

"1.1.1",

"1.0.0",

"0.4.1",

"0.4.0",

"0.2.0"

]

}

Caret Dependencies

If app's package.json contains:

"dependencies": {

"dep1": "^1.1.1"

}

Then npm update will install dep1@1.2.2, because 1.2.2 is latest and 1.2.2 satisfies ^1.1.1.

Tilde Dependencies

However, if app's package.json contains:

"dependencies": {

"dep1": "~1.1.1"

}

In this case, running npm update will install dep1@1.1.2. Even though the latest tag points to 1.2.2, this version does not satisfy ~1.1.1, which is equivalent to >=1.1.1 <1.2.0. So the highest-sorting version that satisfies ~1.1.1 is used, which is 1.1.2.

Caret Dependencies below 1.0.0

Suppose app has a caret dependency on a version below 1.0.0, for example:

"dependencies": {

"dep1": "^0.2.0"

}

npm update will install dep1@0.2.0, because there are no other versions which satisfy ^0.2.0.

If the dependence were on ^0.4.0:

"dependencies": {

"dep1": "^0.4.0"

}

Then npm update will install dep1@0.4.1, because that is the highest-sorting version that satisfies ^0.4.0 (>= 0.4.0 <0.5.0)

Recording Updates with --save

When you want to update a package and save the new version as the minimum required dependency in package.json, you can use npm update -S or npm update --save. For example if package.json contains:

"dependencies": {

"dep1": "^1.1.1"

}

Then npm update --save will install dep1@1.2.2 (i.e., latest), and package.json will be modified:

"dependencies": {

"dep1": "^1.2.2"

}

Note that npm will only write an updated version to package.json if it installs a new package.

Updating Globally-Installed Packages

npm update -g will apply the update action to each globally installed package that is outdated -- that is, has a version that is different from latest.

Creacion de módulos, componentes, servicios, etc.

Crear aplicación

ng new customer-app –routing

con –routing, generará app-roting.module.ts, e incluirá dicho modulo en app.module.ts

Crear modulos

ng generate module customers –routing

Se crea la carpeta customers y crea en ella los ficheros customers.module.ts (CustomersModule) y customers.routing.ts (CustomersRoutingModule).

Crear componentes

Existen tres clases de directivas:

1. Componentes. Que añaden elementos al DOM
2. Directivas de atributo. Que cambian el aspecto del DOM
3. Son directivas de este tipo \*ngIf, \*ngFor, \*ngSwitch. Eliminan o insertan elementos en el DOM.

En este apartado trataremos el primer punto, directivas del tipo componente.

ng generate component customers/customer-list

Tambien nos podemos colocar en la carpeta customer y

Ng generate component customer-list

Crear sevicios

ng generate service servicios/presupuestos

Crear directivas

Crear clases

ng g cl hero, crea la clase hero – ng generate class hero

Utilizar webpack directamente (ng eject)

Lo que hace este comando de angular-cli es poner a nuestra disposición el fichero de configuración de webpack. Lo que cambia en nuestro proyecto es:

1. Creación del fichero webpack.config.js
2. Se modifica el fichero angular-cli.json

- añade “ejected”:true

3. Se nuevas dependencia en el fichero package.json

Se modifica la sección de script:



Se tendrá por tanto que ejecutar un npm install, para que funcione todo.

Módulos, componentes, servicios y directivas.

Módulos

Un módulo en angular es el conjunto de código dedicado a un ámbito concreto de la aplicación o funcionalidad. Es una clase con el decorador @NgModule.

Una app pequeña podrá tener un solo módulo, en una aplicación mayor, como una app social, podrían distinguirse el módulo de post, el módulo de mensajes y el de seguidores, por ejemplo.

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).

D:.

│ app.component.css

│ app.component.html

│ app.component.spec.ts

│ app.component.ts

│ app.module.ts

│

├───model

│ user.ts

│

├───services

│ users.service.ts

│

└───users

users.component.html

users.component.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';

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

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

import { PipeeuroPipe } from './mispipes/pipeeuro.pipe';

import { Pipeeuro2Pipe } from './mispipes/pipeeuro2.pipe';

import { TestproveedoresComponent } from './testproveedores/testproveedores.component';

import { InicioComponent } from './inicio/inicio.component';

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

import { HeaderComponent } from './header/header.component';

import { AddproveeComponent } from './formproveedores/addprovee/addprovee.component';

import { AddpresComponent } from './formreactivopresupuesto/addpres/addpres.component';

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

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

// El servicio ProveedoresService es importado por su cliente mientras que

// PresupuestosService es importado aquí y estará disponimbre para cualquier componente

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

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

import {AutenticacionService} from './servicios/autenticacion.service';

import { ListapresComponent } from './formreactivopresupuesto/listapres/listapres.component';

import { EditpresComponent } from './formreactivopresupuesto/editpres/editpres.component';

import { RegistroComponent } from './autenticacion/registro/registro.component';

const routes: Routes = [

{path:'', component: InicioComponent},

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

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

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

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

{path: 'listapresupuestos', component: ListapresComponent},

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

{path: 'registro', component: RegistroComponent},

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

];

@NgModule({

declarations: [

AppComponent,

CopyrightComponent,

FechaactualComponent,

ViewmodeloComponent,

EjpropertybindingComponent,

EjeventbindingComponent,

Ej2waybindingComponent,

EjdirectivangifComponent,

EjdirectivangstyleComponent,

EjdirectivangclassComponent,

EjdirectivangforComponent,

DirectivamenuDirective,

C1Component,

TestdirectivamenuComponent,

UsodirectivainputpadreComponent,

UsodirectivainputhijoComponent,

PipeeuroPipe,

Pipeeuro2Pipe,

TestproveedoresComponent,

InicioComponent,

HeaderComponent,

AddproveeComponent,

AddpresComponent,

ListapresComponent,

EditpresComponent,

RegistroComponent

],

imports: [

BrowserModule,

RouterModule.forRoot(routes),

FormsModule,

ReactiveFormsModule,

HttpModule

],

providers: [

PresupuestosService,

AutenticacionService

],

bootstrap: [AppComponent]

})

export class AppModule { }

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

Un módulo es una clase con el decorador @NgModule, que recibe un objeto mediante el cual se configuran los metadatos. @NgModule es una función que toma un objeto metadato, cuyas propiedades definen el módulo. Las propiedades son:

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 { }

Modulos lazy

Son módulos cuyos components y servicios se cargan por demanda.

Ejemplo de un módulo de este tipo:

import { CommonModule } from '@angular/common';

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

import { PresupuestosRoutingModule } from './comunicacion-observable.routing';

import { DocFuentesModule } from '../doc-fuentes/doc-fuentes.module';

import { EmisorComponent } from './emisor/emisor.component';

import { Receptor1Component } from './receptores/receptor1/receptor1.component';

import { Receptor2Component } from './receptores/receptor2/receptor2.component';

import { MessageService } from './services/message.service';

@NgModule({

declarations: [

EmisorComponent,

Receptor1Component,

Receptor2Component

],

imports: [

CommonModule,

DocFuentesModule,

PresupuestosRoutingModule

],

providers: [MessageService

],

bootstrap: [EmisorComponent]

})

export class ComunicacionObservableModule { }

Para invocar componentes de este módulo, podemos desde el Modulo routing:

{path: 'comunicacion-observable', loadChildren: './comunicacion-observable/comunicacion-observable.module#ComunicacionObservableModule'},

y estableciendo en el módulo routing de este módulo lazy (ComunicacionObservableModule):

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

import { CommonModule, HashLocationStrategy, LocationStrategy } from '@angular/common';

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

import { EmisorComponent } from '../comunicacion-observable/emisor/emisor.component';

const routes: Routes = [

{ path: '', component: EmisorComponent},

{ path: '\*\*', redirectTo: ''},

];

@NgModule({

imports: [

CommonModule,

RouterModule.forChild(routes)

],

providers: [

{ provide: LocationStrategy, useClass: HashLocationStrategy }

],

exports: [

RouterModule

]

})

export class PresupuestosRoutingModule {

}

Componentes

Un componente controla una zona de espacio de la pantalla.



En el diseño de página de arriaba, podemos distinguir tres componenetes. El correspondiente al Side Bar, el Search Bar y el de Productos. A su vez, dentro del módulo Productos podemos tener otros módulos, como el de Rating. Esta modularización permite la reusabilidad.

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.

Creación de components con angular-cli

Desde la carpeta raíz del proyecto:

ng generate component <carpetadelcomponente/nombrecomponente>

o de forma abreviada:

ng g c <carpetadelcomponente/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 { }

Para tener nombres de clase o componentes con nomenclatura Camel, tendremos que utilizer el guión (‘-‘). Por ejemplo:

ng g component book-detail

creará el componente bookDetailComponent

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';

}

Servicios

Los servicios en angular son clases utilizadas por los componentes que proporcionan una funcionalidad a la aplicación, como puede ser el servio de loggin, el servicio de validación, acceso y actualización de los datos de usuario, etc.

Directivas

Cuando angular renderiza los templates para modificar el DOM, lo hace siguiendo las directivas.

Por ejemplo, cuando utilizamos la directiva autoGrow, angular hace que la caja aumente de tamaño cuando toma el foco.

<input type = “text” autoGrow />

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));

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 crearán los objetos para el paso a la vista y viceversa

Creamos el directorio “modelo” y la clase la definimos en /modelo/desarrollador.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 JavaScript entre dobles llaves, conocida como sintaxis moustache:

{{propiedad/objeto/expresión}}

Ejemplo 1:

creamos el componente viewmodelo

ng g c viewmodelo -spec false

En el fichero 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';

angularLogo = 'https://angular.io/resources/images/logos/angular2/angular.png';

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

}

y en fichero template:

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

<img [src]="angularLogo">

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

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

Se trata de un movimiento de datos de la 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:

[(NgModel)] = “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 y cambiar una clase (css) de un elemento del HTML del DOM.

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>

Si queremos extraer un indice desde la directive \*ngFor podemos obtenerlo así:

\*ngFor = “let curso of cursos; let ind = index”

Uso:

<div class="ui list" \*ngFor="let c of cities; let num = index">

<div class="item">{{ num+1 }} - {{ c }}</div>

</div>

Creación de directivas

Podemos crear una plantilla de directive con Angular CLI:

ng generate directive <nombre-clase-asociada>

Vamos a ver dos ejemplos:

Ejemplo 1. appDirectivamenu (con @HostBinding y @HostListener):

Generamos la plantilla con:

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:

/\* You can add global styles to this file, and also import other style files \*/

.item-orange {

border-left-style: solid;

border-width: 5px;

border-left-color: orange;

padding-left: 10px;

}

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 { }

Ejemplo 2. Directiva con @HostListener y @HostBinding

Con esta directiva se cambiarán atributos del elemento que la utilizan:

Generamos la directiva:

ng g directive misdirectivas/rainbow-directive

completamos /misdirectivas/rainbow-directive.directive.ts:

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

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

@Directive({

selector: '[appRainbowDirective]'

})

export class RainbowDirectiveDirective {

possibleColors = [

'darksalmon', 'hotpink', 'lightskyblue', 'goldenrod', 'peachpuff',

'mediumspringgreen', 'cornflowerblue', 'blanchedalmond', 'lightslategrey'

];

constructor() { }

@HostBinding ('style.color') color: string;

@HostBinding ('style.border-color') borderColor: string;

@HostListener ('keydown') newColor () {

const colorPick = Math.floor(Math.random() \* this.possibleColors.length);

this.color = this.possibleColors [colorPick];

this.borderColor = this.possibleColors [colorPick];

}

}

Creamos componente para test:

ng generate c uso-directiva-rainbow.

Para el test, este componente se puede dejar tal cual lo construye angular-cli.

Y en uso-directiva-rainbow.html:

<p>

uso-directiva-rainbow works!

<input type="text" appRainbowDirective>

</p>

En inicio.component.html, añadimos

<div class="container">

<app-uso-directiva-rainbow></app-uso-directiva-rainbow>

</div>

No olvidar añadir en app.module.ts, la directiva y en componente de test:

@NgModule({

declarations: [

…..

RainbowDirectiveDirective,

UsoDirectivaRainbowComponent

],

….

Ejemplo 3. Paso de datos a una direcitiva (@Directive, @HostListener y @Input)

Definimos la directiva:

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

@Directive({

selector: '[appConfirmDirective]'

})

export class ConfirmDirectiveDirective {

@Input('appConfirmDirective') message = 'Are you sure?';

@Input() functionToRun = () => { };

@HostListener('click')

confirm() {

if (window.confirm(this.message)) {

this.functionToRun();

}

}

}

Creamos el componente para test:

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

@Component({

selector: 'app-uso-directiva-confirm',

template: `

<header>Click the buttons to see the confirmation message.</header>

<p>

<button [appConfirmDirective]="'Are you sure you want to change the color to blue?'"

[functionToRun]="setBoxColor('dodgerblue')"

[disabled]="boxColor==='dodgerblue'">

Blue

</button>

<button appConfirmDirective="Are you sure you want to change the color to red?"

[functionToRun]="setBoxColor('crimson')"

[disabled]="boxColor==='crimson'">

Red

</button>

</p>

<div [style.backgroundColor]="boxColor"

class="box">

</div>

`,

styles: [`

div.box {

height: 50px;

width: 210px;

}

`]

})

export class UsoDirectivaConfirmComponent {

boxColor = 'dodgerblue';

setBoxColor(color: string) {

return () => this.boxColor = color;

}

}

En el ejemplo en nombre de la directiva “appConfirmDirective” se ha hecho coincidir con el alias @input. Por ello es posible utilizar de las dos formas que se muestran en el ejemplo.

Ejemplo 4. Paso de datos desde la directiva al componente.

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

@Directive({

selector: '[appTextSelector]',

exportAs: 'appTextSelector'

})

export class TextSnippetDirective {

@Output() textSelected = new EventEmitter<string>();

private \_snippets: string[] = [];

@HostListener('mouseup')

onSelected() {

const text = document.getSelection().toString();

if (text) {

this.\_snippets.push(text);

this.textSelected.emit(text);

}

}

get snippets(): string[] {

return this.\_snippets;

}

clear(): void {

this.\_snippets = [];

}

}

Desde la directiva se envía al componente una instancia de EventEmitter<string> que se ha decorado con @Output.

Cuando hay texto seleccionado cuando sube el botón de ratón, se utiliza la instancia decorada para emitir un evento con el propio texto seleccionado.

Vemos ahora el usuario de esta directiva:

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

@Component({

selector: 'app-uso-directiva-test-snippets',

template: `

<header [hidden]="quote.snippets.length">Select some of the text below.</header>

<header [hidden]="!quote.snippets.length">Select some more to add to the list.</header>

<p appTextSelector

#quote="appTextSelector"

(textSelected)="onSelected($event)">

{{movieQuote}}

</p>

<p>Last selection: <em>{{text || 'none'}}</em></p>

<div>

<a [hidden]="!quote.snippets.length"

(click)="quote.clear(); false" href="">

Clear

</a>

<ol>

<li \*ngFor="let snippet of quote.snippets">{{snippet}}</li>

</ol>

</div>

`,

styles: [`

em {

color: Green;

font-style: normal;

}

`]

})

export class TextSnippetComponent {

movieQuote = `

Didn’t see the first shark for about a half-hour. Tiger. 13-footer.

You know how you know that in the water, Chief? You can tell by lookin’

from the dorsal to the tail. What we didn’t know, was that our bomb

mission was so secret, no distress signal had been sent. They didn’t

even list us overdue for a week. Very first light, Chief, sharks come

cruisin’ by, so we formed ourselves into tight groups. It was sorta

like you see in the calendars, you know the infantry squares in the

old calendars like the Battle of Waterloo and the idea was the shark

come to the nearest man, that man he starts poundin’ and hollerin’

and sometimes that shark he go away… but sometimes he wouldn’t go away.`;

text: string;

onSelected(text: string) {

this.text = text;

}

}

Decoradores

Comunicación entre componentes

Decorador @Input



Se utiliza para pasar valores a propiedades de componentes hijos. El componente que recibe los valores (hijo) es el que utilizar la el decorador @Input. También es posible pasar funciones como se puede ver en el ejemplo 3 en el capitulo “Creación de directivas”.

ng g c usodirectivainputpadre -- spec false

ng 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

]

Decorador @Output

Decorador @viewChield

Con este decorador podemos acceder a componentes hijos (el padre puede acceder a componentes hijos).

For example, our <user-profile> component can have a method called sendData().

CHILD.COMPONENT.TS

@Component({

selector: 'user-profile'

})

export class UserProfile {

constructor() {}

sendData() {

//send data

}

}

Con el decorador @viewChild es posible acceder al método sendData de dos maneras:

1)

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

import { UserProfile } from '../user-profile';

@Component({

template: '<user-profile (click)="update()"></user-profile>',

})

export class MasterPage {

// ViewChild takes a class type or a reference name string.

// Here we are using the type

@ViewChild(UserProfile) userProfile: UserProfile

constructor() { }

ngAfterViewInit() {

// After the view is initialized, this.userProfile will be available

this.update();

}

update() {

this.userProfile.sendData();

}

}

2)

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

import { UserProfile } from '../user-profile';

@Component({

template: '<user-profile #myProfile (click)="update()"></user-profile>'

})

export class MasterPage {

@ViewChild('myProfile') userProfile: UserProfile

constructor() { }

update(){

this.userProfile.sendData();

}

}

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>

Introducción a los Observables

Angular necesita de los observables para conectarse con los servers del backend de una manera sencilla. Los observables están implementados en una librería llamada “Reactive Extensions” y es independiente de Angular. Para su utilización:

import 'rxjs/Rx';

La idea es que el Observer actúa con los ítems que emite el Observable. Y esto se hace de una manera asíncrona. Para ello el observador tendrá que subscribirse a un Observable.

Vemos un ejemplo:

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

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

import { Observable } from 'rxjs/Rx';

import 'rxjs/add/operator/filter';

import 'rxjs/add/operator/debounceTime';

import 'rxjs/add/operator/distinctUntilChanged';

import 'reflect-metadata';

import 'Zone.js';

@Component({

selector: 'app-root',

template: `

<input class="form-control" type="search"

[formControl]="searchControl">

`

})

export class AppComponent {

searchControl = new FormControl();

constructor(){

this.searchControl.valueChanges

.filter(text => text.length >= 3)

.debounceTime(400)

.distinctUntilChanged()

.subscribe(value => {

console.log(value);

});

}

}

<input class="form-control" type="search" [formControl]="searchControl">

Se utiliza la directiva formControl para enlazar con los componentes observables.

La clase formControl tiene la propiedad valueChanges que devuelve un observable, al cual nos subscribimos con .subscribe y establecemos la función callback que se deberá ejecutar cuando los datos se hayan cargado.

constructor(){

this.searchControl.valueChanges

.filter(text => text.length >= 3)

.debounceTime(400)

.distinctUntilChanged()

.subscribe(value => {

console.log(value);

});

}

En azul hemos marcado los operadores de los observables, con ellos hemos seleccionado las cadenas de más de 2 caracteres, testea cada 400 mseg y descarta si no hay cambios con respecto al último texto tratado

Cancelación de la subscripción

OnDestroy()

Put cleanup logic in ngOnDestroy(), the logic that must run before Angular destroys the directive.

This is the time to notify another part of the application that the component is going away.

This is the place to free resources that won't be garbage collected automatically. **Unsubscribe from Observables** and DOM events. Stop interval timers. Unregister all callbacks that this directive registered with global or application services. You risk memory leaks if you neglect to do so.

export class Receptor1Component implements OnInit, OnDestroy {

message: any;

subscription: Subscription;

constructor(private messageService: MessageService) {

// subscribe to home component messages

this.subscription = this.messageService.getMessage().subscribe(message => { this.message = message; });

}

ngOnDestroy() {

// unsubscribe to ensure no memory leaks

this.subscription.unsubscribe();

}

ngOnInit() {}

}

Servicios e inyección de dependencias - @Inyectable()

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.

Inyección de una instancia de una clase (singleton)

Vamos a crear el servicio proveedores en la carpeta servicios

ng generate service servicios/presupuestos

Al generar el servicio con angular-cli, se incorpora automáticamente la anotación @Inyectable en la clase.

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

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

import 'rxjs/Rx';

@Injectable()

export class PresupuestosService {

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

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

constructor(private http: Http) { }

postPresupuesto( presupuesto: any) { // Retorna Observable

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(); })

}

getPresupuestos () { // Retorna Observable

return this.http.get(this.presURL)

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

}

// también podemos definir esta función de la siguiente manera, si

// hacemos “import { Observable } from ‘rxjs/Rx’;

// getPresupuestos (): Observable<Response> {

getPresupuesto(idPres: string) { // Retorna Observable

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

return this.http.get( url)

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

}

putPresupuesto( presupuesto: any, id$: string) { // Retorna Observable

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();

});

}

delPresupuesto ( id$: string ) {

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

return this.http.delete( url )

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

}

}

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:

Utilización del servicio (aplicando la inyección):

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

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

providers: [PresupuestosService, PresupuestosService ],

Los servicios se inyectan en los componentes a través de métodos.

Creamos un cliente de este servicio con angular cli:

ng generate component formreactivopresupuesto.addpres

En .app/formreactivopresupuesto/addpres/addpres.component.ts :

Deberemos importar nuevamente el servicio:

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

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

@Component({

selector: 'app-addpres',

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

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

providers: [PresupuestosService]// es posible quitarlo de aquí si lo aplicamos en

//providers: de app.module.ts

})

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.module.ts y estará disponible para cualquier componente.***

El constructor de este componente queda:

constructor(private presupuestosService: PresupuestosService) {…

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

Angular para resolver la inyección de dependencias, mira el tipo del parámetro para crear una instancia o reusarla pues son singletos (en este caso de PresupuestosService) .

Inyección de constantes

En nuestro ngModule:

providers: [

{provide:'API\_URL',useValue: 'http://my.api.com/v1'}

]

En los components:

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

exportclass AnalyticsDemoComponent{

constructor(@Inject('API\_URL')apiUrl:string) {

// works!dosomething w/ apiUrl

}

}

De manera análoga, también podemos inyectar una tabla de constantes:

En nuestro ngModule:

export const examples: ExampleDef[] = [

{label: 'Intro', name: 'Root', path: '', component: IntroComponent},

{label: 'Injector', name: 'Injector', path: 'injector', component: UserDemoInjectorComponent},

{label: 'useClass (UserService)', name: 'UseClass', path: 'use-class', component: UserDemoComponent},

{label: 'factory (AnalyticsService)', name: 'UseClass', path: 'factory', component: AnalyticsDemoComponent},

];

…

providers: [

{ provide: APP\_BASE\_HREF, useValue: '/' },

{ provide: LocationStrategy, useClass: HashLocationStrategy },

{ provide: 'ExampleDefs', useValue: examples }

],

Y en el componente:

export class AppComponent {

constructor(

private router: Router,

@Inject('ExampleDefs') public examples: ExampleDef[]) {

}

}

Inyección del valor de retorno de una función.

Routing

Configuración del routing de una aplicación

Para habilitar esta funcionalidad de angular, tendremos que hacer lo 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 estados 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 email, 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>

…

Servicios de conexión con el servidor

En AngularJS se utilizaban las promesas para la comunicación con el servidor y anteriormente, se usaban los callbacks.

Angular utiliza los Observables devueltos por los métodos http para realizar las comunicaciones con el servidor. Estos observables se declaran en los servicios de comunicaciones con el servidor. En los componentes se podrán hacer las subscripciones a estos observables.

// las llamadas devuelven observables

leerDatos(): Observable<Response> {

// Se declara cómo va a ser la llamada

// ocultando los pormenores a los consumidores

return this.http

.get(`${this.urlBase}/recurso`);

// En este momento aún no se efectuó la llamada

}

escribirDatos(unDato): Observable<Response> {

// Los envíos de información deben configurarse a mano

// esto es fácilmente generalizable y reutilizable

let body = JSON.stringify(unDato);

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

let options = new RequestOptions({ headers: headers });

// declarar la llamada y retornar el observable

// las variables de configuración y los datos, van como parámetros

if (unDato.\_id) {

return this.http

.put(`${this.urlBase}/recurso/${unDato.\_id}`, body, options);

} else {

return this.http

.post(`${this.urlBase}/recurso`, body, options);

}

}

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.module.ts y estará disponible 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>

Base de datos en FireBase

https://firebase.google.com

Nos permitirá tener un servidor CRUD de manera sencilla.

Desarrollo cliente y servidor por separado- CORS

Adaptaciones en el servidor tomcat

Tanto el navegador como el servidor web, se protegen y no permiten que se realicen peticiones desde otro host+port distinto desde donde se ha hecho la carga inicial de la aplicación web. Es decir, que si iniciamos la navegación con <http://localhost:8080/CORS> y desde los javascript cargados se invocan servicios o peticiones http a otro host+port, como por ejemplo GET sobre localhost:8082/CORS2, se rechazarán por servidor o por navegador.

Por ejemplo, para permitir al servidor tomcat que atienda peticiones javascript a localhost:8082, (si la aplicación cliente se ha cargado desde localhost:80), podemos establecer el siguiente filtro en /conf/web.xml:

<filter>

<filter-name>CorsFilter</filter-name>

<filter-class>org.apache.catalina.filters.CorsFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>CorsFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

CORS, significa intercambio de recursos de origen cruzado.

De no ser así, obtendríamos un error como este:

Failed to load http://localhost:8082/CORS2/holaAjax.jsp: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:8080' is therefore not allowed access.

Adaptaciones en la aplicación backend con spring

En el controlador spring del backend podemos añadir la notación:

@CrossOrigin(origins = {"http://localhost:8082", "http://localhost:4200"})

Con lo que se permitirán peticiones que aplicaciones descargadas desde localhost:8082 o desde localhost:4200, puedan realizar. En este caso tendríamos clientes que hacen peticiones a localhost:8080, con lo que el servidor tomcat (por ejemplo), estaría escuchando o configurado sobre dicho puerto.

Deploy en Tomcat

Vamos a ver un ejemplo de una app angular sin parte front-end.

1. En tomcat creamos la carpeta …../webapp/Angular-SpringBoot-REST-JWT
2. En el index.html

<base href="/Angular-SpringBoot-REST-JWT/">

Para volver a desarrollo y seguir con las pruebas con “gn serve”, el fichero index.html tendrá que estar así:

<base ref =”/”>

Y cuando tenemos front y back en un mismo proyecto, y el desarrollo angular está en el path /webui, tendremos que poner:

<base href="/webui/">

1. Compilamos y optimizamos el proyecto con

2.1)

Ejecutamos con angular-cli local con:

npm run-script ng build --prod --base-href /Angular-SpringBoot-REST-JWT/

deberemos dejar en este caso el fichero index.html:

Si vamos a probar con ng serve, entonces –base-ref

Si obtenemos error, probar añadiendo - - aot=false

1. copiamos todos los archivos de /dist, que ha generado buid a /webapp/ Angular-SpringBoot-REST-JWT
2. probar con: <http://localhost:8080/Angular-SpringBoot-REST-JWT/>

Deploy en github

**Step 1**

Remove the dist directory from the project’s .gitignore file (it’s ignored by default by Yeoman).

**Step 2**

Make sure git knows about your subtree (the subfolder with your site).

git add dist && git commit -m "Initial dist subtree commit"

**Step 3**

Use subtree push to send it to the gh-pages branch on GitHub.

git subtree push --prefix dist origin gh-pages

Este comando crea la rama y copia el directorio dist en ella.

Para generar este directorio interesa:

ng build --base-href . -prod

Para actulizar los cambios en gh-pages, se ha necesitado eliminar la rama local y remota con git push <remote\_name> --delete <branch\_name>. Aplicado a nuestro caso de ejemplo:

git push origin - - delete gh-pages

Url de ejemplo:

<https://antonio63j.github.io/Proyecto1>

Deploy con NodeJS y Express

1. En index.html de la carpeta /dist dejamos:

Base-href=”/”

2. en /dist, creamos una carpeta con por ejemplo el nombre clientes-app, donde pasaremos todo el contenido de dist, salvo claro está la carpeta que acabamos de crear.

3. En la carpeta /dist, ejecutamos npm init, dejando todas las opciones por defecto, salvo el punto de entrada que asignaremos server.js

4. Ahora en esta misma carpeta /dist, instalamos express:

Npm intall express –save

$ ls -la

total 41

drwxr-xr-x 1 antonio 197121 0 ago 29 00:15 .

drwxr-xr-x 1 antonio 197121 0 ago 29 00:07 ..

drwxr-xr-x 1 antonio 197121 0 ago 29 00:08 clientes-app

drwxr-xr-x 1 antonio 197121 0 ago 29 00:15 node\_modules

-rw-r--r-- 1 antonio 197121 270 ago 29 00:15 package.json

-rw-r--r-- 1 antonio 197121 14299 ago 29 00:15 package-lock.json

En package.json, indicamos en “name”: “clientes-app”

{

"name": "clientes-app",

"version": "1.0.0",

"description": "",

"main": "server.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC",

"dependencies": {

"express": "^4.17.1"

}

}

Creamos el fichero server.js con el siguiente contenido:

let express = require('express');

let path = require('path');

let app = express();

let port = 8060;

app.use(express.static('clientes-app'));

app.get('\*', (req, res, next) => {

res.sendFile(path.resolve('clientes-app/index.html'));

});

app.listen(port, () => {

console.log('\n express escuchando sobre el puerto ' + port)

});

Lanzamos server.js con:

node server.js

Deployment en la nube Heroku

https://www.heroku.com

Otros sitios para hacer deploy

Para front-end

Now

heroku

Para Back-end

Mlab (free - permite base de datos como dbmongo)

FireBase

Herramientas / utilidades

tsun

Sirve para hacer pruebas pequeñas de typescript desde la línea de comandos:

Para instalar:

npm install –g tsun

C:\tmp>tsun

TSUN : TypeScript Upgraded Node

type in TypeScript expression to evaluate

type :help for commands in repl

> function inc(n:number):number{

..return ++n;

..}

undefined

> inc(3);

4