Todo comienza con la aparición AngularJs y otros frameworks como pueden ser KnockoutJs cuando el desarrollo Front-End se basaba casi en su totalidad en Jquery.

## **SPA**

Una ****web SPA**** es aquella web de ****una sola página****, es decir, todo el código html, javascript y css se cargan de una sola vez o se van cargando dinámicamente cuando la página lo requiere. Con el objetivo principal de dar una experiencia al usuario más fluida.

****Angular 2, 4, 5 y ahora 6**** es lo que todo el mundo llama ****Angular.****

IMG_256

****AngularJs y Angular**** son ****dos framework diferentes****, que prácticamente sólo comparten el nombre.

****Angular es un framework multiplataforma**** para crear aplicaciones del ****lado cliente en HTML y JavaScript****. Estas aplicaciones se pueden ejecutar en un dispositivo móvil o en el escritorio con ayuda de otras librerías.

Angular adopta el **estándar de componentes.**

Enfocado para desarrollo rápido y fiable:

* ****Angular es dogmático****: Toma muchas decisiones por defecto como puede ser la conectividad de red, la elección del idioma, herramientas de compilación,…
* ****Angular es escalable****: Permite la participación de muchos desarrolladores en la construcción de una aplicación.
* ****Angular tiene un fuerte ecosistema****: Es Open Source pero es mantenido y desarrollado por Google. Existen multitud de herramientas y bibliotecas reutilizables en internet.

Una ****aplicación Angular**** se compone de un ****set de componentes y servicios**** que proveen funcionalidad a estos componentes

Para ****agrupar**** todos esos ****componentes y servicios**** se definen lo que llamamos ****módulos de Angular****.

Una ****aplicación de Angular siempre**** tendrá por lo menos ****un módulo****, que llamaremos ****root module****, desde el cual arranca la aplicación.

## **¿Qué es un componente Web?**

Un ****componente web**** es una etiqueta ****HTML personalizada, reutilizable y encapsulada**** disponible para usar en cualquier aplicación web.

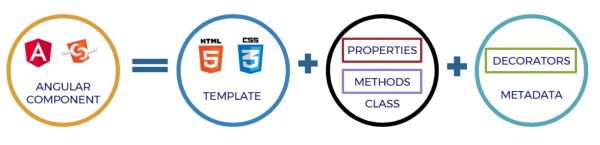
Reducen la complejidad de las aplicaciones web:

* ****Simplifican**** el proceso de ****creación de aplicaciones Web****.
* Promueven los principios de ****reutilización**** de código.
* Proporcionan la importante capacidad de ****encapsular el CSS y el JavaScript en cada componente****.

## **¿Qué es un componente de Angular?**

Un componente de Angular se compone de:

* ****Template****: En el que tenemos el HTML y el CSS
* ****Class****: Una clase en la que se define toda la funcionalidad con propiedades y métodos.
* ****Metadatos****: A través de los decoradores se le indica a Angular qué características y qué propiedades tiene el componente.



****Antes de empezar**** a programar en Angular necesitamos tomar algunas ****decisiones****. Tener un ****buen entorno de desarrollo**** con las correctas herramientas nos permitirá ser mucho más ágiles en el desarrollo.

## **Elección del lenguaje de programación**

Podemos ****programar Angular en Javascript o en Typescript****. Dado todos los beneficios de ****Typescript,****que ahora veremos, y que toda la comunidad, incluso el propio Angular, se desarrolla con Typescript, todo lo que hagamos durante este curso lo realizaremos con Typescript.

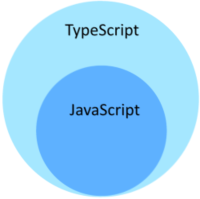
### **¿Qué es Typescript?**

* Typescript es un lenguaje ****open source creado y mantenido por Microsoft****.
* Typescript es un ****superconjunto de Javascript**** que ****compila a Javascript****.
* Typescript ****extiende la sintaxis**** de Javascript.

Typescript nos brinda cuatro ventajas muy importantes para los desarrolladores:

* ****Javascript escalable.****
* ****Sistema de tipado.****
* Buenas ****herramientas**** gracias al sistema de tipado.

****Features del futuro**** hoy.



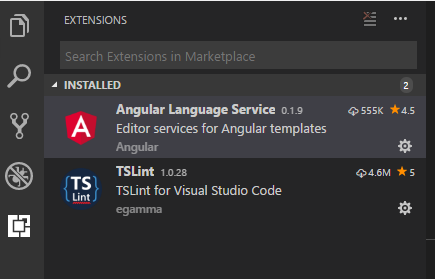
## **Elección de un IDE**

Hay muchos editores que soportan Typescript, incluso podeis escoger un editor que no lo soporte.

Durante el curso ****vamos a utilizar Visual Studio Code****. Este IDE se puede instalar en Linux, Mac o Windows, y trae ****soporte completo para Typescript****.

Lo podéis descargar desde este enlace **[Visual Studio Code](https://code.visualstudio.com/" \o "Visual Studio Code)**

Os recomiendo dos extensiones que podéis instalar en Vs Code que nos ayudan en nuestro día a día con Angular.



## **Instalación de NPM (Node package manager)**

### **¿Qué es NPM?**

Es el ****gestor de paquetes de Javascript****. Gracias a él tenemos disponible una ****línea de comandos****con la que podemos **instalar librerías, paquetes y aplicaciones** con todas sus dependencias.

****Necesitamos NPM**** para instalar todas las **librerías de Angular**, librerías de nuestras aplicaciones y compilar nuestra aplicación.



Lo podemos descarga desde este enlace **[Npm](https://www.npmjs.com/" \o "Npm)** (Ir a la página, bajar al apartado de get started, y en la página de node descargar la versión para vuestro ordenador )

### **CLI (Command line interface)**

En el curso utilizaremos bastante la línea de comandos. Si sois usuarios de Windows, **recomiendo que os descarguéis Cmder**. Cmder es una consola portable y vitaminada para Windows. De todas formas **podéis utilizar vuestro terminal** de comandos de vuestro sistema operativo.

La podeis descargar desde este enlace **[Cmder](http://cmder.net/" \o "Cmder)**

*Una vez tengamos instalado npm e instalado nuestra IDE favorito estamos listos para comenzar la puesta en marcha y la instalación de Angular en la próxima lección.*

Para hacer una aplicación de Angular, siguiendo buenas prácticas, es recomendable empezar con ****Angular CLI****.

Nos vamos a línea de comandos, para comprobar que tenemos instalado NPM con el comando.

npm --version

Durante el curso iremos viendo los comandos de Angular CLI, pero si queréis saber más información la podéis encontrar en **[Angular CLI](https://cli.angular.io/" \o "Angular CLI)**

Volvemos a la línea de comando e instalamos Angular CLI con el comando.

npm install -g @angular/cli

Para comprobar que lo tenemos instalado.

ng --version

Ahora vamos a generar nuestra primera aplicación de Angular. En nuestro command line nos vamos a un directorio que tengamos para realizar esta práctica y escribimos.

ng new my-first-angular-app

Todos los ficheros de la aplicación están dentro de la carpeta generada en la ruta /src. Y los ficheros propios de nuestra aplicación los encontramos en /src/app.

Antes de poder iniciar nuestra aplicación necesitamos instalar todas los ****paquetes**** necesarios de nuestra aplicación, estos se encuentran definidos en el fichero ****package.json****. En él encontramos dos secciones ****“dependencies”**** (todos los paquetes para producción y desarrollo) y ****“devDependencies”**** (todos los paquetes para desarrollo).

Para instalar estas librerías o si añadimos una nueva tenemos que ejecutar el comando.

npm install

****Angular CLI utiliza webpack**** por debajo para optimizar toda nuestra build y el fichero ****angular.json****nos permite tocar ciertas cosas de esa configuración de webpack.

Si queremos dejar de utilizar Angular CLI (poco recomendable y no lo haremos en el curso) utilizaremos el comando.

ng eject

En el fichero ****karma.conf.js**** encontramos toda la configuración necesaria para realizar ****test unitarios****.

El ****package-lock.json**** no tiene nada que ver con Angular CLI, nos lo crea si tenemos una versión nueva de Node. Se genera automáticamente en cada npm install.

El fichero ****protractor.conf.js**** es toda la configuración necesaria para realizar ****test end to end**** de nuestra aplicación, está relacionado con la carpeta que vemos de e2e.

El fichero ****tsconfig.json**** define la ****configuración de typescript****, en él podemos cambiar cosas como el target de nuestra aplicación de es5 a es6.

En el fichero ****tslint.json**** podemos encontrar las ****reglas**** o normas que los ficheros de nuestro proyecto tiene que seguir. El objetivo de Tslint es que todos los programadores de nuestra aplicación sigan las normas establecidas.

Para ejecutar nuestra aplicación en desarrollo, ejecutaremos el siguiente comando, el cual nos levanta un servidor node y nos sirve la aplicación en http://localhost:4200.

ng serve -o

Puedes encontrar los ejemplos usados en **[github](https://github.com/OpenWebinarsNet/Angular-desde-cero)** en la carpeta my-first-angular-app .

Tal y como comentamos anteriormente Typescript es un superconjunto de Javascript y vamos a entenderlo con un ejemplo.

Como hemos visto en la última lección, Angular CLI se encarga de transpilar typescript a javascript y servirlo en el navegador. Pero para trabajar sólo con Typescript necesitamos descargarnos su librería con el siguiente comando:

npm install -g typescript

Comprobamos que lo tenemos correctamente instalado con:

tsc --version

Inicializamos la carpeta en la que estamos como un proyecto de Typescript con el comando:

tsc --init

Para que Typescript transpile todos los archivos cada vez que estos cambien, escribimos en la línea de comandos.

tsc --watch

Definición de tipos:  
****Boolean, Number, String, Array, Tuples, Enums, Any, Void, Null, Undefined****

Definición de ****variables con let****. Let es más seguro utilizarlo.

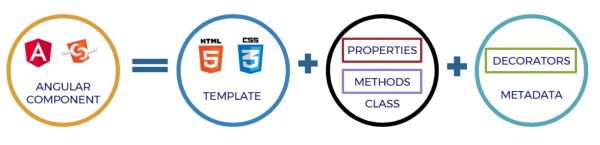
Con Typescript tenemos clases, herencia, polimorfismo y posibilidad de declarar clases abstractas.

Para obtener ayuda F12 y se presentara la documentación: archivo de definiciones

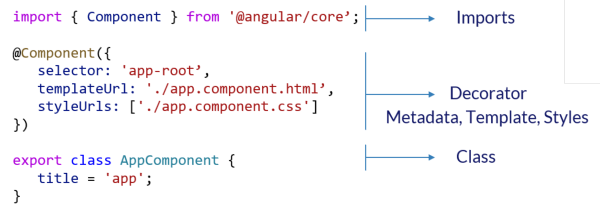
## **¿Qué es un componente de Angular?**

Un componente de Angular se compone de:

* ****Template****: En el que tenemos el HTML y el CSS.
* ****Class****: Una clase en la que se define toda la funcionalidad con propiedades y métodos.
* ****Metadatos****: A través de los decoradores se le indica a Angular que características y que propiedades tiene el componente.



## **Estructura de un componente**



El componente tiene tres bloques diferenciados:

1. Imports.
2. Decorator (Metadata, Template, Styles).
3. Class.

## **Imports**

Utilizaremos la sección de import para importar una librería, un módulo nuestro o un módulo del propio Angular.

En nuestro componente utilizamos el decorador @Component, para ello necesitamos decirle a nuestra clase donde encontrar ese decorador. Lo hacemos importando el miembro Component desde el módulo de ‘@angular/core’ de Angular con la línea:

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

## **Decorator**

Una clase es interpretada por Angular como un componente gracias a su decorador. Este decorador, le provee de los metadatos necesarios para instanciar el componente, contruir la vista y las interaciones del componente.

Recordar que un decorador es una función que añade metadatos a una clase o un miembro.  
Un decorador siempre tiene como prefijo una @.  
Un decorador siempre se posiciona antes de lo que estamos decorando. Es decir, el decorador es toda la configuración de un componente.

@Component({

selector: 'app-root’,

templateUrl: './app.component.html’,

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

})

Propiedades del decorador @Component:

* ****selector****: es el tag html que define como vamos a llamar a nuestro componente para usarlo en nuestra aplicación.
* ****template****: definición inline del HTML de nuestro componente.
* ****templateURL****: nos permite especificar la ruta del archivo que contiene el HTML del componente.
* ****styles****: definición inline de los estilos de nuestro componente.
* ****stylesURL****: nos permite especificar las rutas a las hojas de estilos.
* ****animations****: lista de animaciones de nuestro componente.
* ****viewProviders****: lista de los servicios que el componente puede utilizar.

## **Class**

Si has trabajado con algún lenguaje orientado a objetos esto te resultara familiar.  
Una clase es una estructura que nos permite crear un tipo con propiedades que la definen y métodos los cuales proveen funcionalidad.

export class AppComponent {

title = 'app';

}

Una convención de Angular es nombrar a la clase de nuestro componente por el nombre del archivo y siempre con el sufijo Component. Por ejemplo, si nuestro fichero es weather-info.component.ts, la clase del componente se llamara WeatherInfoComponent.

Esta clase la marcaremos como export para que otros componentes de la aplicación puedan utilizarlo.

En el body de la clase tenemos las propiedades y los métodos.

## **Creación de un componente**

Podemos crear los ficheros a mano, pero Angular CLI nos da la posibilidad de crear componentes y que éstos ya se incluyan dentro del módulo sin que nosotros tengamos que hacer nada más.

Nos vamos a línea de comandos y dentro de la carpeta src/app de nuestra aplicación, escribimos:

ng generate component application-info

Una vez nos hemos creado nuestro primer componente con Angular CLI, vamos a configurarlo y crearlo desde el código, viendo sus propiedades.

Ejemplo de creación de un componente inline (no es recomendable si excede tres líneas de html)

import { Componet } from '@angular/core'

@Component({

selector: 'app-application-info',

template: `

<p>{{info}}</p>

`,

styles: [

`

p {

color: red;

}

`

]

})export class ApplicationInfoComponent {

info: string = 'Información de mi primera aplicación Angular';

}

Ejemplo de creación de un componente

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

@Component({

selector: 'app-application-info',

templateUrl: './application-info.component.html',

styleUrls: ['./application-info.component.css']

})export class ApplicationInfoComponent {

info = 'Información de mi primera aplicación Angular';

}

Para que una aplicación de Angular inicie, necesitamos decirle que ****cargue nuestro componente**raíz** a través de un proceso que se llama ****bootstrapping****.

Lo primero que hacemos es configurar el archivo index.html para alojar nuestra aplicación Angular.

Toda ****aplicación Angular tiene un archivo index.html****, este suele ser la única página web real de la aplicación, por esto se les denomina Single Page Application (SPA). Luego Angular se encarga de ir insertando en el DOM los módulos y componentes que necesitemos.

Dentro del ****tag body**** del fichero ****index.html****, encontramos el ****tag del primer componente que vamos a renderizar**** en nuestra página.

El ****fichero main.ts**** es el ****principal punto de entrada**** de una aplicación Angular, es el encargado de bootstraps el ****módulo inicial (AppModule)****.

En el fichero ****angular.json**** podemos cambiar el nombre o ruta de nuestro index.html o el main.ts.

En el fichero ****app.module.ts**** encontramos la ****propiedad bootstrap****, ésta define el componente de arranque cuando realizamos el bootstrap del módulo.

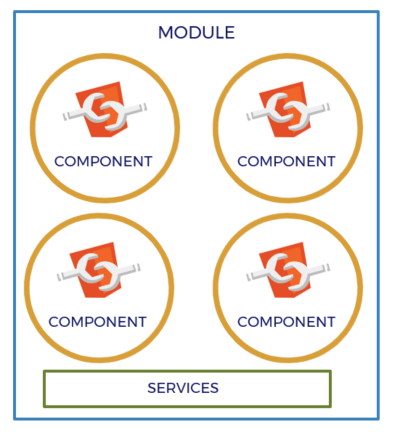
## **Debug nuestra aplicación Angular**

En las developer tools (pulsando F12 en Chrome) y vamos a sources, en la sección de webpack, en la carpeta “./src/app” encontramos nuestros ficheros “.ts”. Podemos hacer puntos de parada en cualquier de ellos, para depurar el código.

*Truco: En las developer tools podemos pulsar******Ctrl + o******para******buscar un archivo******, mucho más rápido!*

## **¿Qué es un módulo de Angular?**

Un módulo de Angular es una clase con el ****decorador NgModule****. Imagina que tenemos una aplicación grande en Angular y tenemos cientos de componentes, eso es algo muy difícil de mantener, por esta razón nacen los módulos.



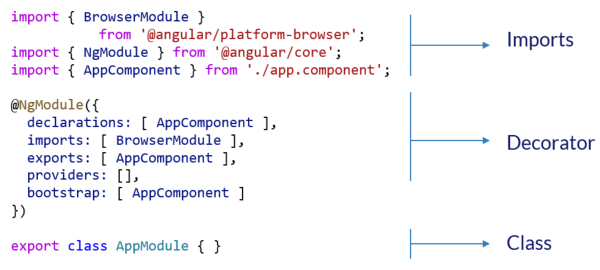
Los módulos sirven para:

* ****Organizar**** componentes, directivas, pipes y servicios en ****bloques lógicos de funcionalidad****.
* ****Extender**** las ****funcionalidades**** con bibliotecas externas.
* ****Importar**** y ****re-exportar clases**** de otros módulos.
* Añadir ****servicios**** a la aplicación.

Los módulos pueden ser cargados cuando la aplicación se inicia, o podemos utilizar lazy-loading para ir cargándolos de forma asíncrona bajo demanda de la aplicación.

Lazy-loading esta fuera del alcance de este curso, si una vez lo termines tienes interés de saber como aplicarlo, he realizado otro curso “Angular buenas prácticas” en el que podrás encontrar como utilizarlo.

## **Estructura de un módulo**



El módulo tiene tres bloques diferenciados:

1. Imports
2. Decorator
3. Class

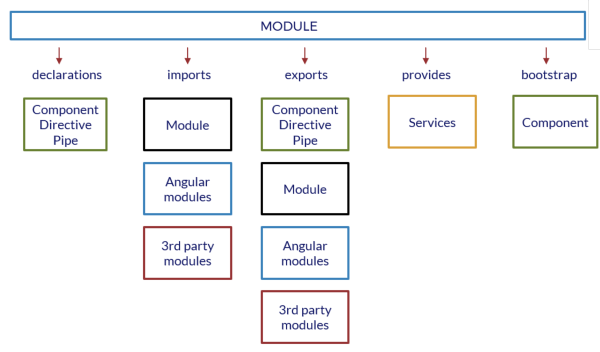
## **Imports**

Como hemos visto en la sección de Typescript, para poder utilizar funciones o clases externas, necesitamos realizar imports.  
Import nos permite utilizar miembros exportables de módulos externos. Utilizaremos la sección de import para importar una librería, un módulo nuestro o un módulo del propio Angular.

En nuestro módulo utilizamos el decorador @NgModule, para ello necesitamos decirle a nuestra clase donde encontrar ese decorador. Lo hacemos importando el miembro NgModule desde el módulo de ‘@angular/core’ de Angular con la línea:

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

## **Decorator**



Propiedades del decorador @NgModule:

* ****declarations****: En esta propiedad ****añadimos los componentes, directivas o pipes**** de este módulo. Hay que tener en cuenta los siguientes puntos:
  + Estos componentes, directivas o pipes son ****privados por defecto****, es decir, sólo son accesibles desde otros componentes, directivas o pipes declarados en el mismo módulo.
  + ****Cada componente, directiva o pipe**** deben pertenecer a ****un módulo****, es decir, no se puede declarar un componente, directiva o pipe en dos módulos.
  + Angular se encarga de la resolución de cada selector de los componentes, directivas o pipes dentro de ese módulo.
* ****imports****: En esta propiedad ****importaremos módulos**** que exportan componentes, directivas o pipes con el fin de ****extender**** la ****funcionalidad**** de nuestro módulo. Angular se encarga de la resolución de cada selector de los componentes, directivas o pipes exportados en cada módulo.
* ****exports****: Con la propiedad exports, Angular nos permite ****compartir**** a otros módulos los ****componentes, directivas o pipes declarados en el módulo****. También nos permite re-exportar módulos con el fin de importarlos en un sólo módulo. Hay que tener en cuenta los siguientes puntos:
  + ****Exporta**** sólo lo que otros ****módulos necesarios****.
  + Podemos exportar módulos sin importarlos.
  + ****No exportar nunca un servicio****. Cada servicio contenido en el array de providers está registrado en el inyector de dependencias de Angular y está disponible para cualquier clase de la aplicación (excepto cuando utilizamos lazy-loading en la aplicación).
* ****providers****: Nos permite ****registrar servicios**** a nivel de módulos, es muy parecido a la propiedad que vimos de los metadatos de un Componente llamada viewProviders.
* ****bootstrap****: Recoge el ****componente**** que es el ****punto de inicio de la**aplicación**, sólo tendremos esta propiedad cuando el módulo sea el inicio de nuestra aplicación.

## **Class**

Al contrario de los componentes en los que la clase tiene propiedades y métodos la clase de un módulo es muy sencilla, ya que el decorador NgModule genera por debajo todo el código necesario por nosotros.

export class AppModule { }

## **Creación de un módulo**

Al igual que los componentes, podemos crear los ficheros de un módulo a mano, pero Angular CLI nos da la posibilidad de crear módulos.

Nos vamos a línea de comandos y dentro de la carpeta src/app de nuestra aplicación, escribimos.

ng generate module login

Una vez nos hemos creado nuestro primer módulo con Angular CLI, vamos a configurarlo y crearlo desde el código, viendo sus propiedades.

Ejemplo de creación del módulo AppModule

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 { }

Ya hemos visto que es y como crear un componente y un módulo. El resto del curso nos vamos a centrar en realizar un ejemplo de aplicación para ir aprendiendo todo lo que nos queda, así que vamos realizar la base del proyecto.

Lo primero que haremos será crearnos una nueva aplicación Angular a la que llamaremos Aweather, vamos a realizar una aplicación con la previsión del tiempo!

Para crear una nueva aplicación, como hemos visto en una sección anterior, ejecutaremos en una carpeta el siguiente comando.

ng new aweather

Nos vamos a la carpeta raíz del proyecto.

cd aweather

Incluiremos la librería de ****Material**** para Angular, la cual nos provee de módulos y componentes creados por Google para crear nuestra UI de forma rápida y eficaz.

El enlace con toda la información es **[Materio IO](https://material.angular.io/)**

En la carpeta raíz del proyecto en la que estamos, ejecutamos en la línea de comandos.

npm install --save @angular/material @angular/cdk

También vamos a incluir una librería de ****iconos del tiempo****, al igual que antes, en la carpeta del proyecto ejecutamos.

Para más información sobre los iconos **[weather icons](http://erikflowers.github.io/weather-icons/)**

npm install weathericons

Otra librería para incluir es ****font-awesome****, es una librería de iconos. **[font-awesome](https://fontawesome.com/)**

npm i @fortawesome/fontawesome-free-webfonts

Ahora para incluir el css referente a estas librería, tenemos que importarlo en nuestro css. Nos vamos al archivo src/styles.css y escribimos las siguientes línea css

@import '../node\_modules/weathericons/css/weather-icons.css';

@import '../node\_modules/weathericons/css/weather-icons-wind.css';

@import '../node\_modules/@fortawesome/fontawesome-free-webfonts/css/fontawesome.css';

@import '../node\_modules/@fortawesome/fontawesome-free-webfonts/css/fa-regular.css';

@import '../node\_modules/@fortawesome/fontawesome-free-webfonts/css/fa-solid.css';

@import '../node\_modules/@fortawesome/fontawesome-free-webfonts/css/fa-brands.css';

@import '~@angular/material/prebuilt-themes/indigo-pink.css';

body {

margin: 0px;

padding: 0;

color: #455a64;

font-family: "Arial" !important;

font-size: 0.9em !important;

font-weight: normal;

line-height: 1.3;

background-color: #fff;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\* MAT CARD \*\*\*\*\*\*\*\*\*\*\*\*\*/

.mat-card {

border-radius: 6px !important;

margin: 10px !important;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\* FIN MAT CARD \*\*\*\*\*\*\*\*\*\*\*\*\*/

Ya tenemos todo listo para empezar con nuestro proyecto del tiempo!

Si teneis cualquier problema, la aplicación base está en **[github](https://github.com/SamazoOo/Angular-desde-cero)** en la carpeta aweather-inicial

**Eventos y ciclos de vida de un Componente**

Ya hemos visto como crear un componente, con los imports necesarios, el decorador y la clase.  
Angular maneja el ciclo de vida de un componente, es decir, Angular lo crea, lo renderiza, y lo destruye removiendolo del DOM.

Angular nos ofrece ****interfaces****, las cuales proveen un ****evento**** en los momentos clave del ****ciclo de vida**** y nos permiten ****actuar cuando ocurren****.

Estos eventos aplican también a directivas.

Los eventos del ciclo de vida que podemos implementar son:

* ****ngOnChanges****: Este evento se ejecuta antes de ngOnInit y cada vez que ocurre un cambio en una de las propiedades de nuestro componente. Recibe el objecto SimpleChanges en el que tenemos el valor actual y el previo.
* ****ngOnInit****: Se ejecuta una sóla vez al inicializar el componente.
* ****ngDoCheck****: Se ejecuta inmediatamente después de ngOnChanges. Es muy parecido a ngOnChanges pero no recibe ningún objeto y sólo tenemos el valora actual y no el previo.
* ****ngAfterContentInit****: Se ejecuta una sóla vez cuando Angular termina de inicializar el componente. Después del primer ngDoCheck.
* ****ngAfterContentChecked****: Se ejecuta después de cada chequeo del componente.
* ****ngAfterViewInit****: Se ejecuta una sóla vez cuando Angular termina de inicializar la vista del componente y sus componentes anidados.
* ****ngAfterViewChecked****: Se ejecuta después de cada chequeo de la vista del componente.
* ****ngOnDestroy****: Se ejecutará justo antes de que Angular destruya el componente.

Los más utilizados e ****importantes**** son ****ngOnInit, ngOnChanges y ngOnDestroy.****

**Configuración de eventos**

Vamos a la aplicación que hemos preparado aweather a la carpeta src/app y escribimos en la línea de comandos:

ng generate component weather-card --spec false

En la carpeta src/app creamos un nuevo fichero llamado “weather.model.ts”, este fichero es nuestro modelo para el componente weather-card.

En el fichero escribimos:

export class Weather {

city: string;

date: string;

temperature: number;

info: string;

windspeed: number;

winddirection: number;

humidity: number;

pressure: number;

maxtemperature: number;

mintemperature: number;

sunrise: string;

sunset: string;

get percentToSunset(): number {

if (this.sunrise === undefined) {

return 0;

}

const sunriseHours = this.sunrise.replace(' am', '').split(':');

const sunsetHours = this.sunset.replace(' pm', '').split(':');

let percentToSunset = 0;

const today = new Date();

const sunriseDate = new Date(today.getFullYear(), today.getMonth(), today.getDate(),

Number(sunriseHours[0]), Number(sunriseHours[1]), 0);

const sunsetDate = new Date(today.getFullYear(), today.getMonth(), today.getDate(),

Number(sunsetHours[0]) + 12, Number(sunsetHours[1]), 0);

const minutesSunAllDay = Math.abs(sunsetDate.getTime() - sunriseDate.getTime());

const minutesToSunset = today.getTime() - sunsetDate.getTime();

if (minutesToSunset > 0) {

return percentToSunset;

}

percentToSunset = 50 - ((Math.abs(minutesToSunset) \* 50) / minutesSunAllDay);

return Math.round(percentToSunset);

}

get hourMessure(): string {

if (this.date === undefined) {

return '';

}

const dateMeassure = new Date(this.date.split(', ')[1].substring(0, this.date.split(', ')[1].lastIndexOf(' ')));

return `${dateMeassure.getHours().toString().padStart(2, '0')}:${dateMeassure.getMinutes().toString().padStart(2, '0')} ${dateMeassure.getHours() > 14 ? 'pm' : 'am'}`;

}

constructor() { }

}

Nos vamos al fichero app.module.ts y vamos añadir módulos que vamos a utilizar de la librería material que instalamos antes. Los módulos que vamos añadir son MatCardModule, MatButtonModule, MatDividerModule, MatGridListModule, MatProgressSpinnerModule.

El fichero debe quedar de la siguiente manera:

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

MatCardModule,

MatButtonModule,

MatDividerModule,

MatGridListModule,

MatProgressSpinnerModule

} from '@angular/material';

import { AppComponent } from './app.component';import { WeatherCardComponent } from './weather-card/weather-card.component';

@NgModule({

declarations: [

AppComponent,

WeatherCardComponent

],

imports: [

BrowserModule,

MatCardModule,

MatButtonModule,

MatDividerModule,

MatGridListModule,

MatProgressSpinnerModule

],

providers: [],

bootstrap: [AppComponent]

})export class AppModule { }

Ahora vamos fichero html del nuevo componente que hemos creado weather-card.component.html y escribimos el siguiente html.

<mat-card class="weather-card">

<mat-card-header class="weather-header">

<div mat-card-avatar class="weather-header-icon">

<i class="wi wi-day-sunny"></i>

</div>

<mat-card-title>Madrid</mat-card-title>

<mat-card-subtitle>14:14 pm</mat-card-subtitle>

</mat-card-header>

<mat-card-content>

<div class="weather-card-content-info">

<mat-grid-list cols="4" rowHeight="30px">

<mat-grid-tile>

<i class="wi wi-strong-wind"></i> 22 km/h</mat-grid-tile>

<mat-grid-tile>

<i class="wi wi-wind from-270-deg"></i>

</mat-grid-tile>

<mat-grid-tile>

<i class="wi wi-humidity"></i> 70 %</mat-grid-tile>

<mat-grid-tile>

<i class="wi wi-barometer"></i> 1002.0

</mat-grid-tile>

</mat-grid-list>

</div>

<mat-grid-list cols="2" rowHeight="30px">

<mat-grid-tile colspan="2" class="weather-info-text">

Soleado

</mat-grid-tile>

<mat-grid-tile class="weather-info-temp-up">

<i class="wi wi-direction-up"></i> 17 ºC

</mat-grid-tile>

<mat-grid-tile class="weather-info-temp-down">

<i class="wi wi-direction-down"></i> 4 ºC

</mat-grid-tile>

</mat-grid-list>

<mat-grid-list cols="1" rowHeight="80px">

<mat-grid-tile class="weather-info-temp">

<div class="number">11</div>

<div class="measure">ºC</div>

</mat-grid-tile>

</mat-grid-list>

<mat-divider></mat-divider>

<mat-grid-list cols="5" rowHeight="120px">

<mat-grid-tile class="weather-info-sunrise">

<div class="hour">6:31 am</div>

<i class="wi wi-sunrise"></i>

</mat-grid-tile>

<mat-grid-tile colspan="3">

<mat-progress-spinner class="sunrise-sunset-spinner" diameter="120" strokeWidth="110" mode="determinate" value="30">

</mat-progress-spinner>

</mat-grid-tile>

<mat-grid-tile class="weather-info-sunset">

<div class="hour">18:48 pm</div>

<i class="wi wi-sunset"></i>

</mat-grid-tile>

</mat-grid-list>

</mat-card-content>

</mat-card>

Ahora vamos agregar el css necesario al componente, por lo que en el archivo weather-card.component.css añadimos el siguiente código.

.weather-card {

width: 400px;

padding: 0px;

}

.weather-header {

position: relative;

padding: 12px;

background:transparent;

min-height: 60px;

}.weather-header:before {

position: absolute;

display: block;

width: 100%;

height: 100%;

left: 0;

top: 0;

content: ' ';

z-index: 1;

opacity: 0.8;

background-image: url('https://drive.google.com/uc?id=1DRGmofyk2KbKvY8HF8vvqcBGj7VRXKnY');

}

.weather-header mat-card-title {

margin-left: 20px;

color: #fff ;

z-index: 2;

position: relative;

}

.weather-header mat-card-subtitle {

margin-left: 20px;

font-size:16px;

color: #fff;

z-index: 2;

position: relative;

}

.weather-header-icon {

font-size:48px;

color: #fff;

z-index: 2;

}

.weather-card-content-info {

background-color: #e0e0e0 !important;

}

.weather-card-content-info .wi{

font-size: 22px;

margin-right: 4px;

}

.weather-info-text {

font-size:22px;

margin-top:4px;

}

.weather-info-temp-up {

font-size:24px;

}

.weather-info-temp-up .wi{

font-size: 30px;

margin-right: 4px;

color:#D32F2F;

}

.weather-info-temp-down {

font-size:24px;

}

.weather-info-temp-down .wi{

font-size: 30px;

margin-right: 4px;

color:#1E88E5;

}

.weather-info-temp .number {

font-size: 80px;

font-weight: bolder;

}

.weather-info-temp .measure {

font-size: 32px;

}

.sunrise-sunset-spinner {

position: absolute;

margin-top: 58px;

transform: rotate(-90deg);

}

:host /deep/ .mat-progress-spinner circle, .mat-spinner circle {

stroke: #ffa828;

stroke-opacity: 0.6;

}

.weather-info-sunrise {

margin-left: 16px;

}

.weather-info-sunrise .wi {

margin-top: 50px;

font-size: 30px;

}

.weather-info-sunrise .hour {

position: absolute;

margin-top: -20px;

}

.weather-info-sunset {

margin-left: -16px;

}

.weather-info-sunset .wi {

margin-top: 50px;

font-size: 30px;

}

.weather-info-sunset .hour {

position: absolute;

margin-top: -20px;

}

Por último iremos al fichero app.component.html y quitamos todo el código html que hay, para renderizar nuestro componente, ¿Como hacemos esto? Como Angular CLI se encargo de añadirlo al módulo y Angular se encarga de leer los selectores sólo tenemos que escribir el tag de html necesario.

El fichero app.component.html debe quedar así:

<app-weather-card></app-weather-card>

Simulamos que hacemos una llamada al servidor para la obtención de datos y los mostramos por consola, para ver que se ejecuta antes y que despues.

El componente de ejemplo es el siguiente:

import { Component, OnInit } from '@angular/core';import { Weather } from '../weather.model';

@Component({

selector: 'app-weather-card',

templateUrl: './weather-card.component.html',

styleUrls: ['./weather-card.component.css']

})

export class WeatherCardComponent implements OnInit {

weather: Weather = new Weather();

constructor() { }

ngOnInit() {

console.log('ngOnInit');

setTimeout(() => {

this.weather.city = 'Madrid';

this.weather.date = 'Tue, 01 May 2018 06:00 PM CEST';

this.weather.humidity = 32;

this.weather.info = 'Nublado';

this.weather.maxtemperature = 24;

this.weather.mintemperature = 10;

this.weather.pressure = 1080;

this.weather.sunrise = '6:58 am';

this.weather.sunset = '11:40 pm';

this.weather.temperature = 18;

this.weather.winddirection = 270;

this.weather.windspeed = 26;

console.log('call to server finalizado');

console.log(this.weather);

}, 3000);

console.log(this.weather);

}

}

Si tenéis cualquier problema, la aplicación base esta en **[github](https://github.com/OpenWebinarsNet/Angular-desde-cero)** en la carpeta aweather-ciclo-de-vida-componentes

**Data binding, Directivas y pipes**

## **Introducción data binding**

Ya hemos empezado a trabajar con componentes, asignándoles un fichero template (html), una hoja de estilos (css) y hemos rellenado un modelo de datos.

Ahora vamos a aprender como dotar a las vistas de interacción y de lógica. Para ello vamos a usar las distintas formas de hacer ****data binding**** o enlazado de datos.

Existen cuatro tipo de enlazado de datos que aprenderemos durante las siguientes lecciones:

* ****Interpolation binding****
* ****Property binding****
* ****Event binding****
* ****Two way binding****

No te preocupes porque las vamos a ver una a una, empezamos con Interpolation binding.

## **Interpolation binding**

Siempre que necesites com****Event binding**** nos permite ****definir**** un ****evento que ocurre en el template y comunicarlo con la clase****.

unicar propiedades (variables, objetos, arrays, etc…) desde la clase del componente al template, podemos usar interpolation.

Por definición la forma de usar ****interpolación**** en un ****template**** es: ****{{ propertyName }}****

Modificaremos nuestro template para poner en práctica interpolation binding, el archivo weather-card.component.html queda así.

<mat-card class="weather-card">

<mat-card-header class="weather-header">

<div mat-card-avatar class="weather-header-icon">

<i class="wi wi-day-sunny"></i>

</div>

<mat-card-title>{{weather.city}}</mat-card-title>

<mat-card-subtitle>{{weather.hourMessure}}</mat-card-subtitle>

</mat-card-header>

<mat-card-content>

<div class="weather-card-content-info">

<mat-grid-list cols="4" rowHeight="30px">

<mat-grid-tile>

<i class="wi wi-strong-wind"></i> {{weather.windspeed}} km/h</mat-grid-tile>

<mat-grid-tile>

<i class="wi wi-wind from-{{weather.windspeed}}-deg"></i>

</mat-grid-tile>

<mat-grid-tile>

<i class="wi wi-humidity"></i> {{weather.humidity}} %</mat-grid-tile>

<mat-grid-tile>

<i class="wi wi-barometer"></i> {{weather.pressure}}

</mat-grid-tile>

</mat-grid-list>

</div>

<mat-grid-list cols="2" rowHeight="30px">

<mat-grid-tile colspan="2" class="weather-info-text">

{{weather.info}}

</mat-grid-tile>

<mat-grid-tile class="weather-info-temp-up">

<i class="wi wi-direction-up"></i> {{weather.maxtemperature}} ºC

</mat-grid-tile>

<mat-grid-tile class="weather-info-temp-down">

<i class="wi wi-direction-down"></i> {{weather.mintemperature}} ºC

</mat-grid-tile>

</mat-grid-list>

<mat-grid-list cols="1" rowHeight="80px">

<mat-grid-tile class="weather-info-temp">

<div class="number">{{weather.temperature}}</div>

<div class="measure">ºC</div>

</mat-grid-tile>

</mat-grid-list>

<mat-divider></mat-divider>

<mat-grid-list cols="5" rowHeight="120px">

<mat-grid-tile class="weather-info-sunrise">

<div class="hour">{{weather.sunrise}}</div>

<i class="wi wi-sunrise"></i>

</mat-grid-tile>

<mat-grid-tile colspan="3">

<mat-progress-spinner class="sunrise-sunset-spinner" diameter="120" strokeWidth="110" mode="determinate" value="30">

</mat-progress-spinner>

</mat-grid-tile>

<mat-grid-tile class="weather-info-sunset">

<div class="hour">{{weather.sunset}}</div>

<i class="wi wi-sunset"></i>

</mat-grid-tile>

</mat-grid-list>

<app-weather-forecast></app-weather-forecast>

</mat-card-content></mat-card>

**Data binding, directivas y pipes Paso de datos a un componente anidado**

Tal y como he comentado anteriormente Property Binding se utiliza para pasar un valor a un componente hijo o anidado.

Vamos a crear un nuevo archivo en la carpeta src/app llamado “forecast.model.ts”.

export class Forecast {

date: string;

maxTemperature: number;

minTemperature: number;

info: string;

constructor() { }

}

Y modificamos el modelo weather.model.ts para incluir una propiedad de este último modelo creado.

import { Forecast } from './forecast.model';

....

....

sunset: string;

forecasts: Forecast;

...

Ahora vamos a crear un nuevo componente para mostrar la previsión del tiempo, en la carpeta src/app escribimos el siguiente comando.

ng generate component weather-forecast --spec false

El archivo css weather-forecast.component.css es:

.forecast-title{

font-size:22px;

background-color: #e0e0e0 !important;

padding: 8px;

}

.forecast-list {

border-bottom: 1px solid #e0e0e0;

}.forecast-date{

font-size:14px;

}.forecast-icon {

font-size: 24px;

}.forecast-temp-up {

font-size:18px;

}

.forecast-temp-up .wi{

font-size: 24px;

margin-right: 4px;

color:#D32F2F;

}

.forecast-temp-down {

font-size:18px;

}

.forecast-temp-down .wi{

font-size: 24px;

margin-right: 4px;

color:#1E88E5;

}

El archivo weather-forecast.component.html es:

<mat-divider></mat-divider><div class="forecast-title">Previsión</div><mat-divider></mat-divider><mat-grid-list cols="5" rowHeight="40px" class="forecast-list">

<mat-grid-tile colspan="2" class="forecast-date">

{{forecast?.day}}, {{forecast?.date}}

</mat-grid-tile>

<mat-grid-tile class="forecast-icon">

<i class="wi wi-day-sunny"></i>

</mat-grid-tile>

<mat-grid-tile class="forecast-temp-up">

<i class="wi wi-direction-up"></i> {{forecast?.maxTemperature}} ºC

</mat-grid-tile>

<mat-grid-tile class="forecast-temp-down">

<i class="wi wi-direction-down"></i> {{forecast?.minTemperature}} ºC

</mat-grid-tile></mat-grid-list>

Ahora declararemos una ****propiedad**** con el decorador ****@Input**** el cual nos permite definir una propiedad de ****entrada al componente****, esta puede ser nullable o no. La declaramos de la siguiente manera:

@Input() forecast: Forecast;

El archivo weather-forecast.component.ts queda de la siguiente manera:

import { Component, Input } from '@angular/core';import { Forecast } from '../forecast.model';

@Component({

selector: 'app-weather-forecast',

templateUrl: './weather-forecast.component.html',

styleUrls: ['./weather-forecast.component.css']

})export class WeatherForecastComponent {

@Input() forecast: Forecast;

constructor() { }

}

Ahora vamos al template de nuestro componente weather-card.component.ts y en el ngOnInit inicializamos la nueva propiedad forecast de nuestro modelo weather.

....

....

this.weather.windspeed = 26;

this.weather.forecast = new Forecast();

this.weather.forecast.date = '04 May 2018';

this.weather.forecast.day = 'Fri';

this.weather.forecast.info = 'Mostly Cloudy';

this.weather.forecast.maxTemperature = 23;

this.weather.forecast.minTemperature = 8;

console.log('call to server finalizado');

....

....

Ahora nos queda escribir el tag del nuevo componente en el componente padre. En el archivo weather-card.component.html escribiremos el tag.

....

....

</mat-grid-list>

<app-weather-forecast [forecast]="weather.forecast"></app-weather-forecast>

</mat-card-content>

</mat-card>

........

**Event binding**

Ya sabemos como mostrar una propiedad en nuestro template, como bindear propiedades a nuestros elementos del template y como pasar un valor del componente padre al hijo.

Ahora lo que nos falta es lo contrario, ****conectar nuestro template con el componente****, es decir, cuando un usuario interactue con un componente genere un evento y nosotros podamos utilizarlo.

****Event binding**** nos permite ****definir**** un ****evento que ocurre en el template y comunicarlo con la clase****.

El evento binding se define con el ****nombre del evento entre paréntesis****, por ejemplo un botón con el evento click sería:

<button (click)="search()">Añadir elemento</button>

Estos eventos nos ****devuelven**** un ****objeto “$event”****, el cual es ****elemento del DOM****, con el que podemos obtener todas sus propiedades.

Los elementos a los Angular tiene acceso los podemos ver en este listado **[Listado eventos](https://developer.mozilla.org/en-US/docs/Web/Events" \t "/home/ezamora/Documentos\\x/_blank)**

<button (click)="search($event)">Añadir elemento</button>

Para probarlo seguiremos con nuestra aplicación y vamos a crear un nuevo componente, en la carpeta src/app, escribiremos en la línea de comandos:

ng generate component weather-search --spec false

Lo primero que haremos es incluir en el fichero app.component.html el selector de nuestro nuevo componente, el archivo quedará:

<app-weather-card></app-weather-card>

<app-weather-search></app-weather-search>

Ahora incluiremos algo de html en nuestro nuevo template, el archivo weather-search.component.html:

<mat-card>

<mat-card-content>

<button mat-raised-button color="primary" (click)="searchCity()">Buscar ciudad</button>

<mat-spinner [style.visibility]="loading ? 'visible' : 'hidden'"></mat-spinner>

</mat-card-content></mat-card>

Y para terminar el archivo weather-card.component.ts con un método que mostrará y ocultará el spinner simulando que estamos buscando.

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

@Component({

selector: 'app-weather-search',

templateUrl: './weather-search.component.html',

styleUrls: ['./weather-search.component.css']

})export class WeatherSearchComponent implements OnInit {

loading: boolean;

constructor() { }

ngOnInit() {

this.loading = false;

}

public searchCity() {

this.loading = !this.loading;

}

}

**Paso de datos desde un componente anidado a su contenedor**

Anteriormente vimos como pasar variables del componente padre al componente hijo. Ahora vamos a ver como realizarlo en la otra dirección, del ****hijo al padre****.

Para actualizar el padre con las acciones que realice el hijo, tenemos que hacerlo a través de eventos y para ello, nos tenemos que crear nuestros ****event binding personalizados****.

Vamos a necesitar algún módulo más para realizar esta práctica por lo que nos dirigimos al archivo app.module.ts para incluir las siguientes líneas:

....

....import { BrowserAnimationsModule } from '@angular/platform-browser/animations';import { FormsModule } from '@angular/forms';

....

MatFormFieldModule,

MatInputModule

} from '@angular/material';

....

....

// en los imports de nuestro decorador

MatFormFieldModule,

MatInputModule,

BrowserAnimationsModule,

FormsModule

Incluimos un input de texto en nuestro template del componente weather-search.component.html, quedará así:

<mat-card>

<mat-card-content>

<mat-form-field>

<input matInput placeholder="Ciudad" name="cityName" [value]="cityName" (input)="cityName=$event.target.value">

</mat-form-field>

<button mat-raised-button color="primary" (click)="searchCity()">Buscar ciudad</button>

<mat-spinner [style.visibility]="loading ? 'visible' : 'hidden'"></mat-spinner>

</mat-card-content>

</mat-card>

Y ahora vamos a crear en el component weather-search.component.ts el evento para buscar una nueva ciudad.

Para crear un evento tenemos que declarar en nuestro componente una ****propiedad**** con el decorador ****@Output****, con la que indicaremos que es un ****parámetro de salida****, de esta manera:

@Output() searchEvent: EventEmitter<string> = new EventEmitter<string>();

Y cuando queramos generar el evento, lo haremos así:

this.searchEvent.emit(this.cityName);

El componente weather-search.component.ts se queda de esta manera:

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

@Component({

selector: 'app-weather-search',

templateUrl: './weather-search.component.html',

styleUrls: ['./weather-search.component.css']

})export class WeatherSearchComponent implements OnInit {

@Output() searchEvent: EventEmitter<string> = new EventEmitter<string>();

loading: boolean;

cityName: string;

constructor() { }

ngOnInit() {

this.loading = false;

}

public searchCity() {

this.loading = !this.loading;

this.searchEvent.emit(this.cityName);

}

}

Ahora tenemos que preparar al componente padre para recibir ese evento, nos vamos al archivo app.component.ts y lo dejamos de la siguiente manera:

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

@Component({

selector: 'app-root',

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

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

})export class AppComponent {

cityName = 'app';

public search(cityName: string): void {

this.cityName = cityName;

}

}

Y ahora en el template tenemos que recoger nuestro event binding personalizado. En el archivo app.component.html escribimos:

<app-weather-card></app-weather-card>

Componente padre, nombre ciudad {{cityName}}

<app-weather-search (searchEvent)="search($event)"></app-weather-search>v

* **Two-way binding**

Ya hemos visto como podemos pasar valores desde nuestro componente al template, y viceversa, utilizando Property Binding y Event Binding.

En algunos casos nos puede ser útil hacer uso de ello en ambos sentidos al mismo tiempo.

En el anterior ejemplo incorporamos un input en el que le pasamos el valor con la propiedad “[value]” y lo recogíamos en el evento “(input)” cuando este se actualizaba.

Otra forma de hacerlo es con Two-way data binding.

La **sintaxis** utilizaba para esto es la ****propiedad entre corchetes y paréntesis a la vez****, de esta manera le indicamos a Angular que vamos a utilizar ****ambos eventos al mismo tiempo**** para actualizar y mostrar el valor.

Por ejemplo en un formulario, con Angular tenemos la propiedad ****ngModel**** con la que podemos manejar el ****valor del elemento****.

Este tipo de binding se utiliza casi en su totalidad para ****formularios****.

Vamos a cambiar el input que escribimos en la práctica anterior por el siguiente.

<input matInput placeholder="Ciudad" name="cityName" [(ngModel)]="cityName">

Todo lo practicado hasta ahora lo podéis encontrar en **[github](https://github.com/OpenWebinarsNet/Angular-desde-cero)** en la carpeta aweather-data-binding

**Preparación del proyecto para directivas**

Antes de empezar con las directivas vamos a organizar un poco el código e incluiremos algunas cosas.

Vamos a crear nuevos módulos y a dar una mejor organización a nuestro código.

Para empezar, nos vamos a la ****carpeta src/app**** de nuestro proyecto y desde la línea de comando:

ng generate module shared

cd shared

ng generate module material-forms

Vamos a ****agrupar**** todo lo referente a ****Material y a los formularios**** en ****un sólo módulo**** y marcarlo como ****exports****, de esta manera no tendremos que declararlo en cada módulo que vayamos a utilizarlo.

El nuevo archivo material-forms.module.ts será:

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

import {

MatCardModule,

MatButtonModule,

MatDividerModule,

MatGridListModule,

MatProgressSpinnerModule,

MatInputModule,

MatSidenavModule

} from '@angular/material';

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';import { FormsModule } from '@angular/forms';

@NgModule({

exports: [

CommonModule,

BrowserAnimationsModule,

// Material

MatCardModule,

MatButtonModule,

MatDividerModule,

MatGridListModule,

MatProgressSpinnerModule,

MatInputModule,

MatSidenavModule,

// Forms

FormsModule

],

declarations: []

})export class MaterialFormsModule { }

Este nuevo módulo lo importamos y exportamos en el archivo shared.module.ts

import { NgModule } from '@angular/core';import { CommonModule } from '@angular/common';import { MaterialFormsModule } from './material-forms/material-forms.module';

@NgModule({

imports: [

CommonModule,

MaterialFormsModule

],

declarations: [],

exports: [MaterialFormsModule]

})export class SharedModule { }

Volvemos a la ****carpeta src/app**** y ejecutamos los siguientes comandos:

ng generate module core

cd core

ng generate component left-menu --spec false

ng generate component top-menu --spec false

En el módulo importaremos algunas librerías que nos hacen falta de Material para el menú que vamos a crear y marcamos como exportable los componentes top y left para utilizarlos en otro módulo. Archivo core.module.ts

import { NgModule } from '@angular/core';import { CommonModule } from '@angular/common';import { TopMenuComponent } from './top-menu/top-menu.component';import { LeftMenuComponent } from './left-menu/left-menu.component';import { MatToolbarModule, MatListModule, MatDividerModule } from '@angular/material';

@NgModule({

imports: [

CommonModule,

MatToolbarModule,

MatListModule,

MatDividerModule

],

declarations: [TopMenuComponent, LeftMenuComponent],

exports: [TopMenuComponent, LeftMenuComponent]

})export class CoreModule { }

Dentro de la carpeta top-menu vamos a rellenar los archivos:

top-menu.component.ts

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

@Component({

selector: 'app-top-menu',

templateUrl: './top-menu.component.html',

styleUrls: ['./top-menu.component.css']

})export class TopMenuComponent implements OnInit {

@Input() sidenav: any;

constructor() { }

ngOnInit() {

}

}

top-menu.component.css

.header\_logo {

padding: 0px;

font-size: 18px !important;

margin-left: 0px;

margin-top: 10px;

}

.fa {

cursor: pointer;

font-size: 1em;

}

.mat-down {

margin-left: 10px;

}

.mat-toolbar .toolbar-right {

-webkit-box-flex: 1;

-ms-flex: 1 1 auto;

flex: 1 1 auto;

}

.toolbar-icons-menu {

margin-top: -8px;

}

.mat-toolbar {

min-height: initial !important;

height: 50px !important;

padding: 0px 5px 0 0px !important;

}

.mat-toolbar .mat-toolbar-row {

align-items: initial !important;

height: 50px;

}

.mat-toolbar .mat-icon-button {

height: 40px;

line-height: 41px;

}

top-menu.component.html

<mat-toolbar color="primary">

<div class="header\_logo">

<img src="https://drive.google.com/uc?id=1UCuwEFL2\_wANdGi7\_6N7\_xs5YDoBSzuq"> </div>

<div class="mat-down">

<i class="fa fa-bars fa-lg" (click)="sidenav.toggle()"></i>

</div>

<span class="toolbar-right"></span>

<div class="toolbar-icons-menu">

<i class="fa fa-sign-out-alt fa-lg"></i>

</div></mat-toolbar>

Hacemos lo mismo el la carpeta left-menu:

left-menu.component.ts

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

@Component({

selector: 'app-left-menu',

templateUrl: './left-menu.component.html',

styleUrls: ['./left-menu.component.css']

})export class LeftMenuComponent implements OnInit {

constructor() { }

ngOnInit() {

}

}

left-menu.component.css

.menu-list-divider {

width: 100%;

text-align: right;

padding-right: 5px;

color: #202020;

font-size: 10px;

background-color: #d3d3d3;

cursor: pointer;

font-weight: bold;

}

:host /deep/ .mat-list-text {

margin-left: 36px;

}

.font-size-20 {

font-size: 20px;

}

:host /deep/ .mat-list-item .mat-list-item-content:hover {

color: #34579c;

background-color: #34589c14;

cursor: pointer;

}

left-menu.component.html

<div class="menu-list-divider"> v.1.0 </div>

<mat-list>

<mat-list-item>

<a mat-button mat-line>Añadir ciudades</a>

<i class="fab fa-cloudversify font-size-20"></i>

</mat-list-item>

<div class="menu-list-divider"> Ciudades </div>

<mat-list-item>

<a mat-button mat-line>Madrid</a>

<i class="fas fa-map-marker-alt font-size-20"></i>

</mat-list-item>

</mat-list>

Ahora nos vamos a crear una nueva carpeta para agrupar la funcionalidad del tiempo, la llamaremos weather.

En ella cortamos y pegamos las carpetas weather-card, weather-forecast, weather-search, el archivo weather.model.ts y forecast.model.ts

Volvemos a la línea de comandos y dentro de la carpeta src/app/weather, escribimos:

ng generate module weather --flat

El archivo weather.module.ts deberá quedar:

import { NgModule } from '@angular/core';import { CommonModule } from '@angular/common';import { SharedModule } from '../shared/shared.module';import { WeatherCardComponent } from './weather-card/weather-card.component';import { WeatherForecastComponent } from './weather-forecast/weather-forecast.component';import { WeatherSearchComponent } from './weather-search/weather-search.component';

@NgModule({

imports: [

CommonModule,

SharedModule

],

declarations: [

WeatherCardComponent,

WeatherForecastComponent,

WeatherSearchComponent

],

exports: [

WeatherCardComponent,

WeatherSearchComponent

]

})export class WeatherModule { }

Una vez configurados todos estos archivos nos toca renderizarlos y terminar de configurar el nuevo menú.

En el archivo app.component.ts añadimos una nueva propiedad, justo debajo de cityName

sidebarMode = 'side';

En el app.module.ts importaremos los nuevo módulos creados y quitaremos los componentes de la carpeta weather porque ya están incluidos en el otro módulo.

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

import { AppComponent } from './app.component';import { CoreModule } from './core/core.module';import { SharedModule } from './shared/shared.module';import { WeatherModule } from './weather/weather.module';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

CoreModule,

SharedModule,

WeatherModule

],

providers: [],

bootstrap: [AppComponent]

})export class AppModule { }

Modificamos el app.component.html para renderizar los componentes del menú.

<app-top-menu [sidenav]="sidenav"></app-top-menu>

<mat-sidenav-container class="app-body">

<mat-sidenav #sidenav [mode]="sidebarMode" opened="true">

<app-left-menu></app-left-menu>

</mat-sidenav>

<div class="main">

<app-weather-card></app-weather-card>

Componente padre, nombre ciudad {{cityName}}

<app-weather-search (searchEvent)="search($event)"></app-weather-search>

</div>

</mat-sidenav-container>

Y por último modificamos el archivo app.component.css

.main {

position: relative;

overflow-x: hidden;

overflow-y: scroll;

-webkit-overflow-scrolling: touch;

-ms-overflow-style: -ms-autohiding-scrollbar;

height: calc(100vh - 40px);

background-color: white;

}

.mat-sidenav {

width: 250px;

box-shadow: 0 3px 5px 3px rgba(0, 0, 0, .2), 0 2px 2px 0 rgba(0, 0, 0, .14), 0 1px 5px 0 rgba(0, 0, 0, .12);

}

Compilamos y veremos el nuevo menú de aplicación!

Si tenéis cualquier problema durante estos pasos, la versión de este código esta en **[github](https://github.com/OpenWebinarsNet/Angular-desde-cero)** con el nombre de aweather-core-shared-modules.

Directivas estructurales

## **Directivas estructurales**

Las ****directivas estructurales**** son aquellas que ****modifican la estructura del html de un componente.****

Angular tiene tres directivas estructurales principalmente:

* ****\*ngIf****
* ****\*ngFor****
* ****\*ngSwitch****

Vamos a explicar una a una para saber como funcionan:

## **ngIf**

La directa ngIf la usamos para añadir o quitar elementos del DOM en función a una expresión. Funciona como un if en cualquier lenguaje de programación.

Por ejemplo, nos vamos al archivo weather-search.component.html y cambiamos la línea.

<mat-spinner [style.visibility]="loading ? 'visible' : 'hidden'"></mat-spinner>

Por una directiva estructural del tipo if.

<mat-spinner \*ngIf="loading"></mat-spinner>

El ****asterisco es obligatorio delante**** de una directiva estructural

## **ngFor**

La directiva ngFor la usaremos para trabajar con Arrays. Funciona como un for en cualquier lenguaje de programación y la podemos definir de la siguiente forma.

Por ejemplo, nos vamos al archivo left-menu.component.ts y nos creamos un array de string, en el que pondremos las ciudades que queremos visualizar.

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

@Component({

selector: 'app-left-menu',

templateUrl: './left-menu.component.html',

styleUrls: ['./left-menu.component.css']

})export class LeftMenuComponent implements OnInit {

cities: Array<string> = new Array<string>();

constructor() { }

ngOnInit() {

this.cities.push('Madrid, ES');

this.cities.push('Barcelona, ES');

this.cities.push('Sevilla, ES');

this.cities.push('Malaga, ES');

this.cities.push('Santander, ES');

}

}

Y ahora en el html, archivo left-menu.component.html realizamos una directiva estructural del tipo \*ngFor.

<mat-list-item \*ngFor="let city of cities">

<a mat-button mat-line>{{city}}</a>

<i class="fas fa-map-marker-alt font-size-20"></i>

</mat-list-item>

## **ngSwitch**

La directiva ngSwitch funciona igual que un switch en cualquier lenguaje de programación, vamos a ver un ejemplo.

Nos vamos a la carpeta src/app/shared y en línea de comandos nos generamos un nuevo componente.

ng generate component weather-icon --spec false

En el componente nuevo weather-icon.component.ts crearemos una propiedad de entrada con @Input(). El fichero quedará de la siguiente manera:

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

@Component({

selector: 'app-weather-icon',

templateUrl: './weather-icon.component.html',

styleUrls: ['./weather-icon.component.css']

})export class WeatherIconComponent implements OnInit {

@Input() weatherInfo: string;

constructor() { }

ngOnInit() {

}

}

Y en el html escribimos una directiva estructural del tipo ngSwitch.

<div [ngSwitch]="weatherInfo">

<i \*ngSwitchCase="'Sunny'" class="wi wi-day-sunny"></i>

<i \*ngSwitchCase="'Cloudy'" class="wi wi-cloudy"></i>

<i \*ngSwitchCase="'Mostly Cloudy'" class="wi wi-day-cloudy-high"></i>

<i \*ngSwitchCase="'Partly Cloudy'" class="wi wi-day-cloudy"></i>

<i \*ngSwitchCase="'Rain'" class="wi wi-day-showers"></i>

<i \*ngSwitchCase="'Showers'" class="wi wi-day-showers"></i>

<i \*ngSwitchCase="'Scattered Thunderstorms'" class="wi wi-day-lightning"></i>

<i \*ngSwitchCase="'Scattered Showers'" class="wi wi-day-rain-mix"></i>

<i \*ngSwitchCase="'Mostly Sunny'" class="wi wi-day-sunny-overcast"></i>

<i \*ngSwitchDefault class="wi wi-day-sunny"></i></div>

Para poder utilizar este componente en otro módulo tenemos que marcarlo como exportable, en el archivo shared.module.ts

exports: [MaterialFormsModule, WeatherIconComponent]

Para hacer uso de ella nos vamos a ir al archivo weather-card.component.html y hacemos uso del nuevo componente con

<div mat-card-avatar class="weather-header-icon">

<app-weather-icon [weatherInfo]="weather.info"></app-weather-icon>

</div>

Y también hacemos uso de ella en el forecast, archivo weather-forecast.component.html

<mat-grid-tile class="forecast-icon">

<app-weather-icon [weatherInfo]="forecast?.info"></app-weather-icon>

</mat-grid-tile>

**Directivas de Atributos**

Las ****directivas de atributo**** son aquellas que ****modifican la apariencia o el comportamiento de un elemento****.

Angular tiene dos directivas de atributo por defecto:

* ****ngClass****
* ****ngStyle****

## **ngStyle**

La directiva ngStyle nos permite asignar valores de estilos a un objeto del DOM.

Por ejemplo vamos a modificar el color de la temperatura de nuestro weather-card.

Agregamos un nuevo método en el componente weather-card.component.ts

public getColorTemperature(): string {

if (this.weather !== undefined && this.weather.temperature >= 24) {

return '#EF6C00';

}

if (this.weather !== undefined && this.weather.temperature <= 10) {

return '#0277BD';

}

return '#212121';

}

Y ahora incoporporamos la directiva de atributo en el archivo weather-card.component.html

<div class="number" [ngStyle]="{color: getColorTemperature()}">{{weather.temperature}}</div>

## **ngClass**

La directiva ngClass nos permite asignar una clase al elemento del DOM de forma dinámica.

Por ejemplo, agregamos el siguiente css al archivo weather-search.component.css

.active {

visibility: visible;

}

.no-active {

visibility: hidden;

}

Y cambiamos el html del botón de búsqueda para usar estas clases según el usuario escriba en el input.

<button mat-raised-button color="primary" (click)="searchCity()" [ngClass]="cityName !== '' ? 'active' : 'no-active'">Buscar ciudad</button>

Directivas personalizadas

Hemos visto las directivas que nos trae Angular de base, pero sabiendo que vamos a necesitar un mayor control sobre nuestros elementos, vamos a ver como crear una directiva personalizada.

Vamos a la carpeta src/app/shared y escribimos los siguientes comandos:

mkdir dropdown

cd dropdown

ng generate directive dropdown --spec false --export true

ng generate directive dropdowntoggle --spec false --export true

El archivo dropdrown.directive.ts quedará de la siguiente manera:

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

@Directive({

selector: '[appDropdown]'

})

export class DropdownDirective {

@HostBinding('class.open') \_open = false;

constructor() { }

public isOpen() {

return this.\_open;

}

public open() {

this.\_open = true;

}

public close() {

this.\_open = false;

}

public toggle() {

if (this.isOpen()) {

this.close();

} else {

this.open();

}

}

}

Y el archivo dropdowntoggle.directive.ts :

import { Directive, HostListener } from '@angular/core';import { DropdownDirective } from './dropdown.directive';

@Directive({

selector: '[appDropdowntoggle]'

})export class DropdowntoggleDirective {

@HostListener('click', ['$event'])

toggleOpen($event: any) {

$event.preventDefault();

this.dropdown.toggle();

}

constructor(private dropdown: DropdownDirective) { }

}

Fijaros en que una directiva tiene la misma estructura que un componente o un módulo.

* imports
* decorator
* class

El selector también es diferente, lo que hemos creado es un atributo para un elemento, pero también podemos hacerlo mediante una clase.

@Directive({

selector: '.appDropdown'

})

Para implementar el drop down creado nos vamos al left-menu.component de nuestro módulo core y modificamos el archivo left-menu.component.html.

<div class="menu-list-divider"> v.1.0 </div>

<mat-list>

<mat-list-item>

<a mat-button mat-line>Añadir ciudades</a>

<i class="fab fa-cloudversify font-size-20"></i>

</mat-list-item>

<div class="menu-list-divider"> Ciudades </div>

<div appDropdown class="drop-down">

<mat-list-item routerLinkActive="active">

<a mat-button mat-line appDropdowntoggle class="drop-down-toggle">Ciudades</a>

</mat-list-item>

<div class="drop-down-items">

<mat-list-item \*ngFor="let city of cities">

<a mat-button mat-line>{{city}}</a>

<i class="fas fa-map-marker-alt font-size-20"></i>

</mat-list-item>

</div>

</div>

</mat-list>

Y también añadimos al css al archivo left-menu.component.css

....

..../\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* DROPDOWN \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.drop-down .drop-down-items {

display: none;

transition: height 3s ease-out;

}

.drop-down .drop-down-items .mat-list-text {

font-size: 12px;

}

.drop-down .drop-down-items .mat-button {

line-height: 40px !important;

}

.drop-down .drop-down-toggle::after {

position: absolute;

right: 12px;

top: 14px;

width: 0;

height: 0;

content: "";

border-bottom: .3em solid;

border-right: .3em solid transparent;

border-left: .3em solid transparent;

transition: .3s;

font-size: 20px;

-webkit-transform: rotate(180deg);

transform: rotate(180deg);

color: #455a64 !important;

}

.drop-down.open .drop-down-items {

display: block;

}

.drop-down.open .drop-down-toggle::after {

-webkit-transform: rotate(0deg);

transform: rotate(0deg);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* FIN DROPDOWN \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Host listener

Ya hemos visto en la lección anterior, un ejemplo de Host listener pero vamos a explicar como funciona.

El ****decorador @HostListener**** asocia una ****función a un evento**** de la siguiente forma.

En el caso del drop down estamos bindeando el evento click con la función justo abajo.

@HostListener('click', ['$event'])

toggleOpen($event: any) {

$event.preventDefault();

this.dropdown.toggle();

}

En el momento que realicemos un click en el elemento que tenga este atributo, se ejecutará la función.

****Host listener y event binding son iguales****, la única diferencia es que ****Host listener se usa en directivas****, mientras que ****event binding sólo se usa en componentes****.

Host binding

Ya hemos visto en la lección de directivas personalizadas, un ejemplo de Host binding pero vamos a explicar como funciona.

El ****decorador @HostBinding detecta los cambios**** sufridos sobre ****una propiedad del elemento**** al que afecta la directiva.

Para configurarlo es necesario ****especificar el atributo**** para detectar un cambio.

En el caso del drop down estamos diciendo, que en el momento que el elemento tenga la clase open, el valor de la propiedad \_open cambiará automáticamente a true.

@HostBinding('class.open') \_open = false;

Pipes

Las pipes o tuberías de Angular nos permiten modificar la forma en la que se van a visualizar los datos en nuestro template y así hacerlos más legibles para el usuario.

Un ejemplo que ocurre en todas las aplicaciones es a la hora de mostrar una fecha, no es lo mismo poner 01/01/2018 que 01-Ene-18.

Angular tiene tuberías ya creadas de base, disponibles para que podamos utilizarlas en cualquier template:

* ****DatePipe****
* ****UpperCasePipe****
* ****LowerCasePipe****
* ****CurrencyPipe****
* ****PercentPipe****
* ****DecimalPipe****
* ****SlicePipe****

Podemos encontrar el listado completo y configuración en **[Angular pipes](https://angular.io/api?type=pipe)**

Todas funcionan igual, detrás del valor escribimos el carácter “|”, seguido del nombre de pipe y su configuración.

## **DecimalPipe, PercentPipe**

Las tuberías decimalPipe y percentPipe sirve para formatear números y en caso de percentPipe, angular nos añade el símbolo de porcentaje.

Por ejemplo, vamos utilizar en nuestro componente weather-card.component las pipes percentPipe y decimalPipe.

<mat-grid-tile>

<i class="wi wi-humidity"></i> {{weather.humidity / 100 | percent:'2.0-2'}}</mat-grid-tile>

<mat-grid-tile>

<i class="wi wi-barometer"></i> {{weather.pressure | number: '4.1-2'}}

</mat-grid-tile>

Estos dos pipes funcionan igual detrás de ellas podemos definir el formato del número a mostrar X.Y-Z donde X es el número de enteros máximo, Y es el número mínimo de decimales y la Z es el número máximo de decimales.

## **UpperCasePipe, LowerCasePipe**

La tubería uppercasePipe y lowercasePipe sirve para poner en mayúsculas o minúsculas el valor.

Nos vamos al mismo archivo que antes, weather-card.component.ts y cambiamos el info a lowercase de la siguiente manera.

<mat-grid-tile colspan="2" class="weather-info-text">

{{weather.info | lowercase}}

</mat-grid-tile>

## **DatePipe**

La tubería datePipe sirve para formatear fechas.

Nos vamos al archivo weather-forecast.component.html y aplicamos a la fecha la pipe.

<mat-grid-tile colspan="2" class="forecast-date">

{{forecast?.date | date: 'fullDate'}}

</mat-grid-tile>

En el caso de esta tubería podemos darle los siguientes formatos.

* ‘short’ (e.g. 6/15/15, 9:03 AM)
* ‘medium’ (e.g. Jun 15, 2015, 9:03:01 AM)
* ‘long’ (e.g. June 15, 2015 at 9:03:01 AM GMT+1)
* ‘full’ (e.g. Monday, June 15, 2015 at 9:03:01 AM GMT+01:00)
* ‘shortDate’ (e.g. 6/15/15)
* ‘mediumDate’ (e.g. Jun 15, 2015)
* ‘longDate’ (e.g. June 15, 2015)
* ‘fullDate’ (e.g. Monday, June 15, 2015)
* ‘shortTime’ (e.g. 9:03 AM)
* ‘mediumTime’ (e.g. 9:03:01 AM)
* ‘longTime’ (e.g. 9:03:01 AM GMT+1)
* ‘fullTime’ (e.g. 9:03:01 AM GMT+01:00)

## **SlicePipe**

La tubería slicePipe nos sirve para hacer el conocido substring de un texto mediante un pipe o para cortar un array.

Vamos al template app.component.html una pipe de ejemplo.

Componente padre, nombre ciudad {{cityName | slice:0:3}}

Pipes personalizadas

Hemos visto como utilizar las pipes que Angular trae por defecto, ahora vamos a ver como crear nuestras propias pipes.

El api que vamos a utilizar para traer la información del tiempo, nos da la temperatura el farenheit, vamos a convertirla a celsius y añadir el signo correspondiente como una pipe.

Vamos a la carpeta src/app/shares y en la línea de comando escribimos lo siguiente:

mkdir pipes

cd pipes

ng generate pipe celsius --spec false --export true

El archivo celsius.pipe.ts queda de la siguiente manera:

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

@Pipe({

name: 'celsius'

})export class CelsiusPipe implements PipeTransform {

transform(value: any, symbol: boolean = true): any {

if (value === undefined || value === null || value === '') {

value = 32;

}

const celsiusValue = (value - 32) \* 5.0 / 9.0;

return `${Math.round(celsiusValue).toString()} ${symbol ? 'ºC' : ''}`;

}

}

Ahora vamos a utilizarlo en en el template weather-card.component.html

<mat-grid-tile class="weather-info-temp-up">

<i class="wi wi-direction-up"></i> {{weather.maxtemperature | celsius}}

</mat-grid-tile>

<mat-grid-tile class="weather-info-temp-down">

<i class="wi wi-direction-down"></i> {{weather.mintemperature | celsius}}

</mat-grid-tile>

....

....

<div class="number" [ngStyle]="{color: getColorTemperature()}">{{weather.temperature | celsius: false}}</div>

Tenemos que cambiar el model a farenheit, en el fichero weather-card.component.ts

....

....

this.weather.maxtemperature = 90;

this.weather.mintemperature = 58;

........

this.weather.temperature = 80;

........

this.weather.forecast.maxTemperature = 88;

this.weather.forecast.minTemperature = 62;

También utilizaremos nuestra nueva pipe en el archivo weather-forecast.component.ts

<mat-grid-tile class="forecast-temp-up">

<i class="wi wi-direction-up"></i> {{forecast?.maxTemperature | celsius}}

</mat-grid-tile>

<mat-grid-tile class="forecast-temp-down">

<i class="wi wi-direction-down"></i> {{forecast?.minTemperature | celsius}}

</mat-grid-tile>

El código de la aplicación hasta este punto lo puedes encontrar en **[github](https://github.com/OpenWebinarsNet/Angular-desde-cero)** en la carpeta aweather-data-binding-directives-pipes

Creación de un servicio

En las lecciones anteriores aprendimos a construir componentes y módulos. Ahora vamos aprender como manejar los datos o la lógica que no esta asociada con un componente o queremos compartirla entre varios componentes, para esto construimos servicios.

Un ****servicio**** es un ****conjunto de métodos**** para gestionar diferentes tipos de datos relacionados con la aplicación.

Creando ****servicios**** estamos ****quitando al componente responsabilidad**** que no le corresponde como puede ser logs, sesiones o servicios externos.

Estructura de un servicio:

* imports
* decorator
* class

## **imports**

Un servicio tiene la sección de la declaración de los imports que ya conocemos.

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

## **decorator**

El decorador de clase del servicio es @Injectable()

## **class**

En la clase pondremos los métodos necesarios del servicio.

Vamos a crear un servicio que nos devuelva el objeto weather, simulando que estamos consumiendo un servicio real.

en la carpeta src/app/weather ejecutamos los siguientes comandos

mkdir services

cd services

ng generate service weather --spec false

El archivo weather.service.ts queda de la siguiente manera:

import { Injectable } from '@angular/core';import { Weather } from '../weather.model';import { Forescast } from '../forescast.model';

@Injectable()export class WeatherService {

constructor() { }

public getWeatherInfo(cityName: string): Weather {

const weather = new Weather();

weather.city = 'Madrid';

weather.date = 'Tue, 01 May 2018 06:00 PM CEST';

weather.humidity = 32.1;

weather.info = 'Cloudy';

weather.maxtemperature = 90;

weather.mintemperature = 58;

weather.pressure = 1080;

weather.sunrise = '6:58 am';

weather.sunset = '23:40 pm';

weather.temperature = 80;

weather.winddirection = 270;

weather.windspeed = 26;

weather.forecast = new Forescast();

weather.forecast.date = '04 May 2018';

weather.forecast.day = 'Fri';

weather.forecast.info = 'Mostly Cloudy';

weather.forecast.maxTemperature = 88;

weather.forecast.minTemperature = 62;

return weather;

}

}

Registro de un servicio

Recuerdo cuando estudiamos los componentes y los módulos sus decoradores tenían dos propiedades para registrar servicios.

En el ****decorador**** de los ****componentes**** la propiedad se llama ****viewProviders****.

Y en el ****decorador**** de los ****módulos**** la propiedad se llama ****providers****.

Hay que diferenciar si registramos un servicio en un componente o en un módulo. Si registramos el servicio en el componente, lo podemos inyectar en el componente y en sus hijos. En cambio, si registramos el servicio en un módulo, lo podemo inyectar en cualquier parte de la aplicación.

Vamos a registrar el servicio que hemos creado, en el módulo weather.

Añadimos la siguiente línea en el decorator de weather.module.ts

import { WeatherService } from './services/weather.service';

....

....

providers: [WeatherService],

Inyección del servicio

Para poder ****utilizar**** un ****servicio**** tenemos que ****inyectarlo en el constructor de un componente****.

Cuando ese componente se instancie, el inyector de dependecias de Angular nos dará una nueva instancia de ese servicio para que podamos utilizarlo.

Con el ejemplo que hemos visto, nos vamos al weather-card.component.ts y modificamos el constructor y el ngOnInit de esta manera:

import { WeatherService } from '../services/weather.service';

....

....

constructor(

private \_weatherService: WeatherService

) { }

ngOnInit() {

this.weather = this.\_weatherService.getWeatherInfo('Madrid');

}

....

....

Consumo de datos con Http Observables y su configuración

Los observables nos ayudan a gestionar datos asíncronos, como los datos que vienen del servidor.

Los observables son una funcionalidad que tiene Angular a través de la librería de RxJS.

Tenemos que entender los observables como un array de elementos que van llegando de forma asíncrona a lo largo del tiempo.

Os recomiendo que miréis esta web para entender el funcionamiento de los observables **[Observables](http://rxmarbles.com/)**

Para poder empezar hacer llamadas HTTP y con ello el uso de observables, tenemos que importar un módulo de angular que se llama HttpClientModule.

Como queremos tenerlo disponible en toda la aplicación lo vamos a exportar en el shared.module.ts

import { NgModule } from '@angular/core';import { CommonModule } from '@angular/common';import { MaterialFormsModule } from './material-forms/material-forms.module';import { WeatherIconComponent } from './weather-icon/weather-icon.component';import { DropdownDirective } from './dropdown/dropdown.directive';import { DropdowntoggleDirective } from './dropdown/dropdowntoggle.directive';import { CelsiusPipe } from './pipes/celsius.pipe';import { HttpClientModule } from '@angular/common/http';

@NgModule({

imports: [

CommonModule,

MaterialFormsModule

],

declarations: [WeatherIconComponent, DropdownDirective, DropdowntoggleDirective, CelsiusPipe],

exports: [

MaterialFormsModule,

WeatherIconComponent,

DropdownDirective,

DropdowntoggleDirective,

CelsiusPipe,

HttpClientModule

]

})export class SharedModule { }

Peticiones y envío de datos en Angular

Cambiamos el weather.model.ts para que ahora el forecast sea un array, ya que el API nos devuelve más de un día.

forecasts: Array<Forescast>;

En el componente weather-forecast.component.ts cambiamos el input para recibir un array.

@Input() forecasts: Array<Forescast>;

Y modificamos el html de weather-forecast.component.html para que ahora renderice varios forescat.

<mat-grid-list cols="5" rowHeight="40px" class="forecast-list" \*ngFor="let forecast of forecasts">

Vamos a terminar de configurar el servicio para consumir el api pública de Yahoo! weather.  
Para ello importamos el HttpClient que no es más que un servicio del módulo HttpClientModule de angular.

import { Injectable } from '@angular/core';import { Weather } from '../weather.model';import { Forescast } from '../forescast.model';import { HttpClient } from '@angular/common/http';import { Observable } from 'rxjs/Observable';

@Injectable()export class WeatherService {

private urlApi = 'https://query.yahooapis.com/v1/public/yql?q=';

private selectApi\_part1 = 'select \* from weather.forecast where woeid in (select woeid from geo.places(1) where text="';

private selectApi\_part2 = '")';

private jsonParam = '&format=json';

constructor(

private \_http: HttpClient

) { }

public getWeatherInfo(cityName: string): Observable<any> {

const url = `${this.urlApi}${this.selectApi\_part1}${cityName}${this.selectApi\_part2}${this.jsonParam}`;

return this.\_http.get(url);

}

public mapResult(result: any): Weather {

const weatherInfo = new Weather();

weatherInfo.city = `${result.location.city}, ${result.location.country}`;

weatherInfo.date = result.item.condition.date;

weatherInfo.humidity = result.atmosphere.humidity;

weatherInfo.info = result.item.condition.text;

weatherInfo.maxtemperature = result.item.forecast[0].high;

weatherInfo.mintemperature = result.item.forecast[0].low;

weatherInfo.pressure = result.atmosphere.pressure;

weatherInfo.sunrise = result.astronomy.sunrise;

weatherInfo.sunset = result.astronomy.sunset;

weatherInfo.temperature = result.item.condition.temp;

weatherInfo.winddirection = result.wind.direction;

weatherInfo.windspeed = result.wind.speed;

weatherInfo.forecasts = new Array<Forecast>();

for (const forecast of result.item.forecast) {

const newForecast = new Forecast();

newForecast.date = forecast.date;

newForecast.day = forecast.day;

newForecast.maxTemperature = forecast.high;

newForecast.minTemperature = forecast.low;

newForecast.info = forecast.text;

weatherInfo.forecasts.push(newForecast);

}

return weatherInfo;

}

}

Y cambiamos el ngOnInit del archivo weather-card.component.ts, para consumir el servicio y obtener el tiempo

ngOnInit() {

this.\_weatherService.getWeatherInfo('Madrid, ES').subscribe(

data => {

if (data['query'].results === undefined) {

alert('La ciudad buscada no existe');

} else {

this.weather = this.\_weatherService.mapResult(data['query'].results.channel);

}

}

);

}

Recogida de errores de la petición

En el caso de que el servicio dé algún error, por ejemplo porque la ciudad que estamos buscando no exista, debemos recoger el error y mostrarlo al usuario.

Esto lo hacemos de manera muy sencilla gracias a los observables de Angular.

Lo hacemos incluyendo error, en el método subscribe. El ngOnInit del archivo weather-card.component.ts se quedaría así.

ngOnInit() {

this.\_weatherService.getWeatherInfo('Madrid, ES').subscribe(

data => {

if (data['query'].results === undefined) {

alert('La ciudad buscada no existe');

} else {

this.weather = this.\_weatherService.mapResult(data['query'].results.channel);

}

},

error => {

alert(error.message);

}

);

}

Todo lo prácticado hasta ahora lo podéis encontrar en **[github](https://github.com/OpenWebinarsNet/Angular-desde-cero)** en la carpeta aweather-servicios

Routing Funcionamiento del routing

Es el momento de empezar añadir rutas con funcionalidad a nuestra aplicación.

Para ello vamos a utilizar el módulo Router de Angular, con la que podemos implementar la navegación a través de URL.

En esencia vamos asociar una url a un componente a renderizar, aunque vamos a ir viendo que hay varios parámetros a tener en cuenta.

Antes de empezar a utilizar las rutas de Angular es muy importante asegurarnos que en nuestro archivo index.html tenemos en el tag head la etiqueta:

<base href="/">

Esto lo que hará será definir una ruta a partir la cual se construirán el resto de rutas, como hemos usado angular cli para crear el proyecto, ya lo tenemos añadido por defecto.

Configuración de rutas

Cada módulo tendrá su configuración de rutas, de esta manera si en un futuro queremos utilizar lazy loading para cargar los módulos nos será mucho más fácil.

Para ello vamos añadir un nuevo archivo al proyecto en la ruta src/app llamado app-routing.module.ts.

Lo que hacemos es crearnos un módulo para definir cada ruta, el cual importaremos en su correspondiente módulo.

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

const appRoutes: Routes = [

{

path: '',

pathMatch: 'full',

redirectTo: 'home/login'

},

{

path: '\*\*',

redirectTo: 'home/login'

}

];

@NgModule({

imports: [

RouterModule.forRoot(appRoutes)

],

exports: [

RouterModule

]

})export class AppRoutingModule { }

Fijaros en como importamos las rutas en Angular, y en el método que utilizamos forRoot. Este método normalmente sólo estará en el módulo que sea nuestro punto de inicio de la aplicación, en el resto de módulos importaremos las rutas con el método forChild. Esto es para que Angular forme una estructura jerárquica de todas nuestras rutas.

Con path identificamos el patrón de la URL al que hace referencia esta ruta.

Path \*\* es cualquier url que no esté contenida en el árbol de rutas de Angular.

La propiedad componente identifica el componente que Angular tiene que renderizar en ese path.

La propiedad redirect to redirección el path a la ruta indicada.

Para probarlo podemos añadir a las rutas un path con el componente AppComponent

{

path: 'home/login', component: AppComponent

}

Router outlet

Ya hemos realizado una configuración de una ruta pero tenemos que informar a Angular donde queremos mostrar el componente asociado con ese path, esto se hace con Router Outlet.

Router outlet no es más que un tag, el cual incorporaremos en un componente para que Angular renderice el componente.

Vamos al archivo app.component.html y añadimos el tag

<div class="main">

<router-outlet></router-outlet>

<!-- <app-weather-card></app-weather-card>

Componente padre, nombre ciudad {{cityName | slice:0-3}}

<app-weather-search (searchEvent)="search($event)"></app-weather-search> -->

</div>

Y ahora para probarlo vamos al archivo app-routing.module.ts y añadimos una ruta para que en home/login el componente sea WeatherCardComponent.

{ path: 'home/login', component: WeatherCardComponent }

Antes de seguir vamos añadir algún módulo más para tener una mejor aplicación y poder ver las siguientes lecciones con claridad.

En la carpeta src/app escribimos en línea de comandos.

ng generate module login

cd login

ng generate component login-container

ng generate component login-card

cd..

ng generate module layout

cd layout

ng generate component main-layout

Lo primero que haremos será copiar el html del archivo app.component.html al archivo main-layout.component.html. Y con el css igual de app.component.css al archivo main-layout.component.css.

El archivo app.component.html lo dejaremos con el siguiente html.

<router-outlet></router-outlet>

Esto lo hacemos con fin de poder manejar más de un layout en nuestra aplicación, cosa muy común en cualquier aplicación web.

Pero fijaros si ahora renderizamos el componente main-layout.component, vamos a tener dos router outlet en la aplicación. Es lo que llamamos herencia de rutas que ahora veremos como configurarlas, pero podemos tener tantos niveles como nuestra aplicación necesite.

En el archivo main-layout.componente.ts ahora necesitamos incorporar la propiedad

sidebarMode = 'side';

Y el archivo main-layout.component importamos los módulos que nos hacen falta y creamos un forChild para poder utilizar el router outlet del template

import { NgModule } from '@angular/core';import { CommonModule } from '@angular/common';import { CoreModule } from '../core/core.module';import { MainLayoutComponent } from './main-layout/main-layout.component';import { SharedModule } from '../shared/shared.module';import { RouterModule, Routes } from '@angular/router';

@NgModule({

imports: [

CommonModule,

CoreModule,

SharedModule,

RouterModule.forChild([])

],

declarations: [MainLayoutComponent],

exports: [MainLayoutComponent]

})export class LayoutModule { }

Nos vamos al módulo de login.

Archivo login-card.component.css

.full-page-background {

position: absolute;

z-index: 1;

height: 100%;

width: 100%;

display: block;

top: 0;

left: 0;

background-size: cover;

background-position: center center;

}

.login-card {

z-index: 2;

background-color:#ffffff !important;

min-height: 200px;

text-align: center;

margin-top: -80px !important;

font-size: 18px;

font-family: "Arial", "Helvetica", "Arial", sans-serif;

font-weight: 300;

width: 500px;

}

.login\_logo {

width: 100%;

height: 60px;

margin-top: 20px;

}

.border-top-input {

border-top: solid 1px #c6c6c6;

padding-top: 20px;

margin-top: 14px;

}

Archivo login-card.component.html

<mat-card class="login-card" (mouseover)='bluringBackground()' (mouseleave)='unbluringBackground()'>

<div class="login\_logo">

<img src="https://drive.google.com/uc?id=1UCuwEFL2\_wANdGi7\_6N7\_xs5YDoBSzuq" alt="Aweather" title="Aweather" />

</div>

<div class="border-top-input">

<form #loginForm="ngForm" (ngSubmit)="loginSubmit(loginForm)">

<mat-card-content>

<mat-grid-list cols="1" rowHeight="60px">

<mat-grid-tile>

<mat-form-field>

<input matInput placeholder="Usuario" name="user" [(ngModel)]="user">

</mat-form-field>

</mat-grid-tile>

<mat-grid-tile>

<mat-form-field>

<input matInput placeholder="Contraseña" name="password" [(ngModel)]="password">

</mat-form-field>

</mat-grid-tile>

</mat-grid-list>

<mat-card-actions class="mat-card-button-login">

<button mat-raised-button type="submit" class="login-button mat-primary">

<i class="fas fa-key fa-lg"></i>

</button>

</mat-card-actions>

</mat-card-content>

</form>

</div></mat-card>

<div class="full-page-background" [@myAnimation]="showBackgroundImage" [ngStyle]="{'background-image': 'url(' + activeBackgroundImage + ')'}"></div><!--PREVENT BLINKING--><img src="https://drive.google.com/uc?id=1wsk0lus\_UmgB1x7YBHZ30PHhd59oI\_QJ" style="display:none;" /><img src="https://drive.google.com/uc?id=1hSNFcWC5LECKPbws1cMJfcmVGiXPe40v" style="display:none;" /><img src="https://drive.google.com/uc?id=13XuzG26KujCSlAqGwAnXMtjNqx5mxYFA" style="display:none;" />

<img src="https://drive.google.com/uc?id=13sGmUKF193j2BDrvOtAuGMV27mEe5HSE" style="display:none;" /><img src="https://drive.google.com/uc?id=1uf\_L2Ms1uGLXMgN5WCrUj0zTVdxN-g2p" style="display:none;" /><img src="https://drive.google.com/uc?id=13nbhGXFU0icwZ1llfTDAFap0r36SbRXX" style="display:none;" />

Archivo login-card.component.ts, el que os dejo una pequeña animación para que si os interesa, tengáis un ejemplo de como realizar animaciones en Angular.

import { Component } from '@angular/core';import { Router } from '@angular/router';import { trigger, state, transition, animate, style } from '@angular/animations';

@Component({

selector: 'app-login-card',

templateUrl: './login-card.component.html',

styleUrls: ['./login-card.component.css'],

animations: [

trigger(

'myAnimation',

[

state('true', style({ opacity: 1 })),

state('false', style({ opacity: 0.3 })),

transition('1 => 0', animate('900ms', style({ opacity: 0.3 }))),

transition('0 => 1', animate('900ms', style({ opacity: 1 })))

])

]

})

export class LoginCardComponent {

backgrounds = [

{ main: 'https://drive.google.com/uc?id=1wsk0lus\_UmgB1x7YBHZ30PHhd59oI\_QJ', blur: 'https://drive.google.com/uc?id=13sGmUKF193j2BDrvOtAuGMV27mEe5HSE' },

{ main: 'https://drive.google.com/uc?id=1hSNFcWC5LECKPbws1cMJfcmVGiXPe40v', blur: 'https://drive.google.com/uc?id=1uf\_L2Ms1uGLXMgN5WCrUj0zTVdxN-g2p' },

{ main: 'https://drive.google.com/uc?id=13XuzG26KujCSlAqGwAnXMtjNqx5mxYFA', blur: 'https://drive.google.com/uc?id=13nbhGXFU0icwZ1llfTDAFap0r36SbRXX' }

];

backgroundIndex = Math.floor(Math.random() \* this.backgrounds.length) + 0;

secondsToChange = 50;

pause = false;

activeBackgroundImage = '';

showBackgroundImage = false;

user = '';

password = '';

constructor(

private \_router: Router,

) {

this.activeBackground(false);

setInterval(() => { this.\_onChangeBackground(); }, 600 \* this.secondsToChange);

}

public loginSubmit() {

}

public activeBackground(withblur) {

this.showBackgroundImage = false;

let back = this.backgrounds[this.backgroundIndex].main;

if (withblur) { back = this.backgrounds[this.backgroundIndex].blur; }

setTimeout(() => {

this.activeBackgroundImage = back;

this.showBackgroundImage = true;

}, 1000);

}

public bluringBackground() {

this.activeBackground(true);

this.pause = true;

}

public unbluringBackground() {

this.pause = false;

this.activeBackground(false);

}

private \_onChangeBackground() {

if (!this.pause) {

this.backgroundIndex++;

if (this.backgroundIndex >= this.backgrounds.length) {

this.backgroundIndex = 0;

}

this.activeBackground(false);

}

}

}

Archivo login-card.container.css

.login-container .full-page-background {

position: absolute;

z-index: 1;

height: 100%;

width: 100%;

display: block;

top: 0;

left: 0;

background-size: cover;

background-position: center center;

}

.login-container {

background-color: #e0e0e0;

}

Archivo login-card.container.html, en el qué, fijaros, incluimos otro router outlet para poder renderizar el login sin que interfiera el menú de la aplicación

<mat-grid-list cols="1" rowHeight="fit" style="min-height: 100vh" gutterSize="0px" class="login-container">

<mat-grid-tile>

<router-outlet></router-outlet>

</mat-grid-tile>

</mat-grid-list>

En la carpeta login nos vamos a crear el módulo para las rutas, creamos un nuevo archivo llamado login-routing.module.ts.

Fijaros que aquí estamos creando nuestra primera herencia de rutas con la propiedad children.

La ruta contenedora renderiza un componente y como dentro de ese componente tenemos un router outlet y en el path del hijo le estamos especificando otro componente, angular se encarga de realizar esta jerarquía en el proceso de renderizado.

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

import { LoginContainerComponent } from './login-container/login-container.component';import { LoginCardComponent } from './login-card/login-card.component';

const loginRoutes: Routes = [

{ path: 'home', component: LoginContainerComponent, children: [

{ path: '', component: LoginCardComponent, pathMatch: 'full' },

{ path: 'login', component: LoginCardComponent }

]

}

];

@NgModule({ imports: [

RouterModule.forChild(loginRoutes)

], exports: [

RouterModule

]

})

export class LoginRoutingModule {

}

Y ahora en el archivo login-card.module.ts importamos el módulo de rutas y el módulo shared.

import { NgModule } from '@angular/core';import { CommonModule } from '@angular/common';import { LoginContainerComponent } from './login-container/login-container.component';import { LoginCardComponent } from './login-card/login-card.component';import { SharedModule } from '../shared/shared.module';import { LoginRoutingModule } from './login-routing.module';

@NgModule({

imports: [

CommonModule,

SharedModule,

LoginRoutingModule

],

declarations: [LoginContainerComponent, LoginCardComponent]

})export class LoginModule { }

Por último en la carpeta weather, nos vamos a crear su correspondiente archivo de rutas, el weather-routing.module.ts, este será muy parecido al del login con rutas hijas.

import { NgModule } from '@angular/core';import { RouterModule, Routes } from '@angular/router';import { AppComponent } from '../app.component';import { WeatherCardComponent } from './weather-card/weather-card.component';import { MainLayoutComponent } from '../layout/main-layout/main-layout.component';

const weatherRoutes: Routes = [

{

path: 'weather',

component: MainLayoutComponent,

children: [

{ path: 'info', component: WeatherCardComponent }

]

}

];

@NgModule({

imports: [

RouterModule.forChild(weatherRoutes)

],

exports: [

RouterModule

]

})export class WeatherRoutingModule { }

Y por último lo importamos en el Módulo, archivo weather.module.ts

import { NgModule } from '@angular/core';import { CommonModule } from '@angular/common';import { SharedModule } from '../shared/shared.module';import { WeatherCardComponent } from './weather-card/weather-card.component';import { WeatherForecastComponent } from './weather-forecast/weather-forecast.component';import { WeatherSearchComponent } from './weather-search/weather-search.component';import { WeatherService } from './services/weather.service';import { WeatherRoutingModule } from './weather-routing.module';import { LayoutModule } from '../layout/layout.module';

@NgModule({

imports: [

CommonModule,

SharedModule,

WeatherRoutingModule,

LayoutModule

],

declarations: [

WeatherCardComponent,

WeatherForecastComponent,

WeatherSearchComponent

],

providers: [WeatherService],

exports: [

WeatherCardComponent,

WeatherSearchComponent

]

})export class WeatherModule { }

Enlaces: Router link

Vamos a ver las dos maneras básicas que tenemos para poder hacer enlaces.

Una es inyectando en un componente el router y otra desde el propio template de html.

Por ejemplo cuando pulse el botón del login.

Previamente tengo que haber inyectado en el componente el Router de angular.

public loginSubmit() {

this.\_router.navigate([`/weather/info`]);

}

(ngSubmit)="loginSubmit()"

Y ahora en el menú top, vamos a volver al login cuando pulse salir de la aplicación.

Para ello tenemos que importar en el core.module.ts las rutas

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

....

....

RouterModule.forChild([])

Y en el html

<i class="fa fa-sign-out-alt fa-lg" [routerLink]="['/home/login']"></i>

Con routerLink podemos especificar el path de la ruta de forma dinámica, si la variable que le pasemos en la ruta cambia su valor, cambiará la ruta también.

Parámetros de rutas

Ahora que hemos visto el funcionamiento de las rutas, vamos a aprender como pasar parámetros a nuestras rutas.

Los parámetros de las rutas se definen en el path con “:” y el nombre del parámetro.

Cambiamos el weather-routing.module.ts para ver un ejemplo:

const weatherRoutes: Routes = [

{ path: 'weather', component: MainLayoutComponent, children: [

{ path: 'search', component: WeatherSearchComponent },

{ path: 'info/:cityName', component: WeatherCardComponent }

]

}

];

Cambiamos la ruta del login para poder entrar a la aplicación

public loginSubmit() {

this.\_router.navigate([`/weather/search`]);

}

Nos vamos al left menu, archivo left-menu.component.html e incorporamos un routerLink con un parámetro variable.

<mat-list-item \*ngFor="let city of cities" [routerLink]="['/weather/info', city]">

<a mat-button mat-line>{{city}}</a>

<i class="fas fa-map-marker-alt font-size-20"></i>

</mat-list-item>

Ahora la siguiente pregunta sería como podemos leer esos valores de nuestra url en nuestro componente para pasárselos a nuestro servicio del tiempo y poder consultar en tiempo en distintas ciudades, esto lo veremos en el siguiente vídeo.

Activación de rutas

Una vez que estamos en una ruta queremos saber si esta activada y, si tiene parámetros, como podemos recogerlos.

Lo primero que vamos a ver es como podemos recoger parámetros de nuestras rutas. Para ello tenemos que inyectar en nuestro componente ActivatedRoute que nos proporciona Angular.

Nos vamos al weather-card.component.ts y el constructor lo inyectamos.

constructor(

private \_weatherService: WeatherService,

private \_activatedRoute: ActivatedRoute

) { }

Ahora para acceder a la ruta en ngOnInit tendremos que hacer lo siguiente:

ngOnInit() {

this.\_activatedRoute.params.subscribe(params => {

this.cityName = <string>(params['cityName'] ? params['cityName'] : 'Madrid, ES');

this.\_weatherService.getWeatherInfo(this.cityName).subscribe(

data => {

if (data['query'].results === undefined) {

alert('La ciudad buscada no existe');

} else {

this.weather = this.\_weatherService.mapResult(data['query'].results.channel);

}

},

error => {

alert(error.message);

}

);

});

}

Angular nos ofrece una manera muy sencilla en el template para saber en que ruta estamos y poder por ejemplo marcar en el menú, algo que se hace muy complicado siempre en la web.

Esto lo podemos hacer con routerlinkactive

Nos vamos al left-menu.component.html y lo dejaremos con el siguiente html.

<div class="menu-list-divider"> v.1.0 </div>

<mat-list>

<mat-list-item routerLinkActive="active">

<a mat-button mat-line [routerLink]="['/weather/search']">Añadir ciudades</a>

<i class="fab fa-cloudversify font-size-20"></i>

</mat-list-item>

<div class="menu-list-divider"> Ciudades </div>

<div appDropdown class="drop-down">

<mat-list-item routerLinkActive="active">

<a mat-button mat-line appDropdowntoggle class="drop-down-toggle">Ciudades</a>

</mat-list-item>

<div class="drop-down-items">

<mat-list-item \*ngFor="let city of cities" routerLinkActive="active">

<a mat-button mat-line [routerLink]="['/weather/info', city]">{{city}}</a>

<i class="fas fa-map-marker-alt font-size-20"></i>

</mat-list-item>

</div>

</div>

</mat-list>

Añadimos algo de estilo en el archivo left-menu.component.css

:host /deep/ .mat-list-item.active {

background-color: #34579c;

color: #fafafa;

}

Guardias en las rutas

Las guardias son muy útiles en Angular, nos permite determinar ciertos comportamientos en las rutas de nuestra aplicación.

Definimos las guardias con un servicio el cual puede implementar CanActivate, CanActivateChild, CanDeactivate, Resolve.

En los métodos que implementa la guardía devolverá true o false.

* CanActivate o CanActivateChild: Nos permite acceder o no a una determinada ruta
* CanDeactivate: Nos permite decidir si el usuario podra dejar esta ruta para ir a otra. Por ejemplo cuando implementamos una pasarela de pago.
* Resolve: Esta ruta no previene la navegación. Su finalidad es la de proporcionar datos antes de que se cargue el componente. Es útil cuando necesitamos que se cargue cualquier información antes que cualquier otra cosa.

Vamos a implementar un servicio con CanActivate.

Nos vamos a la carpeta src/app/core y escribimos los siguientes comandos:

mkdir auth-guard

cd auth-guard

ng generate service auth-guard --spec false

Vamos a simular en el archivo auth-guard.service.ts que no tenemos permiso para acceder a una ruta que implemente este CanActivate.

import { Injectable } from '@angular/core';import { Router, CanActivate } from '@angular/router';

@Injectable()

export class AuthGuardService implements CanActivate {

constructor(

private router: Router

) { }

canActivate() {

if (this.userAuthenticated()) {

return true;

}

this.router.navigate(['/home/login']);

return false;

}

private userAuthenticated(): boolean {

return false;

}

}

Importamos en el módulo core.module.ts este servicio.

providers: [AuthGuardService],

Y en el fichero weather-routing.module.ts cambiamos temporalmente el archivo para simular que no tenemos permiso.

{ path: 'info/:cityName', component: WeatherCardComponent, canActivate: [AuthGuardService] }

Creación de formularios reactivos

Angular tiene dos formas diferentes para crear formularios, template driven y reactive forms.

Nosotros vamos a centrarnos en lo fundamental de los formularios reactivos de Angular forms.

Lo primero que tenemos que hacer para empezar a usar los formularios reactivos, es importar en nuestro módulo materia-forms.module el módulo ReactiveFormsModule.

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

Ahora vamos a definir los FormControls que tenemos en el formulario y el FormGroup.

Vamos a utilizar de ejemplo el formulario del login.

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

....

loginGroup: FormGroup;

....

....

ngOnInit() {

this.loginGroup = new FormGroup({

user: new FormControl(''),

password: new FormControl('')

});

}

public loginSubmit() {

console.log(this.loginGroup.value, this.loginGroup.valid);

// this.\_router.navigate([`/weather/search`]);

}

Y modificamos el template login-card.component.html

<form [formGroup]="loginGroup" >

<mat-card-content>

<mat-grid-list cols="1" rowHeight="60px">

<mat-grid-tile>

<mat-form-field>

<input matInput placeholder="Usuario" formControlName="user">

</mat-form-field>

</mat-grid-tile>

<mat-grid-tile>

<mat-form-field>

<input matInput placeholder="Contraseña" formControlName="password">

</mat-form-field>

</mat-grid-tile>

</mat-grid-list>

<mat-card-actions class="mat-card-button-login">

<button mat-raised-button type="submit" class="login-button mat-primary">

<i class="fas fa-key fa-lg"></i>

</button>

</mat-card-actions>

</mat-card-content>

</form>

Validación y control de errores

Hemos implementado un reactive forms pero sin validación, así que vamos a aprender como hacerlo.

Lo hacemos con la clase validators en la que tenemos multitud de validaciones disponibles.

ngOnInit() {

this.loginGroup = new FormGroup({

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

password: new FormControl('', [Validators.required, Validators.minLength(4)])

});

}

Y ahora podemos hacer cosas, como habilitar el botón sólo cuando el formulario sea válido.

<button mat-raised-button type="submit" [disabled]="loginGroup.invalid" class="login-button mat-primary">

<i class="fas fa-key fa-lg"></i>

</button>

Incluso podemos mostrar los errores, para informar al usuario de los mismos de esta manera.

<form [formGroup]="loginGroup">

<mat-card-content>

<mat-grid-list cols="1" rowHeight="60px">

<mat-grid-tile>

<mat-form-field>

<input matInput placeholder="Usuario" formControlName="user">

<mat-error \*ngIf="loginGroup.get('user').hasError('required')">

El usuario es requerido

</mat-error>

<mat-error \*ngIf="loginGroup.get('user').hasError('email')">

El email del usuario no es válido

</mat-error>

</mat-form-field>

</mat-grid-tile>

<mat-grid-tile>

<mat-form-field>

<input matInput type="password" placeholder="Contraseña" formControlName="password">

<mat-error \*ngIf="loginGroup.get('password').hasError('required')">

El password es requerido

</mat-error>

<mat-error \*ngIf="loginGroup.get('password').hasError('minlength')">

El password no es válido

</mat-error>

</mat-form-field>

</mat-grid-tile>

</mat-grid-list>

<mat-card-actions class="mat-card-button-login">

<button mat-raised-button type="submit" [disabled]="loginGroup.invalid" class="login-button mat-primary">

<i class="fas fa-key fa-lg"></i>

</button>

</mat-card-actions>

</mat-card-content>

</form>

**Envío de un formulario**

Hemos visto como crear un formulario, como validarlo y ahora nos queda enviarlo. Angular nos simplifica todo lo posible el envío de un formulario. Lo único que necesitaremos es tener una función en el controlador para recoger el evento submit y añadir el evento en el template

public loginSubmit(loginGroup: FormGroup) {

console.log(this.loginGroup.value, this.loginGroup.valid);

// this.\_router.navigate([`/weather/search`]);

}

....

<form [formGroup]="loginGroup" (submit)="loginSubmit(loginGroup)">

....

Todo lo practicado hasta ahora lo podéis encontrar en **[github](https://github.com/SamazoOo/Angular-desde-cero)** en la carpeta aweather-final

Compilando para producción

Ya tenemos nuestra aplicación lista para desplegarla en cualquier servidor, para ello Angular nos lo pone extremadamente fácil compilarla.

ng build --prod

El directorio de salida se especifica en el fichero angular.json en la propiedad outputPath.

Librerías para visual Co**@Input**

de

* **TSLint**
* **Angular Language Service**