TUTORIAL ANGULAR con TypeScript.

<https://angular.io/docs/ts/latest/tutorial/>

El Tour de los Héroes, tutorial paso a paso para crear una aplicación Angular con TypeScript.

Caso de uso:

Relación de trabajadores (Héroes) de una agencia de empleo.

La aplicación se implementa a un nivel básico con las siguientes características:

* Listado de héroes.
* Edición de la ficha del héroe seleccionado.
* Navegación por diferentes vistas.

Se cubren los fundamentos básicos de Angular. Se hará uso de los componentes integrados para presentar y ocultar datos en formato de lista. Se crea un componente para crear una ficha de cada héroe y otro para mostrar una lista de héroes. Se enlazarán datos en un solo sentido de solo lectura. Y campos editables para enlace de datos bidireccional. Se crean métodos básicos para control de eventos de teclado y de clic de ratón. Habrá una relación maestro detalle, al seleccionar un elemento de la lista se presentará su ficha o detalle para su edición. Se formatearán los datos mediante “pipes.” Así mismo se creará un servicio compartido para el “ensamble” de los elementos de la lista. Y, por último, se implementan técnicas de enrutamiento para navegar por las diferentes vistas.

[Aquí](https://embed.plnkr.co/?show=preview) se puede ver el ejemplo terminado.

La aplicación irá refinándose paso a paso, refactorizando y ampliando los pasos anteriores.

COMENZAMOS.

Quizás la parte más tediosa y que suele plantear problemas, sobre todos las diferentes versiones de todos los módulos implicados. Se parte de que se ha instalado Node (v4 en adelante) y NPM (v3 en adelante.)

La configuración de un entorno “QuickStart” se ha automatizado en [ésta dirección](https://angular.io/docs/ts/latest/guide/setup.htm).

[en github](https://github.com/angular/quickstart)  se mantiene esta configuración de inicio rápido.

Se obvia aquí la instalación de Node y NPM , (<https://docs.npmjs.com/getting-started/installing-node>)

Comprobar que las versiones de Node y de NPM son como mínimo 4.x.x y 3.x.x o superiores respectivamente. Para ello, en la consola:

node –v y npm -v

En mi caso: v6.9.5 de node y 3.10.10 de npm.

Para la instalación del entorno QuickStart de Angular:

* Crear una carpeta de proyecto (se ha de llamar quickstart y se puede cambiarle el nombre más adelante).
* [Clonar](https://angular.io/docs/ts/latest/guide/setup.html#clone) o [descargar](https://angular.io/docs/ts/latest/guide/setup.html#download) la semilla de inicio rápido en la carpeta de proyecto.
* Instalar [la NGP](https://angular.io/docs/ts/latest/guide/setup.html#install-prerequisites) paquetes.
* Ejecutar npm start para iniciar la aplicación de ejemplo.

En mi caso, instalé previamente el entorno siguiendo los pasos descritos en éste apartado y todo fue bien. Lo subí, hasta el paso 3 del tutorial a <https://github.com/carlgarpel/Angular2Tutorial> y en otro ordenador he vuelto a clonarlo y al ejecutar npm start en la consola ha funcionado correctamente. Eso sí, se han descargado 13.134 ficheros. Esa es una de las cosas que “enfrían” a la hora de abordar este tipo de framework, el sinfín de archivos, versiones y dependencias que los pueden hacer tambalearse ante futuras actualizaciones. Pero en fin, sigamos.

Para editar los ficheros emplearé Sublime Text 3. Y para configurar TypeScript con Intellisense sigo las instrucciones de <https://cmatskas.com/getting-started-with-typescript-and-sublime-text/>

Con Sublime Text abro la carpeta donde he instalado la App, en mi caso:

F:\angular\Angular2Tutorial

Hay que decir que, al ejecutar npm start se inició la aplicación en el navegador predeterminado, en mi caso Crome, y que mientras vayamos modificando la aplicación automáticamente se irá recargando y mostrando las nuevas versiones. <http://localhost:3000/>

En el [inicio rápido](https://angular.io/docs/ts/latest/quickstart.html) se describen los elementos básicos de una aplicación Angular:

Las aplicaciones angular se componen de componentes. Un componente es la combinación de una plantilla HTML y una clase componente que controla una parte de la pantalla. Éste es un ejemplo de un componente que muestra una cadena simple:

En el fichero:

src / app / app.component.ts

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

@Component({

selector: 'my-app',

template: `<h1>Hello {{name}}</h1>`

})

export class AppComponent { name = 'Angular'; }

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Este es un fichero TypeScript, que como leemos en Wikipedia:

*“TypeScript es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft. Es un superconjunto de JavaScript, que esencialmente añade tipado estático y objetos basados en clases.”*

Mi versión en este momento de TypeScript (tsc -v) es v.2.1.6. [Aquí](http://www.jaimeolmo.com/2015/08/primeros-pasos-usando-typescript/), hay otras explicaciones sobre compilación de TypeScript.

Otra aclaración previa:

ECMAScript v6 (Abreviado como ES6 o ES2015) es el estándar que sigue JavaScript a partir de Junio de 2015. Hasta ese momento la versión de JS que estamos usando en nuestros navegadores y Node.js, es la v5.

Y entre las características que incorpora tenemos, como se explica en <https://www.funcion13.com/aprende-es6-las-plantillas-con-cadenas/>

En una plantilla de cadena se utiliza el `(acento)` .

El funcionamiento básico es el de usar el `acento` y otorgaremos a las cadenas los siguientes superpoderes:

* Podemos interpolar variables.
* Incluir expresiones, no solo variables.
* Cadenas multi-línea.
* Formato de cadena.

TypeScript hace uso de plantillas de texto como en este caso:

template: `<h1>Hello {{name}}</h1>`

Como se puede ver, dentro de la plantilla de texto tenemos etiquetas HTML, y variables, en este caso de Angular, una interpolación {{name}}.

En app.compenent.ts tenemos:

Importamos Component de la librería global @angular/core, definimos nuestro componente, que en este caso se aplicará sobre la aplicación y que define una salida de texto. Exportamos la clase que inicializa la variable name con el valor ‘Angular’.

Este componente básico será la raíz de un árbol de componentes anidados a medida que evolucione la aplicación.

app/app.module.ts

Define AppModule, el módulo de raíz que dice angular cómo ensamblar la aplicación. En este momento se declara sólo el AppComponent. Pronto habrá más componentes que declarar.

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

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

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

@NgModule({

imports: [ BrowserModule ],

declarations: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

En toda aplicación además nos encontraremos con otro fichero tipo main.ts:

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

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

platformBrowserDynamic().bootstrapModule(AppModule);

que se utiliza para la compilación.

|  |  |
| --- | --- |
| main.ts | Compila la aplicación con el [compilador JIT](https://angular.io/docs/ts/latest/glossary.html#jit) y [bootstraps](https://angular.io/docs/ts/latest/guide/appmodule.html" \l "main" \o "bootstrap la aplicación) definiendo el módulo principal de la aplicación ( AppModule) para ejecutar en el navegador. El compilador JIT es una opción razonable durante el desarrollo de la mayoría de los proyectos y es la única opción viable para ejecutar en vivo en entornos como Plunker. Se puede profundizar en las alternativas de compilación y  [despliegue](https://angular.io/docs/ts/latest/guide/deployment.html)  más tarde en la documentación.  P |

Pero sigamos con el tutorial.

La estructura que fichero que hemos creado es:

angular-tour-of-heroes

src

app

app.component.ts

app.module.ts

main.ts

index.html

styles.css

systemjs.config.js

tsconfig.json

node\_modules ...

package.json

Hemos compilado y ejecutado mediante npm start.

**PASO 1**

Mostrar un elemento.

Jugando un poco con estilos ccs3, con ayuda de <https://developer.mozilla.org/es/docs/Web/CSS/CSS_Flexible_Box_Layout/Usando_las_cajas_flexibles_CSS>

He creado una plantilla con capas flexibles típica y he adaptado app.components.ts a la presentación de clase Vias, con los siguiente campos:

class Vias {

id: number;

tipoVia: string;

nombreVia: string;

}

Y he modificado el template para que recoja la estructura de capas flexible.

En síntesis, el código

* importa {Component} de Angular
* Crea un @Component que describe:
  + Un selector sobre el que se aplicará el template.
  + Un template o plantilla html en la que se puden crear vínculos direccionales o bidereccionales con, en este caso, la clase Vias.

Decir que con la doble llave {{campo}

} creamos un vínculo en una sola dirección, es decir, de solo lectura; mientras que con <input [(ngModel)]="via.nombreVia" placeholder="via"> estamos creando un vínculo bidireccional que permite editar, modificar el campo nombreVia de la clase via. “placeholder” añade una explicación de solo lectura al elemento imput.

Para poder utilizar la bi-direccionalidad es necesario antes de poder utilizar el enlace de datos bidireccionales en un formulario , necesitamos importar el paquete FormsModule en nuestro módulo app.modulo.ts.

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

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

Añadimos la matriz @NgModuledel. Esta matriz contiene la lista de módulos externos utilizados por nuestra aplicación. Ahora hemos incluido el paquete de formularios ngModel.

app.mocule.ts queda así:

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

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

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

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

@NgModule({

imports: [

BrowserModule,

FormsModule

],

declarations: [

AppComponent

],

bootstrap: [ AppComponent ]

})

El código completo de app.components.ts:

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

export class Vias {

id: number;

tipoVia: string;

nombreVia: string;

}

@Component({

selector: 'my-app',

template: `

<header><h1>{{title}}</h1></header>

<div id='main'>

<article>

<h2>Código: {{via.id}}</h2>

<div class="article"><label>Tipo de vía: </label>{{via.tipoVia}}</div>

<div> <label>Nombre de vía: </label>{{via.nombreVia}} </div>

</article>

<nav>

<label>Edición del Nombre de vía: </label>

<input [(ngModel)]="via.nombreVia" placeholder="via">

</nav>

<aside>Localizador</aside>

</div>

<footer>Vial - 2017</footer>

`

})

export class AppComponent {

title = 'Callejero';

via: Vias = {

id: 1,

tipoVia: 'CALLE',

nombreVia: 'ALTAMIRA'

};

}

**Paso 2**

Hasta ahora solo mostramos una calle y quiere mostrar una lista de ellas. También queremos permitir al usuario seleccionar una vía y mostrar sus detalles.

**Cambio en el .ts y en el .css**

Equivale a una interfaz típica de maestro-detalle, donde un campo hace de clave que se vincula con una lista o tupla de registros.

Para lo que se ha de definir:

* -Una lista de elementos, en este ejemplo array de Vias.
* -Representar esta lista en la plantilla de la vista.

En app.components.ts se añade un array constante de clase Via:

export class Via {

id: number;

tipoVia: string;

nombreVia: string;

distrito: number;

seccion: number;

codigoPostal: string;

}

El array:

const VIAS: Via[] = [

{id: 10, tipoVia: 'CALLE', nombreVia: 'ALTAMIRA', distrito: 7, seccion: 1, codigoPostal: '04006' },

{id: 11, tipoVia: 'CALLE', nombreVia: 'BECERRA', distrito: 8, seccion: 6, codigoPostal: '04006' },

{id: 12, tipoVia: 'CALLE', nombreVia: 'CARRERA DEL DARRO', distrito: 7, seccion: 3, codigoPostal: '04005' },

{id: 13, tipoVia: 'CALLE', nombreVia: 'DINAMARCA', distrito: 7, seccion: 9, codigoPostal: '04005' },

{id: 14, tipoVia: 'AVDA', nombreVia: 'FRANCISCO DE ASÍS', distrito: 7, seccion: 5, codigoPostal: '04003' }

];

Ahora creamos y exponer una propiedad pública en AppComponent que pueda ser enlaza en la plantilla de vista.

cvias = VIAS;

donde cvias es la propiedad pública que estamos definiendo y VIAS el array que creamos anteriormente. No tenemos que establecer que cvias es del tipo via, ya que TypeScript es capaz de inferirlo de VIAS[]. Se podría haber definido aquí la lista, pero no olvidemos que en última instancia obtendremos los datos desde un servidor de datos, y que tiene sentido separar el modelo de datos de la implementación de la clase desde un principio.

Para mostrar los datos se añade a la plantilla una lista desordenada en código html

1. <h2>Listado</h2>
2. <ul class="vias">
3. <li>
4. <!—las “via” de cvias -->
5. </li>
6. </ul>

Queremos enlazar la matriz vías con la plantilla, iterar sobre ella y mostrar todos los elementos de la lista. Para ello haremos uso de métodos de Angular.

ngFor

<li \*ngFor="let via of cvias">

En esta sintaxis de Angular \* indique que el elemento <li> y sus hijos constituyen una plantilla maestra.

ngFor itera sobre cvias[] y devuelve los diferentes ítems de AppComponent.cvias a cada elemento de la lista de la plantilla.

Literalmente ngFor toma cada vía de cvias[], la almacena en local en la variable via y la pone a disposición de la instancia de la plantilla correspondiente.

La palabra reservada lef delante de via indica que ésta es una variable de la plantilla. Podemos hacer referencia a esta variable dentro de la plantilla para acceder a las propiedades de cvias.

Más información en estos enlaces:

[que muestran datos](https://angular.io/docs/ts/latest/guide/displaying-data.html#ngFor) y [plantilla de sintaxis](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngFor).

Dentro del elemento <li></li> se hace referencia a las variables generados en ngFor

<li \*ngFor="let via of cvias">

<span class="badge">{{via.id}} </span> {{via.nombreVia}}

</li>

Los estilos se definen en el fichero styles.css y para no extender esta guía no entraré en los detalles de su implementación.

El siguiente paso será conectar la lista de vías con el elemento de la lista que se muestra en otra parte de la página web. Se trata de enlazar un elemento con el elemento seleccionado de la lista. Para ello creamos la propiedad selectedVia y le implementamos el evento clic.

<li \*ngFor="let via of cvias" (click)="onSelect(via)">

El (evento) representa el evento indicado asociado al element <li>, en este caso (click)

La expresión a la derecha del signo igual llama al método AppComponent onSelect(), al tiempo que se le pasa la variable de entrada “via”, y que es la variable que se ha creado previamente mediante ngFor.

Hay que definir el método onSelect() cuya tarea será exponer el elemento sobre el que se ha hecho clic dentro de la lista. Previamente creamos la variable selectedVia: Via; que ha de representar el elemento de la lista sobre el que se ha clicado. Podríamos inicializar esta variable o no, si lo que queremos es que no se presente el detalle de cada elemento hasta que no se haga clic en uno de ellos.

selectedVia: Via={id: 10, tipoVia: 'CALLE', nombreVia: 'ALTAMIRA', distrito: 7, seccion: 1, codigoPostal: '04006' };

El método se define de la siguiente manera:

onSelect(via: Via): void { this.selectedVia = via; }

es decir, asigna a la variable pública que hemos definido antes la “via” que recibe como parámetro de entrada el método de evento. Con lo que ya podemos hacer algo de este tipo:

<h2>{{selectedVia.nombreVia}}Detalle</h2>

<div><label>id: </label>{{selectedVia.id}}</div>

<div>

<label>name: </label>

<input [(ngModel)]="selectedVia.nombreVia" placeholder="via"/>

</div>

Si decidimos no inicializar la variable selectedVia, cuando iniciemos la aplicación le variable tendrá un valor undefined y, por tanto, la propiedad id o nombreVia serán [null], lo que producirá el siguiente error:

EXCEPTION: TypeError: Cannot read property 'name' of undefined in [null]

Para evitar este error, en los elementos de la plantilla donde hayamos enlazado con selectedVia añadiremos otro \*ng

<aside \*ngIf="selectedVia">Edite el nombre de vía:

<h2>{{selectedVia.nombreVia}}</h2>

<div><label>id: </label>{{selectedVia.id}}</div>

<div>

<label>nombre: </label>

<input [(ngModel)]="selectedVia.nombreVia" placeholder="via"/>

</div>

</aside>

\*ngIf hace que si la variable enlazado no se corresponde con selectedVia, no se mostrarán los elementos secundarios DOM, es decir, todo lo que hay entre <aside></aside> no se mostrará. Es por ello que ngIfy ngForse se conocen como "directivas estructurales", ya que pueden cambiar la estructura de las porciones del DOM. En otras palabras, en el DOM, angular se encarga de añadir y mostrar los elementos.

Más información acerca de ngIf, ngFory otras directivas estructurales en las [Directivas estructurales](https://angular.io/docs/ts/latest/guide/structural-directives.html) y [plantilla de sintaxis](https://angular.io/docs/ts/latest/guide/template-syntax.html#directives) capítulos.

Para definir el estilo de la selección de modo que cuando seleccionemos un registro (hagamos clic en él) se cambie el color del ítem, crearemos un enlace a la propiedad class selected

[class.selected]="via === selectedVia"

Si la variable local via es igual que selectedVia, se aplica class.selected, en caso contrario se elimina esta clase del elemento <li>

Los corchetes indican que se trata de una unión de la propiedad el flujo de datos.

Con lo que el elemento <li> quede den siguiente modo:

<li \*ngFor="let via of cvias"

[class.selected]="via === selectedVia"

(click)="onSelect(via)">

<span class="badge">{{via.id}} </span> {{via.nombreVia}}

</li>

Más información acerca de [los enlaces de la propiedad](https://angular.io/docs/ts/latest/guide/template-syntax.html#property-binding) en el capítulo de plantilla de sintaxis.

**PASO 3**