**DESARROLLO DE APLICACIONES CON ANGULAR**

* Contenido del curso:

1. Introducción a TypeScrypt.
2. Estructura y programación de aplicaciones angular
3. Componentes, servicios y pipes.
4. Navegación entre páginas. Rutas.
5. Peticiones Http. Firebase.
6. Formularios en angular
7. Aplicación CRUD con peticiones a un backend.
8. Preparar aplicación para producción.
9. Optimización de aplicación ángular. Módulos y Lazy Load.

Intentaremos hacer en cada jornada 2 puntos.

**REFERENCIAS PREVIAS:**

<https://gesproy.jccm.es/issues/81550>

Es una aplicación angular semejante a la que haremos en este curso.

Es esta tarea de gesproy también se utiliza un backend basado en spring.

**SOFTWARE QUE VAMOS A NECESITAR EN ESTE CURSO**

* Node. Node es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome , orientado a eventos asíncronos.

Incluye el gestor de paquetes npm, para instalar las dependencias que vayamos a utilizar.

Para instalar node.

Comprobar si tenemos node.

node –versión

Si no lo tenemos, hay que ir a la página de node y descargarlo e instalar.

* TypeScript. No hace falta tenerlo instalado para programar con angular, ya que angular compilará el proyecto , pero para ver algunos ejemplos básicos de TypeScript, lo vamos a usar.

Igual, miramos si lo tenemos instalado con tsc --version

* Microsoft Visual Code. Es el editor de texto que vamos a utilizar para escribir nuestros programas. Se puede utilizar cualquier otro como Eclipse, Atom, notepad++, etc, pero desde luego que este es el recomendable.
* Angular. Claro que tenemos que tener instalado Angular porque es lo mas importante de este curso. Se puede ver con ng –version.
* PostMan. Es el software por excelencia para probar servicios Rest. Nosotros lo vamos a utilizar para comprobar los servicios Rest que vamos a consumir. Ejemplo de apis que podemos utilizar: https://github.com/public-apis/public-apis
* Git. Vamos a utilizar git para el control de versiones en nuestros programas. Veremos algunos de los comandos más importantes de git.

**QUE ES ANGULAR**

Angular es un framework de desarrollo para JavaScript creado por Google. La finalidad de Angular es facilitarnos el desarrollo de aplicaciones web SPA y además darnos herramientas para trabajar con los elementos de una web de una manera más sencilla y óptima.

Otro propósito que tiene Angular es la separación completa entre el front-end y el back-end en una aplicación web.

**¿Qué es una aplicación SPA?**

Una aplicación web SPA creada con Angular es una web de una sola página, en la cual la navegación entre secciones y páginas de la aplicación, así como la carga de datos, se realiza de manera dinámica, casi instantánea, asíncronamente haciendo llamadas al servidor (backend con un API REST) y sobre todo sin refrescar la página en ningún momento.

**Versiones de Angular**

En angular tenemos dos versiones de angular. Angular JS y Angular que es la actual y la que engloba las versiones 2,3,4,5 ….

**¿Cómo nos ayuda Angular?**

Todos los que hemos escrito programas con JavaScript, sabemos lo tedioso que puede resultar a ser. En Angular se escribe el código en TypeScript que como veremos más adelante, nos ayuda mucho en la programación.

**TYPESCRIPT**

TypeScript, es un lenguaje de programación de código abierto desarrollado y mantenido por Microsoft. Es un super conjunto de JavaScript, que esencialmente añade tipado y clases.

Lo que hacemos con TypeScript es escribir el código de forma mas fiable y sencilla y luego ese código es traducido a JavaScript que es realmente el código que conoce el navegador.

Realmente no hace falta conocer muchas cosas nuevas de TypeScript. Hay que decir que todo el código JavaScript, es válido en TypeScript.

En los siguientes ejemplos usaremos características de JavaScript “morderno” ECMAScript 6 y typesrypt.

Comencemos con algunos :

Pongamos el siguiente código en un fichero index.html

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>Document</title>

</head>

<body>

    <script src="basico.js"></script>

</body>

</html>

Haremos una prueba rápida en TypeScript y la compilamos con tsc para generar el js.

*console*.log('Hola desde TypeScript');

*var* valor = 1;

if ( valor > 0 ){

*console*.log('El valor es: ', valor);

*var* valor = 10;

}

*console*.log('El valor termina siendo: ', valor);

/// Qué pasa con let y const. Si ponemos en el anterior lo mismo pero con let:

*let* valorConLet = 1;

if ( valorConLet > 0 ){

*let* valorConLet = 10;

*console*.log('El valorConLet es: ', valorConLet );

}

*console*.log('El let valorConLet termina siendo: ', valorConLet);

// Templates literales

*console*.log(`El valor de valorConLet es ${valorConLet}`);

// Funciones con parámetros opcionales.

saluda('Carlos');

saluda('Carlos', '¿Qué tal?');

*function* saluda( *persona*: *string*, *pregunta*?: *string*  ) {

    if ( !pregunta ){

*console*.log(`Hola ${persona}`)

    } else {

*console*.log(`Hola ${persona}, ${pregunta}`);

    }

}

// Funciones de flecha carácteríatica de ES6

*let* suma = *function* ( *a*: *number*, *b*:*number* ) {

    return a + b;

}

*let* suma2 = (*a*:*number* ,*b*: *number*) *=>* {

    return a + b;

}

*console*.log( 'La suma es: ', suma ( 333, 4));

// Promesas

*let* miPromesa = new Promise(  (*resolve*, *reject* ) *=>*  {

    setTimeout( ()*=>* {

        resolve();

    }, 1500);

});

miPromesa.then( ()*=>*{

*console*.log('La promesa ha terminado bien');

});

Para evitar tener que estar compilando, vamos a iniciar un proyecto TypeScript con tsc –init

Con tsc –w, ponemos a escuchar a TypeScript para que detecte los cambios y compile.

A continuación, vamos a presentar un código que habla por sí solo. Pretende ser una referencia rápida, para recordar llegado el momento como se utilizan o definen objetos, clases, funciones, etc.

Sobre tipos básicos de datos

<http://www.typescriptlang.org/docs/handbook/basic-types.html>

En la página de typescript, vienen 14 tipos básicos. En este ejemplo que pongo a continuación, vamos a mostrar los que se usan o por lo menos los normales.

Además, se incluyen formas de definir y utilizar funciones:

// Boolean, da igual poner Boolean o boolean

*let* esPrivado:*boolean* = true;

*console*.log( esPrivado );

*let* tieneESO = imparteEso();

*console*.log(tieneESO);

*function* imparteEso() {

    return true;

}

////////////////////////////////////////////////

// Tippos numéricos. Enteros y números decimales

////////////////////////////////////////////////

*let* colegios: *number* = 1000;

*let* alumnos = 3500;

*let* alumnas = 3600;

*let* profesores: *number*;

*console*.log( profesores );  // undefined

////////////////////////////////////////////////

// Tipo sting.

////////////////////////////////////////////////

*let* colegio: *string* = 'Infantes';

*let* profesor = 'Juan Carlos';

*let* alumno = `${profesor} trabaja en ${colegio}`;

*console*.log( alumno.toLocaleUpperCase() );

////////////////////////////////////////////////

// Tipo any

////////////////////////////////////////////////

*let* aula: *any*;

aula = 'Primaria';

*console*.log(aula);

aula = 45.36;

*console*.log(aula);

////////////////////////////////////////////////

// Multiples tipos

////////////////////////////////////////////////

*let* numero: *number* | *string*;

numero = 4;

numero = 'Cuatro';

////////////////////////////////////////////////

// Tipo arrays

////////////////////////////////////////////////

*let* primos:*number*[] = [1,2,3,5,7];

primos.push(11);

*let* ciudades = ['Albacete','Ciudad Real','Cuenca','Guadalajara','Toledo'];

ciudades.forEach( ( *ciudad* ) *=>* {

*console*.log(ciudad);

});

////////////////////////////////////////////////

// Funciones. Tipo Funcion

////////////////////////////////////////////////

*function* duplica( *x*: *number* ): *number* {

    return  x \* 2;

}

*let* miFuncion: (*a*: *number* ) *=>* *number*;

miFuncion = duplica;

*console*.log( 'Duplicando : ', miFuncion(4));

////////////////////////////////////////////////

// Funciones. Igual que en javascript . Se pueden definir tipo de parámetros

////////////////////////////////////////////////

*let* capital = 'Madrid';

*console*.log( dimeCapital(capital) );

*function* dimeCapital( *capital* ) {

    return capital;

}

// Parámetros

*function* nombreAlumno

    ( *nombre*: *string*, *apellido1*: *string* , *apellido2*:*string*, *matriculado* = true ):*string* {

    if ( matriculado ){

        return `${nombre} ${apellido1} ${apellido2}, está matriculado `;

    }

    else{

        return `${nombre} ${apellido1} ${apellido2}, no está matriculado `;

    }

}

*let* alumno1 = nombreAlumno('Pedro', 'López', 'Muñoz', false);

*let* alumno2 = nombreAlumno('Pedro', 'López', 'Muñoz');

*console*.log(alumno1);

*console*.log(alumno2);

*function* nombreAlumnoApellidoOpcional

    ( *nombre*: *string*, *apellido1*: *string* , *apellido2*?:*string*, *matriculado* = true ):*string* {

    if ( matriculado ){

        return `${nombre} ${apellido1} ${(apellido2 === undefined) ? '' : apellido2 }, está matriculado `;

    }

    else{

        return `${nombre} ${apellido1} ${(apellido2 === undefined) ? '' : apellido2 }, no está matriculado `;

    }

}

*let* alumno3 = nombreAlumnoApellidoOpcional('Juan', 'Pérez');

*console*.log(alumno3);

Objetos:

// Objetos

*let* coche = {

    marca: 'Citroen',

    potencia: 1500,

    color: 'Blanco'

};

// Objeto de un tipo específico

*let* automovil : {marca: *string*, potencia: *number*, color: *string* } = {

    marca: 'Mercedes',

    potencia: 1000,

    color: 'Blanco',

}

*console*.log(automovil);

// Métodos en objetos

*let* vehiculo1 : {marca: *string*, potencia: *number*, color: *string*, getVehiculo: () *=>* *string* } = {

    marca: 'Mercedes',

    potencia: 1000,

    color: 'Blanco',

    getVehiculo() {

        return this.marca + ' ' + this.color

    }

}

*console*.log( vehiculo1.getVehiculo() );

// otra manera, para si tengo que escribir 2 objetos del mismo tipo que vehiculo

// Definiendo tipos

*type* Vehiculo = {

    marca: *string*,

    potencia: *number*,

    color: *string*,

    getVehiculo: () *=>* *string*

};

*let* vehiculo2: Vehiculo = {

    color: 'Marrón',

    marca: 'Peugeot',

    potencia: 1500,

    getVehiculo() {

        return 'Este es el Peugeot Marrón';

    }

}

*console*.log( vehiculo2.marca + ' ' + vehiculo2.getVehiculo());

Clases

Lo primero, decir que las Clases ya vienen con ES6.

A continuación tienen un código de ejemplo, con definiciones de clases en TypeScript

*class* Mueble {

    nombre: *string* = '';

    color: *string* = '';

    disponible: *boolean* = false;

    peso: *number* = 0;

*constructor*() { }

}

*let* mesa: Mueble = new Mueble;

*console*.log( mesa );

Este ejemplo, contiene una clase Mueble con más opciones, para que la tengan de referencia.

*class* Mueble {

    // si no se pone public, private, etc, por defecto es público

    // protected, hace visible esa propiedad para la clase y sus hijas.

    // static se puede llamar sin instanciar la clase

    public    nombre: *string* = '';

    protected color: *string* = '';

    private   disponible: *boolean* = false;

    private   peso: *number* = 0;

    private   \_procedencia: *string* = 'Perú';

*constructor*(*nombre*: *string*, *color*: *string*, private *fabricante*: *string* = 'Ikea') {

            this.nombre = nombre;

            // Creando una propiedad en el constructor

            this.fabricante = fabricante;

            this.color = color;

            this.disponible = true;

            this.peso = 45;

        }

*set* procedencia(*procedencia*: *string*) {

            this.\_procedencia = procedencia;

        }

*get* procedencia():*string* {

            return this.\_procedencia;

        }

        public nombreMueble(): *string* {

            return this.nombre + ' de color: ' + this.color + ' fabricado por : ' + this.fabricante;

        }

    }

*let* mesa: Mueble = new Mueble('Mesa', 'Amarilla');

*console*.log( mesa );

mesa.nombre = 'Sofá';

*console*.log( mesa.nombreMueble() );

*let* armario: Mueble = new Mueble('Armario', 'Blanco');

*console*.log(armario.nombreMueble());

*let* silla: Mueble = new Mueble('Silla', 'Azul', 'El Corte Inglés');

*console*.log( silla.nombreMueble());

*console*.log( 'Procedencia original: ' + silla.procedencia );

silla.procedencia = 'EEUU';

*console*.log( 'Procedencia definitiva: ' + silla.procedencia );

Otra forma de declarar clases en la que las propiedades van en el constructor.

*class* Coche {

*constructor*(public *marca*: *string*, public *color*: *string* ){}

    getCoche = () *=>* `Tengo un ${this.marca} de color ${this.color}`;

}

*let* miCoche: Coche = new Coche('volvo', 'rojo');

console.log(`Mi coche es un ${miCoche.marca} y es de color ${miCoche.color}`);

console.log(miCoche.getCoche());

Interfaces en TypeScript. Mostramos un ejemplo con varias posibilidades:

*interface* Recipiente {

    nombre: *string*,

    capacidad: *number*,

    material: *string*[],

    color?:*string*,

    fechaFabricacion: Date,

    // Incluso puede tener métodos en la interfaz.

    disminuirCapacidad( *cantidad*: *number* ): *number*

}

*let* botella: Recipiente = {

    nombre : 'Botella',

    capacidad: 150.36,

    material: ['Cristal', 'Madera']  ,

    fechaFabricacion : new Date(),

    disminuirCapacidad( *cant*: *number*  ): *number* {

        return this.capacidad ;

    }

}

*console*.log( botella );

*console*.log( 'getTime: '+ botella.fechaFabricacion.getTime());

*console*.log( 'Disminuimos capacidad de la botella: ', botella.disminuirCapacidad(5));

*console*.log( botella );

Archivo tsconfig.json.

Para generarlo, ponemos tsc –init

¿Qué es lo que tiene y para qué sirve?

Sirve para decirle al compilador de TypeScript, cómo tiene que compilar.

Todas las opciones están en <http://www.typescriptlang.org/docs/handbook/compiler-options.html>

Estas opciones, nos sirven por ejemplo para que no se generen los comentarios en el js, para preparar para la depuración, y muchas otras cosas.

Por ejemplo, vamos a ver este trozo de código ts y vamos a depurarlo.

*let* escrito = 'En un lugar de la mancha de cuyo nombre no quiero acordarme';

*let* escritoArray = escrito.split(' ');

escritoArray.forEach( *palabra* *=>* {

*console*.log( palabra );

});

*console*.log(escritoArray);

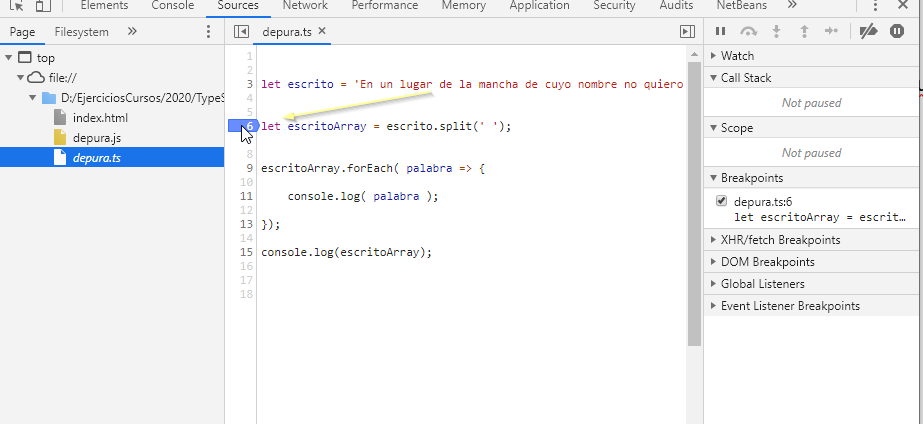
En el fichero tsconfig.json, vamos a poner la propiedad

*"sourceMap"*: true,                     /\* Generates c

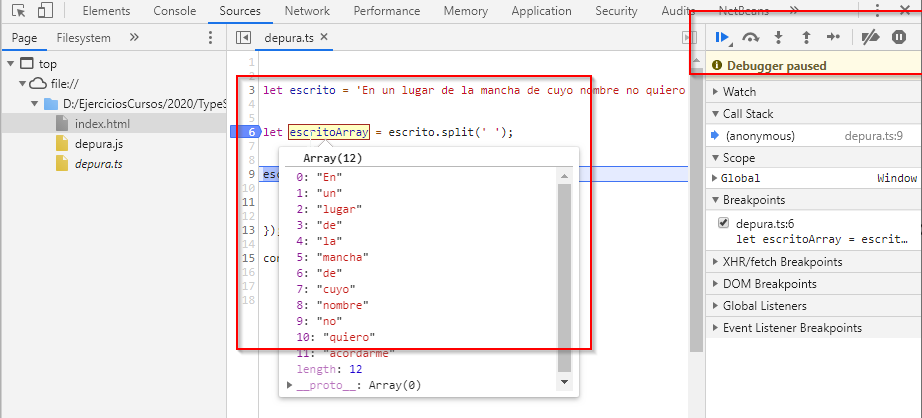
orresponding '.map' file. \*/

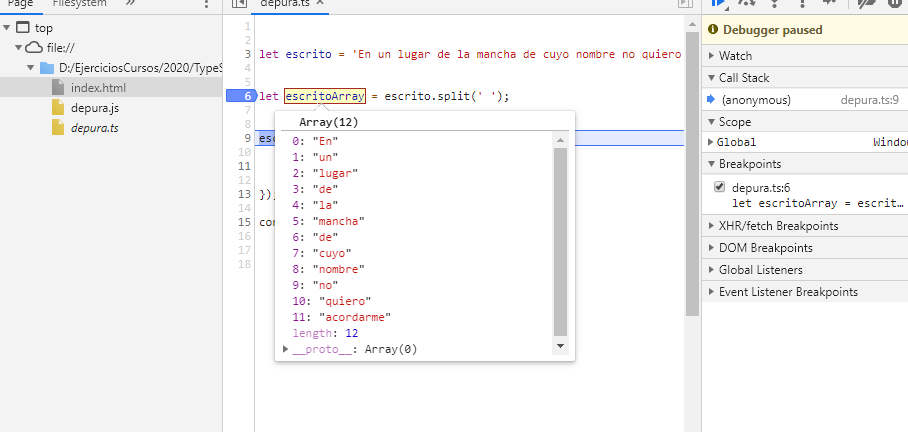
Esto, genera ahora un fichero .map.

Una vez recargado el navegador, en la pestaña de Source, veremos que ha generado un fichero depura.ts y podemos poner un punto de debug.



Ahora, podemos recargar y dejando el ratón encima de una variable podemos ir viendo lo que contiene e ir pasando a otra instrucción, etc.





**COMENCEMOS CON ANGULAR.**

**Tenemos que diferenciar entre Angular JS y Angular.**

Anguar JS, es la versión anterior a Angular y no tiene nada que ver con lo que vemos de Angular 2, 3, etc.

A partir de la versión 2 de angular, todo es Angular.

**Revisemos la página oficial de angular** [**https://angular.io/**](https://angular.io/)

En esta página, cuando buscamos alguna cosa, hay que fijarse que tienen un dibujito y es bueno conocer ese significado.

pk : módulo

i interfax

K constante

f función

e enumeración

p pipes

Además, cuando entramos en la documentación de algún elemento, nos informa entre otras cosas la versión de angular desde que está disponible.

**Aplicaciones de referencia:**

Esta es la aplicación que vamos a hacer todos juntos:

<https://cursoangularjccm.firebaseapp.com/#/home>

Se puede usar sin problemas, sin restricciones. Está para aprender.

**Estructura de una aplicación con angular.**

Vamos a construir nuestra primera aplicación con Angular.

En nuestra carpeta, por ejemplo c:\angualr, vamos a crear nuestra primera aplicación, utilizando el cli de angular.

ng new estructura

Esperamos a que termine

Nos va a preguntar si queremos angular routing y le diremos que sí.

También nos va a preguntar por el formato de hoja de estilos que queremos utilizar. En este caso, seleccionaremos css.

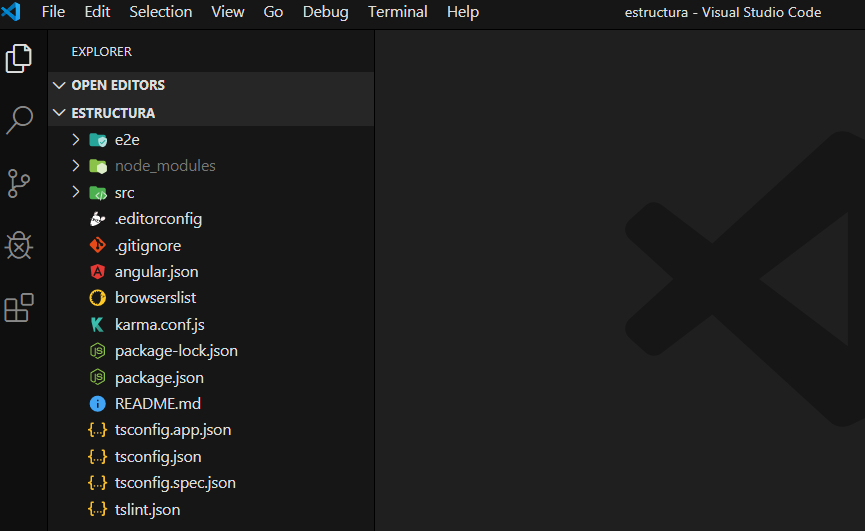
Una vez “aceptado”, se pondrá a realizar esos trabajos de construcción de nuestra primera aplicación angular.

Cuando termine, vamos a abrir nuestro proyecto de angular en el editor Visual Studio Code.

Para ello podemos escribir dentro de la carpeta c:\angular\estructura\ code .

También, podemos optar por abrir el editor Visual Studio Code y abrir la carpeta estructura.

De cualquier manera, tenemos que tener en el Visual Studio Code el proyecto de esta forma:



Aunque esto no es un curso de Visual Studio Code, vamos a comentar brevemente algunas de las opciones que nos ofrece este editor:

En la parte superior, tenemos el menú habitual en un programa de este tipo.

En la parte izquierda, tenemos 5 accesos, que son: Vista de los archivos, Buscar, Git, Debug y mantenimiento de extensiones.

Sobre las extensiones de Visual Code, es muy conveniente tener instaladas algunas que nos harán más fácil el desarrollo con angular.

Yo tengo instaladas estas extensiones:

* Activitus Bar
* Angular 10 Snippets
* Angular 2 TypeSCript
* Angular Language
* BracketP Pair Colorizer 2
* JavaScript ES6
* Material Icon Theme
* Monokai Nignt Theme
* Parse JSON as Code
* Terminal
* TypeSCript Importer

Sobre el mantenimiento de extensiones, habría que instalar las extensiones más recomendadas de typescript, angular, css, html, bootstrap, terminal, etc.

**¿Qué son cada una de las carpetas y ficheros que tenemos en nuestra primera aplicación de angular?**

“e2e”: “end to end”. Para pruebas de integración. No son pruebas unitarias que van por componente.

“node\_modules”. Módulos de node. Librerías. No se guardan en el repositorio. Para volver a generarlos, npm install.

“src”. Es la carpeta source. Contiene el código de la aplicación.

“.gitignore”. Es el fichero de configuración de git, donde especificamos los archivos de los cuales no hay que hacer seguimiento.

“angular.json”. Fichero de configuración. JavaScript, puerto del servidor, estilos…

Para el caso de querer levantar angular en otro puerto, sería dentro de architect -> serve -> options -> port

"options": {

"port": 8100,

"browserTarget": "keycloak4:build"

},

“package.json”: Fichero de configuración de dependencias.

Como podemos observar, estos ficheros están en formato **JSON.**

**Los ficheros JSON (javascript object notation) ,** son fichero con un formato sencillo para el intercambio de información.

Packag.json lock. Es como un historial de como se ha generado el package .json

Dependencias, dependencias de desarrollo.

La carpeta .git. Es el repositorio local. Mas adelante, haremos un par de pruebas para no perder el código que vamos haciendo. Veremos algunos de los comandos principales de git ( git init, git status, git add ., git commit, git checkout -- ) .

Es el momento de probar nuestra aplicación.

ng serve –o

ng serve –ssl –o ( para hacerlo por https)

Haremos algunos cambios en la aplicación para ir explicando el funcionamiento de angular.

<h1>Hola Mundo</h1>

<h3>Bienvenido al curso de {{ curso }}</h3>

Las llaves se llaman interpolación de string. Se pueden poner cualquier expresión que pueda procesar angular. Básicamente cualquier expresión javascript.

En el ts.

curso = 'Angular';

Creamos un componente y vemos que es eso de los componentes.

El selector, el template, templateURL, el css y el cambio que hace en el app.module.

Explicaremos en esta parte el componente principal de la aplicación.

Hablaremos de la carpeta assets y environments.

Hablaremos del módulo principal y del archivo de rutas.

**AUTOEVALUACIÓN 1.**

1. **¿Qué es TypeScript?**
   1. Es un lenguaje de marcas.
   2. Es una versión moderna de javascript
   3. Es un lenguaje libre y de código abierto de Microsoft construido sobre JavaScript.
2. **¿Para qué sirve la carpeta assest en un proyecto de angular?**
   1. Normalmente para guardar los recursos estáticos de la aplicación.
   2. Para guardar la configuración de la aplicación.
   3. Se guardan las rutas de la aplicación.
3. **¿Cómo se crea una aplicación angular desde la terminal?**
   1. ng serve
   2. git init
   3. ng new
4. **¿Entiende el navegador el código escrito en TypeScript?**
   1. No
   2. Si
   3. Únicamente en los navegadores modernos.
5. **¿Cuál es el primer componente que se ejecuta al lanzar una aplicación de angular por defecto?**
   1. App-home
   2. App-navbar
   3. App-Component

**Comencemos con nuestra aplicación.**

**En esta sección, trataremos los componentes, servicios y pipes. También comenzaremos a montar la configuración de las rutas de la aplicación.**

Recuerdo que en la URL <https://github.com/jucarlos/apuntesCursoAngular>, estarán disponibles todos los recursos de ayuda para seguir este curso.

**CREANDO LA APLICACION**

ng new ejercicio

Cuando ejecutamos la creación de la aplicación nos hace unas preguntas.

En la versión actual, nos pregunta si vamos a utilizar angular router, si utilizamos para la hoja de estilos css y esas son las mas importantes.

Una vez creada la aplicación, nos metemos en esa carpeta que nos crea y vamos a levantar el servidor que trae angular. Además, ponemos la opción –o, para que abra directamente el navegador.

ng serve -o

**CAMBIANDO LA APLICACIÓN POR HOLA MUNDO Y BOOTSTRAP**

En app.html

<h1>HOLA MUNDO</h1>

bajamos bootstrap

bajamos https://code.jquery.com/jquery-3.4.1.slim.min.js

bajamos https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js

creamos en assets las carpetas js y css

Una manera de ponerlo es con cdn. Vamos a utilizar angular.json

"styles": [

"src/assets/css/bootstrap.min.css",

"src/styles.css"

],

"scripts": [

"src/assets/js/jquery-3.4.1.slim.min.js",

"src/assets/js/popper.min.js",

"src/assets/js/bootstrap.min.js",

]

En la versión actual de bootstrap, no viene la instalación de jquery. ( Enero de 2021 )

Podemos probar si funciona bien lo de bootstrap poniendo un navbar u otra cosa y luego lo quitamos.

**Estructura de nuestra aplicación:**

CARPETAS DE LA APLICACIÓN

* Dentro de app, creamos:
  + Shared
  + Data
  + Guards
  + Home
  + Interfaces
  + Models
  + Pages
  + Pipes
  + Services

Es una estructura más o menos formal para una aplicación.

**Los componentes de angular**

Los componentes de angular, son pequeñas clases que cumplen con una tarea, por ejemplo sidebar, pie de la aplicación, etc.

Realmente, son clases normales con un con un decorador.

Creando el primer componente. Miramos los cambios que hace en el app.module.

Nuestro primer componente será el **navbar**.

ng g c shared/navbar --skipTests=true --dry-run

Vamos a coger un navbar de bootstrap y lo vamos a poner en el html de nuestra aplicación.

Debajo del navbar, irá el home y todas las páginas de nuestra spa.

Cuando generamos un componente de la forma en la que hemos generado el componente navbar, nos creará tres archivos:

Navbar.component.ts -> que es la clase typescript con un decorador de clase llamado Component y que tiene las directivas selector, templateURL y styleURLs

Navbar.component.html -> Es el html de ese componente.

Navbar.component.css -> Son la hoja de estilos de ese componente

**El componente HOME**

Creamos el componente con el cli. Lo pondremos en la carpeta pages

ng g c pages/home --skipTests=true --dry-run

En el home, pondremos un jumbotron y debajo iremos poniendo información de la aplicación.

**El componente directivas**

Las directivas estructurales se usan en Angular para pintar el html.

Lo haremos directamente creando la página para comprobar las directivas.

ng g c pages/directivas --skipTests=true --dry-run

Ahora que tenemos el componente de las directivas, vamos a terminar el archivo de rutas.

**Sistema de rutas en angulas**

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

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

import { HomeComponent } from './pages/home/home.component';

import { DirectivasComponent } from './pages/directivas/directivas.component';

*const* routes: Routes = [

  { path: 'home', component: HomeComponent },

  { path: 'directivas', component: DirectivasComponent },

  { path: '\*\*', redirectTo: 'home'}

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export *class* AppRoutingModule { }

En algunos casos, será necesaria la utilización de hash en las rutas:

  imports: [RouterModule.forRoot(routes, { useHash: true})],

Para que navegue entre páginas, hay que poner la directiva router-outlet en el app.component.html al igual que pondremos la etiqueta del componente navbar.

<app-navbar></app-navbar>

<router-outlet></router-outlet>

Además de esto, y para que desde el navbar se pueda navegar al home, ponemos:

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

          <a class="nav-link" [routerLink]="['home']">Home <span class="sr-only">(current)</span></a>

        </li>

También se puede poner routerLink = “/directivas”.

Ahora es el momento de terminar la página con las directivas estructurales.

En el ts del componente directivas, crearemos una propiedades con datos para usarlo en los ejemplos que haremos en el curso.

Lo bueno que tiene, es que en los casos de mostrar y ocultar componentes html, esos elementos no están en el html. Podemos comprobarlo inspeccionando esos elementos.

Las directivas que vamos a ver de momento son \*ngFor y \*ngIf

\*ngIf, para mostrar u ocultar un elemento y \*ngFor, para hacer repeticiones.

En el ts tendremos una serie de propiedades que con las haremos estos ejemplos.

  <div \*ngIf="mostrar" class="card mb-3">

                        <div class="card-body">

                          <h5 class="card-title">{{ titulo }}</h5>

                          <p class="card-text">{{ colegio }}</p>

                        </div>

                    </div>

 <div class="col-md-6">

                    <h3>ngFor</h3>

                    <hr/>

                    <ul class="list-group">

                        <li \*ngFor="*let* colegio of colegios; *let* i= index" class="list-group-item">

                            {{ i + ' - '+ colegio }}

                        </li>

                     </ul>

                </div>

**Creamos el componente Temario**

Con este componente, vamos a estudiar nuestro primer servicio.

Vamos a tener en la carpeta data, un fichero con datos en formato json.

Crearemos nuestro **primer servicio**, que leerá esos datos y los pintará en la pantalla.

**Un servicio tiene estas funciones básicamente**

* Mantener la data
* Realizar peticiones CRUD
* Es un recurso reutilizable para la aplicación.

ng g s services/temario --dry-run

En esta primera versión del componente temario, vamos a poner un for, para pintar el html.

Más tarde, volveremos a este componente para practicar con el Input() y OutPut().

El fichero JSON que vamos a utilizar, está en la tarea 107362.

En segundo lugar, inyectaremos el servicio en el componente del temario, para solicitar al servicio que nos devuelva de forma síncrona o asíncrona los datos del temario.

Vamos a hacer el ejemplo todos juntos.

Vamos a incorporar el anímate.css, pero en las versiones nuevas, hay un problemilla que hace que se tenga que comentar la parte esta para que funcione correctamente.

}

/\* @media print, (prefers-reduced-motion: reduce) {

  .animate\_\_animated {

    -webkit-animation-duration: 1ms !important;

    animation-duration: 1ms !important;

    -webkit-transition-duration: 1ms !important;

    transition-duration: 1ms !important;

    -webkit-animation-iteration-count: 1 !important;

    animation-iteration-count: 1 !important;

  } \*/

  .animate\_\_animated[class\*='Out'] {

*opacity*: 0;

  }

/\* Attention seekers  \*/

La clase que vamos a incorporar es:

<div class="animate\_\_animated animate\_\_fadeIn animate\_\_fast">

**Es el momento de cambiar un poco el html de este componente y hacer un componente reutilizable.**

Se trata de hacer un componente tarjeta que reciba los datos que queremos que pinte y que además pueda enviar al padre alguna información.

Imaginemos que vamos a tener una carpeta components que podrían ser reutilizables en toda la aplicación. Para este caso, vamos a tener en esa carpeta un componente llamado tarjeta. Ese componente podría recibir datos de la clase Tema para hacer la presentación de forma centralizada.

Lo haremos juntos y el código quedará de esta forma.

También en este punto y para poner un “Me gusta”, vamos a instalar <https://fontawesome.com/>

Para instalar fontawesome lo haremos con el CDN. Nos pide un correo, pero ya dejo en la tarea estos pasos para la instalación.

Se descomprime la librería en assets y en angular.json, se incluye en el array de styles la entrada: "src/assets/fontawesome/css/all.min.css" y en el array de scripts, se incluye: "src/assets/fontawesome/js/all.min.js"

Aunque más abajo está todo el código, lo haremos paso a paso.

Componente padre.

Html:

<div class="animate\_\_animated animate\_\_fadeIn animate\_\_fast">

  <div class="row mt-3">

    <div class="card border-primary" style="width: 100%">

      <div class="card-body">

        <h6 class="card-subtitle mb-2 text-muted">

          Hay {{ temario.length }} temas

        </h6>

        <p class="card-text text-primary">Me gustan: {{ meGustan }}</p>

      </div>

    </div>

  </div>

  <div class="row mt-3">

    <div \*ngFor="let tema of temario; let i = index" class="col-md-4">

      <app-tarjeta [tema]="tema" [numero]="i+1"

      (cambiaMeGusta)="sumaUnLike($event)"></app-tarjeta>

    </div>

  </div>

</div>

Ts:

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

import { Tema } from '../../models/tema';

import { TemarioService } from '../../services/temario.service';

@Component({

  selector: 'app-temario',

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

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

})

export *class* TemarioComponent implements *OnInit* {

  temario: Tema[] = [];

  meGustan = 0;

*constructor*(public *temaService*: TemarioService) { }

  ngOnInit(): *void* {

    // this.temario = TEMARIO;

    // this.temario = this.temaService.getTemario();

    // Nos suscribimos

    this.temaService.getTemarioAsync()

    .subscribe( *data* *=>* {

      this.temario = data;

    });

  }

  sumaUnLike( *evento* ): *void* {

    this.meGustan = this.meGustan + 1;

  }

}

Componente hijo.

Html:

<div class="card border-primary mb-3">

    <div class="card-header">Tema: {{ numero }}</div>

    <div class="card-body text-primary">

      <h5 class="card-title">{{tema.titulo}}</h5>

      <p class="card-text">{{tema.contenido}}</p>

      <button (click)="meGusta()"

      class="btn btn-primary"><i class="far fa-thumbs-up"></i></button>

    </div>

  </div>

Ts:

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

import { Tema } from '../../models/tema';

@Component({

  selector: 'app-tarjeta',

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

  styles: [

  ]

})

export *class* TarjetaComponent implements *OnInit* {

  @Input() tema: Tema;

  @Input() numero: *number*;

  @Output() cambiaMeGusta: EventEmitter<*number*> = new EventEmitter();

*constructor*() { }

  ngOnInit(): *void* {

  }

  meGusta(): *void* {

    this.cambiaMeGusta.emit(1);

  }

}

**Los Pipes**

Los pipes, sirven para modificar la forma en que se presentan los datos en el html. No modifican los datos.

En este ejemplo vamos a ver los pipes más frecuentes que hay en angular y además vamos a construir uno personalizado.

Comenzamos construyendo la página de los pipes:

ng g c page/pipes

Pipes que vamos a utilizar:

UpperCase, Slice, Decimal, Json, UpperCase, Fecha y alguno mas.

Para la creación de un pipe personalizado, creamos con el cli, el pipe:

ng g p pipes/alreves --skipTests=true --dry-run

Haremos el código todos juntos.

Hasta este momento, ya hemos hecho bastantes cosas y es un buen momento para guardar el código por si tenemos algún problema.

**Ciclo de vida de los componentes de angular**

https://angular.io/guide/lifecycle-hooks

Siempre tenemos por defecto en todos los componentes que creamos el OnInit.

Os invito a probar los demás. Uno muy usaro es OnDestroy.

**Git**

git init. Esto no hace falta, ya que cuando se crea el proyecto angular, se inicializa git.

Vemos como está el proyecto con git

Git status

Vemos que está todo en rojo. Lo subimos al escenario ( stage ). Subimos todo

git add .

Ahora, si ponemos git status, veremos que ya está en verde. Ya está en el escenario.

Es el momento de guardarlo en el repositorio local con

git commit –m “Primer commit”

Podemos ver con git log, el historial de los cambios

En mi caso, no me gusta el comentario que he hecho en el commit y voy a proceder a cambiarlo con

git commit –amend

Poniendo esto, se abre como un editor, donde podemos modificar el comentario que hemos puesto y poner algo mas significativo.

Para guardar y salir, es esc + w ( write ) y q ( salir )

( para cambiar el editor, se puede hacer con git config --global core.editor notepad)

Ahora si ponemos git log, veremos que se ha cambiado.

Vamos a probar a borrar algo o destrozar un trabajo, después basta con poner

Git checkout -- .

Mas adelante subiremos todo esto a un repositorio remoto. Por ejemplo github.

**AUTOEVALUACIÓN 2**

1. **¿Qué archivo contiene la referencia a todos los componentes de la aplicación?**
   1. El fichero app.module
   2. El fichero angular.json
   3. No hace falta
2. **¿Qué directiva es necesaria para repetir datos en una vista?**
   1. \*Ngif
   2. \*ngfor
   3. Ninguna de las dos
3. **¿Cómo se crea un componente con el angular cli?**
   1. ng g c *nombrecomponente*
   2. ng generate component *nombrecomponente*
   3. Las dos anteriores son ciertas
4. **¿Qué directiva hay que poner en el html para que se redibujen en ese sitio los componentes de una SPA?**
   1. App.router
   2. App-component
   3. Router-outlet
5. **¿Cómo se inyecta un servicio en un componente?**
   1. No se puede.
   2. Se le llama en el constructor de la clase del componente
   3. Basta con importarlo en el componente que se va a utilizar.

**Formularios**

**Formularios**

Hay dos formas de trabajar con formularios en angular, una es haciendo prácticamente todo el trabajo en el html ( template ) y otra, haciendo casi todo el trabajo en la parte del TypeScript ( data)

Creamos primero los componentes que vamos a utilizar, y los enlazamos en el menú.

ng g c pages/formularios/formularioHtml --flat -is --skipTests=true --dry-run

ng g c pages/formularios/formularioTs --flat -is --skipTests=true --dry-run

Es importante decir, que para trabajar con formularios tienen que tener en el app.module importado lo siguiente:

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

ReactiveFormsModule es para que funcionen los formularios del lado del TS.

Una vez que se importa el FormsModule, ya deja de hacer el refresh en los submit.

**Vamos a comenzar con este código html**

<div class="animate\_\_animated animate\_\_fadeIn animate\_\_fast">

    <div class="row mt-3">

    <div class="card">

        <h3 class="card-header">Formulario en el Html</h3>

        <div class="card-body">

            <form autocomplete="off">

                <div class="mb-3">

                    <label for="inputNombre">Nombre</label>

                    <input type="text" class="form-control" id="inputNombre">

                    <small class="form-text text-muted">We'll never share your email with anyone else.</small>

                  </div>

                <div class="mb-3">

                  <label for="imputEmail">Email</label>

                  <input type="email" class="form-control" id="imputEmail" >

                  <small class="form-text text-muted">We'll never share your email with anyone else.</small>

                </div>

                <div class="mb-3">

                  <label for="exampleInputPassword1">Password</label>

                  <input type="password" class="form-control" id="exampleInputPassword1">

                </div>

                <div class="mb-3 form-check">

                  <input type="checkbox" class="form-check-input" id="recuerdame">

                  <label class="form-check-label" for="recuerdame">Recuerdame</label>

                </div>

                <button type="submit" class="btn btn-primary">Validar</button>

              </form>

        </div>

    </div>

</div>

</div>

**Sobre los formularios**

Inspeccionamos el elemento de un input tal cual y luego pondremos en el elemento nombre el ngModel y solucionaremos el error (poner el name ) que nos muestra y volveremos a inspeccionar el elemento para comprobar los cambios realizados.

En el inspector de elementos, tiene muchas clases. Importantes: untouched ( el usuario no ha tocado ), pristine ( valor original ) , valid ( que está válido ) . Si se hace algo, pues ya tiene dirty, el untouched cuando sales a otro campo pues lo cambia. Mejor lo vemos con ejemplos

Angular, al ver que hay un form en el html, creo todo lo referente a los formularios.

Para tener la información del formulario pondremos en el form el (ngSubmit) con información de todo el formulario. Eso se hace con una referencia local al formulario. En nuestro caso, en la etiqueta form ponemos: #formulario=”ngForm” . ngForm es un tipo de angular.

Algunas validaciones para campos:

* required .
* Para el correo: pattern="[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,3}$"
* minLength="3" en el nombre.

Los navegadores, para que validen o no validen si ponemos novalidate="" en el form, pues no valida el html.

Para establecer valores por defecto crearemos en el ts las propiedades que se verán en el html.

El binding se hace a través del ngModel. Si se pone [ngModel]=dato, nos indica que está recibiendo lo que está en la propiedad dato.

Si se pone [(ngModel)]=dato, hacemos que la información fluya en los dos sentidos; recibe y emite. No se recomienda esta forma. Es más seguro ver luego los datos que se envían desde el formulario.

**Validaciones en un campo de texto.**

Usaremos los estilos del bootstrap para marcar cuando los campos son inválidos.

Hay que poner una referencia local al campo en concreto.

Para ello, utilizaremos la clase “is-invalid”. Lo pondremos cambiando la clase del bootstrap dinámicamente.

Hay dos formas de hacerlo:

[class.is-invalid]="true".

[ngClass]="{'is-invalid': true}"

En lugar de true, hay que poner una expresión javascript que devuelva true o false

Lo hacemos juntos.

**En el campo del correo,** emplearemos una validación basada en una expresión regular:

"[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,3}$"

Hacemos juntos la validación del correo de forma parecida a la que hemos hecho con el nombre pero añadiendo la validación de la expresión regular en la propiedad pattern.

Hay una cuestión cuando están ya puestas todas las validaciones.

Si pulsamos el botón validar, no aparecen las validaciones.

La forma más correcta de hacerlo, es poner en el método de guardar, haríamos esto:

Object.values( forma.contros ).forEach….

**Datos por defecto.**

en el ts, creamos las variables:

Persona = {

nombre: "Juan carlos",

email: "okjuaancarlos@gmail.com"

}

Para saber en el html, si hay problemas en los campos:

en controls.apellidos(u otros).errors, está el motivo por el que no valida.

Si que se puede hacer una referencia local de cada elemento del formulario al form-control que crea angular.

por ejemplo

#nombre="ngModel"

y entonces, usar:

ngIf="nombre.errors?.required ( ? por si no existe )

Los de minLength, nombre.errors?.minlength y además, se pueden interpolar los datos para informar de los tamaños máximos y mínimos.

Aplicando clases para las validaciones de bootstrap.

<form novalidate="" #f="ngForm" (ngSubmit)="validar(f)">

                <div class="form-group">

                    <label for="inputNombre">Nombre</label>

                    <input type="text"

                            [ngClass]="

                            { 'is-invalid' : nombre.invalid && nombre.touched,

                              'is-valid' : nombre.valid }"

                            class="form-control" id="inputNombre"

                            name="nombre"

                            [ngModel]="persona.nombre"

                            required

                            minlength="3"

                            #nombre="ngModel">

                    <div class="invalid-feedback" \*ngIf="nombre.errors?.required">

                      El nombre es requerido

                    </div>

                    <div class="invalid-feedback" \*ngIf="nombre.errors?.minlength">

                      El nombre tiene que tener al menos 3 caracteres

                    </div>

                </div>

[ngClass]="{'has-danger': nombre.errors?.required || nombre.errors?.minlength ( incluso son && nombre.touched}

El botón, debería ser disabled si el formulario es inválido.

[disabled]="!f.valid"

Los demás ejemplos y comentarios en el código.

**FORMULARIOS REACTIVOS.**

Para utilizar los formularios en tl, primero tenemos que importar en app.module el módulo ReactiveFormsModule de anguar/forms

En este tipo de formularios se hará casi todo el trabajo en la clase del componente.

También se pueden poner validaciones en la parte del html, pero se recomienda hacer todo en la clase.

En primer lugar, hay que construir el formulario antes de que se construya el HTML, por eso el lugar indicado es en el constructor de la clase. Nos vamos a ayudar de FormBuilder que nos proporciona angular para facilitar la construcción.

Veremos la agrupación de campos, validaciones síncronas y asíncronas, validaciones personalizadas y algunas cosas más.

Veamos el ejemplo todos juntos.

**En esta sección haremos peticiones a un servicio REST.**

**El objetivo será crear una página de mantenimiento de tipos de vehículos.**

**En esta sección también protegeremos la ruta para que únicamente los usuarios autenticados puedan entrar y por lo tanto mantener el tipo de vehículos.**

Lo que vamos a hacer está en:

<https://cursoangularjccm.firebaseapp.com/>

en el menú Vehículos -> Tipos.

Para entrar es necesario estar autenticado. Podemos utilizar las credenciales:

Usuario: carlos@gmail.com

Clave: 123456

Como es necesario estar autenticado, por lo tanto, antes de nada haremos todo lo referente a los usuarios de la aplicación.

Primero, haremos la **pantalla de registro**, para lo cual, utilizaremos este “end-point”: POST

<https://backend-marco.herokuapp.com/usuarios>

en la que hay que enviar en el body ( x-www-form-urlencoded) los parámetros nombre, email y password y nos devolverá un objeto con un error o con el ok.

Después, **haremos la pantalla de login.**

Utilizaremos este “end-point”, para dar hacer el login

https://backend-marco.herokuapp.com/login

Es una petición POST

En el body body ( x-www-form-urlencoded), hay que enviar las propiedades email y password.

Luego, el login nos devolverá un token que será el método para llamar a los end-point que están protegidos mediante un token. Usaremos un token jwt.

**Estructura de la pantalla de registro y login**

El componente del registro, así como lo referente a login y lo que se necesite de usuarios, lo vamos a crear dentro de pages, en una carpeta que se llame usuarios.

Creamos el componente registro y hacemos la entrada en el fichero de rutas para ver si nos funciona correctamente.

Ng g c pages/usuarios/registro --skipTests=true

Cuando tengamos claro que funciona, haremos el formulario de registro con los campos nombre, email y password.

Ya de paso vamos a aprovechar para hacer comentarios sobre los formularios en angular, aunque veremos un apartado un poco más detallado.

Vamos a crear nuestro **servicio de usuarios**, donde haremos todo lo necesario para el control de los mismos.

Ng g s services/usuario

Esta es la primera vez que vamos a realizar peticiones http. Por lo tanto hay que importar en el app.module el módulo HttpClientModule. En el servicio, hay que importar el HttpClient.

Se puede ver el código terminado. En la clase, lo haremos todos juntos.

Para hacer la aplicación un poco más amigable, **instalaremos Sweetalert 2.** Es un sistema de alertas bastante chulo.

<https://sweetalert2.github.io/>

Lo instalamos según pone en las instrucciones y terminamos el componente de registro todos juntos.

Para el modal, haremos algo diferente. Un Modal.

**Un modal para el LOGIN**

Haremos un modal de bootstrap de forma que cuando se pulse en entrar en el navbar nos abra el modal y nos pida las credenciales.

Para ello centralizaremos todo lo necesario en un servicio llamado usuarioService.

Tienen todo el código que haremos en el repositorio.

ng g c components/modal --skipTests=true -is --flat --dry-run

**Guardando los cambios en Git.**

* Vemos que tenemos pendiente git status.
* Subimos todo al state git add .
* Hacemos commit poniendo un comentario git commit –m “Formularios y validación de usuarios”
* Hacemos un push a nuestro repositorio de git en internet.

En mi caso, lo he creado y al crear el proyecto, nos dice lo que tenemos que hacer.

Nos dirá el git remote, que hay que ponerlo una vez y el git push. Si es la primera vez que se hace desde nuestro ordenador, nos pedirá nuestras credenciales de git.

* Por fin, lo subimos al repositorio. git push –u origin master

**Mantenimiento de Tipos de vehículos, módulos independientes y guards**

En este punto, podemos observar que el fichero app.module, está creciendo mucho.

Eso es así ya que según vamos creando módulos, pipes y otros componentes de la aplicación pues hay que registrarlos en ese fichero.

Pero como buena práctica y es lo que haremos a continuación, habría que crear módulos independientes por tipo de componentes o funcionabilidades de la aplicación.

Aunque más adelante en otro ejercicio, haremos un ejemplo con 3 módulos independientes, esto es un avance.

Utilizando el angular cli, vamos a crear el módulo llamado vehiculos, donde pondremos luego todos los componentes que se refieren a la gestión de vehiculos.

Crearemos el módulo

ng g m pages/vehiculos --dry-run

Seguidamente, crearemos los componentes para hacer el listado de los tipos de vehículos y la pantalla de edición de un tipo de vehículo.

ng g c pages/vehiculos/tipoVehiculo

ng g c pages/vehiculos/tiposVehiculos

Podemos observar que ahora cuando se crean estos componentes, se actualizan en el módulo de vehículos y solamente este módulo es el que se incorpora en al app.module. (aunque parece que no hace falta. No sé si son por alguna actualización de angular )

**Pantalla con la lista de tipos de vehículos**

Si probamos en postman la url

<https://backend-marco.herokuapp.com/tipovehiculo>

Veremos que nos dice que tenemos que estar autenticados o que hay que mandar un token.

Ese token, nos dice la gente del backend, hay que enviarlo en la cabecera ( headers) con el nombre token y el valor del token, por ejemplo lo tenemos en el storage.

Si probamos ahora, veremos que ya tenemos la lista de vehículo.

Por eso para entrar en esta página hay que estar autenticado y eso, lo haremos con nuestro primer **Guards**

Creamos nuestro primer guards con el angular cli:

Ng g g guards/auth

Cuando lo creamos, nos pregunta por la interfaz que tiene que implementar. Nosotros para nuestro ejercicio, usaremos CanActivate, que viene siendo que si puede o no activar una ruta.

El método canactivate(), tienen las propiedades next y state, con sus tipos.

El tipo  ActivatedRouteSnapshot, como pone en la documentación tiene información sobre la ruta a la que se protege.

El tipo RouterStateSnapshot, Representa el estado del enrutador en un momento determinado.

Luego, viene el tipo de dato que puede devolver.

Nosotros vamos a quitar todos y poner un boolean para que nos diga si el usuario está autenticado o no. Para ello, necesitaremos inyectar el servicio de usuarios.

En el fichero de rutas, donde se hace referencia a la ruta de tipos de vehículos hay que poner que sea controlada por el guard.

{ path: 'tiposvehiculos', component: TiposVehiculosComponent, canActivate: [AuthGuard] },

Se ponen los guards entre corchetes porque se pueden poner varios guards.

Lo hacemos juntos y el código lo tienen en el repositorio.

El html del listado de tipos de vehiculos.

Partiremos de este código:

<div class="card animate\_\_animated animate\_\_fadeIn animate\_\_fast mt-3">

  <div class="card-header">Tipos de Vehículos</div>

  <div class="card-body text-primary">

    <div class="alert alert-warning mt-2 text-center" role="alert">

      En esta página, estamos probando el envío al servidor del token para ver

      los datos. Tienes que loguearte.

    </div>

    <button class="btn btn-outline-primary">Nuevo Tipo de Vehículo</button>

    <hr />

    <h5 class="card-title">

      Listado de tipos de Vehículos (<small> 5 </small>)

    </h5>

    <div class="alert alert-warning mt-2 text-center" role="alert">

      Tienes que estar autenticado para ver esta información

    </div>

    <table class="table table-striped table-bordered">

      <thead>

        <tr>

          <th>#</th>

          <th>Nombre</th>

          <th>Acciones</th>

        </tr>

      </thead>

      <tbody>

        <tr>

          <td class="w40 align-middle">1</td>

          <td>Camion</td>

          <td class="w70 align-middle">

            <button class="btn btn-outline-primary btn-sm">

              <i class="fa fa-edit"></i>

            </button>

            <button class="btn btn-outline-danger btn-sm">

              <i class="fa fa-trash"></i>

            </button>

          </td>

        </tr>

      </tbody>

    </table>

  </div>

</div>

Ahora todos juntos, vamos a dar sentido a esta pantalla.

Tendremos en cuenta que necesitamos traer los datos de los tipos de vehiculos almacenados en la base de datos a través de un servicio, que necesitaremos controlar si el usuario está autenticado para que vea estos mensajes alert y alguna cosa mas.

Vamos con ello.

Pasos que vamos a dar:

1. Rellenar la tabla con los registros que hay en la base de datos.
2. Hacer el método para borrar tipos de vehículos
3. Crear la página de creación de un nuevo tipo de vehículo

En este punto, hay que tener especial cuidado, ya que cómo hemos creado un nuevo módulo, pues tiene que conocer las rutas de la aplicación para poder hacer un routerLink en el botón. Para ello, en el vehículos module, hay que importar AppRoutingModule.

También nos pasa algo semejante al querer incorporar el formulario. En este módulo tendremos que incorporar el FormModule

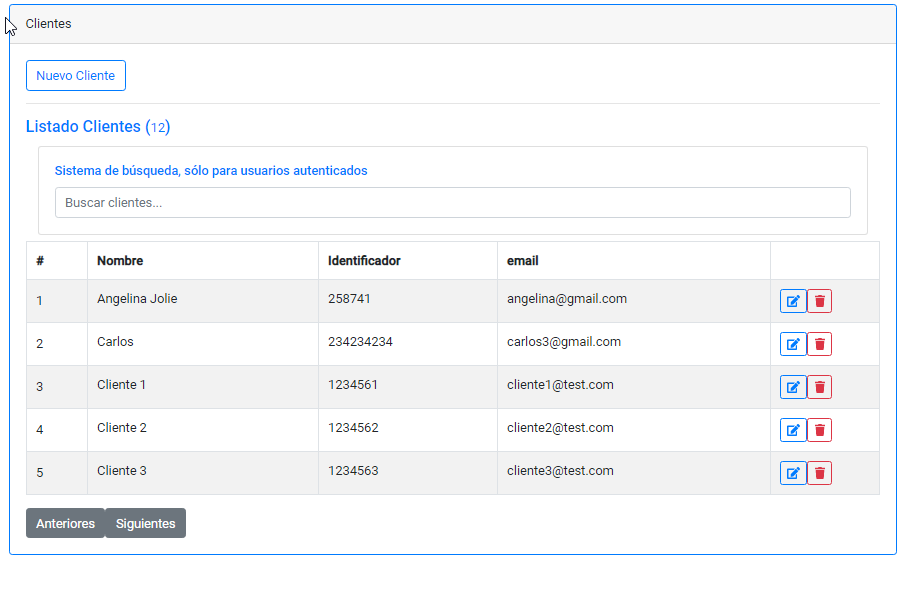
Esta página, sí que la vamos a controlar desde el guard y así probamos el state del guard.

1. Reconfigurar la página anterior para que nos sirva de edición del registro.

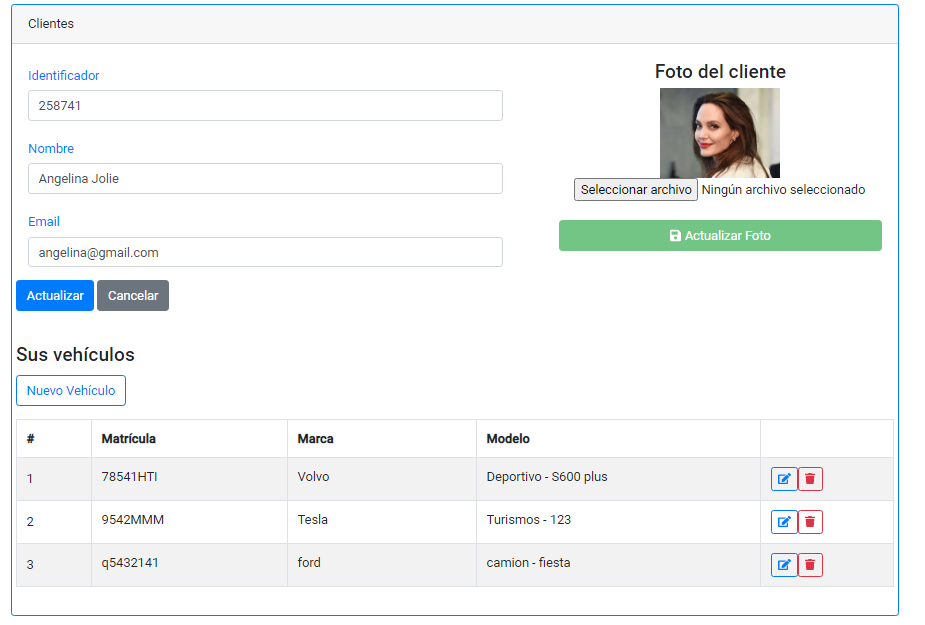
**Terminada toda la parte del CRUD de registro de Tipos de Vehículos, vamos a comenzar con el CRUD de clientes y los vehículos que tiene.**

Se trata de una pantalla del tipo maestro – detalle para gestionar los vehículos que tiene un cliente y la foto del cliente.

Primero tendríamos una página principal de clientes en la que nos ayuda incluso a buscar un cliente y paginado



Luego, al ver el detalle de un cliente, nos aparece una página del tipo maestro / detalle con la información del cliente y sus vehículos.



Para la parte inferior, podremos dar de alta/modificar o quitar los vehículos de un cliente concreto.

Al igual que hicimos con los tipos de vehículos y como buena práctica, toda esta gestión de clientes lo meteremos en un módulo nuevo.

Aprenderemos también a enviar ficheros a un servidor, por ejemplo una foto.

Por lo tanto, los pasos que daremos serán:

1. Crear el módulo clientes
2. Crear el componente clientes (es el principal )
3. Crear el componente para el detalle del cliente.
4. Crear el componente vehículos del cliente
5. Crear el componente del detalle de los vehículos del cliente
6. Crear el servicio cliente service y el servicio vehículos service.

Estas son las URL de los servicios que vamos a utilizar:

<https://backend-marco.herokuapp.com/vehiculo>

<https://backend-marco.herokuapp.com/cliente>

En estos casos, no hace falta mandar el token. El bakend está hecho con fines formativos por lo que aunque parezca raro, pues así lo tenemos.

Vamos creando el módulo, el servicio y todos los componentes que hemos mencionado.

ng g m pages/clientes

ng g c pages/clientes/clientes –skipTests

ng g c pages/clientes/clienteDetalle –skipTests

ng g c pages/clientes/vehiculosCliente –skipTests

ng g c pages/clientes/vehiculosDetalleCliente –skipTests

ng g s services/cliente

creo que podemos separarar el servicio y tener otro con los vehículos por eso, creamos otro servicio.

ng g s services/vehiculosCliente

A veces, cuando creamos los servicios, no los reconoce angular y hay que reiniciar el servidor. Lo hacemos y vamos ir poniendo los html base de cada componente.

Usaremos las siguientes páginas base que las podéis descargar de <https://github.com/jucarlos/apuntesCursoAngular>

Empezamos con la pantalla de clientes. Vamos a utilizar la plantilla llamada clientes.html

En esta pantalla haremos el trabajo siguiendo estos pasos.

1. Haremos la tabla de clientes
2. Implementaremos los botones anterior y siguiente.
3. Activaremos el sistema de búsqueda de clientes
4. Haremos el botón borrar
5. Finalmente haremos la navegación para ir a la página del detalle del cliente.

El servidor tiene un endpoint para que permite opcionalmente enviar los parámetros limite y desde para decirle que nos devuelva los clientes desde un número y con un límite de registros. Sería algo así: desde=4&limite=2

Para ello necesitamos el servicio que hemos creado y vamos con el trabajo.

Subida de archivo.

El servidor nos ofrece un endpoint para subir archivos. En este caso, el servidor obliga a que sean imágenes.

También nos ofrece otro servicio para obtener las imágenes. Este servicio tiene la particularidad de que si no existe la imagen nos devuelve una por defecto.

Los pasos que vamos a seguir para construir esto serán:

1. Creamos el servicio para que nos ayude con la subida de imágenes

ng g s services/subirArchivo --skipTests

1. Crearemos un pipe para que nos ayude con el momento de presentar la imagen del cliente o una por defecto.

ng g p pipes/imagen --skipTests

1. Terminaremos el html y la clase ts.

Preparando la aplicación para producción.

ng build –prod

Probamos la aplicación dentro de dist/ejercicio con un servidor local

<https://www.npmjs.com/package/http-server>

en la carpeta raíz del proyecto, poner http-server y nos dará información de las urls donde podemos entrar.

Para desplegar la aplicación en un servidor sería la carpeta dist.

https://www.npmjs.com/package/http-server

**Desplegando en firebase**

ng build --prod

1. Nombre del proyecto xxxxxxx

2. En mi caso, no habilito ananitics .

3. siguiente y siguiente

4. vamos a hosting

En el ordenador

instalamos firebase

npm install -g firebase-tools

Luego iniciamos sesión en firebase

firebase login

**luego en dist**

donde está el public o ejercicio

creamos el repositorio firebase con firebase init

Decimos que es Hosting

Decimos si a SPA

Al preguntar sobrescribir decimos que No

Por último hacer el deploy

firebase delploy

Cuando termine, nos informará de las rutas de la aplicación.

**Angular Material**

**Vamos a instalar, para probar el angular –material**, para probar la instalación de librerías de terceros y hacemos alguna prueba.

En concreto, probamos un **input con un snack-bar**

<https://material.angular.io/>

Primero creamos el componente donde vamos a hacer la prueba.

ng g c pages/angularMaterial -is --skipTests=true

Según la página de materias, agregamos material al proyecto con

ng add @angular/material

Hemos contestado N al gesturs reconizers y Y al animations

En app.module module, hemos puesto

import { MatSnackBarModule } from '@angular/material/snack-bar';

import { MatFormFieldModule } from '@angular/material/form-field';

import { MatInputModule } from '@angular/material/input';

MatSnackBarModule,

MatFormFieldModule,

MatInputModule,

Y el mismo código que viene de ejemplo en la página es el que hemos usado en nuestra aplicación.

**Interceptores**

Un interceptor no deja de ser un servicio que configuraremos de tal manera para que todas las peticiones http pasen por él.

Las dos ventajas principales para implementar los interceptores son las de añadir a las peticiones http una cabecera y otra muy importante es la de gestionar los errores producidos de forma centralizada.

Para este ejercicio, vamos a crear un nuevo proyecto de angular que hará una petición http <https://restcountries.eu/rest/v2/lang/es>

Los pasos a seguir serán:

1. Crear el servicio interceptor.
2. Configurar en el app.module el interceptor para que angular lo conozca.

**Socket**

**La librería que parece mas habitual para trabajar con socket es** ngx-socket-io

**Optimización del código.**

Creamos el módulo para las páginas. Optimizaciones

porque el modulo principal es muy grande

vamos a quitar todos los componentes de las páginas.

creamos el módulo de las pages

ng g m pages/pages --flat --dry-run

creamos también un módulo para los pipes

SWAGER

https://app-ejemplo-utilidades-matg09.cm-pre.jccm.es/swagger-ui.html?urls.primaryName=app-ejemplo#/Vehiculos

https://app-ejemplo-utilidades-matg09.cm-pre.jccm.es/api/vehiculos?matricula=contains(AAA)

https://app-ejemplo-utilidades-matg09.cm-pre.jccm.es/api/vehiculos?marca=Seat

https://app-ejemplo-utilidades-matg09.cm-pre.jccm.es/api/vehiculos?marca=ne(Seat)

RESUMEN DE LOS CLI QUE HEMOS UTILIZADO

================================================

ng new ejercicio

ng serve -o

ng g c components/navbar --skipTests=true --dry-run

ng g c home --skipTests=true --dry-run

ng g c pages/directivas --skipTests=true --dry-run

ng g c pages/temario --skipTests=true

ng g c pages/temario/temario-tarjeta --flat --skipTests=true --dry-run

ng g s services/temario --dry-run

ng g c pages/pipes --skipTests=true

ng g p pipes/alreves --skipTests=true --dry-run

ng g c pages/tipo-vehiculo/edicionTipoVehiculo --flat -is --skipTests=true --dry-run

ng g c pages/clientes --skipTests=true -is

ng g s services/cliente --flat --skipTests=true --dry-run

ng g s services/subirArchivo --flat --skipTests=true --dry-run

ng g c pages/clientes/vehiculosCliente --flat -is --skipTests=true

ng g c pages/taller -is --skipTests=true

ng g g guards/login --skipTests=true

**Soluciones a los cuestionarios de autoevaluación:**

Autoevaluación 1.

1c, 2a, 3c, 4a, 5c

Autoevaluación 2.

1a, 2b, 3c, 4c, 5b