PIPES

Los pipes, son un mecanismo que tiene Angular, para la transformación visual de los datos.

Hay que tener en cuenta que no modifica los datos, únicamente se utilizan para mostrarlos de otra forma.

Por ejemplo, si tenemos una variable llamada ciudad que tiene el valor Toledo, y queremos mostrarla en nuestro html, pondríamos

{{ ciudad }}

Se mostraría Toledo

Si pasamos la variable ciudad por unos de los pipes de Angular como es uppercase, se mostrará todo en mayúsculas.

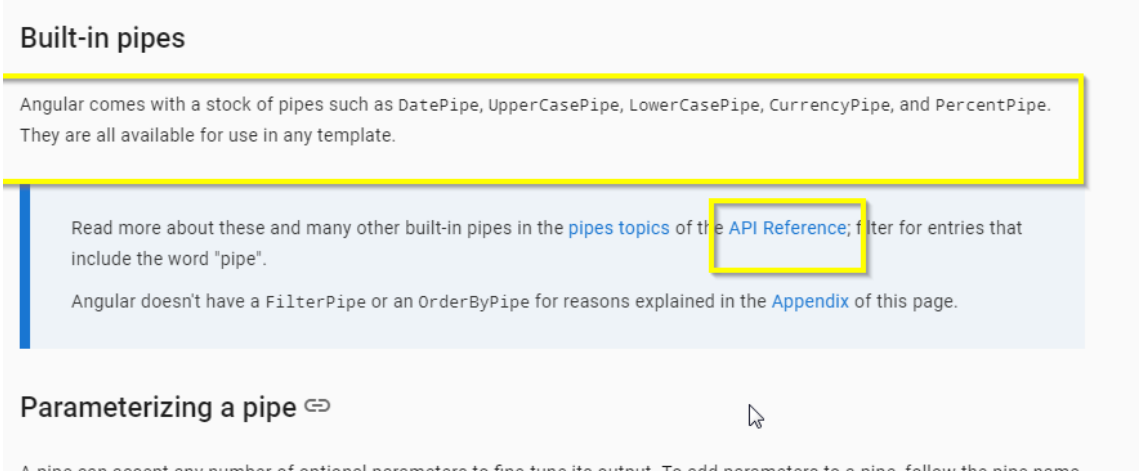
Se haría así:

{{ ciudad | uppercase }}

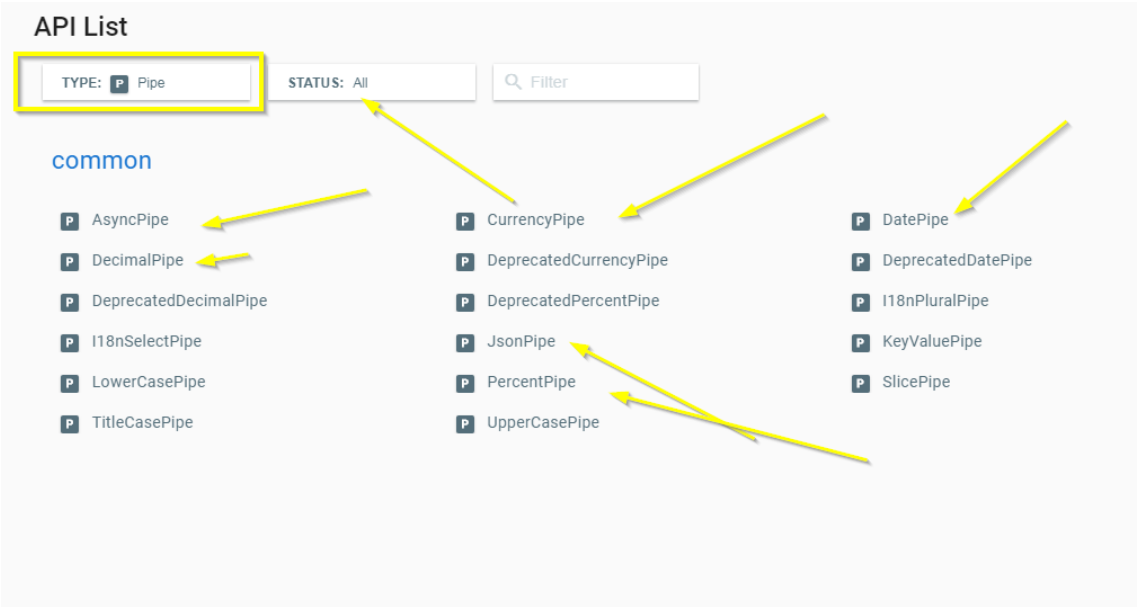
Se mostraría TOLEDO

Pero vamos al lío.

En https://angular.io/ buscamos pipes y en la página https://angular.io/guide/pipes



Aquí podemos ver algunos de los pipes que mas se utilizaan, como es el DatePipe, CurrencyPipe, PercentPipe y pinchando en API Reference, podemos ver mas en detalle cada uno de ellos.



Nosotros, vamos a realizar unos ejemplos para cubrir estos pipes y además algo que me he encontrado sobre mostrar por ejemplo un pdf externo o un video u otro recurso de forma segura.

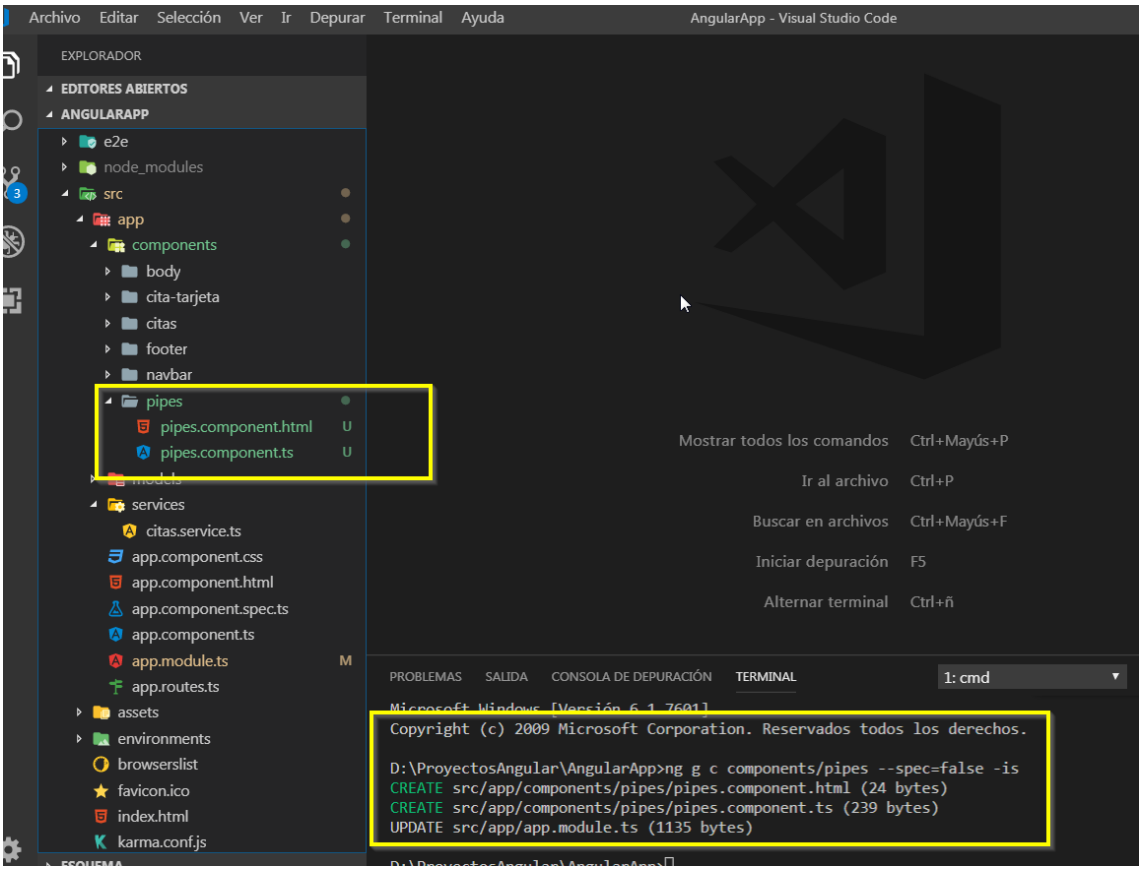
Para ello, en nuestra aplicación, vamos a poner otra opción de menú que la llamaremos Pipes y que tendrá una tabla bootstrap.

1. Preparar la opción de menú

   Abrimos con VSC y vamos a crear un componente llamado pipes en la carpeta components.

   Para ello decimos y escribimos (Angular, genera un componente llamado pipes en la carpeta components y no me hagas el fichero de pruebas y sin estilos): ng g c components/pipes –spec=false –is

Si no hay problemas, nos quedará algo así:



Abrimos ahora el fichero de las rutas de angular e incorporamos la ruta a este componente.

Mi fichero de rutas app.routes.ts, va teniendo esta forma:

```
import { RouterModule, Routes } from '@angular/router';
import { BodyComponent } from './components/body/body.component';
import { CitasComponent } from './components/citas/citas.component';
import { CitaComponent } from './components/citas/cita.component';
import { MostrarCitasComponent } from './components/citas/mostrar-
citas.component';
import { PipesComponent } from './components/pipes/pipes.component';

const APP_ROUTES: Routes = [
    { path: 'body', component: BodyComponent },
    { path: 'citas', component: CitasComponent },
    { path: 'cita/:id', component: CitaComponent },
    { path: 'mostrar/:termino', component: MostrarCitasComponent },
    { path: 'pipes', component: PipesComponent },
    { path: '**', pathMatch: 'full', redirectTo: 'body' }
];

export const APPROUTING = RouterModule.forRoot(APP_ROUTES);
```

Ahora en el html del navbar, ponemos otra opción para ir a pipes.

Por ejemplo debajo de Citas.

En mi caso, el fichero navbar.component.html, va quedando así

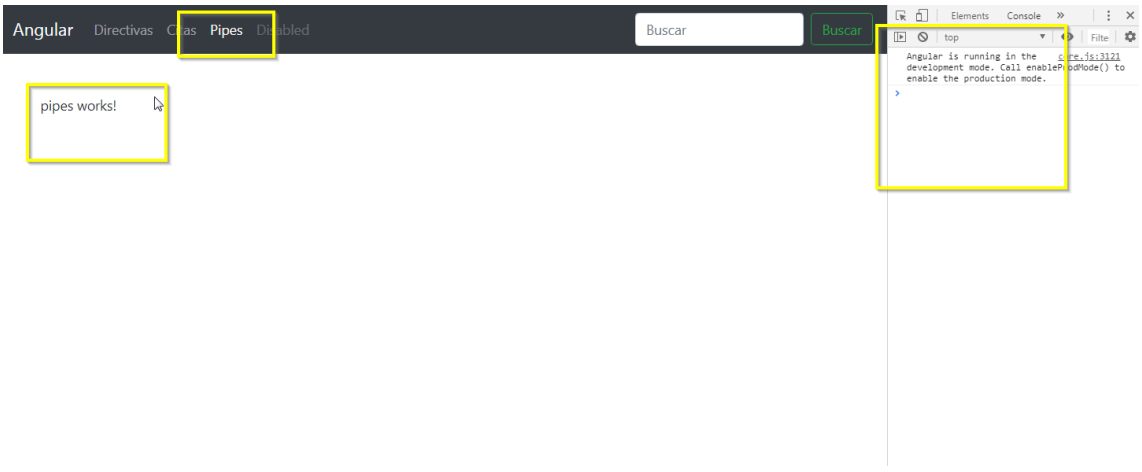
```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    <a class="navbar-brand" href="#">Angular</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse"
data-target="#navbarSupportedContent" aria-
controls="navbarSupportedContent" aria-expanded="false" aria-
label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>

    <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item">
                <a class="nav-link" routerLinkActive="active"
[routerLink]="['body']">Directivas <span class="sr-
only">(current)</span></a>
```

```
            </li>
            <li class="nav-item">
                <a class="nav-link" routerLinkActive="active"
[routerLink]="['citas']">Citas</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" routerLinkActive="active"
[routerLink]="['pipes']">Pipes</a>
            </li>
            <li class="nav-item">
                <a class="nav-link disabled" href="#" tabindex="-1" aria-
disabled="true">Disabled</a>
            </li>
        </ul>
        <form class="form-inline my-2 my-lg-0">
            <input class="form-control mr-sm-2" type="text"
placeholder="Buscar" aria-label="Search" #textoBuscar>
            <button (click)="buscarCita(textoBuscar.value)" class="btn
btn-outline-success my-2 my-sm-0" type="button">Buscar</button>
        </form>
    </div>
</nav>
```

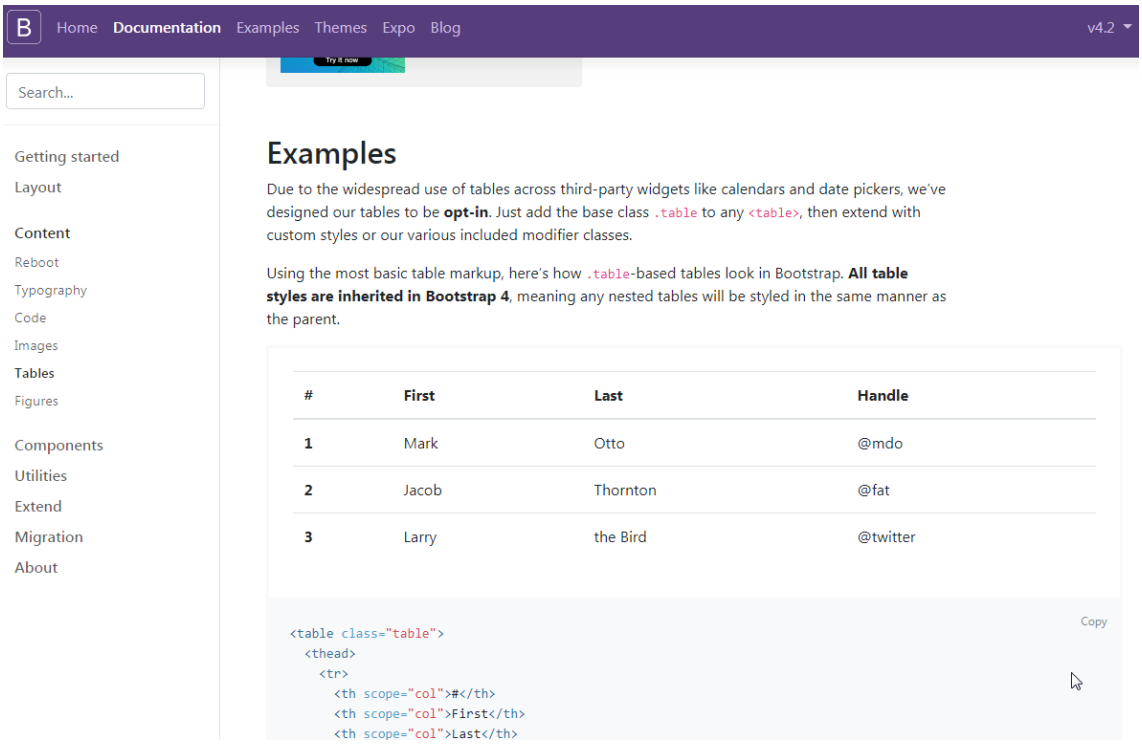
Ahora, vamos a probar iniciando el servidor con ng serve –o en la carpeta del proyecto.

Tiene que salirnos otra opción de menú y además se tiene que poder navegar a ella.

Tenemos que tener esto y sin errores en la consola.

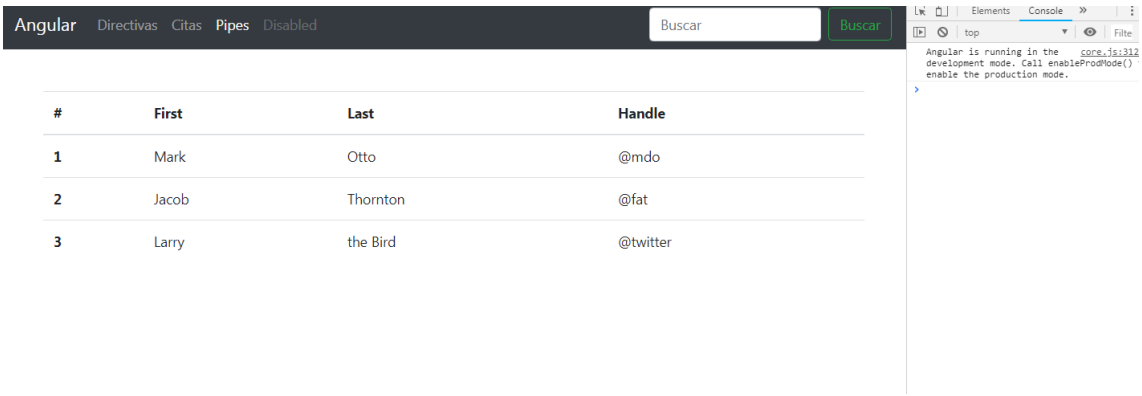


2. Preparar la tabla bootstrap
   En la página de bootstrap, buscamos una table y cogemos por ejemplo la primera que viene



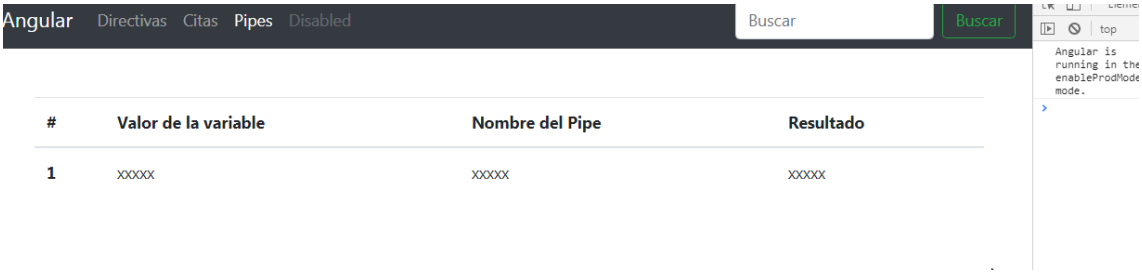
Hacemos un copy/paste en nuestro archivo pipes.component.html y pegamos esta tabla sustituyendo lo que tenemos de pipes Works.

Tenemos que tener esto:



Lo que vamos a hacer es poner los títulos de los pipes que vamos a utilizar y como se muestra el dato con el valor que tiene y por defecto y como se muestra cuando lo pasamos por el pipe. Recuerdo **que los pipes no cambian los valores** únicamente los muestra de otra forma.

Cambiamos un poco los títulos:



Lo dejamos así y seguimos con los primeros pipes.

3. Pipes uppercase y lowercase

En el ts del componente creamos la variable ciudad y lo igualamos a Toledo. Quedará así:

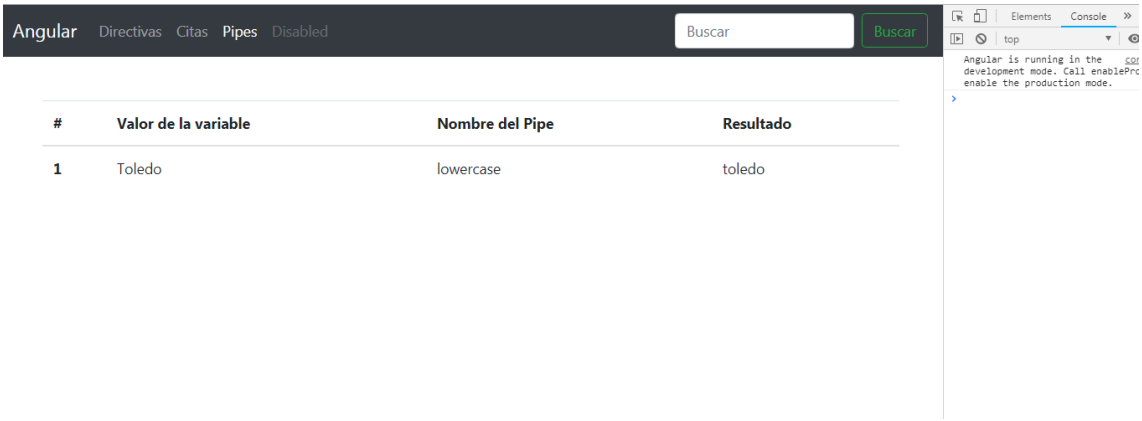
```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-pipes',
  templateUrl: './pipes.component.html',
  styles: []
})
export class PipesComponent implements OnInit {

  ciudad = 'Toledo';
```

```
  constructor() { }

  ngOnInit() {
  }

}
```

En el archivo html del componente ponemos:

```
<table class="table">
    <thead>
        <tr>
            <th scope="col">#</th>
            <th scope="col">Valor de la variable</th>
            <th scope="col">Nombre del Pipe</th>
            <th scope="col">Resultado</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <th scope="row">1</th>
            <td> {{ ciudad }}</td>

            <td>lowercase</td>
            <td>{{ ciudad | lowercase}}</td>
        </tr>

    </tbody>
</table>
```

Como se puede ver, estamos mostrando el varlor de la variable ciudad y la pasamos por el pipe lowercase.
El resultado:



En otra fila de la tabla, usamos el pipe uppercase

```
<tr>
            <th scope="row">1</th>
            <td> {{ ciudad }}</td>

            <td>lowercase</td>
            <td>{{ ciudad | uppercase}}</td>
        </tr>
```

4. Pipe Slice

   Este pipe sirve para cortar una cadena de caracteres, pero a este pipe, se le tienen que pasar parámetros.
   Por ejemplo:

   Vamos a seguir con la variable ciudad y escribimos el pipe así:

{{ ciudad | slice:1 }} aquí lo que le estamos diciendo es que elimine la primera posición de la cadena de texto.

Si por ejemplo ponemos {{ ciudad | slice:0:2 }} lo que estamos diciendo es que queremos que muestre desde la posición 0 a la 2.

```
<tr>
            <th scope="row">1</th>
            <td>{{ ciudad }}</td>
            <td>Slice</td>
            <td>{{ ciudad | slice:0:3}}</td>
        </tr>
```

Vamos a crear otra variable, que será un array con números primos, de la forma siguiente:

```
  ciudad = 'Toledo';
  primos = [1, 2, 3, 5, 7, 11, 13];
```

Y queremos que muestre desde el primero al cuarto

```
<tr>
            <th scope="row">1</th>
            <td>{{ primos }}</td>
            <td>Slice</td>
            <td>{{ primos | slice:0:4}}</td>
        </tr>
```

5. Pipe Json

Vamos a crear en el ts un objeto con datos, por ejemplo:

```
  alumno = {
    nombre: 'Angelina',
    apellidos: 'Jolie',
    edad: 45
  };
```

6. Pipe Decimal

Es para mostrar números con un determinado formato.

Este pipe es un poco más raro, pero está muy bien explicado en esta parte de la documentación de angular:

https://angular.io/api/common/DecimalPipe

DecimalPipe

PIPE

Transforms a number into a string, formatted according to locale rules that determine group sizing and separator, decimal-point character, and other locale-specific configurations.

See more...

```
{{ value_expression | number [ : digitsInfo [ : locale ] ] }}
```

NgModule

-

```
CommonModule
```

Input value

value	any	The number to be formatted.

Parameters

digitsInfo	string	Decimal representation options, specified by a string in the following format: `{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}`. • `minIntegerDigits`: The minimum number of integer digits before the is `1`. • `minFractionDigits`: The minimum number of digits after the decimal • `maxFractionDigits`: The maximum number of digits after the decimal Optional. Default is `undefined`.
locale	string	A locale code for the locale format rules to use. When not supplied, uses the valu which is `en-US` by default. See Setting your app locale.

Optional. Default is `undefined`.

Y además con varios ejemplos. Peo bueno, lo hacemos nosotros con algún caso:

Creamos una variable llamada cambio con el valor 166.386

```
ciudad = 'Toledo';
  primos = [1, 2, 3, 5, 7, 11, 13];
  cambio = 166.386789;
```

y vamos a poner en el html estos ejemplos

```
<tr>
            <th scope="row">5</th>
            <td>{{ cambio }}</td>
            <td>Decimal</td>
            <td>{{ cambio | number }}</td>
        </tr>

        <tr>
            <th scope="row">6</th>
            <td>{{ cambio }}</td>
            <td>Decimal</td>
            <td>{{ cambio | number:'4.0-2' }}</td>
        </tr>
```

El primer pipe sin parámetros, deja el número con 3 decimales (por defecto)

En el segundo, le estamos diciendo que queremos 4 dígitos en la parte entera y 0 o como mucho 2 decimales.

No creo que esto sea complejo, pero es bueno saber que existe.

7. Pipe Percent
   https://angular.io/api/common/PercentPipe

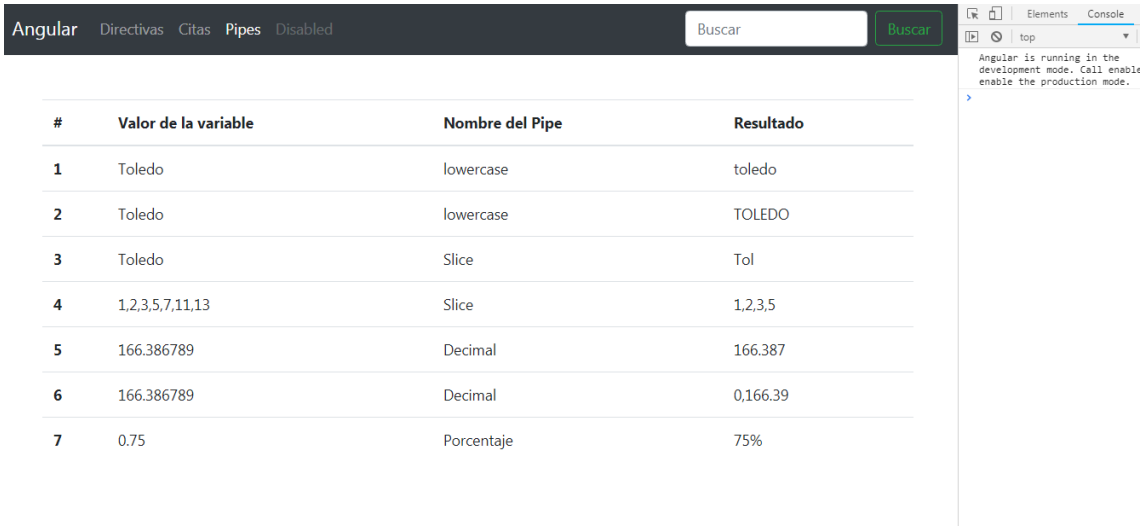
En el ts, ponemos la variable:

```
porcentaje = 0.75;
```

y en la vista:

```
8.  <tr>
9.              <th scope="row">7</th>
10.             <td>{{ porcentaje }}</td>
11.             <td>Porcentaje</td>
12.             <td>{{ porcentaje | percent }}</td>
13.         </tr>
```

Tiene que salir:



#	Valor de la variable	Nombre del Pipe	Resultado
1	Toledo	lowercase	toledo
2	Toledo	lowercase	TOLEDO
3	Toledo	Slice	Tol
4	1,2,3,5,7,11,13	Slice	1,2,3,5
5	166.386789	Decimal	166.387
6	166.386789	Decimal	0,166.39
7	0.75	Porcentaje	75%

Con un poco de formato, ponemos por ejemplo en la vista:

Tiene los parámetros de forma muy parecida al pipe Decimal por ejemplo

Vamos a poner otra variable con el valor 0.4587

```
porcentaje2 = 0.4587;
```

y en la vista:

```
<tr>
            <th scope="row">8</th>
            <td>{{ porcentaje2 }}</td>
            <td>Porcentaje</td>
            <td>{{ porcentaje2 | percent:'2.2-2' }}</td>
        </tr>
```

Le estamos diciendo que queremos 2 enteros y 2 decimales.

14. Pipe Currency (El de la moneda)

    En la documentación pone:

Transforms a number to a currency string, formatted according to locale rules that determine group sizing and separator, decimal-point character, and other locale-specific configurations.

```
{{ value_expression | currency [ : currencyCode [ : display [ : digitsInfo
[ : locale ] ] ] ] }}
```

    https://angular.io/api/common/CurrencyPipe

    Conviene mirarlo, para ver com ova el tema de los países y como muestra por ejemplo el símbolo de Euro.

    Al lío:

    Pongamos una variable precio con el valor 1533.56.

```
precio = 1533.56;
```

    en la vista:

```
<tr>
            <th scope="row">9</th>
            <td>{{ precio }}</td>
            <td>Currency</td>
```

```
            <td>{{ precio | currency }}</td>
        </tr>
```

Sale algo así:

9	1533.56	Currency	$1,533.56

Si queremos que salga con el euro, pondremos:

```
<td>{{ precio | currency:'EUR' }}</td>
```

De esta forma, si que sale el símbolo del euro.
Hay muchas combinaciones, que os invito a probar.

15. Pipe Date

    Aquí tenemos la información.

https://angular.io/api/common/DatePipe

Pongamos en nuestro ts, una fecha y vamos a mostrarla de varias formas, para ver cómo va este pipe.

En el ts, agregamos la variable:

```
  fecha = new Date();
```

Por cierto, que en el ts, no estamos utilizando el constructor, tampoco el ngOninit() y se podría quitar.

Y en el html, hemos puesto estas pruebas.

```
    <tr>
            <th scope="row">13</th>
            <td>{{fecha}}</td>
            <td>Fecha</td>
            <td>{{ fecha }}</td>
        </tr>

        <tr>
            <th scope="row">14</th>
            <td>{{fecha}}</td>
            <td>Fecha</td>
            <td>{{ fecha | date }}</td>
        </tr>

        <tr>
            <th scope="row">16</th>
            <td>{{fecha}}</td>
            <td>Fecha</td>
            <td>{{ fecha | date:'fullDate' }}</td>
        </tr>
```

PERO ESTÁ EN INGLÉS!!!

Bueno, para esto, hay que poner en el app.module.ts lo siguiente.

(Seguro que habrá mas formas de hacerlo, pero esto funciona)

Se importa la constante LOCALE_ID de @angular/core y se da de alta en los providers (que es donde se darán de alta los servicios siempre que no vienen configurados con providedIn: 'root' con la anotación @Inyectable).

En el app.module.ts, agregamos las importaciones:

```
import { LOCALE_ID } from '@angular/core';

import localeEs from '@angular/common/locales/es';
import { registerLocaleData } from '@angular/common';
registerLocaleData( localeEs );
```

y en el providers:

```
{ provide: LOCALE_ID, useValue: 'es' }
```

De tal forma que nuestro app.module.ts, va quedando así:

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

import { AppComponent } from './app.component';
import { NavbarComponent } from './components/navbar/navbar.component';
import { BodyComponent } from './components/body/body.component';
import { FooterComponent } from './components/footer/footer.component';
import { CitasComponent } from './components/citas/citas.component';
import { APPROUTING } from './app.routes';
import { CitaComponent } from './components/citas/cita.component';
import { MostrarCitasComponent } from './components/citas/mostrar-
citas.component';
import { CitaTarjetaComponent } from './components/cita-tarjeta/cita-
tarjeta.component';
import { PipesComponent } from './components/pipes/pipes.component';

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

import localeEs from '@angular/common/locales/es';
import { registerLocaleData } from '@angular/common';
registerLocaleData( localeEs );


@NgModule({
  declarations: [
    AppComponent,
    NavbarComponent,
    BodyComponent,
    FooterComponent,
    CitasComponent,
    CitaComponent,
    MostrarCitasComponent,
    CitaTarjetaComponent,
    PipesComponent
  ],
```

```
  imports: [
    BrowserModule,
    APPROUTING
  ],
  providers: [
    { provide: LOCALE_ID, useValue: 'es' }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

16. Pipe asíncrono

    Este pipe, sirve para que se muestre un valor cuando termina de ejecutarse una promesa o viene el valor de un observable. Esto de los observables, los veremos con más detalle cuando hagamos las llamadas a los servicios Rest de un backend que crearemos con Spring boot.
    De momento, en nuestro ts, vamos a crearnos una promesa llamada promesa, que tiene los parámetros resolve y reject para devolver si se realiza bien o mal. Bien con resolve y reject si se hace mal.
    Simulamos una petición que se retrasa 4 segundos y que cuando termina, el dato que viene es el string Terminó!!!

    En el ts

```
promesa = new Promise ( (resolve, reject ) => {

    setTimeout ( () => {
      resolve( 'Terminó !!!');
    }, 4000);

  });
```

Estamos diciendo que cuando terminen los 4 segundos, que devuelva la frase Terminó!!!

Si en el html ponemos

```
<tr>
            <th scope="row">12</th>
            <td>{{promesa}}</td>
            <td>Asíncrono</td>
            <td>{{ promesa }}</td>
        </tr>
```

Nos muestra esto:

12	[object Promise]	Asíncrono	[object Promise]

Pero si ponemos esto en el html,

```
<td>{{ promesa | async}}</td>
```

Nos saldría así

12	[object Promise]	Asíncrono	Terminó !!!

Pasados los 4 segundos.

17. Pipes personalizados

Hagamos un pipe personalizado que escriba una cadena de texto pero al revés.

Ya sé que es una cosa sin mucho sentido, pero lo importante es saber cómo se utilizan.

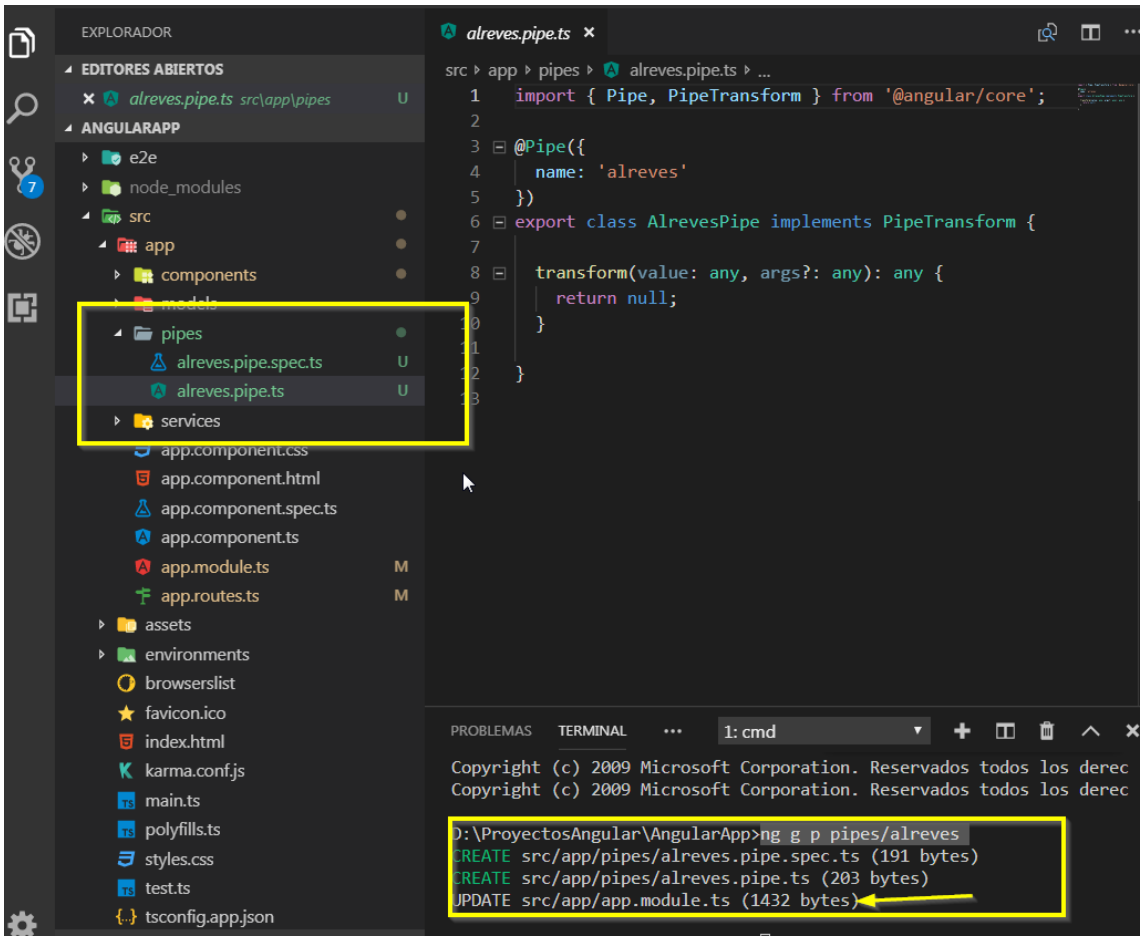
Al lío,

Creamos una variable en nuestro ts

```
cadenaDeTexto = 'Saber que se sabe lo que se sabe y que no se sabe lo que
no se sabe; he aquí el verdadero saber. (Confucio)';
```

Ahora, con la terminal y con el cli de angular generamos un pipe.

Por eso, decimos y escribimos (Angular, genérame un pipe dentro de la carpeta pipes, que se llame alreves) ng g p pipes/alreves



Cómo se puede observar, nos ha creado la carpeta pipes y nos ha generado de igual forma el ts del pipe anotado con @Pipe.

También nos ha generado el archivo de pruebas unitarias. (Si estos documentos tiene éxito, os escribo cómo va esto de las pruebas en angular, que es muy chulo).

Por otra parte, nos ha modificado el app.module.ts, para informar a angular que tenemos un pipe.

Pero, ¿Qué es lo que tiene un Pipe?

Este es el código del Pipe que se ha generado de forma automática

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

@Pipe({
  name: 'alreves'
})
export class AlrevesPipe implements PipeTransform {

  transform(value: any, args?: any): any {
    return null;
  }

}
```

Lo que tenemos aquí es lo siguiente:

Lo primero es la anotación @Pipe, en la que ponemos el nombre del pipe que luego se utilizará en el html

Luego viene el nombre de la clase , que implementa PipeTransforms e implementamos la función transform, que devuelve un valor de tipo any. Lo vamos a cambiar, diciendo que devuelve un string, además, vamos a poner ya en el html

```
        <tr>
            <th scope="row">16</th>
            <td>{{cadenaDeTexto}}</td>
            <td>Alreves - Personalizado </td>
            <td>{{ cadenaDeTexto | alreves }}</td>
        </tr>
```

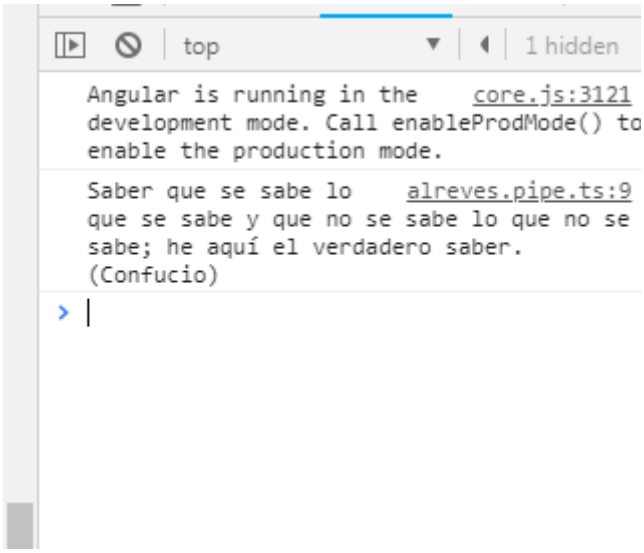
Y dentro del transform pondremos un console.log, para ver lo que viene.

El ts de nuestro pipe quedará así:

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

@Pipe({
```

```
  name: 'alreves'
})
export class AlrevesPipe implements PipeTransform {

  transform(value: any, args?: any): string {
    console.log( value );
    return null;
  }

}
```

Y en la consola, sale el valor que viene al pipe



Ahora, vamos a realizar nuestro código para que devuelva el valor alrevés.

Quedará así:

(Seguro que hay mejores formas de hacerlo, pero esta funciona)

Ponemos que el value es string, para que typescript nos ayude con los métodos de un string.

Y el código quedaría así:

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

@Pipe({
  name: 'alreves'
})
export class AlrevesPipe implements PipeTransform {

  transform(value: string, args?: any): string {
    console.log( value );

    let alreves = '';

    let x = value.length;

    while ( x >= 0) {
      alreves = alreves + value.charAt(x);
      x--;
    }


    return alreves;
  }

}
```

No estaría mal, preguntar si el parámetro value, trae algo escrito, pero bueno.

FORMULARIOS EN ANGULAR

Lo primero que hay que tener en cuenta es que angular utiliza dos formas para trabajar con los formularios. Una es en el html y la otra es en la parte del ts.

Comenzaremos a realizar unos ejercicios en la parte del html.

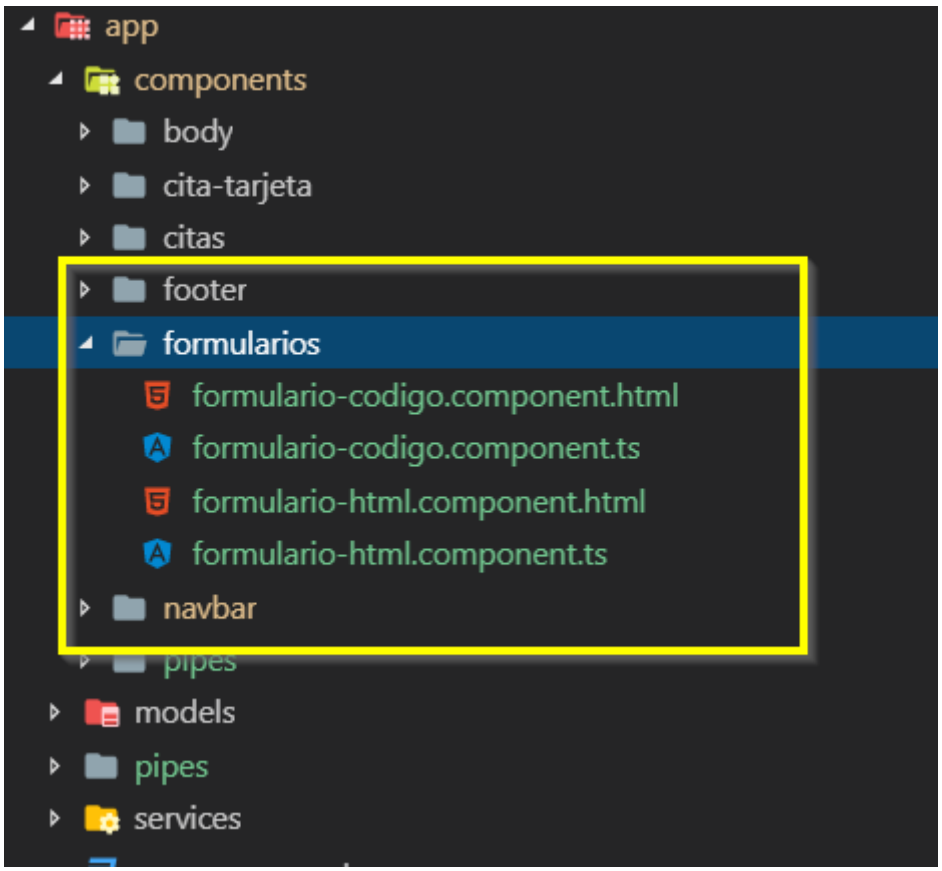
Como tarea, aunque luego tendrán el código, vamos a crear otra opción de menú llamada Formularios y que tenga un desplegable (se puede copiar de bootstrap) con dos opciones. Una que se llame Html y otra Código Ts.

Además, hay que crear dos componentes que llamados formularioHtml y formularioCodigo dentro de una carpeta components/formularios (en esto os pongo la instrucción del angular cli, que es:

ng g c components/formularios/formularioHtml --spec=false -is --flat

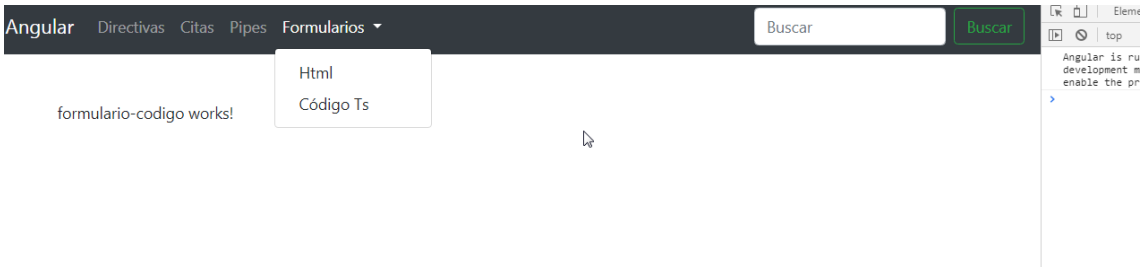
ng g c components/formularios/formularioCodigo --spec=false -is –flat

La bandera flat es para que no genere mas directories y guarde los dos componente en el mismo directorio, quedando así:



) y también actualizar el fichero app.routes.ts, para que cuando se pinche es esas opciones se muestren los componentes que hemos hecho.

Tiene que quedar así:



Y claro, sin errores.

Para comenzar, vamos al **componente formulario html y** vamos a ir creando un formulario con algunos campos

Vamos a comenzar con este formulario base en el html que ha sido generado desde los forms de bootstrap

```
<h4>Formularios <small>dentro el html</small></h4>
<hr>
<form>

    <div>

        <div class="form-group row">
            <label class="col-2 col-form-label">Nombre</label>
            <div class="col-8">

                <input class="form-control" type="text"
placeholder="Nombre">
            </div>
        </div>

        <div class="form-group row">
            <label class="col-2 col-form-label">Apellido</label>
            <div class="col-8">

                <input class="form-control" type="text"
placeholder="Apellido">
            </div>
        </div>

    </div>

    <div class="form-group row">
        <label class="col-2 col-form-label">Correo</label>
        <div class="col-md-8">

            <input class="form-control" type="email" placeholder="Correo
electrónico">
        </div>
    </div>
```

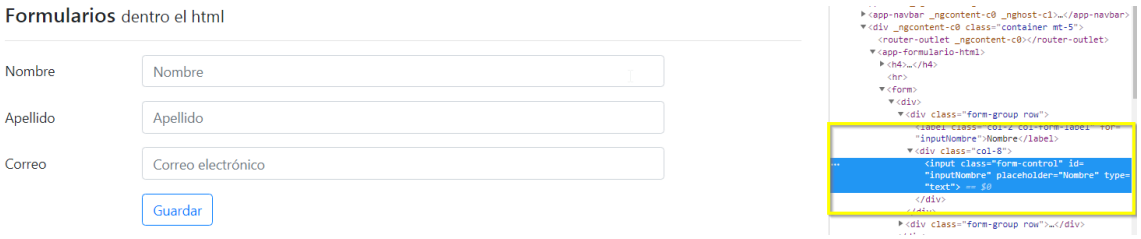
```
    <div class="form-group row">
        <label class="col-2 col-form-label"> </label>
        <div class="input-group col-md-8">
            <button type="submit" class="btn btn-outline-primary">
        Guardar
      </button>
        </div>
    </div>

</form>
```

En el ts, no tenemos nada, incluso y ya que no estamos utilizando el OnInit, lo podríamos quitar, dejando el ts de la forma:

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

@Component({
  selector: 'app-formulario-html',
  templateUrl: './formulario-html.component.html',
  styles: []
})
export class FormularioHtmlComponent {

  constructor() { }


}
```

La verdad, es que es un poco complicado explicar esto de forma escrita, después de que uno se acostumbra a youtube, pero lo intentaré.

Lo primero es que vamos a inspeccionar con las herramientas de desarrollo de crhome, el campo nombre de nuestro formulario (botón derecho sobre el componente e inspeccionar elemento) , para ver las clases que le ha incorporado.



Como se puede observar, no hay ninguna clase rara que llame la atención, todas son típicas de un input.

En angular, los formularios tienen una propiedad clave que se pone en el componente que queremos que sea nuestro formulario. Esa palabra es ngModel.

Pues bien, vamos a ponerlo en nuestro componente nombre. Y además, esa propiedad, nos dice la documentación, tiene que ir con otra propiedad llamada name. Tienen que estar las dos. Si no se pone el name, podéis ver en la consola, que muestra un error diciendo precisamente que tienen que estar.

Al lío.

Una vez que ponemos en el input, el ngModel y el name, al inspeccionar deberíamos ver un montón de nuevas propiedades en el elemento nombre. Veríamos esto:

En el caso de que no se vea, puede ser que la versión que tenemos de angular, no ha incorporado el módulo de formularios en la aplicación.

Creo que hasta la versión 5 lo incorporaba y después, lo tenemos que incorporar nosotros, pero es simple. En el app.module, importamos :

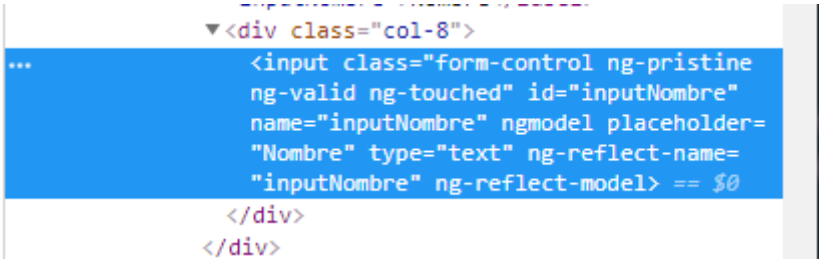
```
import { FormsModule } from '@angular/forms';
```

y en los imports, ponemos ese módulo.

```
FormsModule
```

```
<input class="form-control"
                type="text" placeholder="Nombre"
                id="inputNombre"
                name="inputNombre"
                ngModel>
```

Lo del id, no se utilza, lo podríamos quitar.



Las nuevas propiedades que tiene el campo hacen referencia a si el campo es válido (ng-valid), si está limpio (que no se ha tocado)si está touched, etc.

Ahora, vamos a poner unas reglas de validación para ver que el ng-valid no es tal si no se cumplen y además, vamos a poner en el form el submit, para que cuando se pulse al botón aceptar se ejecute el “Submit”.

Para ello, ponemos en la etiqueta form, lo siguiente:

```
<form (ngSubmit)="guardar(formulario)" #formulario="ngForm">
```

Le estamos diciendo, que el submit, ejecute una función que está en el ts y que le mande un objeto formulario del tipo ngForm, es decir, del tipo de formulario de angular. La referencia local a un elemento se hace con #.

Luego, en el ts, tenemos que tener la función guardar, que recibe un parámetro de tipo formulario. Aunque se puede dejar como any (cualquier cosa)

```
  guardar( miFormulario: any) {
    console.log('Submit');
    console.log(miFormulario);
  }
```

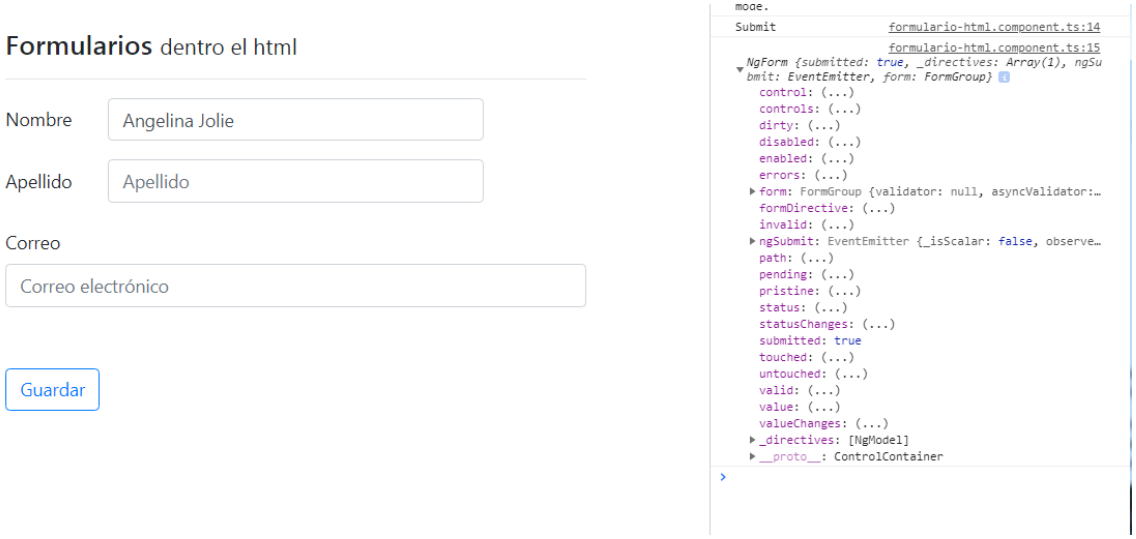
Lo mejor, es poner que es de tipo formulario de angular, para que de esa forma, tengamos la ayuda del editor de código con las propiedades de un elemento de tipo formulario de angular.

De cualquier forma, cuando hacemos el console.log del formulario que nos envía, vamos a obtener un montón de propiedades a las que podemos acceder, para ver si el formulario tiene errores, es válido, los componentes que tiene , etc.

```
guardar( miFormulario: NgForm) {
  console.log('Submit');
  console.log(miFormulario);
}
```

Y tenemos que importar, si no lo hace por nosotros el VSC:

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



Vamos a terminar este pequeño formulario con estos cambios.

Le vamos a poner unos datos por defecto, le vamos a poner validaciones a los campos nombre, apellidos y email.

Además, le vamos a poner que el botón se desactive si el formulario no es válido.

Al lío

Antes de comenzar, no está de mas, quitar las validaciones del html 5, para ello en el form, le ponemos la bandera novalidate, quedando así:

```
<form (ngSubmit)="guardar(formulario)" #formulario="ngForm" novalidate>
```

El primer paso, ha sido crear un objeto que contiene algunas propiedades de un alumno:

```
  alumno = {
    nombre: 'Angelina',
    apellidos: 'Jolie',
    email: 'angelina@gmail.com'
  };
```

Luego, en el html, he puesto el ngModel de la forma [(ngModel)], de esta forma, hacemos un binding al ts. Si ponemos únicamente [ngModel], se cargarían por defecto como en una copia, podéis hacer pruebas poniendo un console.log en el método guardar con los datos del alumno teniendo en ngModel con () y con [()].

En fin, ponemos también en correo electrónico, que sea válido según una expresión regular.

La expresión regular, está buscada y pegada de google.

El html quedará así y el ts quedará así.

```
<h4>Formularios <small>dentro el html</small></h4>
<hr>
<form (ngSubmit)="guardar(formulario)" #formulario="ngForm" novalidate>

    <div>

        <div class="form-group row">
            <label for="inputNombre" class="col-2 col-form-
label">Nombre</label>
            <div class="col-8">

                <input class="form-control" [ngClass]="{ 'is-invalid' :
nombre.errors && nombre.touched} " type="text" placeholder="Nombre"
id="inputNombre" name="inputNombre" [(ngModel)]="alumno.nombre" required
minlength="3" #nombre="ngModel">
```

```
                <div *ngIf="nombre.errors?.required" class="invalid-
feedback">
                    El nombre es necesario
                </div>
                <div *ngIf="nombre.errors?.minlength" class="invalid-
feedback">
                    El nombre tiene que tener al menos {{
nombre.errors.minlength.requiredLength }} caracteres
                </div>
            </div>
        </div>

        <div class="form-group row">
            <label for="inputApellido" class="col-2 col-form-
label">Apellido</label>
            <div class="col-8">

                <input class="form-control" [ngClass]="{ 'is-invalid' :
apellidos.errors } " type="text" id="inputApellido"
placeholder="Apellido" name="apellido" [(ngModel)]="alumno.apellidos"
#apellidos="ngModel">

            </div>
        </div>

    </div>

    <div class="form-group row">
        <label for="inputCorreo" class="col-2 col-form-
label">Correo</label>
        <div class="col-md-8">

            <input class="form-control" [ngClass]="{ 'is-invalid' :
correo.errors } " type="email" id="inputCorreo" placeholder="Correo
electrónico" name="email" [(ngModel)]="alumno.email" pattern="[a-z0-
9._%+-]+@[a-z0-9.-]+\.[a-z]{2,3}$" #correo="ngModel">
            <div *ngIf="correo.errors?.pattern" class="invalid-feedback">
                El correo tiene que estar bien
            </div>
        </div>
    </div>
```

```
    <div class="form-group row">
        <label class="col-2 col-form-label"> </label>
        <div class="input-group col-md-8">
            <button type="submit" [disabled]="!formulario.valid"
class="btn btn-outline-primary">
        Guardar
      </button>
        </div>
    </div>

</form>
```

Y el ts quedará así:

```
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-formulario-html',
  templateUrl: './formulario-html.component.html',
  styles: []
})
export class FormularioHtmlComponent {


  alumno = {
    nombre: 'Angelina',
    apellidos: 'Jolie',
    email: 'angelina@gmail.com'
  };


  constructor() { }
```

```
  guardar( miFormulario: NgForm) {
    console.log('Submit');
    console.log( miFormulario );
    console.log ( this.alumno);
  }

}
```

Cómo se puede ver, la mayoría del código está en el html y no tenemos casi nada en el ts.

Paso a comentar el html:

En primer lugar hemos puesto (ngSubmit), para que cuando se pulse al botón “Guardar”, llame a la función Guardar y le pase como parámetro el formulario completo, al que le hemos hecho una referencia local con #formulario. Además, le hemos puesto el novalidate para que las reglas del html 5 no las pase .

Hemos incorporado en el input, un concepto que no había comentado es la directiva de clase ngClass, que lo que hace es una validación en la que incorporará la clase is-invalid si se cumple que el nombre tiene errores y además el campo ha sido tocado.

La clase is-invalid es de bootstrap y se encuantra buscando en la página de bootstrap validators.

Debajo, hemos puesto unos div, que se activan si hay errores. En estos div, se pueden ver las condiciones que se han puesto. Además, hemos hecho también referencias locales con # a cada campo, para que nos sea cómodo buscar los errores o validaciones de esos campos.

También, hemos puesto en los avisos de los erroes la clase invalid-feedbach, según la documentación de bootstrap.

Luego, en el correo, hemos incorporado una validación con una expresión regular.

El botón, tiene también un disabled manejado por angular, por eso lo de meterlo entre corchetes, en el que se activará o no según el formulario sea válido.

Aunque todo esto es un poco lioso, espero haberme explicado más o menos.

En cualquier caso, me podéis consultar.

La pega que tiene este tipo de formularios es que la parte del html es muy grande y la parte del ts no.

Ahora, vamos a ver el mismo formulario, pero haremos toda la parte con el ts.

Formularios por código

En esta parte, vamos a poner nuestro código base de formulario html en el componente formulario – código.html

En este tipo de funcionamiento de formularios, hay que incluir en el app.module.ts la importación:

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

Y luego en los imports

```
  imports: [
    BrowserModule,
    APPROUTING,
    FormsModule,
    ReactiveFormsModule
```

Una vez esto, voy a crear un formulario sencillo y os lo explico. Creo que es mejor y así pueden probar poniendo y quitando cosas.

El html lo he dejado así:

```
<h4>Formularios <small>dentro el html</small></h4>
<hr>
<form (ngSubmit)="guardar(formulario)" #formulario="ngForm" novalidate>

    <div>

        <div class="form-group row">
            <label for="inputNombre" class="col-2 col-form-
label">Nombre</label>
            <div class="col-8">
```

```
                <input class="form-control" [ngClass]="{ 'is-invalid' :
nombre.errors && nombre.touched} " type="text" placeholder="Nombre"
id="inputNombre" name="inputNombre" [(ngModel)]="alumno.nombre" required
minlength="3" #nombre="ngModel">
                <div *ngIf="nombre.errors?.required" class="invalid-
feedback">
                    El nombre es necesario
                </div>
                <div *ngIf="nombre.errors?.minlength " class="invalid-
feedback">
                    El nombre tiene que tener al menos {{
nombre.errors.minlength.requiredLength }} caracteres
                </div>
            </div>
        </div>

        <div class="form-group row">
            <label for="inputApellido" class="col-2 col-form-
label">Apellido</label>
            <div class="col-8">

                <input class="form-control" [ngClass]="{ 'is-invalid' :
apellidos.errors } " type="text" id="inputApellido"
placeholder="Apellido" name="apellido" [(ngModel)]="alumno.apellidos"
#apellidos="ngModel">

            </div>
        </div>

    </div>

    <div class="form-group row">
        <label for="inputCorreo" class="col-2 col-form-
label">Correo</label>
        <div class="col-md-8">

            <input class="form-control" [ngClass]="{ 'is-invalid' :
correo.errors } " type="email" id="inputCorreo" placeholder="Correo
electrónico" name="email" [(ngModel)]="alumno.email" pattern="[a-z0-
9._%+-]+@[a-z0-9.-]+\.[a-z]{2,3}$" #correo="ngModel">
            <div *ngIf="correo.errors?.pattern" class="invalid-feedback">
                El correo tiene que estar bien
```

```
            </div>
        </div>
    </div>



    <div class="form-group row">
        <label class="col-2 col-form-label"> </label>
        <div class="input-group col-md-8">
            <button type="submit" [disabled]="!formulario.valid"
class="btn btn-outline-primary">
        Guardar
      </button>
        </div>
    </div>

</form>
```

El código ts, lo he dejado así.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';



@Component({
  selector: 'app-formulario-codigo',
  templateUrl: './formulario-codigo.component.html',
  styles: []
})
export class FormularioCodigoComponent  {


  formulario: FormGroup;


  alumno = {
    nombre: 'Richard',
    apellidos: 'Gere',
    correo: 'rgere@gmail.com'
```

```
};


constructor() {

  this.formulario = new FormGroup({
    'nombre': new FormControl('', [
      Validators.required,
      Validators.minLength(3),
      this.noAlumnoPedro
    ]),
    'apellidos': new FormControl('', Validators.required),
    'correo': new FormControl('', [
      Validators.required,
      Validators.pattern('[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,3}$'),
     ],
     this.existeCorreo)
  });

  this.formulario.setValue(this.alumno);

}


guardar() {
  console.log ( 'Submit ');
  console.log ( this.formulario.value );
  console.log ( this.formulario.valid );
  console.log ( this.formulario );

  // Reseteamos el formulario
  // Se queda limpio, y sin tocar

  // this.formulario.reset();


}

// Validador que devuelve un par de valores

noAlumnoPedro ( control: FormControl ): { [s: string ]: boolean } {

  if ( control.value === 'Pedro') {
```

```
      return {
        nopedro: true
      };
    }
      return null;
  }

  existeCorreo( control: FormControl ) : Promise<any> {

    return  new Promise( (resolve, reject ) => {

      setTimeout( () => {
         if ( control.value === 'carlos@gmail.com') {
           resolve( {
             existe: true
            } );
         } else {
           resolve ( null );
         }
      }, 3000);
    } ) ;

  }


}
```

Paso a explicar brevemente lo que hemos hecho.

En el html.

Hemos puesto el formulario base y le hemos configurado el submit que llamará a una función del ts. Además, hemos puesto que el formulario se conecte al formulario que hemos declarado en el ts.

Por otra parte, hemos conectado todos los campos del formulario con el ts.

Se puede ver que hay dos maneras de hacerlo, con .get y con controls['xxxx']

También hemos incorporado algo de bootstrap para que los errores de validación se muestren de forma adecuada.

En la parte de abajo del formulario, hemos puesto un {{ formulario.status }}, que nos informa si hay algo pendiente por el tema de una validación asíncrona.

En la parte del código.

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

Hemos importado lo necesario para crear un formulario y las validaciones.

Por eso, hemos creado un formulario llamado formulario, que es de tipo FormGroup. Conviene poner los tipos aunque funcione con tipo any, ya que de esa forma el VSC, nos ayuda con el código.

Luego, hemos configurado nuestro formulario, con los controles y las validaciones.

Como **se puede ver, cada formControl, tiene un parámetro con el valor por defecto, luego un array de las validaciones y por último otro array de valicaciones asíncronas.**

Hemos puesto validadores que vienen por defecto, como el required o el minlength y luego hemos creado el validador personalizado noAlumnoPedro y el validador asíncrono existeCorreo. Este validador, devuelve una promesa (Podría ser un observable), que hace una comprobación y espera 3 segundos

Por eso en el html, hemos puesto el formulario.status, para ir viendo como lo hace.

No hemos puesto un validador en el botón para que se desactive cuando el formulario no es válido, para poder dar al botón y ver en la consola todas las propiedades del formulario.

Supongo que eso ha sido un poco lío, pero os invito a “cacharrear” con esto.

Por último, subimos todos los cambios al repositorio.

git status

git add .

git status (tiene que salir verde)

git commit –m “Pipes y Formularios”

git push origin master

Gracias.

(En el próximo document, haremos el backend con spring boot conun CRUD de una tabla alumnus, por ejemplo y probaremos todo con Postman)