Angular 2+

Angular es un framework para aplicaciones web desarrollado en **TypeScript**, de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página. Su objetivo es aumentar las aplicaciones basadas en navegador con capacidad de Modelo Vista Controlador (MVC), en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles.

Typescript

Typesript es Javascript con algunos añadidos y fue desarrollado por Microsoft. Aunque se puede trabajar en Angular con Javascript, lo habitual es emplear Typescript. La extensión predeterminada para Typescript es ts. Básicamente lo que se hace es añadir ciertas funcionalidades a Javascript en su versión 6. Dado que los navegadores no son capaces de interpretar typescript, es necesario traducir typescript a Javascript. A este proceso se le llama transpilación (es una compilación de un lenguaje a otro de similar nivel) y la herramienta encargada de hacerlo en Angular es **Webpack**.

La transpilación puede hacerse a la versión 6 de Javascript o, más habitualmente, a la versión 5 (está mucho mejor soportada por todos los navegadores). En la fase de creación de la aplicación se realizará (entre otras tareas) toda la traducción del código typescript a javascript.

Instalación de Angular

Para instalar (https://angular.io/guide/quickstart) Angular, el primer paso es instalar node.js y npm. Node.js es un programa capaz de ejecutar código Javascript (una especie de servidor web). Npm es un sistema de descarga e instalación de paquetes.

Descargamos Node.js de la página (https://nodejs.org/en/download/). Esto también nos descarga **npm**. Ahora abrimos una ventana en el navegador y escribimos:

```
npm install -g @angular/cli
```

que instala Angular de manera **global** (accesible desde cualquier sitio).

Creación de una aplicación Angular

Una vez instalado Angular, debemos situarnos en la carpeta **padre** en la que queremos instalar la aplicación (cada aplicación de Angular dispone de su propia carpeta). Para crear una aplicación:

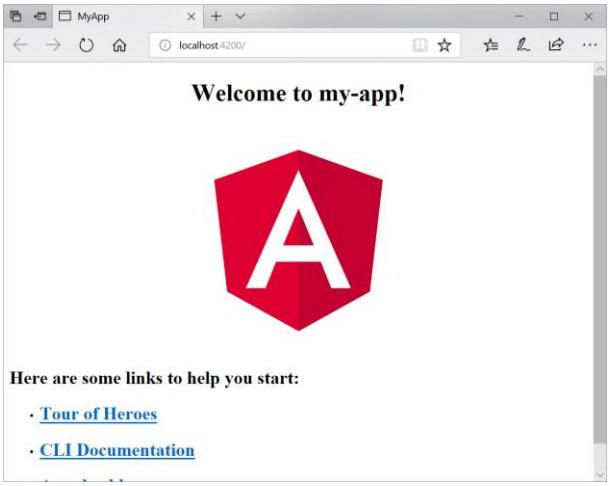
```
ng new my-app
```

Se descargarán una serie de paquetes y se creará una estructura con los ficheros y directorios básicos para una aplicación Angular básica.

Una vez creada la aplicación podemos ejecutarla a través de Node.js con los siguientes comandos:

```
cd my-app
ng serve --open
```

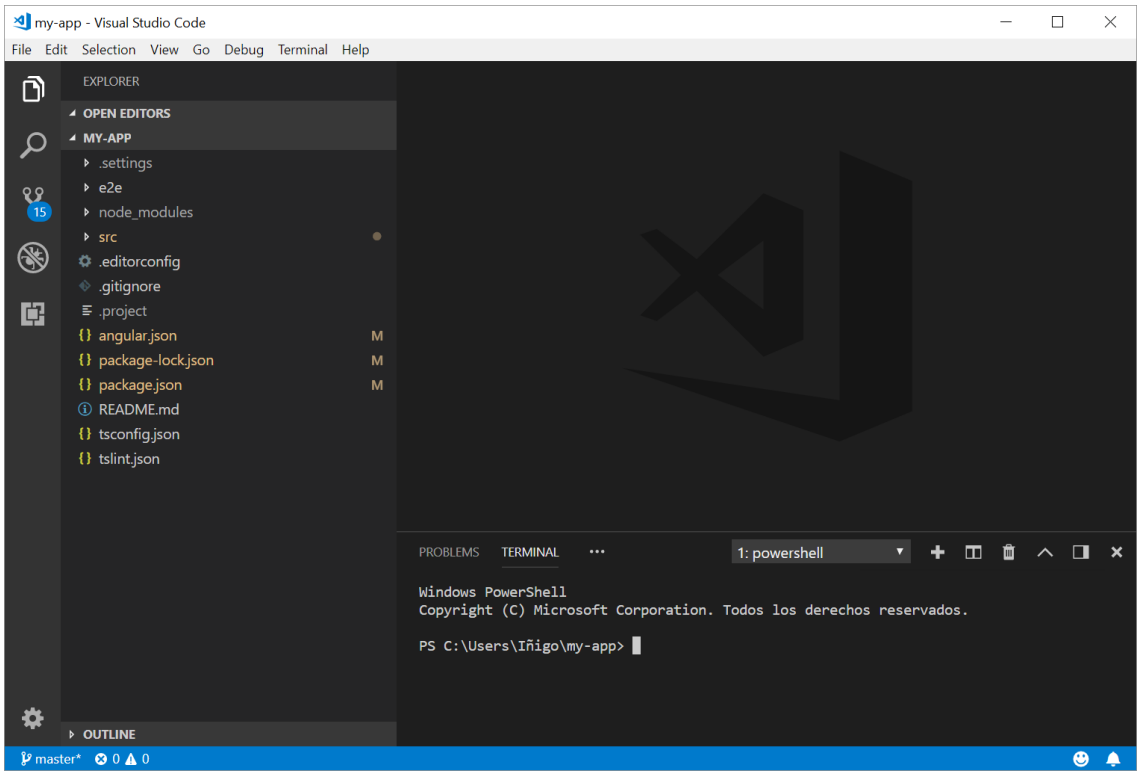
La opción **--open** (o **-o**) abre una ventana del navegador con la aplicación en ejecución.



Instalación de un IDE para Angular

Vamos a emplear como IDE, **Visual Studio Code** (https://code.visualstudio.com/) dado que tiene un buen soporte para **Typescipt**.

Una vez instalado el IDE podemos emplear **Angular CLI**, seleccionando **Terminal → New terminal** y abrir la capeta del proyecto con **File → Open folder**. El resultado será similar al siguiente:



Estructura de la aplicación

Si miramos la estructura de la aplicación podemos ver, entre otros, las siguientes capetas:

- **raíz**: En la carpeta raíz de la aplicación se encuentran diferentes ficheros de configuración. Entre ellos:

    - **angular.json**: Contiene directivas de configuración de la aplicación (como compilar, dónde se encuentran los diferentes elementos…)
    - **package.json, package-lock.json**: Información sobre los paquetes que contienen los módulos que podemos necesitar para desarrollar nuestra aplicación. Es información para el **npm** que es el gestor de paquetes que se encarga de descargar y gestionar estos paquetes y sus dependencias
    - **tsconfig.json**: Configuración para **Webpack** el transpilador de Typescript incluido en Angular. Un elemento importante es **target** que indica que se debe generar Javascript vesión 5.
    - **tslint.json**: Configuración para TSLint que es una herramienta que comprueba que el código Typescript sigue unas ciertas normas
    - **index.html**: El fichero HTML base de la aplicación
    - **main.ts**: El código principal del proyecto
- **e2e**: Carpeta con ficheros para la depuración de la aplicación
- **node_modules**: En esta capeta se encuentran todos los módulos descargados desde Internet por npm
- **src**: La carpeta donde se encuentra el código de nuestra aplicación. Dentro de ella destacan:
    - **app**: La carpeta con el código de nuestra aplicación base:
        - **app.component.css**: El CSS del componente raíz
        - **app.component.html**: El HTML. La vista en el patrón MVC
        - **app.component.ts**: El Typescript asociado al componente (su funcionalidad). Sería una mezcla entre el controlador y el modelo en el patrón MVC
        - **app.component.specs.ts**: Typescript para la realización automática de tests del componente
        - **app.module.ts**: El módulo base de la aplicación encargado, entre otras cosas, de definir el componente principal de la misma
    - **assets**: Carpeta donde situar los recursos estáticos de la aplicación (imágenes, videos, fuentes…)
    - **environments**: Contiene información sobre el entono de la aplicación. En el fichero **environment.ts** se emplea cuando la aplicación está en desarrollo y **environment.prod.ts** cuando está en producción. En esos ficheros, por ejemplo, se pueden definir variables con diferentes valores para cada entorno. Para indicar que queremos construir la aplicación con el entorno producción emplearíamos **ng build—env=prod** (si añadimos el parámetro—**prod** además optimizará la aplicación para que pese menos)

Componentes en Angular

Los componentes son la base del desarrollo en Angular. Un componente es un elemento que puede insertarse, a través de un tag personalizado, en el HTML. Cuando se sirve la aplicación, Angular convierte ese tag en HTML + Javascript para realizar la funcionalidad deseada.

La idea es que nuestra aplicación esté formada por diferentes componentes que generan un árbol de componentes (a partir de un componente **raíz** que es el único que aparece en el fichero **index.html**).

Un componente, básicamente, está formado por dos elementos:

- **HTML + CSS**: Conforman la vista del componente

- **Typescript / Javascript**: Especifican el comportamiento del mismo

Es habitual que los componentes se definan en varios ficheros como hemos visto en el ejemplo anterior. Podemos crear una estructura básica para un componente mediante Angular CLI:

```
ng generate component componente
```

Veremos este proceso más en detalle más adelante.

NOTA: Además de los componentes, mediante el comando generate podemos crear **directivas, rutas, pipes, servicios, clases, interfaces** y **enumeraciones**. El comando **generate** crea los elementos siguiendo las guías de estilo de Angular lo que facilita el trabajo de los programadores cuando acceden a proyectos Angular no desarrollados por ellos.

Los componentes se agrupan en módulos (**ngModules**).

Ejecución de la aplicación de ejemplo

Toda aplicación Angular dispone de un **módulo raíz** (habitualmente llamado **AppModule**) que es el que proporciona el mecanismo para ejecutar la aplicación. Este módulo se carga en el fichero **main.ts** que es el punto de entrada a la aplicación y se encarga de cargar el módulo raíz:

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

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

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

import { environment } from './environments/environment';

if (environment.production) {

enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule)

.catch(err => console.log(err));
```

Como se puede ver tenemos los siguientes elementos:

- Una serie de **importaciones** de los elementos que vamos a emplear en el fichero (entre ellos, el **módulo raíz** que está en **app/app.module**. Como podemos ver no se indica la extensión ya que, ahora es **ts** pero acabará siendo **js**.
- La indicación de cuál es el módulo a ejecutar para iniciar todo el proceso de carga y ejecución de la aplicación (**bootstrapModule**)

NOTA: Angular no sólo se emplea en páginas web sino que la idea es que sea más versátil. Es por eso que para iniciar el proceso necesitamos indicar en qué plataforma vamos a trabajar (`platformBrowserDynamic()`). El método `catch` se ejecuta si en el método al que se asocia se ha producido un error (existe otro método `then` que se ejecutaría cuando finalice el método al que está asociado)

El siguiente paso es cargar el módulo ejecutando **app.module.ts**:

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

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

```
import { AppComponent } from './app.component';
@NgModule({
declarations: [
AppComponent
  ],
imports: [
BrowserModule
  ],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

- Se definen las importaciones de los elementos (entre ellos el componente raíz **AppComponent**)
- Luego tenemos el **decorador @NgModule** que indica que la clase que le sigue es un módulo Angular. El decorador tiene los siguientes elementos:
    - **declarations**: Una matriz con **los componentes del módulo**
    - **imports**: Una matriz con los módulos requeridos por este módulo para trabajar (en este caso para poder trabajar con servidores web, hay otros para formularios, enrutamiento, etc)
    - **providers**: Proveedores de servicios
    - **bootstrap**: El componente **raíz** de la aplicación
- Por último, tenemos la clase del módulo. La palabra **exports** indica que esta clase **puede ser importada** desde otros ficheros Typescript (en nuestro caso desde **main.ts**)

NOTA: Los elementos Angular tienen siempre este formato. Un decorador que define qué tipo de elemento es y sus propiedades y la clase asociada donde especificamos el código asociado al elemento.

Como vemos el módulo carga el componente raíz de la aplicación (**AppComponent**). Si miramos **app.component.ts**:

```
import { Component } from '@angular/core';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    title = 'my-app';
}
```

- El decorador **@Component** tiene los siguientes elementos:
    - **selector**: Identifica el **tag** a emplear en el HTML para **insertar este componente**. En este caso es **app-root**. Si abrimos el fichero **index.html** veremos:

```
<body>
<app-root></app-root>
</body>
```

    - **templateUrl**: El fichero donde se especifica el HTML a mostrar en el tag (lo vemos más adelante)
    - **styleUrls**: Una matriz con los ficheros CSS asociados al componente
- La clase que representa el componente. En el ejemplo, la clase únicamente tiene una **propiedad** que es **title**.

Si miramos el fichero **app.component.html** ahí encontraremos el código HTML que se inserta en el tag **<app-root>** del fichero **index.html**:

```
<!--The content below is only a placeholder and can be replaced.-->
<div style="text-align:center">
<h1>
  Welcome to {{ title }}!
</h1>
<img width="300" alt="Angular Logo"
src="
vMjAwMC9zdmciIHZpZXdCb3g9IjAgMCAyNTAgMjUwIj4KICAgIDxwYXRoIGZpbGw9IiNER
DAwMzEiIGQ9Ik0xMjUgMzBMMzEuOSA2My4ybDE0LjIgMTIzLjFMMTI1IDIzMGw3OC45LTQ
zLjcgMTQuMi0xMjMuMXoiIC8+CiAgICA8cGF0aCBmaWxsPSIjQzMwMDJGIiBkPSJNMTI1I
DMwdjIyLjItLjFWMjMwbDc4LjktNDMuNyAxNC4yLTEyMy4xTDEyNSAzMHoiIC8+CiAgICA
8cGF0aCAgZmlsbD0iI0ZGRkZGRiIgZD0iTTEyNSA1Mi4xTDY2LjggMTgyLjZoMjEuN2wxM
S43LTI5LjJoNDkuNGwxMS43IDI5LjJIMTgzTDEyNSA1Mi4xem0xNyA4My4zaC0zNGwxNy0
0MC45IDE3IDQwLjl6IiAvPgogIDwvc3ZnPg==">
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener"
href="https://angular.io/tutorial">Tour of Heroes</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener"
href="https://github.com/angular/angular-cli/wiki">CLI
Documentation</a></h2>
  </li>
```

```
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
  </li>
</ul>
```

Como se ve es código HTML normal junto con elementos Angular. El texto **`{{ title }}`** es una **expresión Angular** que se evalúa y muestra su resultado. En nuestro caso, hay que tener en cuenta que el HTML está asociado a la clase **AppComponent** por lo que la expresión **title** hace referencia a la **propiedad title** de la clase. Por tanto, al visualizar la página, se mostrará el texto de dicha propiedad.

Podemos probar a modificar el HTML del componente y a cambiar el valor de la propiedad **title**. Al servir la aplicación los cambios se **actualizan dinámicamente en el navegador**.

NOTA: Comprobar la codificación del fichero en AngularIDE en las propiedades de la carpeta y cambiarlo a UTF-8 si es necesario.

Usar Bootstrap

Bootstrap es una librería CSS con Javascript que permite el diseño de páginas HTML. Incluye una serie de componentes que permiten diseñar aplicaciones modernas. Para incorporarlo a nuestra aplicación, instalamos Bootstrap y jquery mediante npm:

```
npm install bootstrap
npm install jquery
```

Y, tras ello, modificamos el fichero **angular.json** para que use el CSS y el JS de Bootstrap:

```
"styles": [
"src/styles.css",
"node_modules/bootstrap/dist/css/bootstrap.css"
            ],
"scripts": [
"node_modules/jquery/dist/jquery.js",
"node_modules/bootstrap/dist/js/bootstrap.js"
             ]
```

Si ahora recargamos de nuevo la aplicación veremos que ha cambiado la fuente y el espaciado y ya podemos emplear los componentes y clases de Bootstrap.

Creación y uso de un componente

Vamos a añadir un nuevo componente a nuestra aplicación. El componente va a mostrar información sobre un empleado y va a insertarse en el tag **<empleado>**.

En primer lugar empleamos Angular CLI para generar los ficheros necesarios para definir y usar el componente:

```
ng generate component Empleado
```

La salida

```
CREATE src/app/empleado/empleado.component.html (27 bytes)

CREATE src/app/empleado/empleado.component.spec.ts (642 bytes)

CREATE src/app/empleado/empleado.component.ts (277 bytes)

CREATE src/app/empleado/empleado.component.css (0 bytes)

UPDATE src/app/app.module.ts (404 bytes)
```

Como se ve el proceso crea una carpeta **empleado** dentro del módulo raíz (**app**) y dentro diferentes ficheros (que ya hemos comentado previamente). En particular, nos interesan los siguientes ficheros:

empleado.component.html: Actualmente simplemente tiene el html para mostrar que funciona el componente

empleado.component.ts: La definición y funcionalidad del componente:

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

@Component({

    selector: 'app-empleado',

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

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

})

export class EmpleadoComponent implements OnInit {

    constructor() { }

    ngOnInit() {

   }

}
```

- Se importan los elementos a usar en el componente (Component, OnInit)
- Mediante el decorador Component se definen las propiedades del componente:
    - **selector**: el tag a emplear para invocar el componente
    - **templateUrl**: la URL del fichero HTML (el visto en el apartado anterior)
    - **styleUrls**: Una matriz con los estilos a emplear
- Tras ello se declara la clase propiamente dicha (**EmpleadoComponent**):
    - La clase se **exporta** para poderse emplear desde otros componentes (en nuestro caso la importaremos en **AppComponent**
    - Se define un **constructor** sin argumentos que se ejecutará **cuando se crea** un componente
    - Se implementa el interfaz **OnInit** que implica redefinir el método **ngOnInit** que será **invocado después de crear** el componente

NOTA: Se supone que la inicialización básica del componente de debería hacer en el constructor, mientras que inicializaciones complejas debieran definirse en el método **ngOnInit**. Por ejemplo, asignar valores estáticos a un componente se haría en el constructor, pero recuperar dichos datos de un servicio debiera hacerse en ngOnInit.

Para que nuestro componente pueda ser cargado por Angular hay que definirlo en el fichero **app.modules.ts** (eso lo hace el comando **ng** automáticamente):

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { EmpleadoComponent } from './empleado/empleado.component';
@NgModule({
  declarations: [
    AppComponent,
    EmpleadoComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Una vez definido el componente podemos emplearlo dentro de otros componentes creando el árbol de componentes que define nuestra aplicación. Por ejemplo, modificamos **app.component.html** para que haga uso de dicho componente:

```
<div class="container">
<h1>{{title}}</h1>
<app-empleado></app-empleado>
</div>
```

Si servimos la página:

Mi Aplicación

Empleado works!

Parámetros de ng generate para componentes

El comando ng generate admite las siguientes opciones:

- **spec (true|false)** : Indica si se debe crear o no el fichero de tests (componente.spec.ts)
- **flat (true|false)**: Indica si se debe (false) o no (true) crear un directorio para el componente
- **inline-style (true|false)**: Indica si los estilos se definen en el propio componente o en un fichero css externo

- **prefix ('texto'|false)**: Indica el prefijo a emplear (o si emplear) en el selector del componente
- **inline-template (true|false)**: Indica si la plantilla HTML se define en el propio componente o en un fichero html externo

Acceso desde la vista a propiedades y métodos del componente

A la hora de renderizar la vista del componente, necesitaremos mostrar las propiedades o acceder a los métodos de la misma. En la práctica dichos valores los podemos emplear para:

- Mostrarlos como información. Para ello emplearemos expresiones (**{{}}**). A este proceso se la llama **interpolación**
- Utilizarlos en los atributos HTML. Para ello emplearemos el **binding** (**[]**).

Vemos un ejemplo.

Vamos a definir un par de propiedades y un método en la clase **EmpleadoComponent**. Dado que empleamos Typescript podemos indicar el tipo de la propiedad (**string, number, boolean**…). Recordemos que no existen los tipos en Javascript.

```
export class EmpleadoComponent implements OnInit {

    private nombre: string = 'Juan';

    private apellido = 'López';

    private departamento = 'Administración';

    constructor(){

    }

    ngOnInit() {

    }


    nombreCompleto(): string {

      return this.nombre +  ' ' + this.apellido;

  }

}
```

En este caso hemos declarado tres propiedades de tipo **private** (Typescript) y en una de ellas, hemos indicado explícitamente que es de tipo **string** (Typescript). Si no lo indicamos expresamente (como en las otras dos) Typescript **inferirá** el tipo de datos en función del valor asignado (es decir, apellido y departamento se asumen que son de tipo **string**). Cualquier intento de asignar un valor que no sea una cadena provocará un **error** en tiempo de compilación

Modificamos el fichero **empleado.component.html** :

```
<div class="card" style="width: 20em;">

  <div class="card-body">

    <h5 class="card-title">{{nombreCompleto()}}</h5>
```

```
    <p     class="card-text">{{nombre.toLowerCase()}}     trabaja     en
{{departamento}}.</p>

  </div>

</div>
```

La salida:

Como ya hemos comentado, la sintaxis **{{}}** permite indicar expresiones Typescript que serán evaluadas y sustituidas por su resultado. Esto incluye propiedades, llamadas a métodos y cualquier tipo de expresión válida. **NOTA**: El método **toLowerCase** es propio de Javascipt y convierte una cadena a minúsculas.

En ocasiones, lo que queremos es emplear las propiedades y/o los métodos de la clase para modificar los **atributos HTML** (en realidad, las **propiedades del DOM**). En estos casos deberemos emplear la sintaxis:

```
[atributo]="expresiónTypescript"
```

Por ejemplo, supongamos que queremos que el cuerpo del componente card de un empleado u otro pueda mostrarse o no en función de una propiedad del componente. Vamos a incluir la propiedad en la clase **EmpleadoComponente** del fichero **empleado.componente.ts**:

```
private ocultar=false;
```

Ahora modificamos el empleado.componente.html:

```
<p     class="card-text"     [hidden]="ocultar">{{nombre.toLowerCase()}}
trabaja en {{departamento}}.</p>
```

La salida:

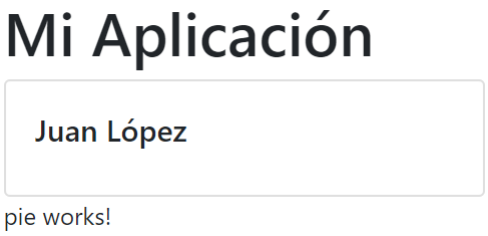
NOTA: Como se ha dicho este mecanismo permite modificar las **propiedades DOM** y **no los atributos HTML**. Esto es importante porque, por ejemplo, la propiedad **hidden del DOM muestra / oculta** el elemento cuando su valor es **false / true**. Sin embargo, el **atributo hidden de HTML** oculta el elemento **cuando existe independientemente del valor que tenga** asignado (o incluso si no tiene asignado ningún valor).

NOTA2: Dado que, en Javascript, cualquier valor se considera **true** y ausencia de valor (**0**, **''**, **undefined, null, false** y **NaN**) se considera **false**.

ACTIVIDAD

Queremos definir un componente básico que va a ser el pie de la aplicación. Dicho componente va a estar definido en la clase **Pie**. Los pasos a dar son:

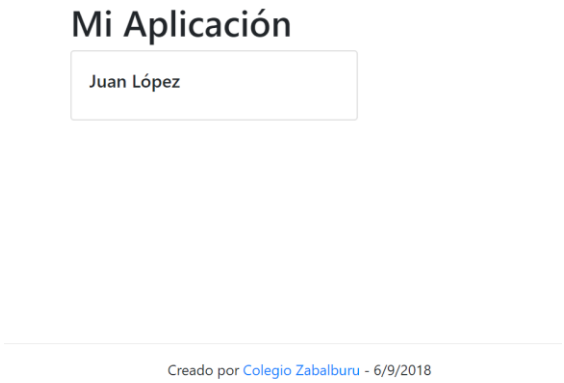
- Abrir un nuevo terminal
- **Generar** el nuevo componente
- Conseguir que se muestre en la página de inicio. Debería verse lo siguiente:



- **Añadir** propiedades al componente para que pueda almacenar el nombre de una **empresa**, la **url** de la misma y la **fecha** de hoy. Declarar las variables **private** y de tipo **string** las dos primeras y **Date** la de fecha
- **Modificar** el constructor de la clase para que añada el código necesario para asignar **Colegio Zabalburu** a la **empresa**, **http://www.zabalburu.net** a la **url** y **new Date()** a la fecha (crea un objeto fecha con el día de hoy)
- **Añadir** un método a la clase **getFechaFormateada** que retorne un **String** con la **fecha en formato local** (buscar información en Internet sobre cómo mostrar un **Date** en **Javascript** con dicho formato)
- **Modificar** el CSS del componente para que incluya el siguiente estilo

```
.footer {
position: fixed;
left: 0;
bottom: 2em;
width: 100%;
text-align: center;
}
```

- **Modificar** el HTML del componente para conseguir que se vea de la siguiente forma (empleando las propiedades y el método anteriormente creado):



Colegio Zabalburu es un enlace a la URL del componente.

Enlazando eventos

Podemos asociar los **eventos** emitidos por los elementos del DOM mediante la sintaxis **(evento)**. Por ejemplo, supongamos que queremos que, cuando el usuario haga clic en un botón se **muestre / oculte** el detalle del empleado.

Definimos en el html del componente:

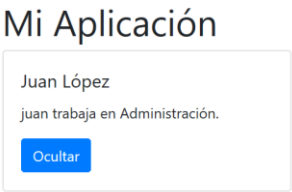
```
<div class="card" style="width: 20em;">

<div class="card-body">

<h5 class="card-title">{{nombreCompleto()}}</h5>

<p class="card-text" [hidden]="ocultar">{{nombre.toLowerCase()}} trabaja en {{departamento}}.</p>

<button class='btn btn-primary' (click) = 'changeVisibility()'>{{(ocultar)?'Ver':'Ocultar'}}</button>

</div>

</div>
```

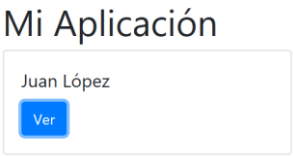
- **(click)** se asocia con el **evento onClick** del botón
- **changeVisibility()** es una llamada a un método del componente (lo veremos ahora)
- **{{(ocultar)?'Ver':'Ocultar'}}** es una expresión que retorna **Ver** u **Ocultar** en función del estado de la propiedad **ocultar**

Ahora modificamos el typescript del empleado:

```
@Component({

selector: 'app-empleado',

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

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

})

export class EmpleadoComponent implements OnInit {

changeVisibility():void{

this.ocultar = !this.ocultar;

  }

}
```

Si cargamos la aplicación:





Input Binding

Ya hemos visto cómo podemos pasar información desde el modelo/controlador (Clase) a la vista (HTML). En ocasiones, sin embargo, necesitamos pasar **información desde la vista al controlador / modelo**.

Para ello basta con utilizar en las propiedades de la clase el decorador **@Input**. A las propiedades identificadas con ese decorador se les **puede asignar un valor desde la vista** especificando dicha **propiedad como un atributo del tag** (emplearemos la misma sintaxis que para asignar un valor a una propiedad del DOM **[propiedad]='expresión'**)

Por ejemplo, modificamos la clase **Empleado** para que sus propiedades sean de entrada (además de poner la propiedad **ocultar** a **false** si no lo está:

```
export class EmpleadoComponent implements OnInit {
@Input() private nombre: string = 'Juan';
@Input() private apellido = 'López';
@Input() private departamento = 'Administración';
@Input() private ocultar = false;
constructor(){
  }

ngOnInit() {
  }

nombreCompleto(): string {
return this.nombre +  ' ' + this.apellido;
  }

}
```

NOTA: Otra forma es indicar las propiedades de entrada en el decorador:

```
@Component({
selector: 'app-empleado',
templateUrl: './empleado.component.html',
styleUrls: ['./empleado.component.css'],
inputs: ['nombre','apellido','departamento','ocultar']
})
export class EmpleadoComponent implements OnInit {
private nombre: string = 'Juan';
private apellido = 'López';
```

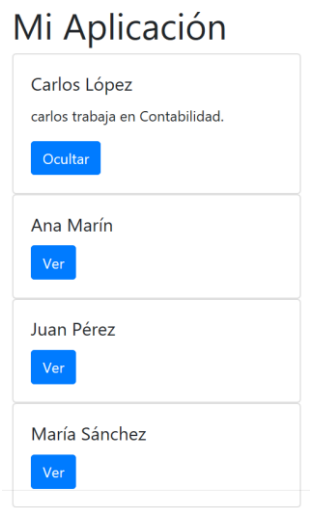
```
private departamento = 'Administración';

private ocultar = false;
```

Ahora disponemos en el tag **app-empleado** de las **propiedades** para ser usadas como **atributos**. Al ser un atributo, podemos acceder al mismo con **binding** (**[expresión]**). Por ejemplo, podemos modificar **app-component.html** para crear varios empleados con valores constantes:

```
<div class="container">

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

<app-empleado [nombre]="'Carlos'" [apellido]="'López'" [departamento]="'Contabilidad'" [ocultar]=false></app-empleado>

<app-empleado [nombre]="'Ana'" [apellido]="'Marín'" [departamento]="'Ventas'" [ocultar]="true"></app-empleado>

<app-empleado [nombre]="'Juan'" [apellido]="'Pérez'" [departamento]="'Ventas'" [ocultar]="true"></app-empleado>

<app-empleado [nombre]="'María'" [apellido]="'Sánchez'" [departamento]="'Contabilidad'" [ocultar]="true"></app-empleado>

<app-pie></app-pie>

</div>
```

La salida:

Como se puede ver las propiedades especificadas con **@Input()** pueden recibir valores de entrada.



Organizando la información

El siguiente paso que vamos a da es crear un componente **empleado-list** que es el que va a gestionar la lista de empleados. Adicionalmente, crearemos una **clase Empleado** para gestionar el modelo de la aplicación y emplearemos objetos de tipo Empleado para mostrar la información.

El primer paso es crear la clase **Empleado**. Para ello podemos usar el **CLI** de Angular:

```
ng g class Empleado
CREATE src/app/empleado.ts (26 bytes)
```

Si queremos poner la clase en un directorio específico:

```
ng g class modelo/Empleado
CREATE src/app/modelo/empleado.ts (26 bytes)
```

Con lo que se creará la clase en la capeta **modelo**.

Si miramos la clase veremos que es una clase **vacía** que únicamente tiene la etiqueta **export**. Añadimos el código de la clase:

```
export class Empleado {
private nombre:string;
private apellido:string;
private departamento:string;
private foto:string;

constructor(nombre, apellido,departamento,foto){
this.nombre = nombre;
this.apellido = apellido;
this.departamento = departamento;
this.foto = foto;
    }

public nombreCompleto():string{
return this.nombre + " " + this.apellido;
    }
}
```

También podemos declarar las propiedades a la vez que las utilizamos en el constructor simplemente indicando su visibilidad:

```
export class Empleado {
constructor(
private nombre:string,
private apellido:string,
private departamento:string,
private foto:string)
    {};
public nombreCompleto():string{
```

```
return this.nombre + " " + this.apellido;
    }
}
```

Como se puede ver, en este caso no es necesario declarar previamente los campos (se deducen del constructor)

NOTA: Si no hay que especificar ningún **método**, también podríamos declarar el Empleado como un **Interface**.

El siguiente paso es modificar el componente **empleado-component.ts** para que haga uso de un objeto de tipo Empleado:

```
import { Component, OnInit } from '@angular/core';
import {Input} from '@angular/core'
import { Empleado } from 'src/app/modelo/empleado';
@Component({
selector: 'app-empleado',
templateUrl: './empleado.component.html',
styleUrls: ['./empleado.component.css']
})
export class EmpleadoComponent implements OnInit {
@Input() private empleado:Empleado;
@Input() private ocultar = false;
constructor(){
this.empleado = new Empleado('Juan','Lopez','Contabilidad',"");
  }

ngOnInit() {
  }

changeVisibility():void{
this.ocultar = !this.ocultar;
  }

}
```

Como se puede ver, necesitamos **importar** la clase **Empleado**, **declaramos** una variable de tipo Empleado y eliminamos el método **nombreCompleto** (dado que ahora está en la clase). Además, en el constructor, hemos creado un empleado de prueba.

Modificamos el empleado.component.html:

```
<div class="card text-center mx-auto" style="width: 15em;">

<img class="card-img-left mx-auto mt-3 rounded-circle" style='width: 40%' src={{empleado.foto}}>

<div class="card-body">

<h5 class="card-title">{{empleado.nombreCompleto()}}</h5>

<p class="card-text" [hidden]="ocultar">{{empleado.nombre}} trabaja en {{empleado.departamento}}.</p>

<button class='btn btn-primary' (click) = 'changeVisibility()'>{{(ocultar)?'Ver':'Ocultar'}}</button>

</div>

</div>
```

Por último, cambiamos **app.component.html** para que sólo muestre un empleado:

```
<div class="container">

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

<app-empleado></app-empleado>

<app-pie></app-pie>

</div>
```

La salida:

Directiva *ngFor

El siguiente paso es crear el componente **lista-empleados**. Para ello:

```
ng g c lista-empleados

CREATE src/app/lista-empleados/lista-empleados.component.html (34 bytes)

CREATE src/app/lista-empleados/lista-empleados.component.spec.ts (685 bytes)

CREATE src/app/lista-empleados/lista-empleados.component.ts (304 bytes)

CREATE src/app/lista-empleados/lista-empleados.component.css (0 bytes)

UPDATE src/app/app.module.ts (590 bytes)
```

Modificamos de nuevo **app-component.html**:

```
<div class="container">
```

```
<h1>{{title}}</h1>
<app-lista-empleados></app-lista-empleados>
<app-pie></app-pie>
</div>
```

Y, si lo ejecutamos:

Mi Aplicación

lista-empleados works!

Si ahora modificáramos **lista-empleados.component.html** para que se muestre el empleado:

```
<app-empleado></app-empleado>
```

Obtendríamos el mismo resultado del paso anterior.

¿Cómo podemos hacer que el componente empleado muestre la información del empleado (o empleados) que queramos?

Deberemos pasar **un objeto Empleado** al componente. Para ello, necesitamos que el componente **lista-empleados** tenga una **lista de empleados**. En nuestro caso vamos a emplear una **matriz de empleados**.

En lista-empleados.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { Empleado } from '../modelo/empleado';
@Component({
selector: 'app-lista-empleados',
templateUrl: './lista-empleados.component.html',
styleUrls: ['./lista-empleados.component.css']
})
export class ListaEmpleadosComponent implements OnInit {
private empleados:Empleado[];
constructor() {
const url:string='https://randomuser.me/api/portraits';
this.empleados = [
new Empleado('Juan','López','Contabilidad',url+'/men/57.jpg'),
new Empleado('Ana','Miró','Ventas',url+'/women/12.jpg'),
new Empleado('Carlos','Simón','Contabilidad',url+'/men/1.jpg'),
new Empleado('Lucía','Sanchez','Ventas',url+'/women/4.jpg'),
new Empleado('Manuel','Gamez','Contabilidad',url+'/men/12.jpg')
    ];
  }
```

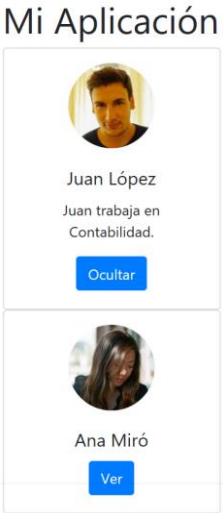
```
ngOnInit() {
  }


}
```

Como vemos, creamos una propiedad llamada **empleados** y le asignamos una **matriz de empleados**. Ahora podemos pasar los empleados de la matriz a la página html:

```
<app-empleado [empleado]='empleados[0]'></app-empleado>
<app-empleado [empleado]='empleados[1]' [ocultar]=true></app-empleado>
```

Observar que el componente empleado tiene **dos propiedades de entrada** que son **empleado** y **ocultar**.

La salida:



¿Cómo podemos automatizar este proceso? Mediante el uso de **directivas**. Una **directiva** permite asociar ciertos comportamientos a las etiquetas HTML. De hecho, los **componentes** son un tipo de directivas. Además tenemos las directivas **estructurales** (que modifican el DOM) como **ngFor** y **ngIf** y las **directivas de atributos** (que no modifican la estructura sino el aspecto) como **ngClass** y **ngStyle**.

En nuestro caso vamos a emplear la directiva **ngFor** cuya sintaxis es :

```
*ngFor let variable of coleccion
```

donde **colección** es una colección de elementos (por ejemplo una matriz) y **variable** es una variable que es **visible dentro del bloque donde se define el ngFor**.

Podemos coger el **índice** de la colección (empieza desde 0) con la expresión

```
*ngFor let variable of colección, let indice = index
```

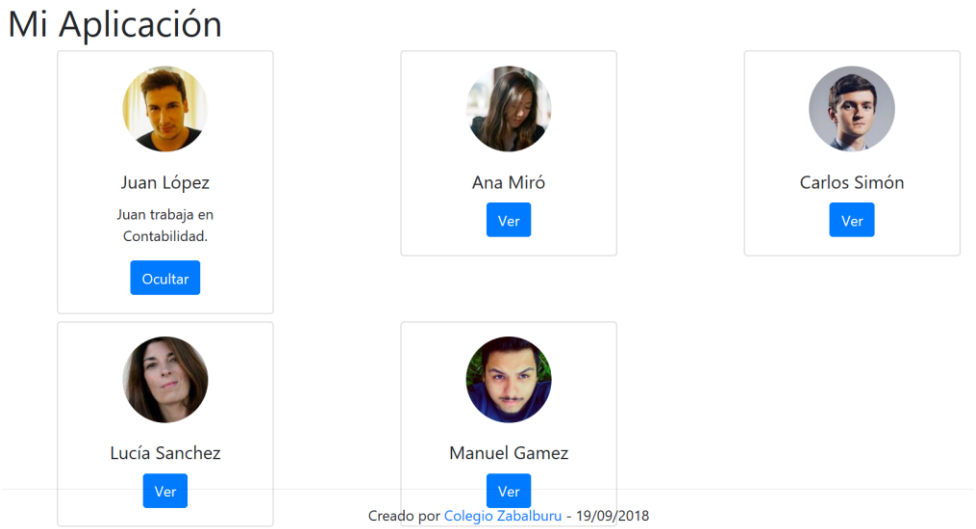
En este caso, la variable **índice** almacena el índice de cada elemento de la colección.

De la misma forma podemos obtener el **primer** y **último** índice (**first**, **last**) y si la fila es **par o impar** (**even, odd**).

Por ejemplo:

```
<div class='row'>
<app-empleado *ngFor="let empleado of empleados, let primero=first"
class='col-4 mb-2' [empleado]='empleado'
[ocultar]='(primero)?false:true'>
</app-empleado>
</div>
```

La salida



Directiva *ngIf

Otra directiva que podemos emplear es la directiva estructural **ngIf** que añade o no elementos al DOM. No hay que confundirla con **ocultar elementos** (propiedad **hidden**). Si se **oculta** un elemento, **sigue estando en el DOM**.

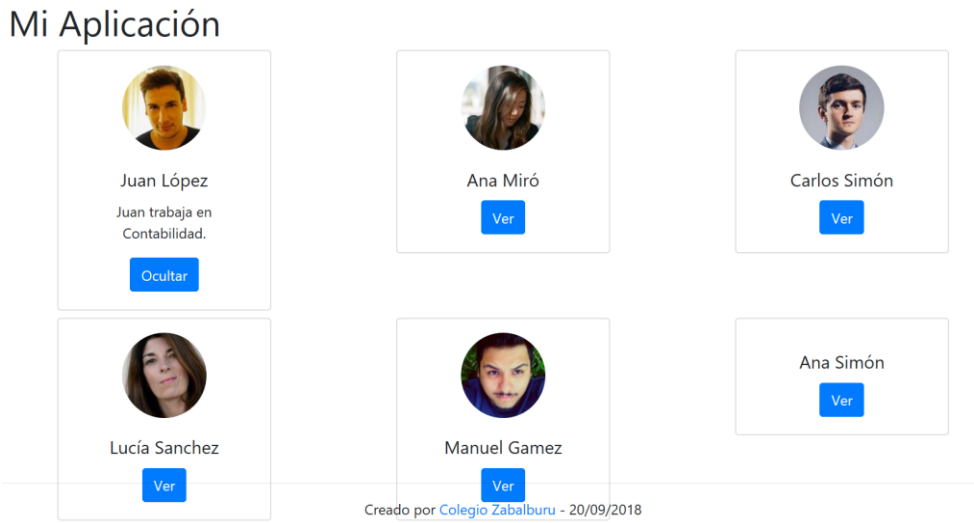
La sintaxis básica:

```
*ngIf='condición; else=marcador'
```

La directiva se coloca en el elemento que queremos **mostrar/ocultar**. Si se especifica el **else** se debe crear un elemento **ng-template** con ese marcador (**#marcador**) que es lo que se mostrará. Por ejemplo, supongamos que queremos que se muestre una imagen estándar para los empleados **sin foto**. Modificamos **lista-empleados.component.ts** para añadir uno a la matriz:

```
new Empleado('Ana','Simón','Contabilidad',null)
```

Si ejecutamos ahora:



Cambiamos empleado.component.html:

```
<img class="card-img-left mx-auto mt-3 rounded-circle" style='width: 40%'
```

```
*ngIf="!empleado.foto; else sifoto"
```

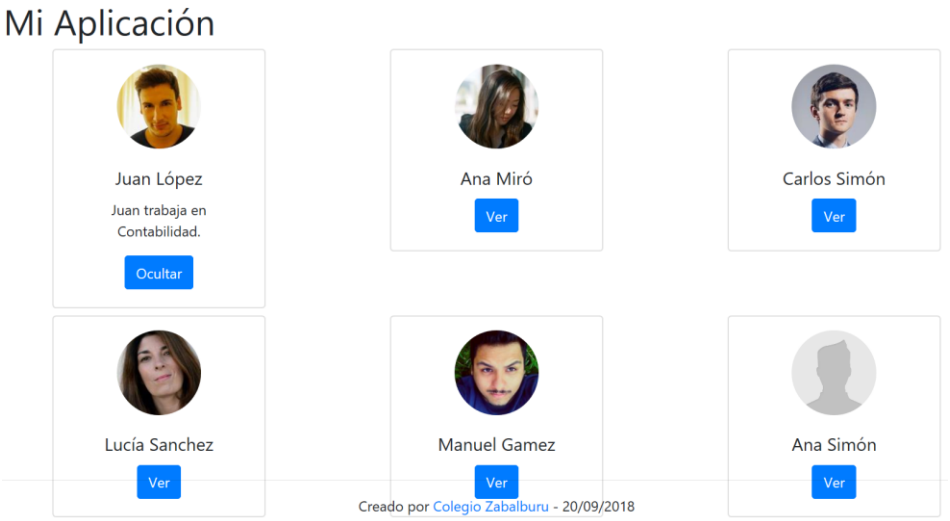
```
src="https://upload.wikimedia.org/wikipedia/commons/f/f4/Nelson_Neves_picuture.gif">
```

```
<ng-template #sifoto>
```

```
<img  class="card-img-left mx-auto mt-3 rounded-circle" style='width: 40%' src={{empleado.foto}}>
```

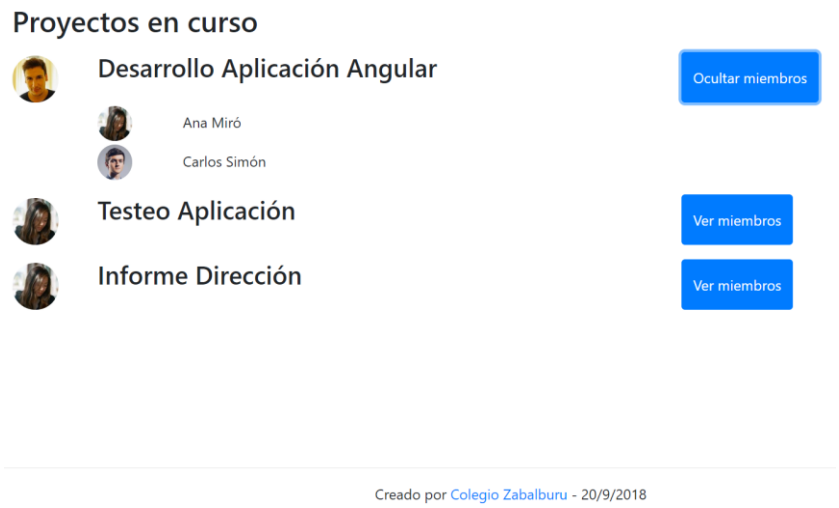
```
</ng-template>
```

La salida ahora:



Actividad 2

Se desea crear los componentes necesarios para mostrar una página similar a la siguiente:



En un principio se mostrará un listado con los proyectos (sin mostrar los miembros del proyecto)

Habrá que definir una **clase Proyecto** (dentro de **modelo**) que almacene la siguiente información:

- el **id** del proyecto
- el **nombre** del proyecto,
- el **empleado responsable** (un objeto **Empleado**)

una matriz de objetos Empleado con los miembros del proyecto (los empleado La aplicación deberá emplear tres componentes:

- Un componente **lista-proyectos** que almacenará **todos los proyectos** y mostrará, por cada uno, el componente adecuado (deberá almacenar, internamente, los empleados en una **matriz** y lo mismo con los **proyectos**.
- Un componente **proyecto** que mostrará la información de un proyecto y que hará uso de un componente para mostrar los miembros del proyecto
- Un componente **miembro-proyecto** que mostrará una fila con la información de cada miembro del proyecto.

Output Binding – Eventos en componentes

De la misma forma que un componente puede recibir información del exterior (mediante el decorador **@Input**) también puede informar a otros componentes de su estado mediante la **emisión de eventos**. Básicamente lo que se hace es generar un **evento personalizado** que es lanzado cuando sucede algo.

Los componentes interesados en dicho evento lo registran del mismo modo que lo harían con un evento DOM (mediante **(evento)**).

Por ejemplo, supongamos que, en nuestra aplicación de proyectos, queremos dar la posibilidad de **quitar** un empleado de un proyecto. La cuestión es:

- El botón de **quitar** el empleado del proyecto debería estar en **miembro-proyecto-component**

Sin embargo, los **datos de los empleados del proyecto** los tenemos en **lista-proyectos** Para lanzar un evento, tenemos que dar los siguientes pasos:

- Declarar una **propiedad** con el decorador **Output** y asignarle un **EventEmitter**

- Emplear el método **emit** sobre dicha **propiedad** cuando queramos lanzar el evento.

Los pasos a dar son los siguientes:

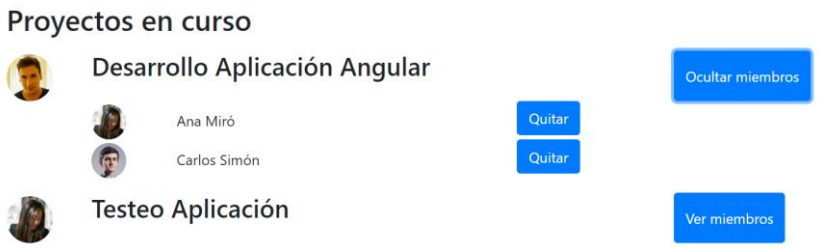
- Añadimos a la clase **Proyecto** un **método** para **quitar** un empleado **en base a su id**:

```
public quitarEmpleado(idEmpleado:number){
let i:number;
for(i=0;i<this.miembros.length && idEmpleado!=this.miembros[i].getId();
            i++);
if (i<this.miembros.length){
this.miembros.splice(i,1);
    }
}
```

- En el componente **miembro-proyecto.component.html** añadimos un botón para quitar el componente

```
<div class='row'>
  …
<div class="col-2">
<button class="btn btn-primary" (click)='quitaEmpleado()'>
            Quitar</button>
</div>
</div>
```

La salida:



- Ahora tenemos que **lanzar** el evento en el método **quitarEmpleado** del fichero **miembro-proyecto.component.ts**

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';
import { Empleado } from '../modelo/empleado';
@Component({
selector: 'app-miembro-proyecto',
templateUrl: './miembro-proyecto.component.html',
styleUrls: ['./miembro-proyecto.component.css']
```

```
})
export class MiembroProyectoComponent implements OnInit {
@Input() private empleado:Empleado;
@Output() empleadoQuitado = new EventEmitter();
constructor() { }
ngOnInit() {
  }

quitaEmpleado(){
console.log("Quitado " + this.empleado);
this.empleadoQuitado.emit(this.empleado);
  }
}
```

- Ahora, desde el componte que llama a **miembro-proyecto**, es decir desde **proyecto** podemos **suscribirnos al evento** como lo haríamos con cualquier **evento DOM** poniendo **(evento)** y asociándole código (una llamada). Por tanto, lo primero es modificar **proyecto.component.html** para suscribirnos al evento:

```
<div class="row mt-3">
    …
</div>
<app-miembro-proyecto [hidden]='ocultarEmpleados'
*ngFor="let e of proyecto.miembros" [empleado]='e'
(empleadoQuitado)='empleadoQuitado($event)'>
</app-miembro-proyecto>nto
```

Como se puede ver nos suscribimos al evento. El parámetro **$event** nos permite acceder a la **infomación emitida por el evento** (en nuestro caso, el **empleado**).

- Por último, definimos el método **empleadoQuitado** dentro de **proyecto.component.ts**:

```
public empleadoQuitado(empleado:Empleado){
this.proyecto.quitarEmpleado(empleado.getId());
  }
```

Y ya está. Cuando hacemos clic en el botón **Quitar** del componente **miembro-proyecto**, éste lanza un **evento** con dicho empleado que es recogido por el componente **proyecto** mediante **binding de eventos**. El componente proyecto se encarga de **eliminar el empleado** del proyecto llamando al método correspondiente de la clase **Proyecto**.

Hacer un sistema similar para **eliminar proyectos de la lista de proyectos**.

Rutas en Angular

Nuestro proyecto tiene ahora un componente que permite ver todos los empleados y otro que permite ver los proyectos asociados a los mismos. Vamos a ver cómo podemos añadir una barra de navegación que nos permita alternar entre un componente y otro.

En primer lugar vamos a añadir un componente **barra-navegacion** a nuestro proyecto:

```
ng g c barra-navegacion

CREATE src/app/barra-navegacion/barra-navegacion.component.html (35 bytes)

CREATE src/app/barra-navegacion/barra-navegacion.component.spec.ts (692 bytes)

CREATE src/app/barra-navegacion/barra-navegacion.component.ts (308 bytes)

CREATE src/app/barra-navegacion/barra-navegacion.component.css (0 bytes)

UPDATE src/app/app.module.ts (1210 bytes)
```

Y en el fichero **barra-navegacion.component.html** copiamos el código de ejemplo del componente **navbar** de Bootstrap eliminando lo que no nos interesa:

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">

<a class="navbar-brand" href="#">Empleados</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 active">

<a class="nav-link" href="#">Home <span class="sr-only">(current)</span></a>

</li>

<li class="nav-item">

<a class="nav-link" href="#">Proyectos</a>

</li>

</ul>

<form class="form-inline my-2 my-lg-0">

<input class="form-control mr-sm-2" type="search" placeholder="Buscar" aria-label="Buscar">
```

```
<button class="btn btn-outline-success my-2 my-sm-0" type="submit">Buscar</button>
</form>
</div>
</nav>
```

NOTA: Es conveniente cambiar las etiquetas **href** por etiquetas **routerLink** (sobre todo cuando queramos emplear parámetros como veremos más adelante). El componente **routerLink** recibe una matriz con

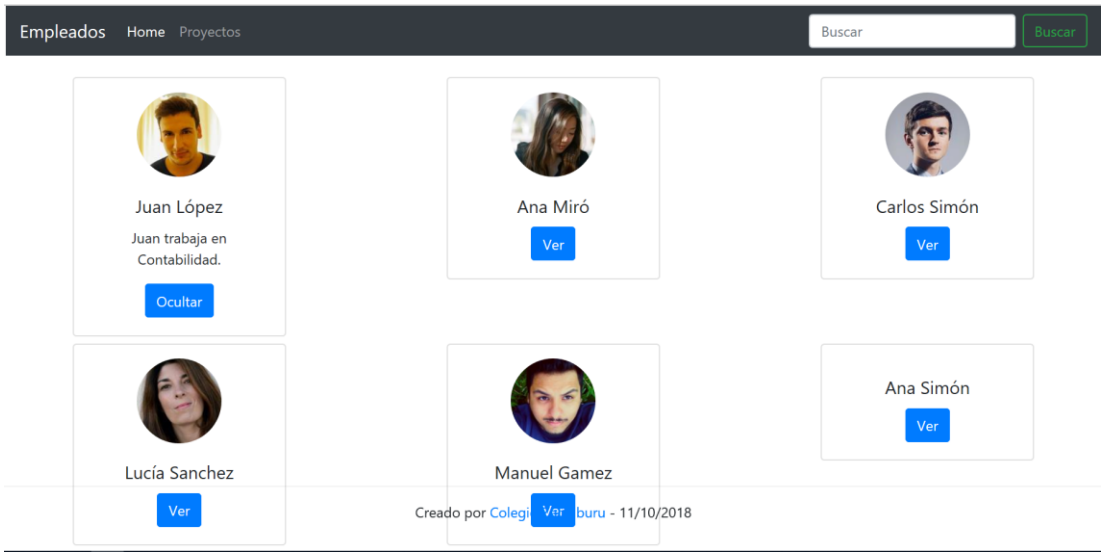
las diferentes partes de una URI dinámica. Por tanto podríamos poner:

```
<ul class="navbar-nav mr-auto">
  <li class="nav-item active">
    <a class="nav-link" [routerLink]="['/home']">Home <span class="sr-only">(current)</span></a>
  </li>
  <li class="nav-item">
    <a class="nav-link" [routerLink]="['/proyectos']">Proyectos</a>
  </li>
 </ul>
```

Añadimos el componente a **app.component.html**:

```
<div class="container-fluid">
<app-barra-navegacion></app-barra-navegacion>
<app-lista-empleados class="m-4"></app-lista-empleados>
<!--app-lista-proyectos></app-lista-proyectos>-->
<app-pie></app-pie>
</div>
```

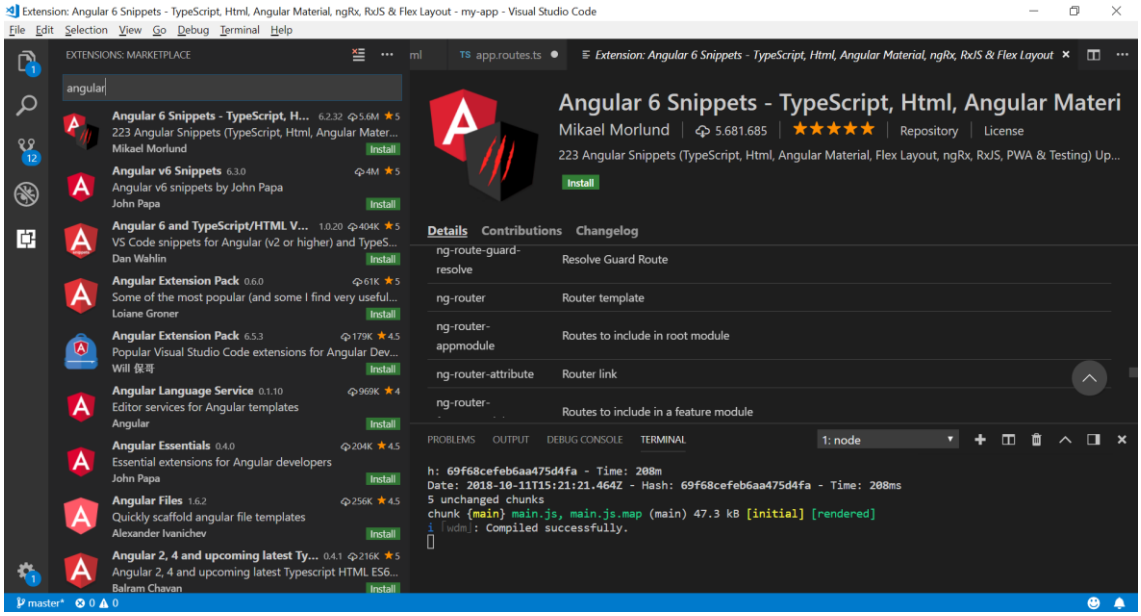
La salida:



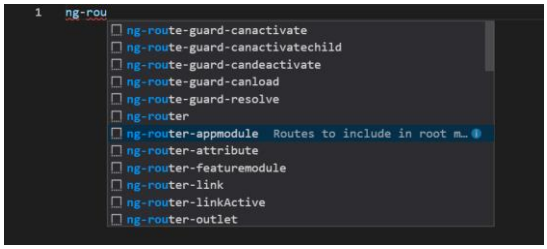
Las **rutas** nos proporcionan un mecanismo para diseñar **rutas de navegación entre componentes** que se cargan dinámicamente en una SPA (Single Page Aplication).

El primer paso es crear el archivo de **rutas** en el raíz de la aplicación. Habitualmente se llama a este archivo **app.routes.ts**.

NOTA: Para simplificar la escritura de código podemos instalar unos **snippets** de VisualStudio para Angular. En concreto vamos a instalar **Angular 6 Snippets**. Para ello vamos a **File ❼ Preferences ❼ Extensions** y escribimos **Angular**:



Podemos ver en **Details** cuáles son los atajos de teclado que incorpora. Lo instalamos y en el fichero escribimos **ng-router-app-module**:



Al pulsar **Enter** nos generará el siguiente código:

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

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

const routes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'path', component: FeatureComponent },

{ path: '**', component: PageNotFoundComponent },

];


@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule {}
```

En el que aparecen los siguientes elementos:

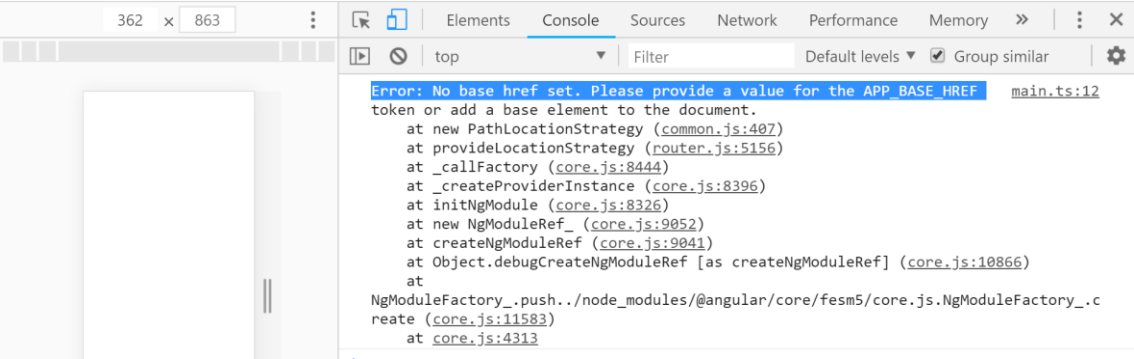
- Una matriz con las **rutas de la aplicación**. Cada ruta tiene dos elementos:
  - **path** : La URI asociada a la ruta
  - **component** : El componente a mostrar cuando se selecciona esa ruta
  - **redirectTo** : Una URI a la que redirigir cuando se llega a la ruta
- La última de las rutas representa cualquier ruta no incluida previamente
- La directiva **@NgModule** indica que estamos creando un nuevo módulo sólo para las rutas

Modificamos el fichero para dejarlo de la siguiente forma:

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

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

import { ListaEmpleadosComponent } from './app/lista-empleados/lista-empleados.component';

import { ListaProyectosComponent } from './app/lista-proyectos/lista-proyectos.component';

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

const routes: Routes = [

{ path: 'home', component: ListaEmpleadosComponent },
```

```
{ path: 'proyectos', component: ListaProyectosComponent },
{ path: '**', redirectTo: 'home' }
];

@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule],
providers: [{provide: APP_BASE_HREF, useValue : '/' }]
})
export class AppRoutingModule {}
```

NOTA: El apartado **APP_BASE_HREF** es necesario si, tras configurar las rutas aparece el siguiente error:



Otra posibilidad es añadir la etiqueta **base href='/'** en el **HEAD** del fichero **index.html**:

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <base href="/">
   <link rel="icon" href="favicon.ico" type="image/x-icon" >
    <title>Hello, world!</title>
  </head>
  <body>
    <app-root></app-root>
```

```
   </body>
</html>
```

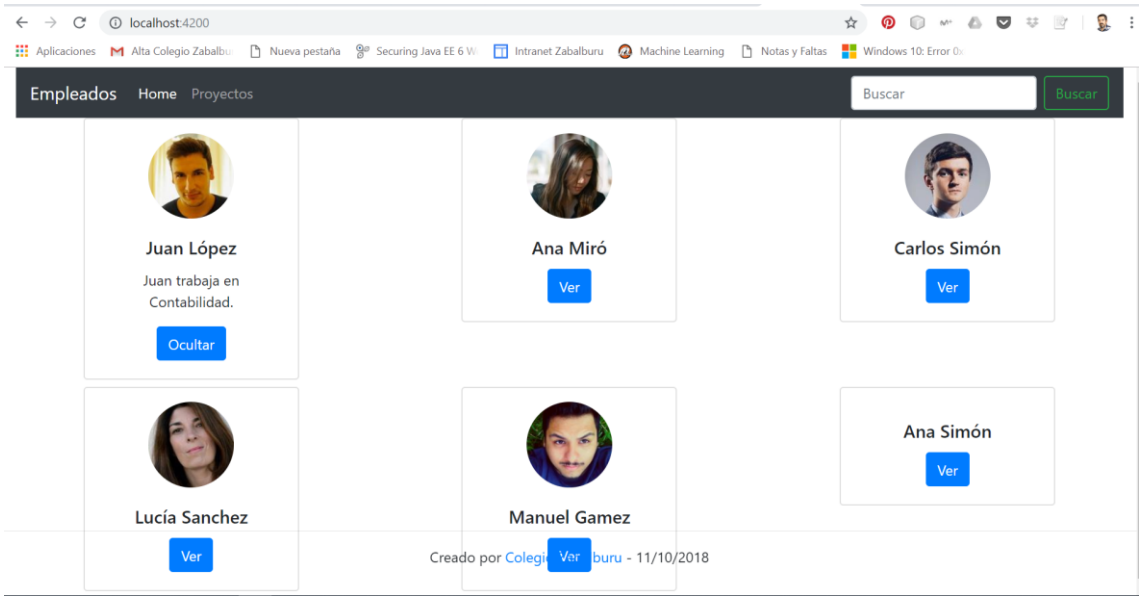
El siguiente paso es cargar dicho módulo en el fichero **app.module.ts**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
…
import { BarraNavegacionComponent } from './barra-navegacion/barra-navegacion.component';
// Rutas
import { AppRoutingModule } from './app.routes';
@NgModule({
declarations: [
AppComponent,
EmpleadoComponent,
    …
  ],
imports: [
BrowserModule,
FormsModule,
AppRoutingModule
  ],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

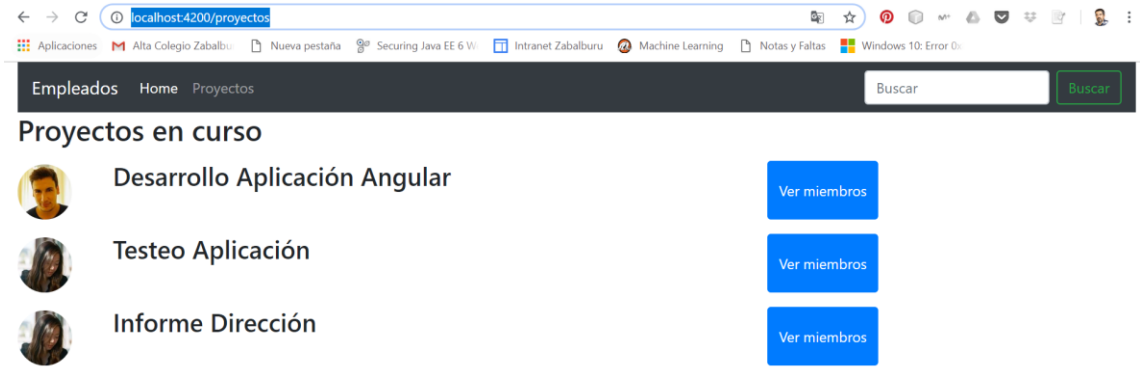
Todavía queda un último paso: decirle a Angular **dónde** debe ubicar los componentes a a los que de la navegación. Para ello es necesario modificar el **app.component.html** para añadir una etiqueta **router-outlet** donde queramos nuestros componentes:

```
<div class="container-fluid">
<app-barra-navegacion></app-barra-navegacion>
<router-outlet></router-outlet>
<app-pie></app-pie>
</div>
```

Si cargamos la URL **localhost:4200**



Y localhost:4200/proyectos:



Cualquier URL no contemplada nos llevará a la página de inicio.

Ahora Podemos modificar los enlaces de la barra de navegación para que todo funcione correctamente:

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <a class="navbar-brand" [routerLink]="['/']">Empleados</a>
  …
  <div class="collapse navbar-collapse" id="navbarSupportedContent">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item">
        <a class="nav-link" routerLink="/home" >Home </a>
      </li>
      <li class="nav-item">
        <a                                          class="nav-link"
     [routerLink]="['/proyectos']">Proyectos</a>
      </li>

    </ul>
    …
    </form>
  </div>
</nav>
```

Marcar un enlace como activo

Si queremos resaltar el enlace actualmente activo podemos emplear la directiva **routeActive** en la que especificamos la **clase** que queremos emplear cuando dicha ruta es la ruta actual.

Por ejemplo, Podemos emplear la clase **active** de **Bootstrap** que resalta el enlace.

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

    <a class="nav-link" routerLink="/home" >Home </a>

</li>

<li class="nav-item" [routerLinkActive]="['active']">

    <a class="nav-link" [routerLink]="['/proyectos']">Proyectos</a>

</li>
```

Como se puede ver, la propiedad puede recibir una clase o una matriz de clases.

Rutas con parámetros

Supongamos que queremos mostrar los datos de un empleado individualmente. Vamos a cambiar el componente **empleado** para que **el botón Ver** nos lleva a un nuevo componente llamado **empleado-detalles**.

En nuestro caso la ruta a dicho empleado sería **empleados/*idEmpl*** donde **idEmp** sería el id del empleado a mostrar.

En primer lugar creamos un nuevo componente:

```
ng g c empleado-detalles

CREATE  src/app/empleado-detalles/empleado-detalles.component.html  (36
bytes)
```

```
CREATE src/app/empleado-detalles/empleado-detalles.component.spec.ts
(699 bytes)
CREATE src/app/empleado-detalles/empleado-detalles.component.ts (312
bytes)
CREATE   src/app/empleado-detalles/empleado-detalles.component.css   (0
bytes)
UPDATE src/app/app.module.ts (1601 bytes)
```

Luego modificaremos este componente.

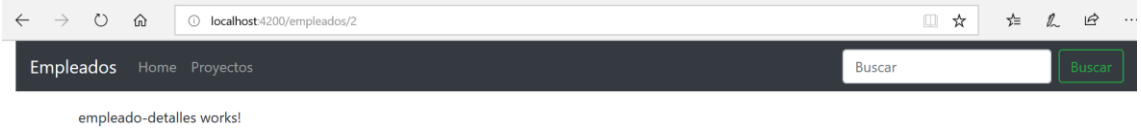
Creamos la ruta en **app.routes.ts**:

```
import { NgModule } from '@angular/core';
…
import    {    EmpleadoDetallesComponent    }    from    './empleado-
detalles/empleado-detalles.component';

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

const routes: Routes = [
    { path: 'home', component: ListaEmpleadosComponent },
    { path: 'proyectos', component: ListaProyectosComponent },
    { path: 'empleados/:id', component: EmpleadoDetallesComponent}
    { path: '**', redirectTo: 'home' }
];
…
```

Ahora modificamos el componente **empleado** (**empleado.component.html**) para que el botón enlace a dicha ruta:

```
<div class="card text-center mx-auto" style="width: 15em;">
  <img class="card-img-left mx-auto mt-3 rounded-circle" style='width:
40%' src={{empleado.foto}}>
  <div class="card-body">
    <h5 class="card-title">{{empleado.nombreCompleto()}}</h5>
    <p     class="card-text"     [hidden]="ocultar">{{empleado.nombre}}
trabaja en {{empleado.departamento}}.</p>
    <a  [routerLink]="[  '/empleados',  empleado.id  ]"  class='btn  btn-
primary' >Ver más</a>
  </div>
</div>
```

Como se puede ver, pasamos el **id** del empleado en la ruta. Si ahora vamos a **/empleados/2** :



Del mismo modo, si en la lista de empleados pulsamos en **Ver más** sobre cualquier empleado deberíamos ver la página correspondiente.

Si queremos seguir utilizando el botón tendríamos que hacer la redirección por código. Para ello, en **empleado.component.html**:

```
<button class='btn btn-primary' (click) = 'verEmpleado(empleado.id)'>Ver</button>
```

Y en **empleado.component.ts** necesitamos un objeto **Router** (que es un servicio inyectable) para la navegación:

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

@Component({
  selector: 'app-empleado',
  templateUrl: './empleado.component.html',
  styleUrls: ['./empleado.component.css']
})
export class EmpleadoComponent implements OnInit {
   @Input() private empleado:Empleado;
   @Input() private ocultar = false;

  constructor(private router:Router){
    this.empleado = new Empleado(1,'Juan','Lopez','Contabilidad','https://randomuser.me/api/portraits/men/79.jpg');
  }

  ngOnInit() {
  }

  …
```

```
  verEmpleado():void{
    this.router.navigate(['/empleados',this.empleado.getId()]);
  }
}
```

Y el resultado es el mismo.

¿Cómo **recuperamos** el **parámetro** del URL? Mediante un objeto **ActivatedRoute**. Dicha clase tiene una **propiedad observable** llamada **params**.

Observables

Un **observable** en Angular es un elemento que se emplea para **pasar mensajes entre emisores y suscriptores**. Un observable no empieza a funcionar hasta que no hay un suscriptor y, una vez que lo hay, le envía información hasta que la función termine o el suscriptor abandone su suscripción.

Para suscribirse a un observable se emplea el método **subscribe** que tiene como parámetro la función a ejecutar cada vez que llegue un mensaje. En el caso de la propiedad **params** ese mensaje es un **objeto con todos los parámetros de la ruta** (en nuestro caso sólo hay uno, pero puede haber más).

Por ejemplo, si queremos mostrar en el log el parámetro:

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

@Component({
  selector: 'app-empleado-detalles',
  templateUrl: './empleado-detalles.component.html',
  styleUrls: ['./empleado-detalles.component.css']
})
export class EmpleadoDetallesComponent implements OnInit {

  constructor(private activatedRoute:ActivatedRoute) {
    this.activatedRoute.params.subscribe( function(params) {
      console.log(params);
    });
  }
}
```

Podemos emplear la función **flecha**:

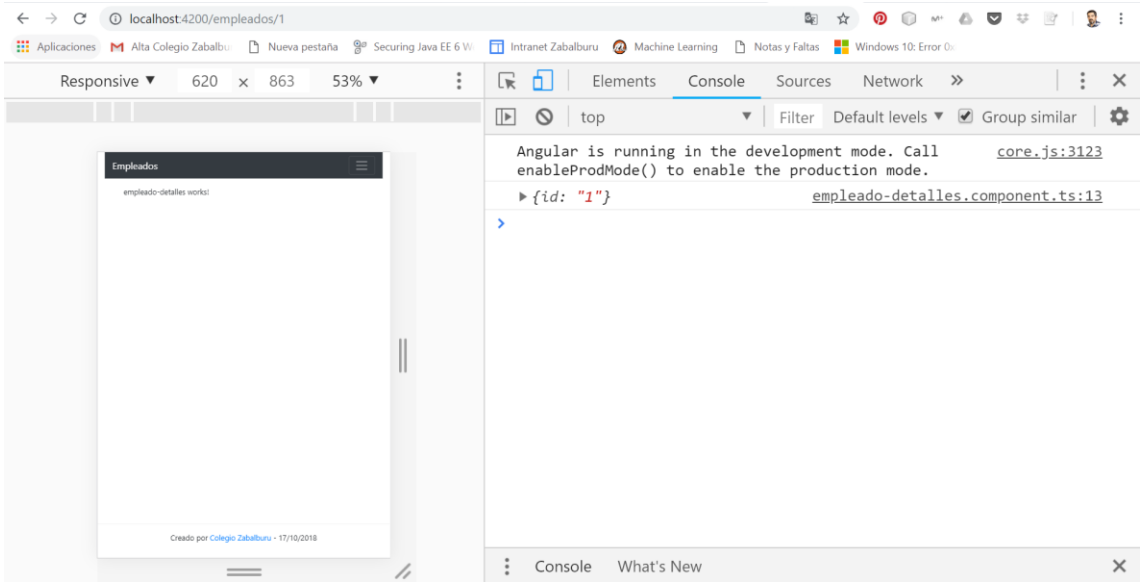
```
this.activatedRoute.params.subscribe( (params) => {
      console.log(params);
```

```
    });
```

NOTA: La sintaxis de flecha tiene una ventaja importante sobre la función y es que es capaz de acceder a las propiedades del objeto en el que se define. El problema con las funciones javascript estándar es que, si dentro de la función suscriptora, empleamos **this** nos estamos refiriendo a la propiedad **params y no al componente**.

Si empleamos la función flecha, **this** representa al componente en el que estamos.

La salida:

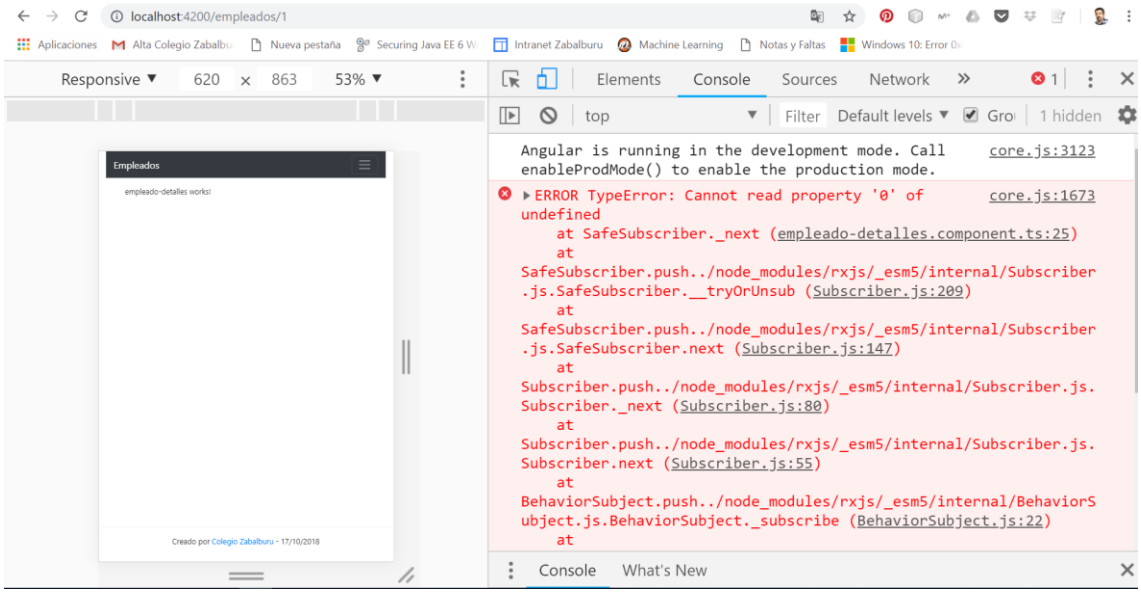


Como se puede ver, la función recibe un objeto con los parámetros de la misma. Vamos a almacenar el empleado correspondiente al id:

```
export class EmpleadoDetallesComponent implements OnInit {
  private empleado:Empleado;
  private empleados;
  constructor(private activatedRoute:ActivatedRoute) {
    const url:string='https://randomuser.me/api/portraits';
    this.empleados = [
      new Empleado(1,'Juan','López','Contabilidad',url+'/men/57.jpg'),
      new Empleado(2,'Ana','Miró','Ventas',url+'/women/12.jpg'),
      new
Empleado(3,'Carlos','Simón','Contabilidad',url+'/men/1.jpg'),
      new Empleado(4,'Lucía','Sanchez','Ventas',url+'/women/4.jpg'),
      new
Empleado(5,'Manuel','Gamez','Contabilidad',url+'/men/12.jpg'),
      new Empleado(6,'Ana','Simón','Contabilidad',null)
    ];
    this.activatedRoute.params.subscribe( params => {
```

```
        let i:number;
        for(i=0;params.id != this.empleados[i].getId();i++);
        this.empleado = this.empleados[i];
      });
  }
```

Si, en lugar de emplear la función **flecha** empleáramos **function**:



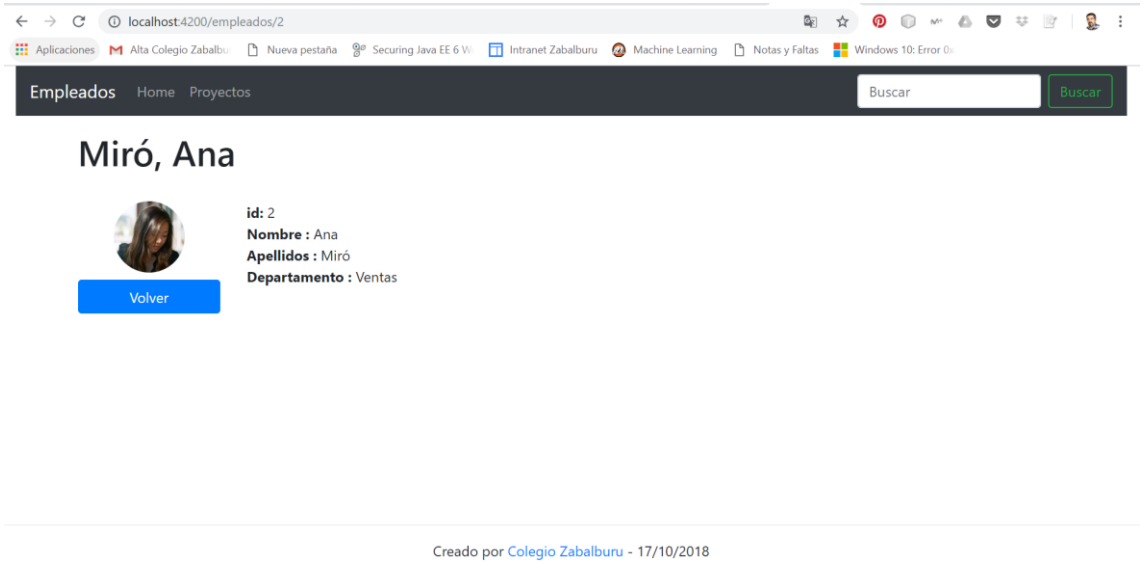
El problema es que **this no se refiere al componente** por lo que **this.empleados** está **indefinido**.

Dejamos la función flecha y modificamos el HTML:

```
<h1>{{ empleado.apellido + ', ' + empleado.nombre }}</h1>
<br>
<div class="row">
  <div class="col-2 text-center">
    <img src='{{ empleado.foto }}' class="rounded-circle" width="50%">
    <br>
    <a [routerLink]="['/empleados']" class="btn btn-primary btn-block mt-2">Volver</a>
  </div>
  <div class="col-10">
    <span class='font-weight-bold'>id: </span>{{ empleado.id }}<br>
    <span class='font-weight-bold'>Nombre : </span> {{ empleado.nombre }}<br>
    <span class='font-weight-bold'>Apellidos : </span> {{ empleado.apellido }}<br>
```

```
    <span    class='font-weight-bold'>Departamento    :    </span>    {{
empleado.departamento }}
  </div>
</div>
```

La salida:



Al pulsar el botón **Volver** deberíamos regresar al componente **lista empleados**.

Servicios

Hasta ahora la información la hemos gestionado desde los propios componentes pero, como ya sabemos, esto tiene múltiples desventajas (sólo es accesible desde el componente, hay que pasarla manualmente a otros componentes y necesitamos **output binding** para realizar las operaciones de mantenimiento debido a este sistema).

Además, nos limita gravemente a la hora de modificar la forma de acceder a la información y rompe con el modelo MVC.

La solución, evidentemente, pasa por emplear una capa de servicio para la aplicación encargada de proporcionar los métodos adecuados para acceder a los datos.

Vamos a crear un servicio para los **empleados**. Podemos hacerlo empleando el **ng**:

```
ng g service services/empleado --spec=false
```

```
CREATE src/app/services/empleado.service.ts (137 bytes)
```

El fichero creado:

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


@Injectable({
  providedIn: 'root'
```

```
})
export class EmpleadoService {


  constructor() { }
}
```

Donde:

- **@Injectable**: decorador que indica que estamos en un elemento que va a ser proporcionado a otros elementos sin que éstos se encarguen de su gestión (permite **inyectar** el servicio en otras clases)
- **providedIn: 'root'**: indica que el servicio está asociado al proveedor raíz. Un proveedor es el encargado de crear y mantener el servicio. Empleando el proveedor raíz el servicio puede ser inyectado en cualquier componente y sólo se crea una **instancia compartida del mismo**.
- **EmpleadoService**: es la clase del servicio propiamente dicha. Aquí deberán definirse los métodos de acceso a los datos del mismo

Definimos los elementos del servicio, un método para retornar los empleados y otro para retornar un empleado en base a su **id**:

```
import { Empleado } from '../modelo/empleado';


@Injectable({
  providedIn: 'root'
})
export class EmpleadoService {
  private empleados:Empleado[];
  constructor() {
    const url:string='https://randomuser.me/api/portraits';
    this.empleados = [
      new Empleado(1,'Juan','López','Contabilidad',url+'/men/57.jpg'),
      new Empleado(2,'Ana','Miró','Ventas',url+'/women/12.jpg'),
      new
Empleado(3,'Carlos','Simón','Contabilidad',url+'/men/1.jpg'),
      new Empleado(4,'Lucía','Sanchez','Ventas',url+'/women/4.jpg'),
      new Empleado(5,'Manuel','Gamez','Contabilidad',url+'/men/12.jpg'),
      new Empleado(6,'Ana','Simón','Contabilidad',null)
    ];
  }
```

```
  getEmpleados():Empleado[]{
    return this.empleados;
  }
  getEmpleado(id:number):Empleado {
    let i:number;
    for(i=0;id != this.empleados[i].getId();i++);
    return this.empleados[i];
  }
}
```

Una vez creado el servicio es necesario informar a Angular de que dicho servicio está disponible y, como siempre, lo hacemos en el **app.modules.ts**:

```
import { BrowserModule } from '@angular/platform-browser';
…
// servicios
import { EmpleadoService } from './services/empleado.service';

@NgModule({
  declarations: [
    AppComponent,
    …
  ],
  imports: [
    BrowserModule,
    …
  ],
  providers: [
    EmpleadoService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Como se ve, los servicios se especifican en la sección **poviders** indicando que son servicios que se **proveen a los componentes de la aplicación**.

El último paso es cómo podemos inyectar los servicios en los componentes. Esto se hace en el **constructor** del componente. Por ejemplo, modificamos **lista-empleados** para que use el servicio:

```
import { Component, OnInit } from '@angular/core';
import { Empleado } from '../modelo/empleado';
import { EmpleadoService } from '../services/empleado.service';

@Component({
  selector: 'app-lista-empleados',
  templateUrl: './lista-empleados.component.html',
  styleUrls: ['./lista-empleados.component.css']
})
export class ListaEmpleadosComponent implements OnInit {
  private empleados:Empleado[];

  constructor(servicio:EmpleadoService) {
    this.empleados = servicio.getEmpleados();
  }

  ngOnInit() {
  }
}
```

Como se puede ver, el servicio **simplemente se declara** en el constructor y ya podemos usarlo. Evidentemente el resultado es el mismo.

Inyección de dependencias

El que un servicio sea **inyectable** quiere decir que puede emplearse en un componente **sin necesidad de instanciarlo** en dicho componente (sin emplear el operador **new**). Será el **proveedor del servicio** (en nuestro caso **root**) el que se encargue de instanciar y proporcionar dicho servicio a los componentes. A este proceso se le llama **inyección de dependencias**.

También vamos a modificar **empleado-detalles.component.ts** para que haga uso del servicio:

```
export class EmpleadoDetallesComponent implements OnInit {
  private empleado:Empleado;
  constructor(private activatedRoute:ActivatedRoute,
      private empleadoService:EmpleadoService) {
    this.activatedRoute.params.subscribe( (params) =>  {
      this.empleado = this.empleadoService.getEmpleado(params.id);
```

```
    })
  }
}
```

ACTIVIDAD

Definir un **nuevo servicio** para la parte de los proyectos de la aplicación. Este servicio hará uso del **servicio EmpleadoService** y proporcionará métodos para **recuperar todos los proyectos**, para **localizar un proyecto en base a su id** para **eliminar** un miembro de un proyecto (este método recibirá el **idProyecto** y el **idEmpleado** a eliminar)

Modificar los componentes de proyectos para hacer uso de este servicio.

```
ng g service services/proyecto --spec=false
CREATE src/app/services/proyecto.service.ts (137 bytes)
```

proyecto.service.ts

```
import { Injectable } from '@angular/core';
import { Proyecto } from '../modelo/proyecto';
import { EmpleadoService } from './empleado.service';
import { Empleado } from '../modelo/empleado';

@Injectable({
  providedIn: 'root'
})
export class ProyectoService {
  private proyectos:Proyecto[];

  constructor(private empleadoServicio:EmpleadoService) {
    let empleados:Empleado[];
    empleados = empleadoServicio.getEmpleados();
    this.proyectos = [
      new Proyecto(1,'Desarrollo Aplicación Angular',empleados[0],
        [empleados[1],empleados[2]]),
      new Proyecto(2,'Testeo Aplicación',empleados[1],
        [empleados[2],empleados[4]]),
      new Proyecto(3,'Informe Dirección',empleados[1],
        [empleados[1],empleados[2], empleados[3]])
    ]
```

```
  }

  getProyectos():Proyecto[] {
    return this.proyectos;
  }

  eliminarMiembroProyecto(idProyecto:number,idEmpleado:number):void{
    let i:number;
    for(i=0;i<this.proyectos.length &&
    this.proyectos[i].getId();i++);
    if (i<this.proyectos.length){
      this.proyectos[i].quitarEmpleado(idEmpleado);
    }
    console.log(this.proyectos[i]);
  }

}
```

app.module.ts

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

// Rutas
import { AppRoutingModule } from './app.routes';

// servicios
import { EmpleadoService } from './services/empleado.service';
import { ProyectoService } from './services/proyecto.service';

@NgModule({
  declarations: [
    …
  ],
  imports: [
    …
  ],
```

```
  providers: [
    EmpleadoService,
    ProyectoService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

lista-proyectos.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Proyecto } from '../modelo/proyecto';
import { Empleado } from '../modelo/empleado';
import { ProyectoService } from '../services/proyecto.service';

@Component({
  selector: 'app-lista-proyectos',
  templateUrl: './lista-proyectos.component.html',
  styleUrls: ['./lista-proyectos.component.css']
})
export class ListaProyectosComponent implements OnInit {
  private proyectos:Proyecto[];
  constructor(proyectoService:ProyectoService) {
    this.proyectos = proyectoService.getProyectos();
   }

  ngOnInit() {
  }
  }
```

proyecto.component.ts

```
import { Component, OnInit, Input, Output, EventEmitter } from
'@angular/core';
import { Proyecto } from '../modelo/proyecto';
import { Empleado } from '../modelo/empleado';
```

```
import { ProyectoService } from '../services/proyecto.service';

@Component({
  selector: 'app-proyecto',
  templateUrl: './proyecto.component.html',
  styleUrls: ['./proyecto.component.css']
})
export class ProyectoComponent implements OnInit {
  @Input() private proyecto:Proyecto;
  private ocultarEmpleados:boolean = true;
  private proyectoService:ProyectoService;
  constructor(proyectoService:ProyectoService) {
    this.proyectoService = proyectoService;
   }

  ngOnInit() {
  }

  public empleadoQuitado(empleado:Empleado){
    this.proyectoService.eliminarMiembroProyecto(
            this.proyecto.getId(),empleado.getId());
  }
}
```

Referencia de controles

Queremos añadir la funcionalidad de **buscar un empleado**. Queremos que la búsqueda se inicie cuando el usuario pulse **Enter** en el campo de búsqueda. Para ello necesitamos capturar el evento **onKeyup** para llamar en él a una función. Como ya sabemos esto lo hacemos con **keyup** pero ¿cómo indicamos que queremos controlar la tecla **Enter**? Simplemente añadiendo **enter** a dicha expresión **(keyup.enter)**. Evidentemente tenemos literales para cada posible tecla.

La cuestión es ¿cómo podemos enviar el valor del cuadro de texto a la función de búsqueda?.

Emplear $event

Para obtener información sobre el evento podemos pasar el objeto **$event** a la función que vamos a asignar al mismo. **$event** representa el **evento DOM** correspondiente (en nuestro caso un **KeyboardEvent**). Todo evento DOM tiene una propiedad **target** que identifica el elemento en el que se produce el evento (en nuestro caso, el cuadro de texto). Adicionalmente los controles disponen de la propiedad **value** que retorna / modifica el valor asociado a los mismos.

Modificamos el código de **barra-navegacion.component.html** :

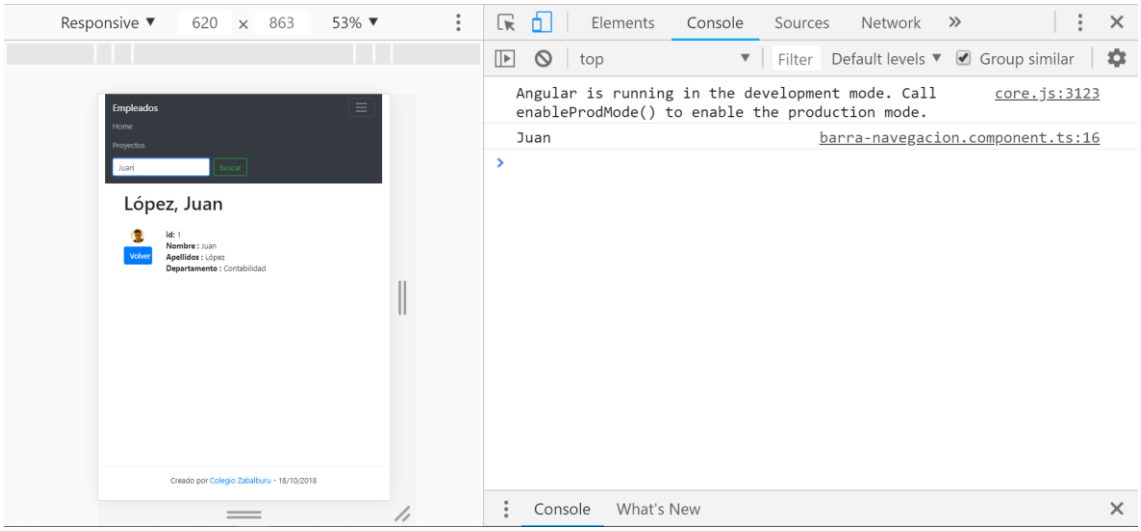
```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">

    …

    </ul>

    <form class="form-inline my-2 my-lg-0">

      <input class="form-control mr-sm-2" type="search" placeholder="Buscar" (keyup.enter)="onEnter($event)" aria-label="Buscar">

      <button class="btn btn-outline-success my-2 my-sm-0" type="button">Buscar</button>

    </form>

  </div>

</nav>
```

Y ponemos la función en **barra-navegacion.component.ts**:

```
export class BarraNavegacionComponent implements OnInit {

  …

  onEnter(evento:KeyboardEvent):void{

    let elemento:HTMLInputElement = <HTMLInputElement> evento.target;

    console.log(elemento.value);

  }

}
```

NOTA: La sintaxis **<clase>** es un **casting** en **Typescript** y no sería necesario hacerlo para que el código funcione (lo hacemos simplemente para demostrar lo explicado anteriormente).

Si escribimos algo en el cuadro de búsqueda y pulsamos **Enter**:



Emplear una variable de referencia de plantilla

En Angular podemos emplear una variable de referencia de plantilla para identificar elementos que puedan ser pasadas a métodos. Las variables de referencia se especifican con **#** (hemos empleado ya este tipo de variables en **ngIf/else**):

El código de **barra-navegacion.component.html** :

```
<form class="form-inline my-2 my-lg-0">
      <input class="form-control mr-sm-2" type="search" #buscar placeholder="Buscar" (keyup.enter)="onEnter(buscar)" aria-label="Buscar">
      <button class="btn btn-outline-success my-2 my-sm-0" type="button">Buscar</button>
    </form>
```

Ahora la función recibe directamente el control:

```
onEnter(control:HTMLInputElement){
    console.log(control.value);
}
```

Vamos a poner la misma llamada en el botón:

```
<button class="btn btn-outline-success my-2 my-sm-0" type="button" (click)="onEnter(buscar)">Buscar</button>
```

Ahora funciona en ambos casos.

El siguiente paso es modificar el **servicio (empleado.service.ts)** para añadir una función de búsqueda de empleados:

```
buscaEmpleados(busqueda:string):Empleado[]{
    let emple:Empleado[] = [];
    busqueda = busqueda.toLowerCase();
    for(let emp of this.empleados){
      if (emp.nombreCompleto().toLowerCase().indexOf(busqueda)!=-1){
        emple.push(emp);
      }
    }
    return emple;
  }
```

Creamos el componente **resultado-busqueda.component**.

```
ng g c resultado-busqueda
CREATE src/app/resultado-busqueda/resultado-busqueda.component.html (37 bytes)
```

```
CREATE  src/app/resultado-busqueda/resultado-busqueda.component.spec.ts
(706 bytes)

CREATE  src/app/resultado-busqueda/resultado-busqueda.component.ts  (316
bytes)

CREATE  src/app/resultado-busqueda/resultado-busqueda.component.css  (0
bytes)

UPDATE src/app/app.module.ts (1729 bytes)
```

Luego lo modificaremos.

Añadimos una ruta en **app.routes.ts**:

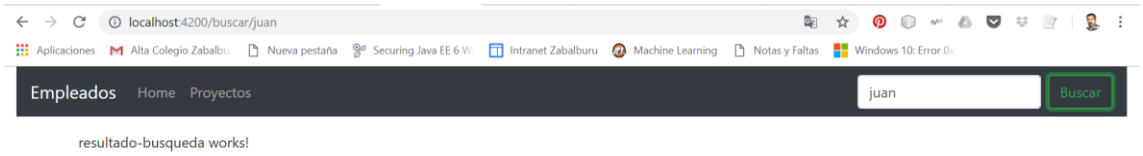
```
import { NgModule } from '@angular/core';

…

import { ResultadoBusquedaComponent } from './resultado-
busqueda/resultado-busqueda.component';


const routes: Routes = [

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

    { path: 'proyectos', component: ListaProyectosComponent },

    { path: 'empleados/:id', component: EmpleadoDetallesComponent},

    { path: 'buscar/:busqueda', component: ResultadoBusquedaComponent}

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

];

…

export class AppRoutingModule {}
```

Ahora modificamos el código de **barra-navegacion.component.ts** para hacer uso del servicio de rutas:

```
export class BarraNavegacionComponent implements OnInit {


  constructor(private rutas:Router) {

   }


  ngOnInit() {

  }


  onEnter(elemento:HTMLInputElement){

    this.rutas.navigate(['/buscar',elemento.value]);

  }
```

```
}
```

Ahora ya debería funcionar la búsqueda y mostrarnos el componente:



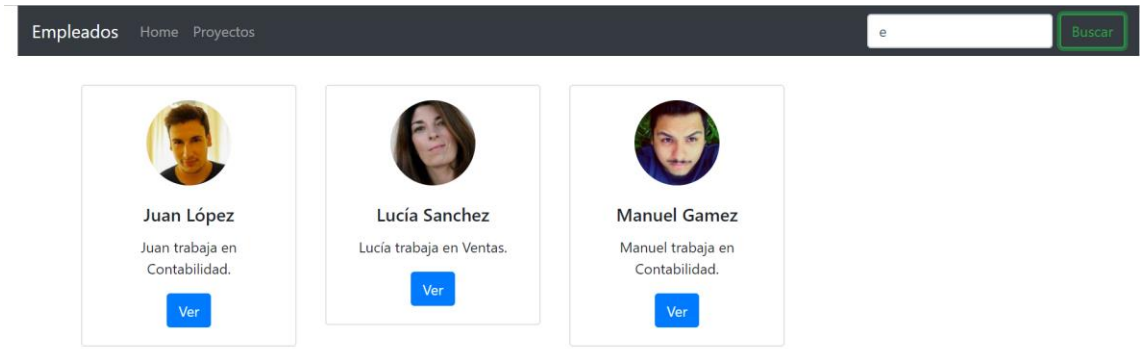
Ahora modificamos **resultado-busqueda.ts** para hacer uso del servicio y recuperar el parámetro:

```
import { Component, OnInit } from '@angular/core';
import { EmpleadoService } from '../services/empleado.service';
import { ActivatedRoute } from '@angular/router';
import { Empleado } from '../modelo/empleado';

@Component({
  selector: 'app-resultado-busqueda',
  templateUrl: './resultado-busqueda.component.html',
  styleUrls: ['./resultado-busqueda.component.css']
})
export class ResultadoBusquedaComponent implements OnInit {
  private empleados:Empleado[];
  constructor(servicio:EmpleadoService, ruta:ActivatedRoute) {
    ruta.params.subscribe((params) => {
        this.empleados = servicio.buscaEmpleados(params.busqueda);
    } )
  }
  ngOnInit() {
  }
}
```

Ya sólo queda especificar **resultado-busqueda.component.html**:

```
<div class="row">
    <h4 *ngIf="empleados.length ==0">No hay resultados</h4>
  <app-empleado class='m-3' *ngFor="let e of empleados" [empleado]="e"></app-empleado>
</div>
```

La salida:



Introducción a las peticiones Http

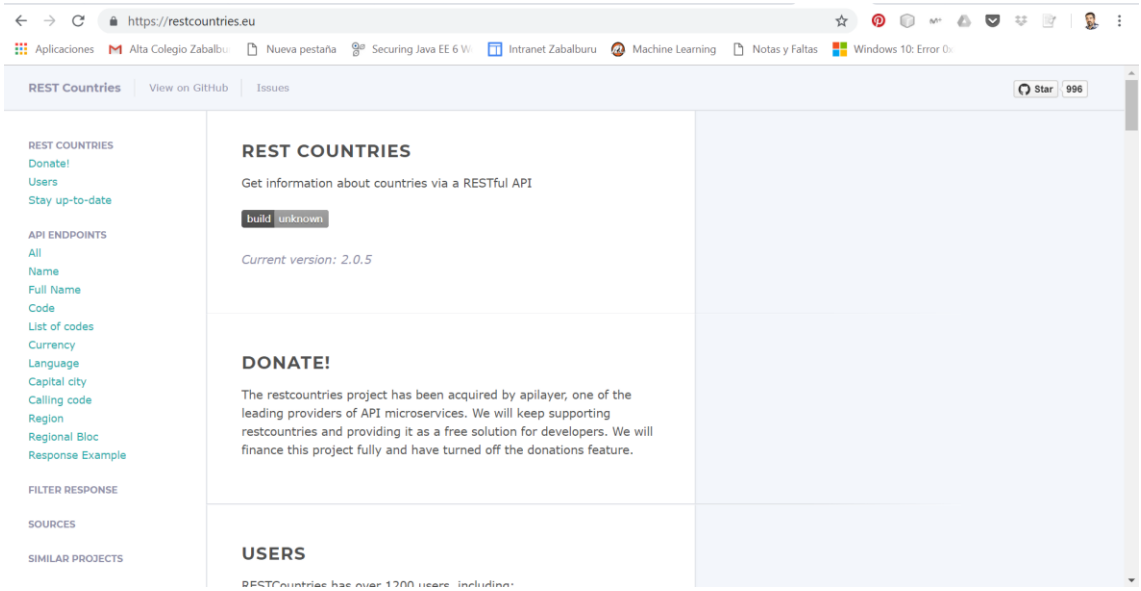
En muchas ocasiones, los datos los vamos a tener que recuperar de servicios operando en Internet. Para poder acceder a dicha información necesitamos importar **HttpClientModule** a nuestra aplicación.

Añadimos dicho modulo a nuestro proyecto en el fichero **app.modules.ts**:

```
import { BrowserModule } from '@angular/platform-browser';
…
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  declarations: [
    AppComponent,
    …
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
  providers: [
    EmpleadoService,
    …
  ],
```

```
  bootstrap: [AppComponent]
})
export class AppModule { }
```

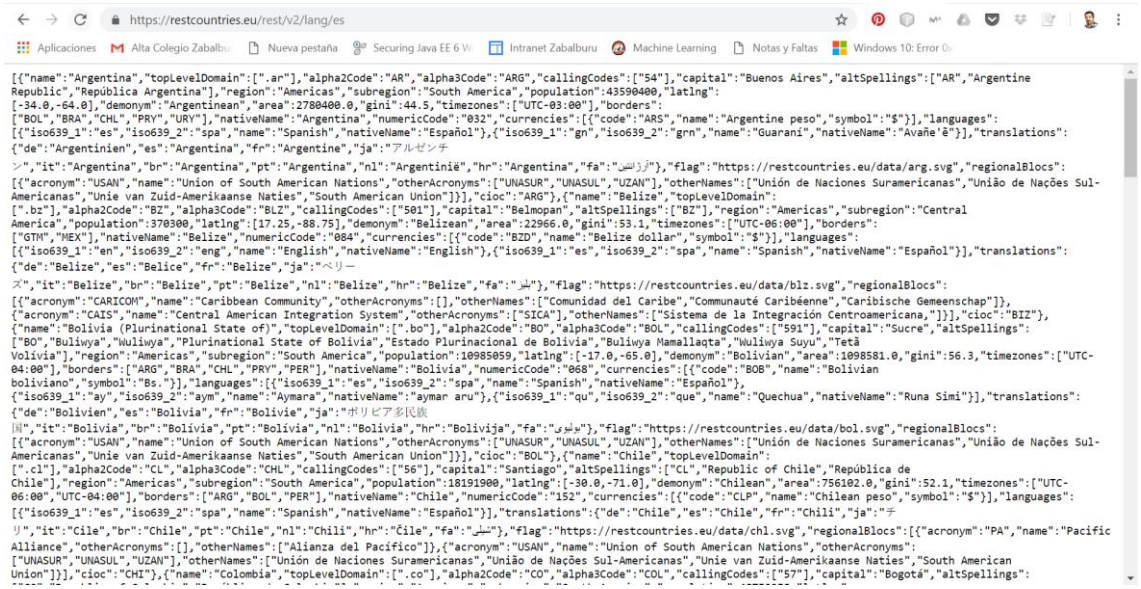
Vamos a hacer un ejemplo recuperando datos de países de **restcountries.eu** que es un servicio REST que nos proporciona información sobre los países del mundo.



Supongamos que queremos coger todos los países que hablan castellano. La petición a hacer sería:

https://restcountries.eu/rest/v2/lang/es

que nos devuelve un JSON que es una matriz de objetos JSON con información de cada país. Si, por ejemplo, lo introducimos en el navegador:



Para poder acceder a esta información desde Angular, vamos a crear un componente **países**:

```
ng g c paises --spec=false -is
CREATE src/app/paises/paises.component.html (25 bytes)
CREATE src/app/paises/paises.component.ts (242 bytes)
UPDATE src/app/app.module.ts (2053 bytes)
```

Normalmente, los datos deben obtenerse desde un **servicio**, así que vamos a crear el servicio **paises**:

```
ng g service service/paises --spec=false
CREATE src/app/service/paises.service.ts (135 bytes)
```

En dicho servicio, definimos hay que inyectar el servicio **HttpClient** de **@angular/common/http** en el constructor.

```
import { Injectable } from '@angular/core';
import { HttpClient } from 'selenium-webdriver/http';

@Injectable({
  providedIn: 'root'
})
export class PaisesService {

  constructor(private http:HttpClient) { }
}
```

La clase **HttpClient** tiene métodos específico para cada **comando HTTP** (GET, PUT, POST, DELETE,…). En el caso del GET se debe pasar la URL donde se encuentran los datos a recuperar, lo que nos devuelve un **observable** al que podemos **suscribirnos**.

El **servicio** simplemente va a retornar dicho observable para que sea consumido por los componentes:

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class PaisesService {
```

```
  getPaises():Observable<any>{
    return  this.http.get('https://restcountries.eu/rest/2/lang/es');
  }

  constructor(private http:HttpClient) { }
}
```

El observable devuelto dispone de un **parámetro** donde se reciben los datos y, opcionalmente, podemos detectar si se han producido **errores** e, incluso, **reintentar** la conexión. En **países.component.ts**:

```
export class PaisesService {

  constructor(private http:HttpClient) {
    this.http.get('https://restcountries.eu/rest/2/lang/es')
    .subscribe(
      (datos) => {
        console.log(datos);
      } ,
      error => {
        console.error('ERROR:  '  +  error.status  +  "  -  "  +
error.message)
      }
    );
  }

  ngOnInit() {
  }

}
```

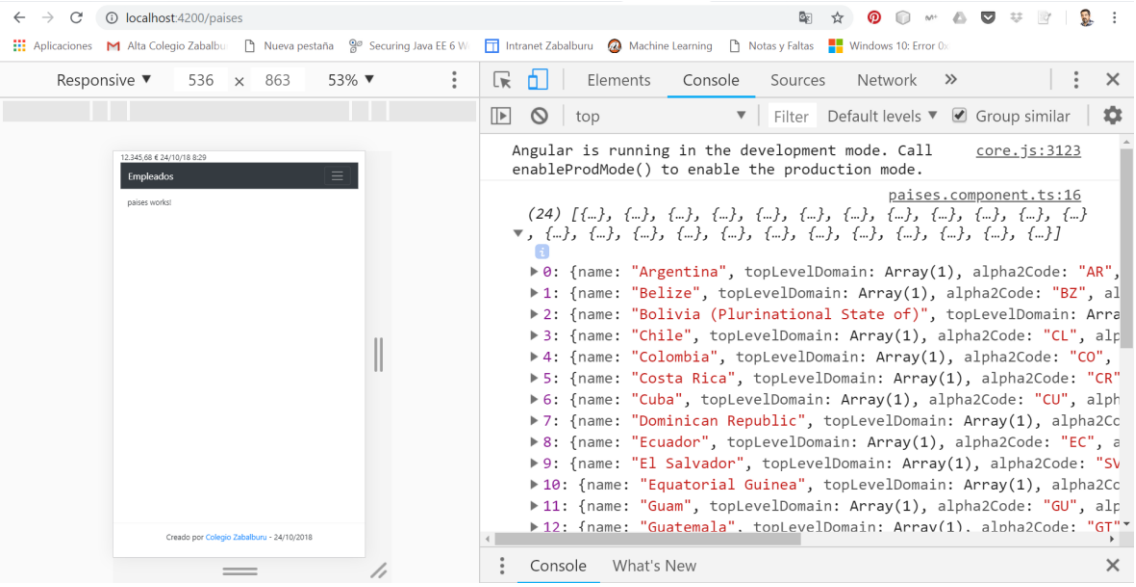
Como podemos ver añadimos un suscriptor que, si recibe parámetros los muestra por consola y, si se produce un error, muestra el código y el mensaje por consola.

El último paso es hace disponible este componente mediante una ruta. Modificamos **app.routes.ts**:

```
import { NgModule } from '@angular/core';
…
import { PaisesComponent } from './paises/paises.component';
```

```
const routes: Routes = [
    { path: 'home', component: ListaEmpleadosComponent },
    …
    { path: 'paises', component: PaisesComponent}
    { path: '**', redirectTo: 'home' }
];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule],
})
export class AppRoutingModule {}
```

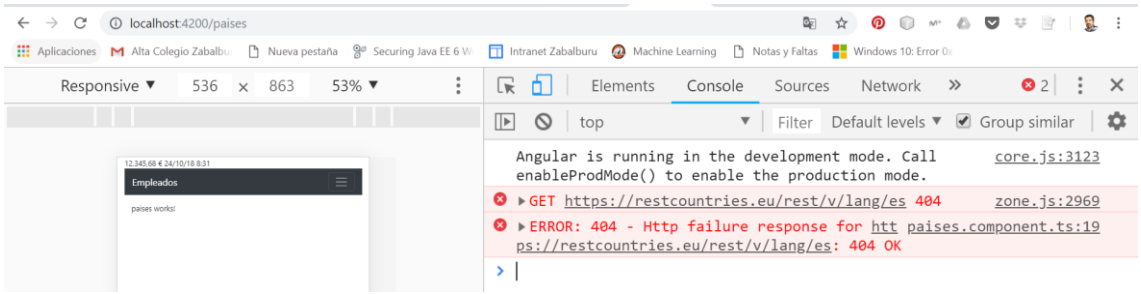
Si vamos a http://localhost:4200/paises:



Vamos a cambiar la URL en el servicio para que no se encuentre:

```
this.http.get('https://restcountries.eu/rest/v/lang/es')
```

El resultado:



Dado que la información la tenemos en **datos** y es una matriz de objetos, podemos manipularla directamente. Modificamos el código:

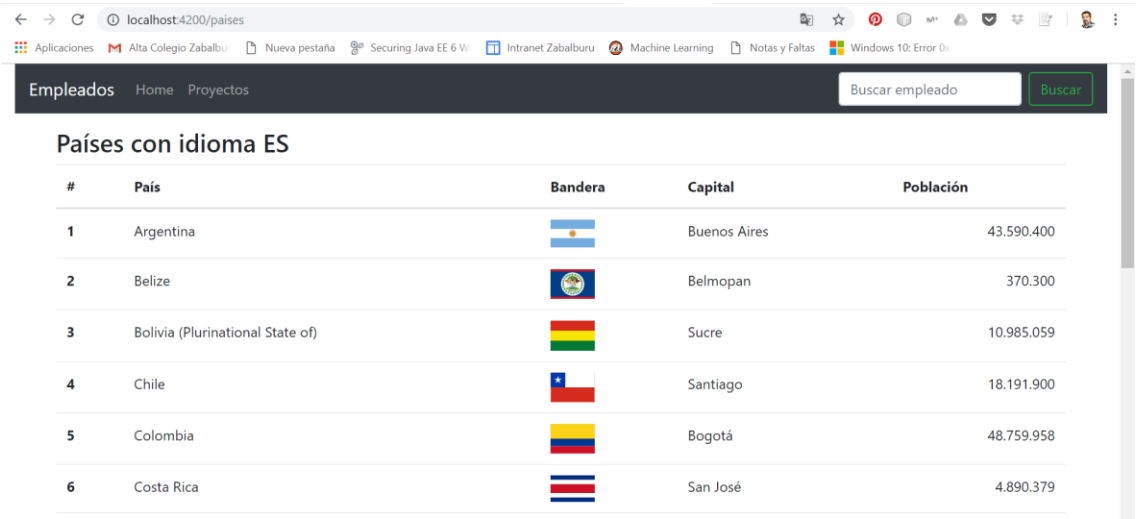
```
export class PaisesComponent implements OnInit {

  private paises:any[];

  constructor(private service:PaisesService) {

    this.service.getPaises().subscribe(

      (datos:any[]) => {

        //console.log(datos);

        this.paises = datos;

      } ,

      error => {

        console.error('ERROR: ' + error.status + " - " + error.message)

      });

    }


  ngOnInit() {

  }


}
```

Y, en **países.component.html**:

```
<div class="row">

  <h3>Países con idioma ES</h3>

  <table class="table">

      <thead>

        <tr>

          <th scope="col">#</th>

          <th scope="col">País</th>

          <th scope="col">Bandera</th>
```

```
          <th scope="col">Capital</th>
          <th scope="col">Población</th>
        </tr>
      </thead>
      <tbody>
        <tr *ngFor="let p of paises; let i=index">
          <th scope="row">{{ i+1 }}</th>
          <td>{{ p.name }}</td>
          <td><img [src]='p.flag' style="width:50px"> </td>
          <td>{{ p.capital }}</td>
          <td class='text-right'>{{ p.population.toLocaleString() }}</td>
        </tr>

      </tbody>
    </table>

</div>
```

La salida:



Enviar parámetros en la petición

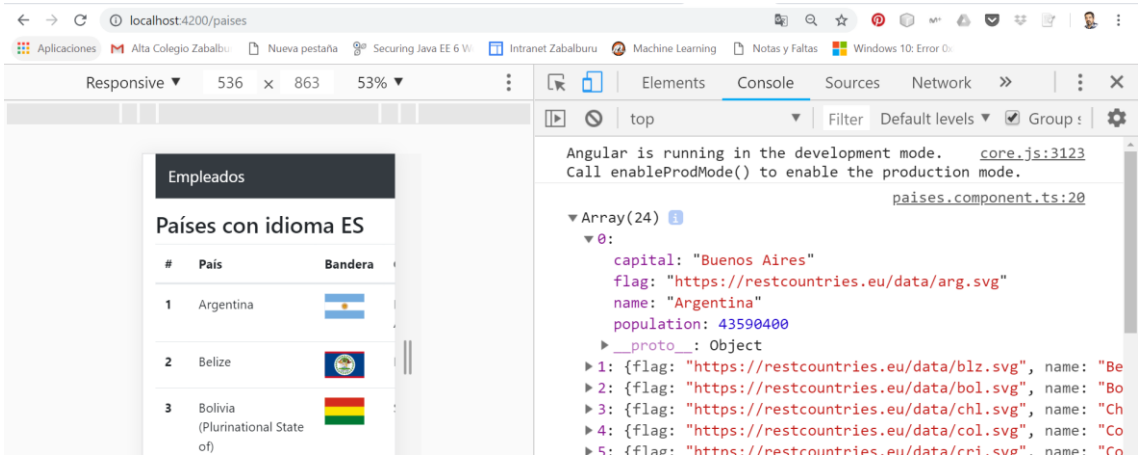
A la petición se le puede añadir un **objeto con las opciones de la misma (parámetros, cabeceras http…)**. Entre ellas, se encuentra la opción **params** que, a su vez, recibe un **objeto con los parámetros a enviar y sus valores**.

Por ejemplo, en la petición del ejemplo, se pueden filtrar los datos a mostrar añadiendo el parámetro **fields** con una lista de los campos separados por **;**:

```
export class PaisesService {
```

```
  getPaises():Observable<any>{
    return this.http.get('https://restcountries.eu/rest/v2/lang/es', {
      params : {
        fields :'name;flag;capital;population'
      }
    });
  }

  constructor(private http:HttpClient) { }
}
```

El resultado obtenido es el mismo que en el caso anterior:



Proyecto

Empleando el **API** de **TvRage** (https://www.tvmaze.com/api) se desea realizar una aplicación que permita ver información sobre programas de televisión. Dado que hay poca información sobre las series en España, emplearemos los datos de EEUU (valores por defecto).

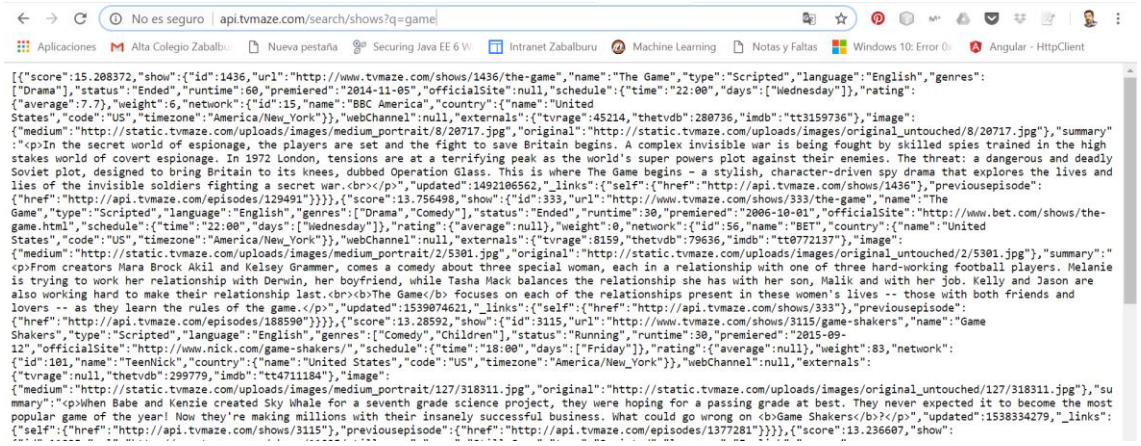
La aplicación deberá disponer de un **modelo de datos** y gestionar la información mediante un **servicio**.

Al menos constará de los siguientes elementos:

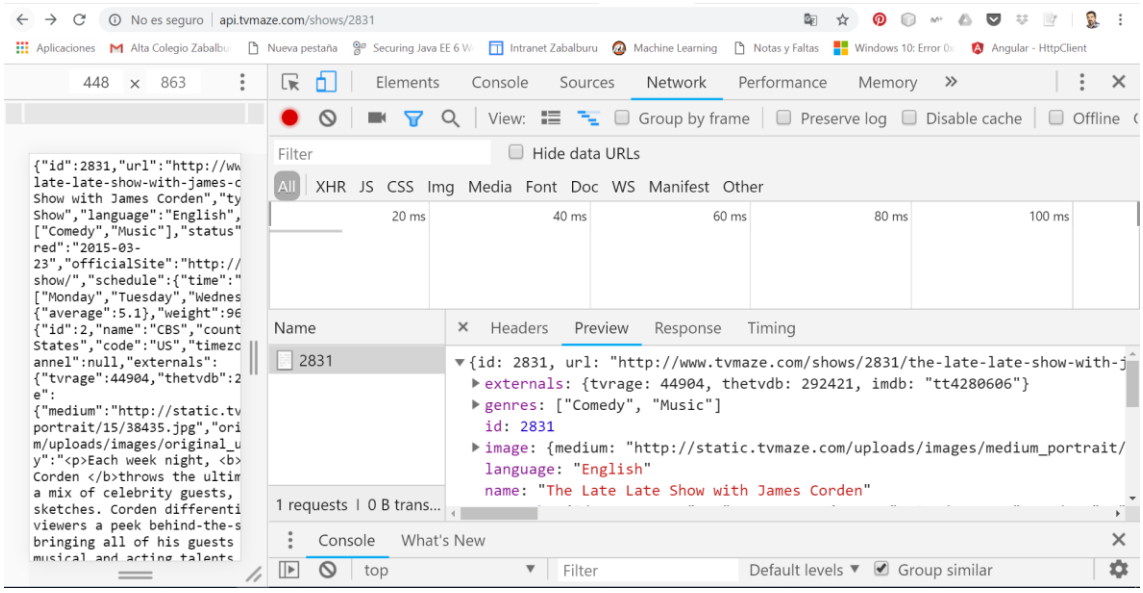
- Una página inicial donde se vean los **próximos episodios de series** (por ejemplo, http://api.tvmaze.com/schedule?country=US&date=2018-10-30)
- De cada episodio se mostrará la información que consideréis adecuada
- Cada episodio deberá permitir acceder a la información del programa de televisión correspondiente (http://api.tvmaze.com/shows/2831)

- Deberá proporcionarse un mecanismo para buscar programas de televisión en base a su nombre (http://api.tvmaze.com/search/shows?q=game)
- Opcionalmente se podría mostrar, desde el programa, información sobre las temporadas (http://api.tvmaze.com/shows/1/seasons) y/o los episodios de las mismas (http://api.tvmaze.com/seasons/1/episodes)

NOTA: Todas las consultas se pueden ejecutar desde el navegador para ver qué información proporcionan:

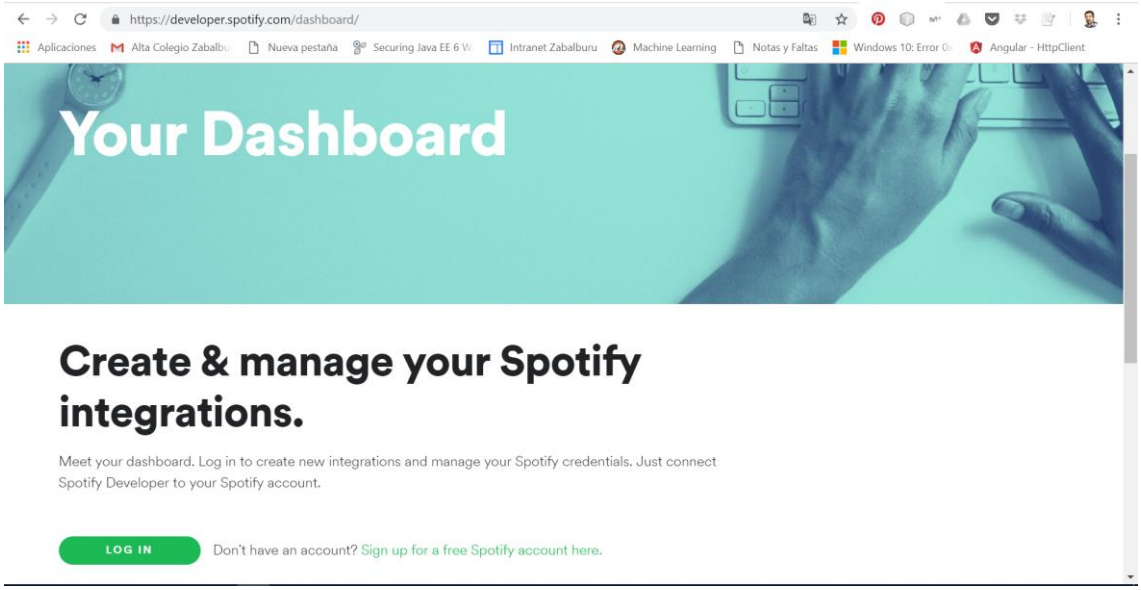


La información puede verse mejor empleando la ventana para desarrolladores de Chrome (**Network** → Seleccionando el elemento en **Name** y viendo la pestaña **Preview**)

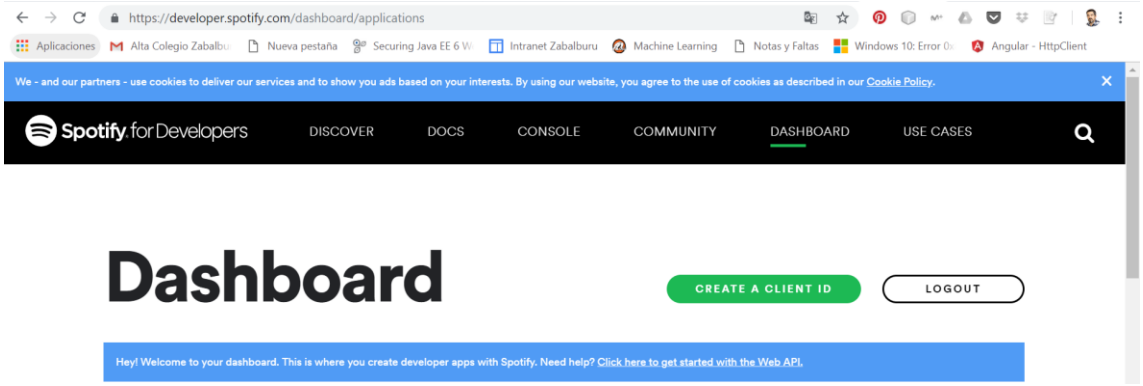


Creación de una aplicación básica basada en Spotify

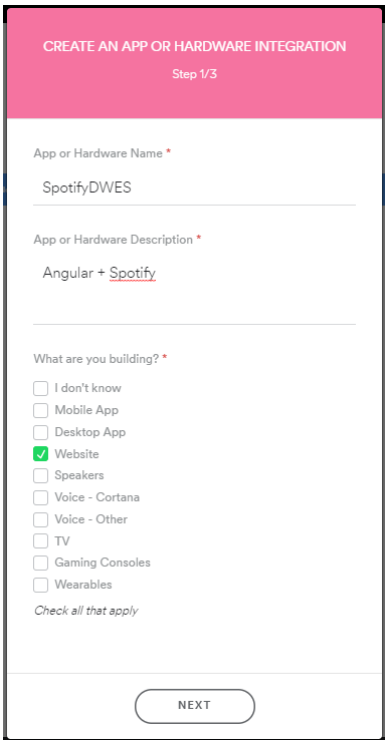
Vamos a crear una aplicación básica que emplee todo lo visto hasta ahora y algunas características nuevas. El primer paso es ir a **https://developer.spotify.com/dashboard/** para loguearnos o crear una nueva cuenta:



Una vez logueados deberemos crear un nuevo **clientId**:



Rellenamos los campos:



Y confirmamos los términos.

Tras ello creamos nuestra aplicación:

```
ng new SpotifyDWES
```

Vamos a crear un componente home, otro para búsquedas y otro con información del artista. Los vamos a almacenar en una carpeta componentes:

```
ng g c componentes/home --spec==false -is
ng g c componentes/busqueda --spec==false -is
ng g c componentes/artista --spec==false -is
ng g c componentes/barra-navegacion --spec==false -is
```

Abrimos la aplicación y la preparamos para Bootstrap. En este caso simplemente vamos a modificar **index.html** para que cargue la librería:

```
<!doctype html>
<html lang="es">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <title>SpotifyDWES</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
```

```
    <link rel="icon" type="image/x-icon" href="favicon.ico">

    <!-- Bootstrap CSS -->

    <link                                                rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap
.min.css"                                              integrity="sha384-
MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
crossorigin="anonymous">




  </head>

  <body>

    <app-root></app-root>

    <!-- Optional JavaScript -->

    <!-- jQuery first, then Popper.js, then Bootstrap JS -->

    <script      src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
integrity="sha384-
q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>

    <script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/poppe
r.min.js"                                              integrity="sha384-
ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIPm49"
crossorigin="anonymous"></script>

    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.m
in.js"                                                 integrity="sha384-
ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy"
crossorigin="anonymous"></script>

  </body>

</html>
```

Añadimos las rutas

Creamos el archivo **app.routes.ts**:

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

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

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

import { ArtistaComponent } from './componentes/artista/artista.component';

import            {            BusquedaComponent            }            from
'./componentes/busqueda/busqueda.component';

import { BarraNavegacionComponent } from

                  './componentes/barra-navegacion/barra-navegacion.component';
```

```
const routes: Routes = [
    { path: 'home', component: HomeComponent },
    { path: 'artista', component: ArtistaComponent },
    { path: 'busqueda', component: BusquedaComponent },
    { path: '**', redirectTo: 'home' }
];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
})
export class FeatureRoutingModule {}
```

El siguiente paso es importar el módulo de rutas en **app.module.ts**:

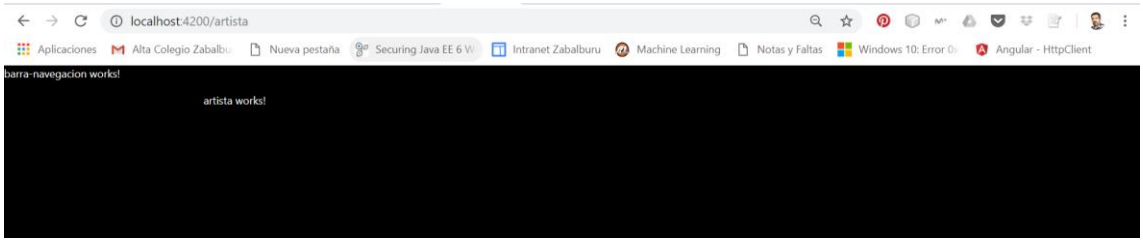
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { HomeComponent } from './componentes/home/home.component';
import { BusquedaComponent } from
'./componentes/busqueda/busqueda.component';
import { ArtistaComponent } from
'./componentes/artista/artista.component';
import { BarraNavegacionComponent } from './componentes/barra-
navegacion/barra-navegacion.component';
import { FeatureRoutingModule } from './app.routes';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    BusquedaComponent,
    ArtistaComponent,
    BarraNavegacionComponent
  ],
  imports: [
    BrowserModule,
```

```
    FeatureRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Por último, añadimos **router-outlet** en **app.component.html**:

```
<app-barra-navegacion></app-barra-navegacion>
<div class="container">
  <router-outlet></router-outlet>
</div>
```

Y ya podríamos acceder a las diferentes URLs. Por ejemplo:



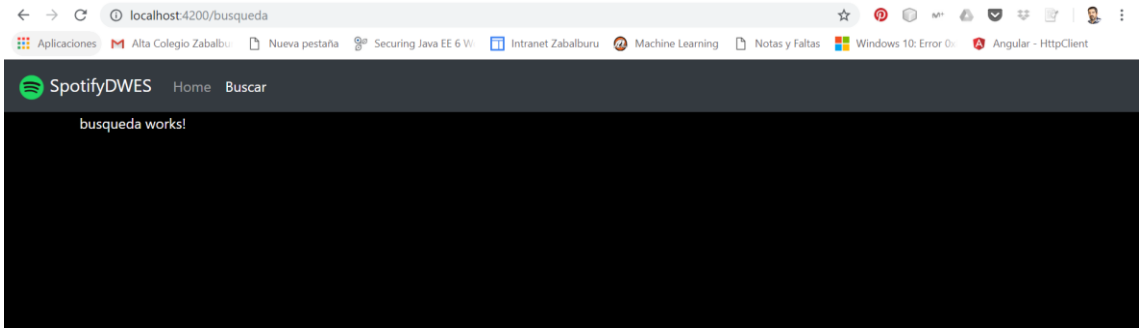
Ahora modificamos el componente **barra-navegación** a partir de la plantilla de la web de Bootstrap:

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
   <a class="navbar-brand" routerLink="home">
      <img    src="/src/assets/imagenes/banner-ico.png"    width="30"
height="30" alt="">
     SpotifyDWES</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" routerLinkActive='active'>
```

```
        <a class="nav-link" routerLink='home' >Home </a>
      </li>
      <li class="nav-item" routerLinkActive='active'>
        <a class="nav-link" routerLink='busqueda'>Buscar</a>
      </li>
    </ul>


  </div>
</nav>
```

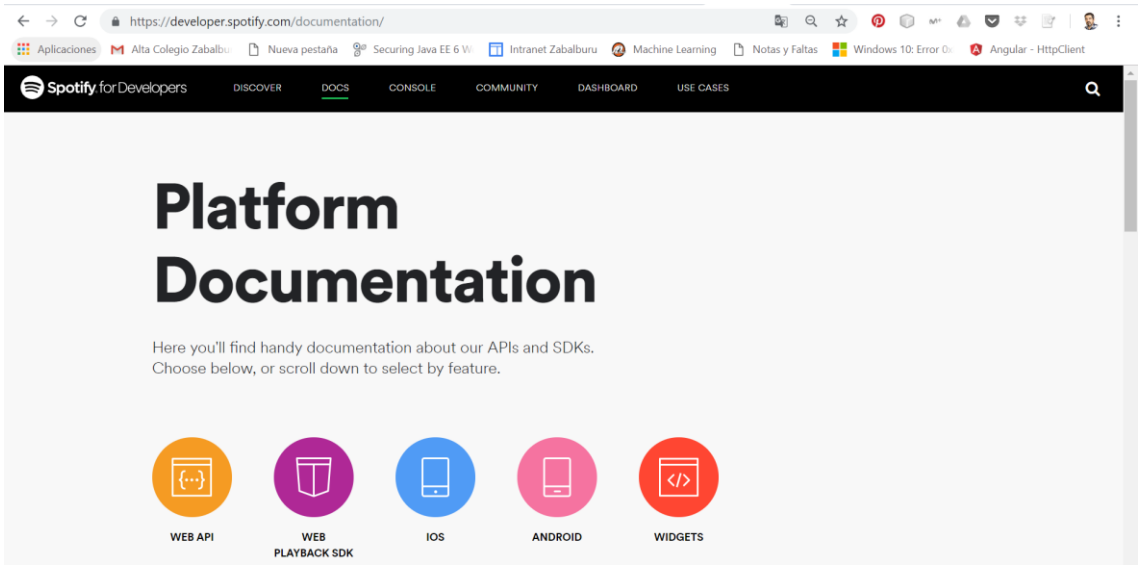
NOTA: Hay que copiar una imagen para el logo en la carpeta **assets/imágenes**.

La salida:



API de Spotify

En la pestaña **Docs** del **dashboard** de Spotify podemos acceder a toda la documentación sobe el API de Spotify:



En nuestro caso vamos a trabajar con **Web API** (y, posteriormente, con los **widgets**). Si entramos en **Web API** podemos ver toda la documentación sobre el proceso necesario para usar el API vía peticiones web.

Específicamente, en la parte **Guides → Authorization Guide** es donde tenemos información sobre el proceso de autenticación y obtención del **token** necesario para realizar las consultas.

Siguiendo esa documentación encontramos que, para obtener el token, tenemos varias opciones. Entre ellas, vamos a seleccionar **Client Credentials Flow** que es la más simple dado que no requiere **autenticación** por parte del usuario (por tanto no se pueden realizar tareas que impliquen gestionar los datos del usuario). En esta situación, tenemos que hacer la siguiente petición:

```
POST https://accounts.spotify.com/api/token
```

En la cabecera de la petición se debe enviar el parámetro **Authorization** con el **id y la contraseña del cliente** codificados en Base64 (**client_id:client_secret**).

En el cuerpo de la petición, se debe enviar el parámetro **grant_type** con el texto **client_credentials**.

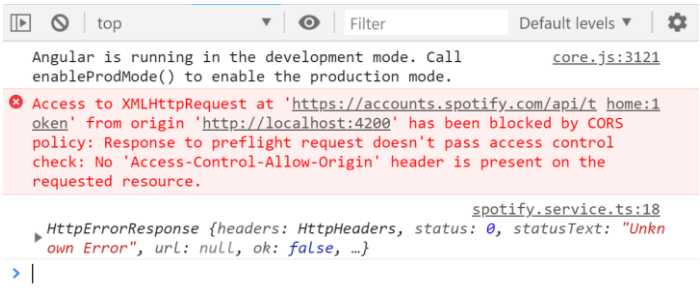
El resultado será un **JSON** con información sobre el **token**.

CORS

Si intentamos acceder a la autenticación de Spotify desde la aplicación Angular comprobaremos que no es posible, dado que nos da un error de **CORS** (Cross Origin Resource Shaing). Esto es debido a que los navegadores no permiten acceder a recursos **que no estén en su propio dominio** (su propio servidor). Para solucionar esto, los servidores se pueden configurar para permitir dicho acceso a través de CORS.

Dado que todos los navegadores modernos implementan CORS, cuando hacen una petición a otro dominio comprueban si, **en la respuesta aparece una cabecer Access-Control-Allow-Origin** con los dominios a los que se les permite el acceso al recurso (***** si se permite a cualquiera).

De este modo, si el servido no envía dicha cabecera (Spotify no lo hace), el navegador nos dará un error:



Para solucionar este error necesitamos un elemento intermedio (un servidor) que se encargue de hacer la petición por nosotros. Podríamos hacer, por ejemplo, un **servlet** aunque, en nuestro caso, vamos a optar por hacer una aplicación **node.js**. Como sabemos **node.js** es el servidor basado en javascript sobre el que se ejecuta Angular.

Los pasos a dar son los siguientes:

- Creamos una carpeta en el equipo (por ejemplo SpotyAuth)
- En dicha carpeta creamos los ficheros **package.json** con las dependencias del proyecto y **app.js** con la aplicación propiamente dicha

package.json

```
{
  "author": "ichueca",
  "name": "SpotiAuth",
  "description": "Conexion a Spotify con Client Credentials",
  "version": "1.0.0",
  "private": true,
  "dependencies": {
    "express": "~4.16.0",
    "request": "~2.83.0"
  }
}
```

El paquete **express** nos permite una forma simplificada de desarrollar aplicaciones web con node.js. Por su parte, **request** nos permite realizar peticiones web

app.js

```
/**
*/
var express = require('express');
var request = require('request');

var client_id = 'client_id'; // De la web de Spotify
var client_secret = 'client_secret'; // Lo mismo

// Opciones de autenticación
var authOptions = {
  url: 'https://accounts.spotify.com/api/token',
  headers: {
    'Authorization': 'Basic ' +
      (new Buffer(client_id + ':' + client_secret).toString('base64'))
  },
  form: {
    grant_type: 'client_credentials'
  },
  json: true
```

```
};

var app = express();
var token;
var expires = Date.now();
app.get('/', function(req, res) {
    res.header("Access-Control-Allow-Origin", "*");
    if (Date.now() > expires) {
        request.post(authOptions,
            function(error, response, body) {
                token = body;
                res.send(token);
                expires = Date.now() + body.expires_in*1000;
            });
      } else {
            res.send(token);
      }
});

console.log('Escuchando en 8888');
app.listen(8888);
```

Básicamente la aplicación realiza las siguientes tareas:

- Declara las librerías necesarias (express / request)
- Almacenamos en variables las credenciales de Spotify y dos variables para almacenar el **token** y el momento en que **expira**
- Definimos las cabeceras para la autenticación como se indica en la web de Spotify (https://developer.spotify.com/documentation/general/guides/authorization-guide/) en la parte **Client Credentials flow**
    - Empleamos una cabecera **Authentication** con **Basic cadena** donde **cadena** es **client_id:client_secret** codificado en Base64
    - En el **cuerpo** del mensaje enviamos **grant_type=client_credentials**
    - Indicamos que esperamos un **json**
- Creamos una **aplicación de Express** y la almacenamos en la variable **app**. Esta aplicación es capaz de responder a diferentes peticiones (**GET, POST…**) en **diferentes rutas** (incluso dinámicas)
- Añadimos una **ruta** para una petición **GET** al **raíz** y le asociamos una función que recibe la petición y la respuesta de / al cliente :

- **Si hay que obtener un nuevo token**
    - Añadimos la cabecera **Access Control Allow Origin** para indicar al navegador que aceptamos peticiones de otros dominios
    - Realiza un **POST** mediante la librería **request** a la URL de autenticación de Spotify para solicitar un **token**.
    - A dicha petición se le pasa la autorización creada anteriormente y se le asigna otra función que recibe tres parámetros cuando finaliza la petición: **error**: información sobre los errores que hayan podido suceder, **response** : información sobre la respuesta del servidor y **body**: la respuesta propiamente dicha.
    - En dicha función
        - Si **no hay errores** y el **código de la respuesta** es **200** (OK)
            - Reenviamos el **body** al cliente mediante la **respuesta del cliente**
            - Almacenamos el **token** y **calculamos la nueva fecha de expiración**
- **Si no hay que renovar el token**, enviamos el que tenemos **almacenado**

Una vez creados los ficheros debemos instalar las dependencias así que, desde el directorio de la aplicación ejecutamos:

```
npm install
```

```
npm notice created a lockfile as package-lock.json. You should commit this file.

added 96 packages in 12.743s
```

Por ultimo, basta con ejecutar la aplicación **node.js**:

```
node app.js
```

```
Escuchando en 8888
```

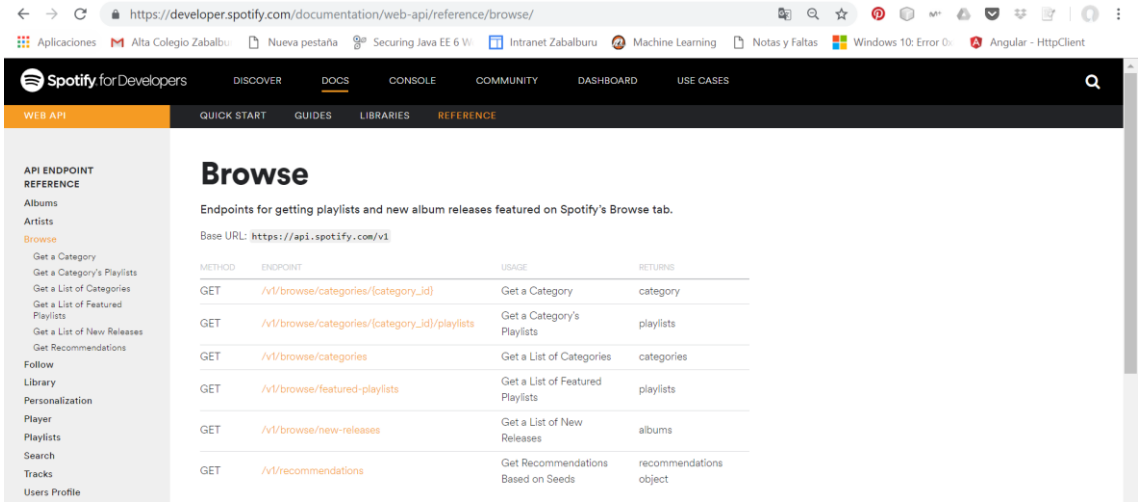
Si vamos a http://localhost:8888:

Ahora podemos, desde nuestro servicio, acceder al servidor **node.js** para que haga de intermediario y nos devuelva el token de Spotify (NOTA: como se puede ver el token sólo es válido durante **3600 segundos** (una hora)).

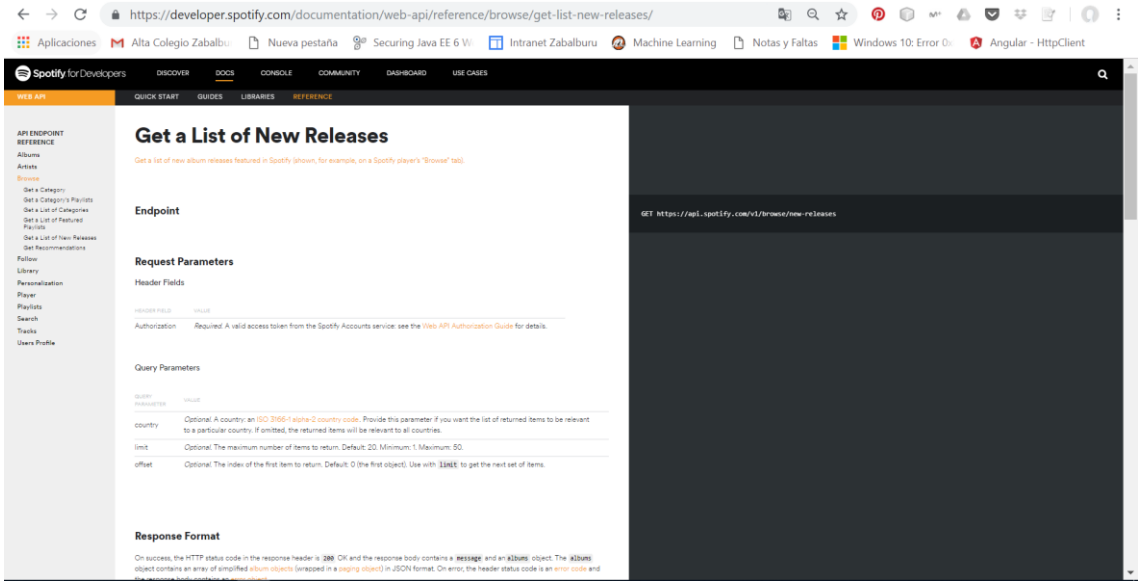
Obtener las novedades

Una vez que podemos obtener el token, vamos a ve un pequeño ejemplo de cómo podemos obtener las novedades de Spotify.

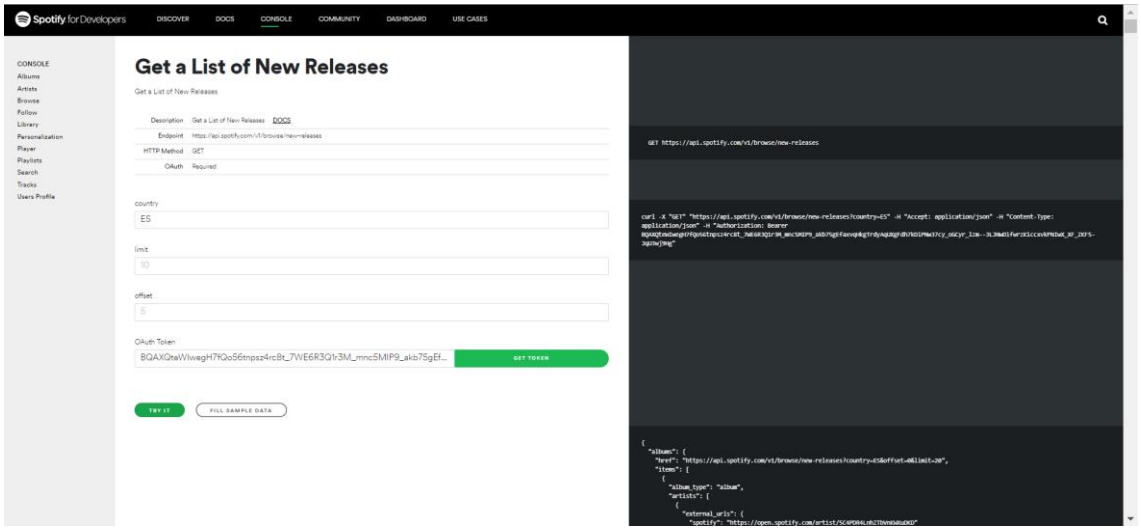
En la página de Spotify, desde **Docs → Web API → Reference** podemos obtener información sobre cualquier aspecto de Spotify. Por ejemplo, en la categoría **Browse** tenemos las peticiones necesarias para acceder a los diferentes **endpoints** de Spotify:



Si nos centramos en **new-releases** podemos ver información sobre las cabeceras a enviar, los parámetros disponibles y el formato de la respuesta:



Incluso disponemos de un botón para probar el API:



Con la documentación podemos observar que la petición nos retorna un objeto JSON que incluye un objeto **albums** con los siguientes elementos destacables:

- **href** : La URL con la petición realizada
- **ítems**: Una matriz con todos los álbumes encontrados. De cada álbum:
    - **album_type** : Puede ser **"album", "single"** o **"compilation"**
    - **artists** : Una matriz con los artistas que participan en el álbum. De cada artista:
        - **external_urls** : Matriz con enlaces a información del artista
        - **href**: Enlace a la página del artista en Spotify
        - **id**: Identificador del artista en Spotify (SpotifyId)
        - **name**: Nombre
        - **type**: Tipo de objeto (**"artist"**)
        - **uri**: Uri del artista en Spotify (SpotifyURI)
    - **available_markets**: Una matriz con los códigos de los países donde está disponible el álbum
    - **external_urls**: URLs externas a Spotify con información del álbum
    - **href**: URL al endpoint de Spotify con información detallada del álbum
    - **id**: El id de Spotify del álbum
    - **images**: Una matriz con imágenes de la portada del álbum. De cada imagen
        - **heigth**: Altura de la imagen
        - **width**: Ancho de la imagen
        - **url**: URL a la imagen
    - **name**: El nombre del álbum
    - **type**: El tipo de objeto (**"album"**)
    - **uri**: La Spotify URI del álbum
- **limit**: Cuántos álbumes queremos recuperar
- **next**: URL para solicitar el siguiente grupo de álbumes
- **offset**: El desplazamiento respecto al origen de los datos
- **previous**: URL para solicitar los álbumes anteriores
- **total**: Número total de álbumes

Para poder hacer la petición **GET** es necesario pasar un parámetro en la cabecera que sea:

Authorization:Bearer *token*

¿Cómo podemos acceder a la información desde la aplicación? Tendríamos dos opciones:

- Acceder directamente a Spotify desde nuestra aplicación
- Crear nuestro propio API empleando Node.js (evitando así las restricciones de acceso)

En cualquiera de los casos, tenemos un problema:

Las consultas http son **operaciones asíncronas**, es decir que si ejecutáramos las siguientes peticiones:

- Definir una variable **token**
- Solicitar **token** a Spotify y almacenarlo en **token**
- Solicitar álbumes **empleando el token**

La segunda petición se realizaría **antes de recuperar el token** (no hay forma con **HttpClient** de realizar consultas síncronas dado que podrían bloquear la ejecución de la aplicación.

¿Cómo podemos solucionar este problema?

- Una posible solución sería solicitar los álbumes cuando recibimos el token (es decir en el método **subscribe**). El problema es que, si queremos hacer esto en el servicio propagamos el problema porque **no podemos retornar** el Observable de la **segunda petición** desde el servicio (sí se podría hacer en el componente pero esto rompe la arquitectura de la aplicación).
- La solución que vamos a utilizar es emplear **Promesas**. Las **Promesas** (**Promise**) es código que se va a ejecutar en algún momento y del que, en algún momento, podemos saber cuándo **ha finalizado** (o si se ha rechazado). Una promesa no deja de ser un tipo de observable.

Promesas

Para crear una promesa empleamos el siguiente código:

```
promesa:Promise = new Promise ( (resolv,reject) => {
    código;
    resolv(datos); ❼ Ha hábido éxito y se retorna un resultado
    reject(error); ❼ No ha habido éxito y se retorna un error
});
```

Cuando queramos ejecutar código cuando finalice la promesa:

```
promesa.then(datos => código).catch(error => código)
```

Acceder al API de Spotify directamente con promesas

En primer lugar en **app.module.ts** importamos el módulo **HttpClientModule**:

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

import { AppComponent } from './app.component';
…
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent,
    …
  ],
  imports: [
```

```
    BrowserModule,
    FeatureRoutingModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

creamos el servicio para Spotify:

```
ng g s services\spotify
```

En nuestro caso, empleando **promesas**, el código podría ser similar al siguiente:

```
getNovedades():Promise<any[]>{
    return new Promise((resolv,response) =>
      {
        this.http.get("http://localhost:8888").subscribe(
          (datos:any) => {
            this.http.get('https://api.spotify.com/v1/browse/new-
releases',
              {
                headers:{'Authorization':'Bearer
'+datos.access_token},
                params:{'country':'ES'}
              }
          ) .subscribe((datos:string) => resolv(datos.albums.items));
        });
      })
    }
```

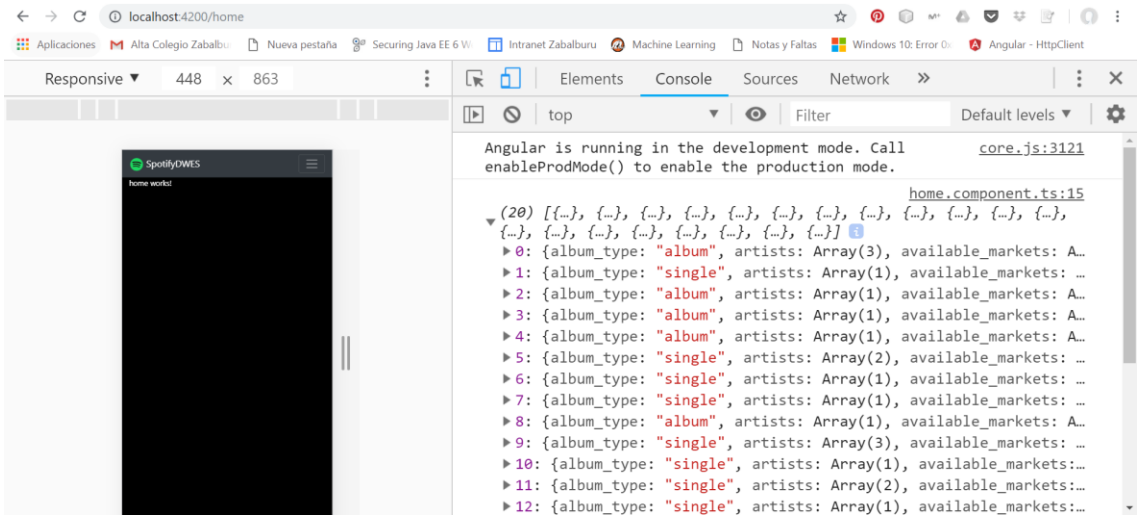
Como se puede ver, básicamente creamos una promesa que **se resuelve cuando recibimos la respuesta desde Spotify**. La promesa se resuelve en una **matriz de objetos** con los **álbumes** que son novedad (en la propiedad **ítems** dentro del objeto **albums** devuelto por Spotify).

Para poder probarlo vamos a usar el servicio en **home.component.ts**:

```
import { Component, OnInit } from '@angular/core';
import { SpotifyService } from 'src/app/services/spotify.service';


@Component({
```

```
  selector: 'app-home',
  templateUrl: './home.component.html',
  styles: []
})
export class HomeComponent implements OnInit {
  private novedades:any[] = [];
  constructor(service:SpotifyService) {
      service.getNovedades().then( data => {
        this.novedades = data;
        console.log(this.novedades);
      })
        ;
   }

  ngOnInit() {
  }

}
```

La salida:

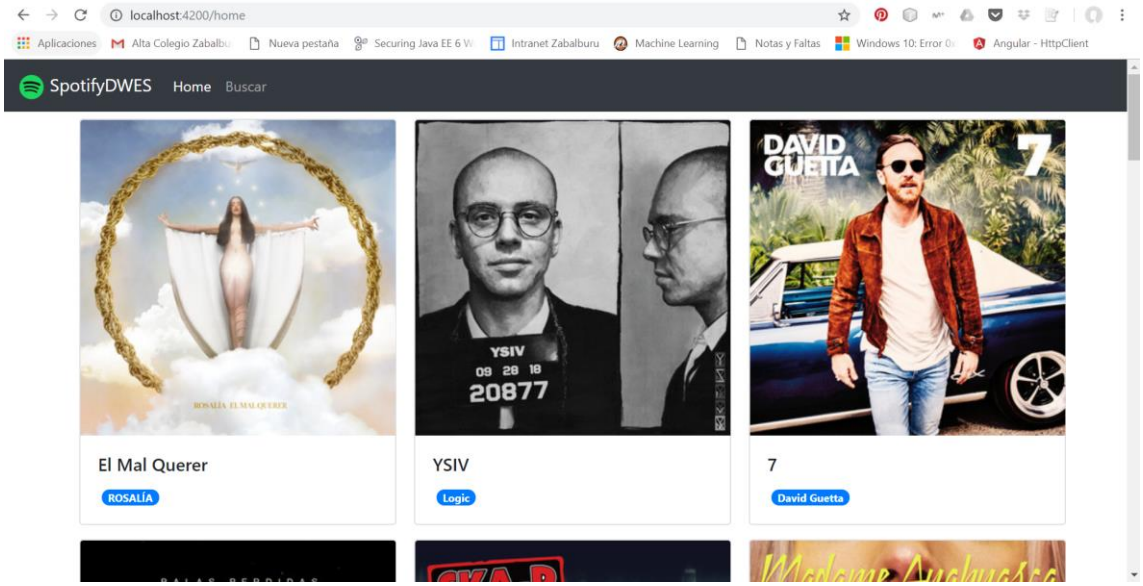


Ahora ya podemos mostrar los álbumes con elementos **card** de **Bootstrap**:

```
<div class="card-columns">
  <div *ngFor="let novedad of novedades" class="card m-2">
    <img  class="card-img-top"  [src]="novedad.images[1].url"  alt="{{ 'Imagen del álbum ' + novedad.name  }}">
    <div class="card-body">
```

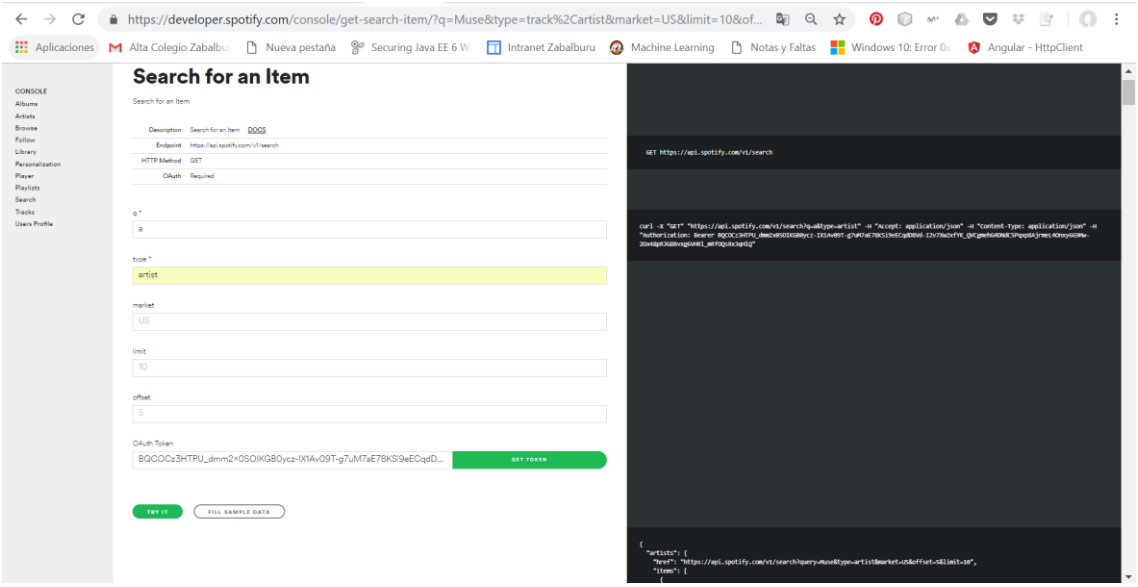
```
      <h5 class="card-title">{{novedad.name}}</h5>


      <p class="card-text">
        <span *ngFor="let artista of novedad.artists" class="badge badge-pill badge-primary ml-1">
          {{artista.name}}
        </span>
      </p>
    </div>
  </div>
</div>
```

La salida



Acceder al API de Spotify desde el servidor (Búsqueda de Artista)

El siguiente paso es permitir la búsqueda de artistas en la opción **Buscar**. En la web de Spotify tenemos el punto de acceso para diferentes tipos de búsquedas:



Como se puede ver, básicamente es un GET sobre https://api.spotify.com/v1/search pasándole la autorización y una serie de parámetros (emplearemos **q** y **type**).

La respuesta consta de la siguiente información:

- **external_urls**: Direcciones del artista externas a Spotify
- **followers**: Un objeto con información de los seguidores:
    - **href**: Actualmente siempre es **null** (se supone que llevará a una página con información sobre seguidores)
    - **total**: Número total de seguidores
- **genres**: Géneros musicales (matriz)
- **href**: URL de Spotify del artista
- **id**: El SpotifyID del artista
- **images**: Igual que los albums
- **name**: Nombre del artista
- **popularity**: Popularidad del artista en Spotify (un valor entre 0 y 100)
- **type**: artist
- **uri**: Uri de Spotify del artista

En este caso lo vamos a hacer en la parte de **Node.js**. En este caso vamos a hacer dos consultas anidadas (una para obtener el **token** del propio servidor y otra para acceder a la API de Spotify):

```
app.get('/search/:artist', (req, res) => {
        res.header("Access-Control-Allow-Origin", "*");
        var auth={
                url:'http://localhost:8888/',
                json:true
        }
        request.get(auth, (error,response,body) => {
                var token = body.access_token;
```

```
            auth.url = 'https://api.spotify.com/v1/search';
            auth.headers = {'Authorization':'Bearer '+token};
            auth.qs = { 'q':req.params.artist,
                        'type':'artist'};
            request.get(auth, (error,response,data) => {
                  res.send(data.artists.items);
            }
            )
      })
});
```

Como se puede ver, la url incluirá un parámetro que podemos leer mediante la propiedad **params** del objeto **req**. Por otro lado, para enviar parámetros al servidor (**q** y **type**) empleamos la propiedad **qs** (query string) del objeto **auth** del **request**.

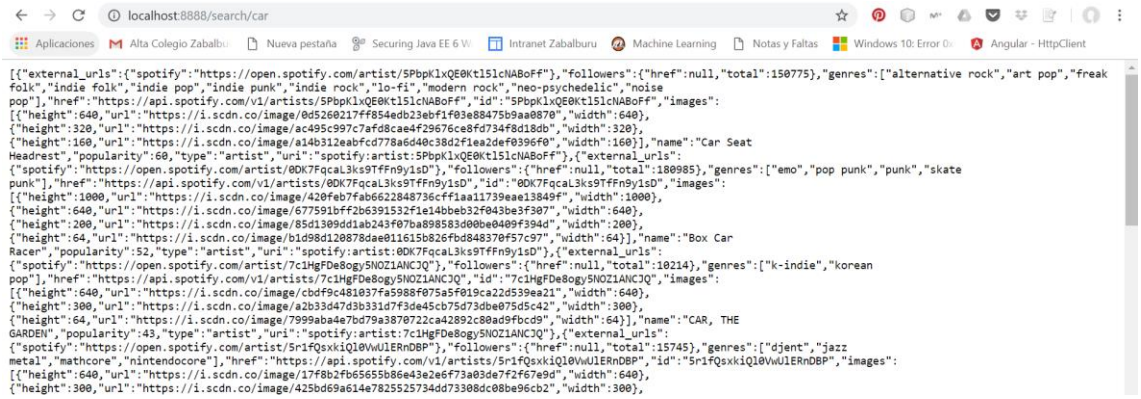
NOTA: También podríamos haber hecho lo mismo mediante **Promesas**. En este caso:

```
app.get('/searchPromise/:artist', (req, res) => {
      res.header("Access-Control-Allow-Origin", "*");
      var auth={
            url:'http://localhost:8888/',
            json:true
      }
      let p = new Promise( (resolv,reject) => {
                  request.get(auth, (error,response,body) =>{
                              resolv(body.access_token);
                        })
                  }
            );
      p.then( token => {
            auth.url = 'https://api.spotify.com/v1/search';
            auth.headers = {'Authorization':'Bearer '+token};
            auth.qs = { 'q':req.params.artist,
                                    'type':'artist'};
            request.get(auth, (error,response,data) => {
                        res.send(data.artists.items);
                  }
            )
```

```
        });
});
```

Reiniciamos el servidor.

Si vamos a la URL http://localhost:8888/search/car (o http://localhost:8888/searchPromise/car):



Ahora ya podemos acceder a dicha información desde el servicio:

```
buscarArtista(artista:string):Observable<any[]>{
    return <Observable<any[]>> this.http.get(
`http://localhost:8888/search/${artista}`); }
```

NOTA: En este caso hemos hecho uso de las llamadas **plantillas de cadena de texto** en las que se puede **incrustar expresiones dentro de una cadena** empleando **${espresion}**. Para indicar que queremos usar una plantilla de texto se emplea el delimitador **` (back tick)**.

NOTA: Dado que retornamos una matriz de artistas, lo indicamos en el método indicando Observable<any[]>.

El siguiente paso es modificar el componente de búsqueda.

Empezamos creando el componente de búsqueda:

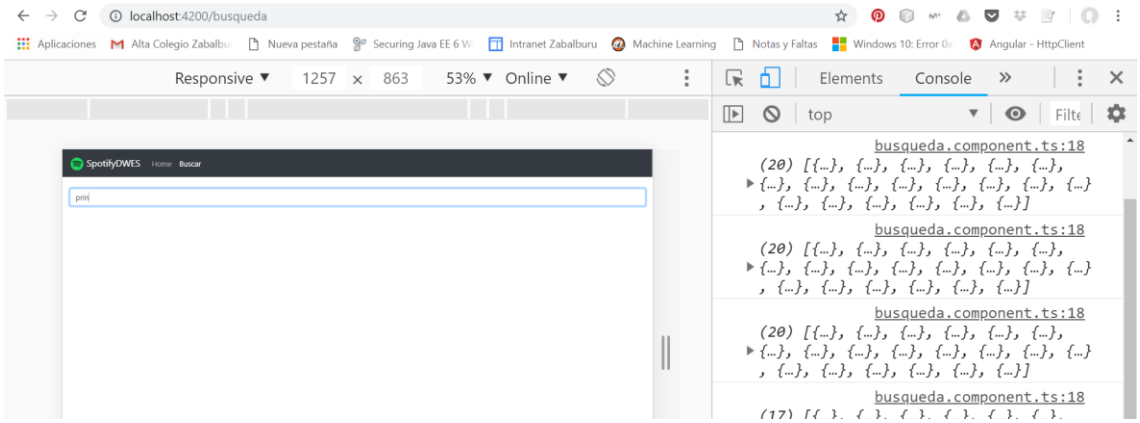
```
<div class="row">
  <div class="col">
    <input #artista type="text" (keyup)="buscar(artista)" class="form-control" placeholder="Artista">
  </div>
</div>
```

En nuestro caso, vamos a llamar al método **buscar** pasando una referencia al **input** a través de la variable de plantilla **artista**. El código, de momento sólo va a llamar al servicio y retornar los datos por consola:

```
import { Component, OnInit } from '@angular/core';
import { SpotifyService } from 'src/app/services/spotify.service';

@Component({
```

```
  selector: 'app-busqueda',
  templateUrl: './busqueda.component.html',
  styles: []
})
export class BusquedaComponent implements OnInit {

  constructor(private service:SpotifyService) { }

  ngOnInit() {
  }

  buscar(artista:HTMLInputElement){
    this.service.buscarArtista(artista.value).subscribe(
      (data) => console.log(data)
    );
  }
}
```

Si probamos:



Cada vez que pulsamos una tecla se envía una petición al servidor que nos devuelve los artistas con ese texto. Vamos a guardar la información en una matriz:

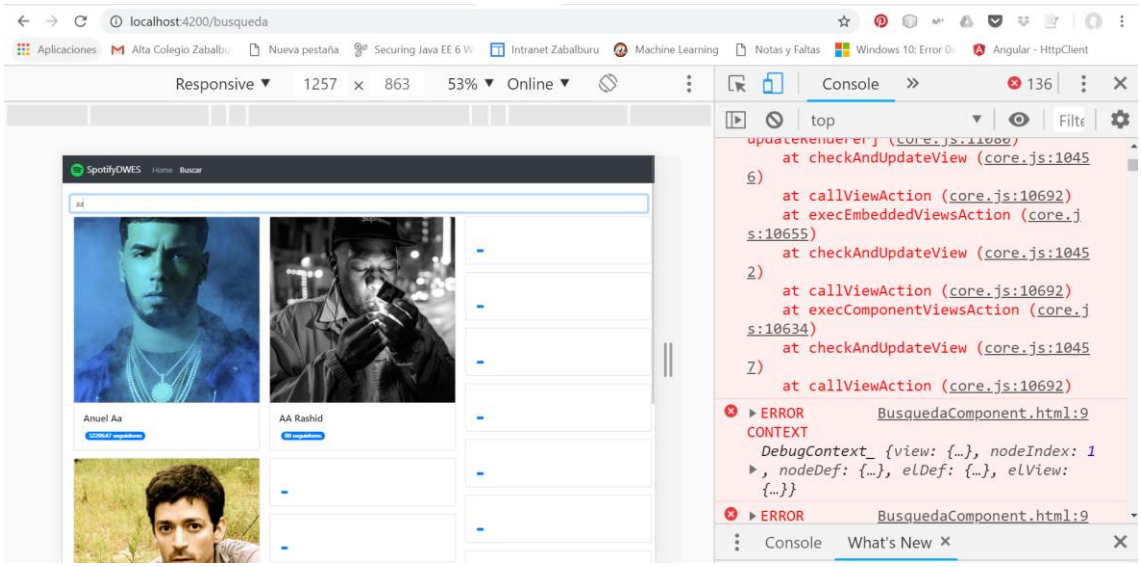
```
export class BusquedaComponent implements OnInit {
  private artistas:any[] = [];
  constructor(private service:SpotifyService) { }

  ngOnInit() {
  }
```

```
  buscar(artista:HTMLInputElement){
    this.service.buscarArtista(artista.value).subscribe(
      (data:any[]) => this.artistas = data
    );
  }
}
```

Ahora modificamos el HTML para que nos muestre la información del artista:

```
<div class="row">
  <div class="col">
    <input #artista type="text" (keyup)="buscar(artista)" class="form-
control" placeholder="Artista">
  </div>
</div>


<div class="card-columns">
    <div *ngFor="let artista of artistas" class="card m-2">
      <img class="card-img-top" [src]="artista.images[1].url" alt="{{
'Imagen del artista ' + artista.name }}">
      <div class="card-body">
        <h5 class="card-title">{{artista.name}}</h5>


        <span class="badge badge-pill badge-primary ml-1">
            {{artista.followers.total}} seguidores
        </span>
      </div>
    </div>
</div>
```

La salida tecleando **aa**:



Como podemos ver no aparecen todos los artistas y nos lanza un error. El problema esta en que hay artistas de los que **no hay imágenes** y por eso el código falla y no continúa mostrando su información.

Podríamos solucionar esto con un **ngIf** pero, dado que también se pueden dar en las novedades y en otras imágenes podemos hacerlo de una mejor manera mediante el uso de los **pipes**.

Pipes

Los **pipes** no son más que una clase de Angular capaz de **transformar una información en otra**. Para aplicar un **pipe** a un elemento se emplea el operador **|** y Angular viene con un serie de pipes **predefinidos**. Entre ellos:

- **CurrencyPipe**: Convierte el dato a una cadena con formato monetario. Se puede pasar un parámetro con **currency[:{'code' | 'symbol'}[:maxEnt.minDec-maxDec[:locale]]**
- **DatePipe**: A una fecha **date[:formato[:timezone[:locale]]]** donde **formato** puede ser **short**, **médium**, **long**, **full**, **shortDate**, **mediumDate**, **longDate**, **fullDate**, **shortTime**, **mediumTime**, **longTime**, **fullTime** o una **cadena de formato** (como la de la clase **DateFormart** de Java)
- **DecimalPipe**: El formato es **number[:maxEnt.minDec-maxDec[:locale]]**
- **JsonPipe**: Convierte un objeto a JSON. Formato **json**
- **LowerCasePipe / UpperCasePipe**: A minúsculas / mayúsculas. Fomato **lowercase** / **uppercase**
- **PercentPipe**: Multiplica el número por 100 y le pone el símbolo **%**. Formato **percent[:maxEnt.minDec-maxDec[:locale]]**
- **SlicePipe**: Extrae parte de una colección. Formato **slice:'inicial:final'** (**final**, si es negativo, comienza a contar desde el final de la colección)
- **AsyncPipe**: Se ejecuta sobre una promesa o un observable y retorna el resultado de la misma

Ejemplo:

12345.678 | currency	$12,345.68
12345.678 | currency:'EUR':'code'	EUR12,345.68
12345.678 | currency:'EUR':'symbol'	€12,345.68
fecha | date	Nov 9, 2018
fecha | date:'short'	11/9/18, 1:24 PM
fecha | date:'medium'	Nov 9, 2018, 1:24:28 PM
fecha | date:'long'	November 9, 2018 at 1:24:28 PM GMT+1
fecha | date:'full'	Friday, November 9, 2018 at 1:24:28 PM GMT+01:00
fecha | date:'shortDate'	11/9/18
fecha | date:'shortTime'	1:24 PM
fecha | date:'dd/MM/yyyy'	09/11/2018
12345.678 | number	12,345.678
12345.678 | number:'6.0-0'	012,346
12345.678 | number:'5.2-2'	12,345.68
12345.678 | number:'3.2-4'	12,345.678
objeto | json	{ "nombre": "Juan", "edad": 20 }
'Juan' | lowercase	juan
'Juan' | uppercase	JUAN
0.12345 | percent:'3.0-2'	012.35%
[1,2,3,4,5] | slice:1	2,3,4,5
[1,2,3,4,5] | slice:1:4	2,3,4
[1,2,3,4,5] | slice:1:-1	2,3,4
(novedad | async)[0].name	El Mal Querer

Donde **novedad** es un observable que devuelve las novedades de Spotify

Localización de los pipes

Los pipes emplean el **locale por defecto** de Angular. En principio, sólo se incluye en el proyecto el **locale** en inglés (**en**). Es por esto que los mensajes y los formatos aparecen en inglés.

Si queremos emplear otros locales, deberemos incluirlos en el fichero **app.module.ts**:

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

```
import { AppComponent } from './app.component';
…
import { registerLocaleData } from '@angular/common';
import localeEs  from '@angular/common/locales/es';

registerLocaleData(localeEs);
…
```

Podemos incluir multiples locales y emplearlos en los pipes con posibilidad de locale:

```
Ahora
{{ 12345.678 | currency:'EUR':'symbol':'4.2':'es'}} ❼ 12.345,68€
{{ fecha | date:'long':undefined:'es'}} -> 9 de noviembre de 2018, 13:38:46 GMT+1
```

Si lo que queremos es emplear el castellano como idioma predeterminado debemos modificar de nuevo **app.module.ts**:

```
import { BrowserModule } from '@angular/platform-browser';
…
import { registerLocaleData } from '@angular/common';
import localeEs  from '@angular/common/locales/es';

registerLocaleData(localeEs);

@NgModule({
  declarations: [
 …
  ],
  imports: [
…
  ],
  providers: [{ provide: LOCALE_ID, useValue: 'es' }],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Si volvemos a cargar la página de prueba:

12345.678 | currency	12.345,68 US$
12345.678 | currency:'EUR':'code'	12.345,68 EUR
12345.678 | currency:'EUR':'symbol'	12.345,68 €
fecha | date	9 nov. 2018
fecha | date:'short'	9/11/18 13:43
fecha | date:'medium'	9 nov. 2018 13:43:26
fecha | date:'long'	9 de noviembre de 2018, 13:43:26 GMT+1
fecha | date:'full'	viernes, 9 de noviembre de 2018, 13:43:26 (GMT+01:00)
fecha | date:'shortDate'	9/11/18
fecha | date:'shortTime'	13:43
fecha | date:'dd/MM/yyyy'	09/11/2018
12345.678 | number	12.345,678
12345.678 | number:'6.0-0'	012.346
12345.678 | number:'5.2-2'	12.345,68
12345.678 | number:'3.2-4'	12.345,678
objeto | json	{ "nombre": "Juan", "edad": 20 }
'Juan' | lowercase	juan
'Juan' | uppercase	JUAN
0.12345 | percent:'3.0-2'	012,35 %
[1,2,3,4,5] | slice:1	2,3,4,5
[1,2,3,4,5] | slice:1:4	2,3,4
[1,2,3,4,5] | slice:1:-1	2,3,4
(novedad | async)[0].name	El Mal Querer

Pipes personalizados

Los pipes no son más que clases con un decorador que implementan un método específico y que transforman una entrada en una salida.

Volviendo a nuestro problema queremos un pipe que recibe un parámetro que **si no existe o está vacío** deberá retornar una ruta a una **imagen de no foto**.

Creamos el pipe con el CLI:

```
ng g pipe pipes/imageURL --spec=false
CREATE src/app/pipes/image-url.pipe.ts (205 bytes)
UPDATE src/app/app.module.ts (1149 bytes)
```

Como vemos, hay que modifica el fichero **app.module.ts** para poder usa el pipe (lo hace el CLI por nosotros):

```
…
import { ImageURLPipe } from './pipes/image-url.pipe';
…

@NgModule({
  declarations: [
    AppComponent,
…
    ImageURLPipe
  ],
…
```

En nuestro caso, el problema (en artistas y álbumes) es que recuperamos la imagen con:

artista.images[1].url ó album.images[1]

y, en ambos casos, **images[1]** puede **no existir**. La solución pasa por, en ese caso, enviar otra URL:

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

@Pipe({
  name: 'imageURL'
})
export class ImageURLPipe implements PipeTransform {

  transform(value: any, args?: any): any {

    if (!value || value.length == 0){
      value = "…C";
    } else {
```

```
      value = value.url; // queremos la url de la imagen
    }
    return value;
  }
}
```

En **búsqueda.component.html** ponemos el código para mostrar los artistas (incluyendo el nuevo pipe). Empleamos, además, una **variable de plantilla** para acceder al valor del cuadro de texto desde el código del componente:

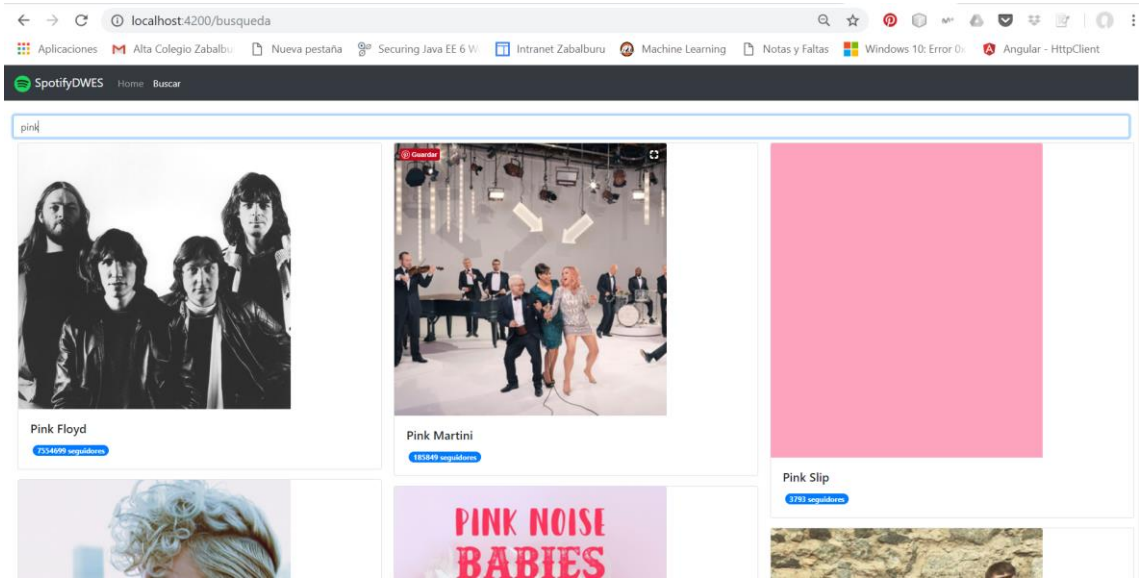
```
<div class="row">
  <div class="col">
    <input #artista type="text" (keyup)="buscar(artista)" class="form-control" placeholder="Artista">
  </div>
</div>


<div class="card-columns">
  <div *ngFor="let artista of artistas" class="card m-2">
    <img class="card-img-top" style="width:75%" [src]="artista.images[1] | imageURL" alt="{{ 'Imagen del artista ' + artista.name }}">
    <div class="card-body">
      <h5 class="card-title">{{artista.name}}</h5>


      <span class="badge badge-pill badge-primary ml-1">
        {{artista.followers.total}} seguidores
      </span>


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

Por ultimo, en **búsqueda.component.ts**:

```
import { Component, OnInit } from '@angular/core';
import { SpotifyService } from 'src/app/services/spotify.service';


@Component({
  selector: 'app-busqueda',
```

```
  templateUrl: './busqueda.component.html',
  styles: []
})
export class BusquedaComponent implements OnInit {
  private artistas:any[] = [];
  constructor(private service:SpotifyService) { }

  ngOnInit() {
  }

  buscar(artista:HTMLInputElement){
    this.service.buscarArtista(artista.value).subscribe(
      (data) => this.artistas = data
    );

  }
}
```

La salida:



Página del Artista

Tanto desde los **badges** de la página principal como desde las imágenes de la página de búsqueda, vamos a permitir al usuario ir a la página del artista correspondiente. Para ello, tendremos que añadir un método a nuestro servicio que nos permita **buscar un artista en base a su sid** (SpotifyId). Si miramos el API veremos que la petición es GET https://api.spotify.com/v1/artists/{id}

El resultado:

artist object (full)

KEY	VALUE TYPE	VALUE DESCRIPTION
external_urls	an external URL object	Known external URLs for this artist.
followers	A followers object	Information about the followers of the artist.
genres	array of strings	A list of the genres the artist is associated with. For example: `"Prog Rock"` , `"Post-Grunge"`. (If not yet classified, the array is empty.)
href	string	A link to the Web API endpoint providing full details of the artist.
id	string	The Spotify ID for the artist.
images	array of image objects	Images of the artist in various sizes, widest first.
name	string	The name of the artist
popularity	int	The popularity of the artist. The value will be between 0 and 100, with 100 being the most popular. The artist's popularity is calculated from the popularity of all the artist's tracks.
type	string	The object type: `"artist"`
uri	string	The Spotify URI for the artist.


```
{
  "external_urls" : {
    "spotify" : "https://open.spotify.com/artist/0OdUWJ0sBjDrqHygGUXeCF"
  },
  "followers" : {
    "href" : null,
    "total" : 306565
  },
  "genres" : [ "indie folk", "indie pop" ],
  "href" : "https://api.spotify.com/v1/artists/0OdUWJ0sBjDrqHygGUXeCF",
  "id" : "0OdUWJ0sBjDrqHygGUXeCF",
  "images" : [ {
    "height" : 816,
    "url" : "https://i.scdn.co/image/eb266625dab075341e8c4378a177a27370f91903",
    "width" : 1000
  }, {
    "height" : 522,
    "url" : "https://i.scdn.co/image/2f91c3cace3c5a6a48f3d8e2fd21364d4911b332",
    "width" : 640
  }, {
    "height" : 163,
    "url" : "https://i.scdn.co/image/2efc93d7ee8843511609327498bf04ebceb7b527",
    "width" : 200
  }, {
    "height" : 52,
    "url" : "https://i.scdn.co/image/4f25297750dfa4051195c36809a9049f6b841a23",
    "width" : 64
  } ],
  "name" : "Band of Horses",
  "popularity" : 59,
  "type" : "artist",
  "uri" : "spotify:artist:0OdUWJ0sBjDrqHygGUXeCF"
}
```


Que nos devuelve la misma información que la empleada en la búsqueda.

En primer lugar vamos a modificar la ruta a la página del artista para que reciba el **sid**. En **app-routes.ts**:

```
…
const routes: Routes = [
    { path: 'home', component: HomeComponent },
    { path: 'artista/:id', component: ArtistaComponent },
    { path: 'busqueda', component: BusquedaComponent },
    { path: '**', redirectTo: 'home' }
];
```

El siguiente paso es pasar el **sid** en los **badges** del **HomeComponent** cuando el usuario haga clic en ellos:

```
<div class="card-columns">
  <div *ngFor="let novedad of novedades" class="card m-2">
  …
      <p class="card-text">
        <span *ngFor="let artist of novedad.artists" style='cursor: pointer;' (click)='verArtista(artist.id)' class="badge badge-pill badge-primary ml-1">
          {{artist.name}}
        </span>
      </p>
    </div>
  </div>
</div>
```

Y añadimos el código en **home.component.ts** para que se redireccione a la página del artista pasando el id:

```
import { Component, OnInit } from '@angular/core';
import { SpotifyService } from 'src/app/services/spotify.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styles: []
})
export class HomeComponent implements OnInit {
  private novedades:any[] = [];
  private fecha = new Date();
  private objeto = {nombre:'Juan', edad:20};
  private novedad;
  constructor(private service:SpotifyService,
      private router:Router) {
      this.novedad = service.getNovedad();
      service.getNovedad().then( data => {
        this.novedades = data;
        console.log(this.novedades);
      })
        ;
   }

  verArtista(id:string){
    this.router.navigate(['artista',id]);
  }

  ngOnInit() {
  }
}
```

Algo similar hacemos en la página de búsqueda:

```
<div class="row">
  <div class="col">
    <input #artista type="text" (keyup)="buscar(artista)" class="form-control" placeholder="Artista">
  </div>
</div>


<div class="card-columns">
  <div *ngFor="let artista of artistas"
(click)="verArtista(artista.id)" style='cursor: pointer' class="card m-2">
    <img class="card-img-top" style="width:75%" [src]="artista.images[1] | imageURL" alt="{{ 'Imagen del artista ' + artista.name }}">
    <div class="card-body">
      <h5 class="card-title">{{artista.name}}</h5>


      <span class="badge badge-pill badge-primary ml-1">
        {{artista.followers.total}} seguidores
      </span>


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

El código es igual al caso anterior:

```
import { Component, OnInit } from '@angular/core';
import { SpotifyService } from 'src/app/services/spotify.service';
import { Router } from '@angular/router';


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

```
  private artistas:any[] = [];
  constructor(private service:SpotifyService,
    private router:Router) { }
…

  verArtista(id:number){
    this.router.navigate(['artista',id]);
  }
}
```

Definimos en el servicio el código para buscar la información del artista:

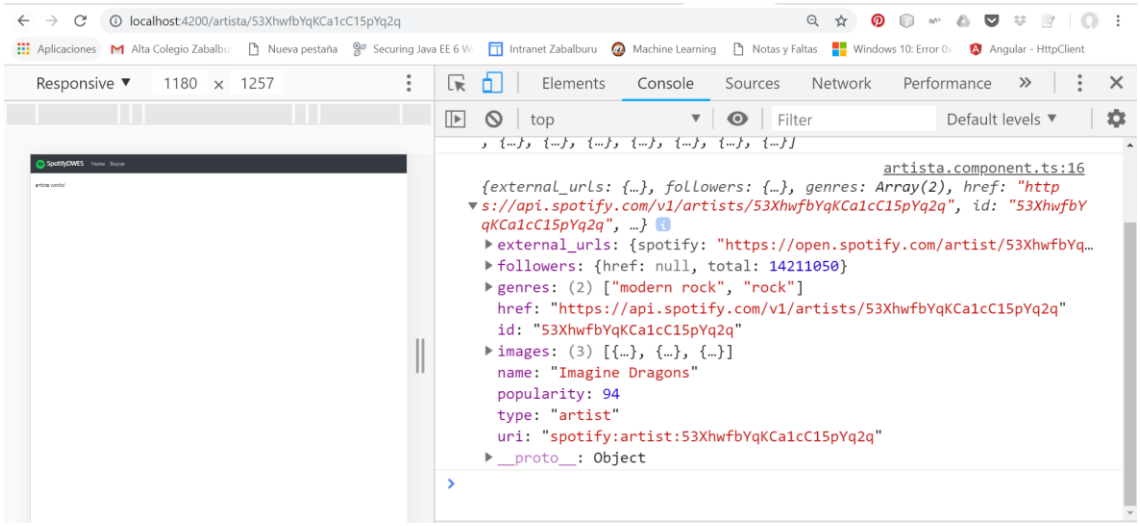
```
getArtista(id:any):Promise<any>{
      return new Promise((resolv,response) =>
        {
          this.http.get("http://localhost:8888").subscribe(
            (datos:any) => {

this.http.get(`https://api.spotify.com/v1/artists/${id}`,
                {
                  headers:{'Authorization':'Bearer
'+datos.access_token},
                  params:{'country':'ES'}
                }
            ) .subscribe((datos:any) => resolv(datos));
          });
        })
      }
```

Y ahora podemos llamar al servicio desde el componente artista. Comprobamos primero que funciona:

```
import { Component, OnInit } from '@angular/core';
import { SpotifyService } from 'src/app/services/spotify.service';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-artista',
  templateUrl: './artista.component.html',
```

```
  styles: []
})
export class ArtistaComponent implements OnInit {

  constructor(service:SpotifyService, link:ActivatedRoute) {
    link.params.subscribe(
      params => {
        service.getArtista(params.id).then(
          datos => console.log(datos)
        )
      }
    )
  }

  ngOnInit() {
  }

}
```

Si seleccionamos un artista:



Ahora sólo queda diseñar la página y modificar el código para pasar la información del artista:

```
import { Component, OnInit } from '@angular/core';
import { SpotifyService } from 'src/app/services/spotify.service';
import { ActivatedRoute } from '@angular/router';
```

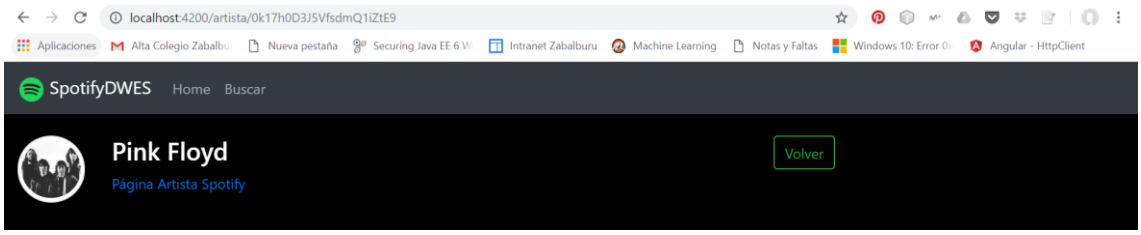
```
@Component({
  selector: 'app-artista',
  templateUrl: './artista.component.html',
  styles: []
})
export class ArtistaComponent implements OnInit {
  private artista:any;
  constructor(service:SpotifyService, link:ActivatedRoute) {
    link.params.subscribe(
      params => {
        service.getArtista(params.id).then(
          datos => {
            this.artista = datos;
          }
        )
      }
    )
  }
}
```

Y modificamos el HTML:

```
<div class="row">
  <div class="col-1 text-center">
    <img [src] ="artista.images[1] | imageURL"
[style.width]="'100px'" class="img-thumbnail rounded-circle">
  </div>
  <div class="col">
    <h3>{{ artista.name }}</h3>
    <p>
      <a [href]="artista.external_urls.spotify" target="_blank">Página
Artista Spotify</a>
    </p>
  </div>
  <div class="col-4">
    <button class="btn btn-outline-success"
routerLink="buscar">Volver</button>
  </div>
```

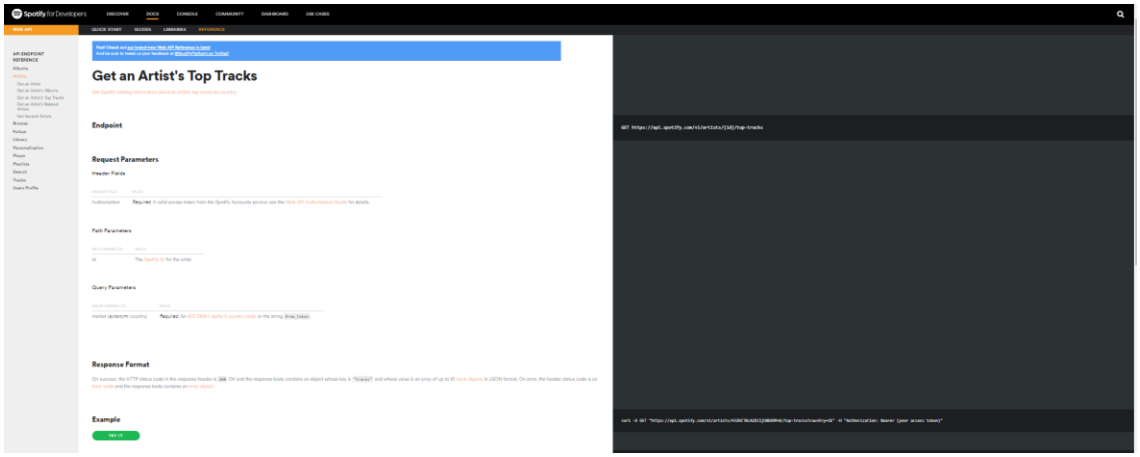
```
</div>
```

La salida ahora:



Mejores canciones

Para completar la página del artista vamos a localizar sus mejores canciones. Para ello vamos al Dashboard para ver la documentación de la búsqueda **top-tracks**:



La URL es https://api.spotify.com/v1/artists/{id}/top-tracks y como parámetro debemos proporcionar **market** con el código del país (**ES** para España) o **from_token** si no queremos especificar un país concreto.

El resultado es una **matriz** de objetos (**track**) con los siguientes campos:

- **album**: Un objeto de tipo Album
- **artists**: Una matriz de objetos de tipo Artista
- **available_markets**: Mercados en los que está disponible (matriz de strings)
- **disc_number**: El número del disco (normalmente 1)
- **duration_ms**: Duración en milisegundos
- **explicit**: Si las letras son explícitas (**true**) o no (**false**)
- **external_urls**: URLs externas (de momento **Spotify**)
- **href**: Enlace a información sobre la canción en Spotify
- **id**: El SID de la canción
- **name**: El nombre de la canción
- **popularity**: Un valor entre 0 y 100

- **preview_url**: URL de los primeros 30 segundos de la canción
- **track_number**: Número de la canción en el disco
- **type**: Es **track**
- **uri**: La URI de la canción en Spotify

Añadimos un nuevo método al servicio (o al servidor Node):

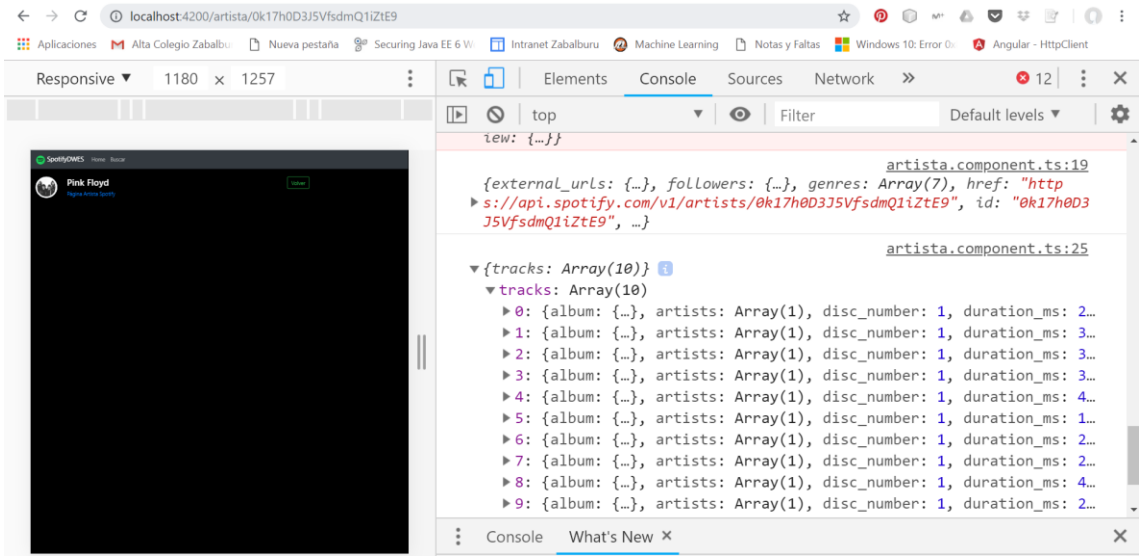
```
getCancionesArtista(id:any):Promise<any>{
        return new Promise((resolv,response) =>
          {
            this.http.get("http://localhost:8888").subscribe(
              (datos:any) => {

this.http.get(`https://api.spotify.com/v1/artists/${id}/top-tracks`,
                  {
                    headers:{'Authorization':'Bearer
'+datos.access_token},
                    params:{'market':'ES'}
                  }
              ) .subscribe((datos:any) => resolv(datos));
            });
          })
        }
```

Y lo leemos en el componente:

```
export class ArtistaComponent implements OnInit {
  private artista:any;
  private canciones:any[];
  constructor(service:SpotifyService, link:ActivatedRoute) {
    link.params.subscribe(
      params => {
        service.getArtista(params.id).then(
          datos => {
            this.artista = datos;
            console.log(datos);
          }
        );
        service.getCancionesArtista(params.id).then(
```

```
            datos => {
              this.canciones = datos.tracks;
              console.log(datos);
            }
          )
        }
      )
    }
```

La salida:



Ahora vamos a mostrar las canciones en una tabla:

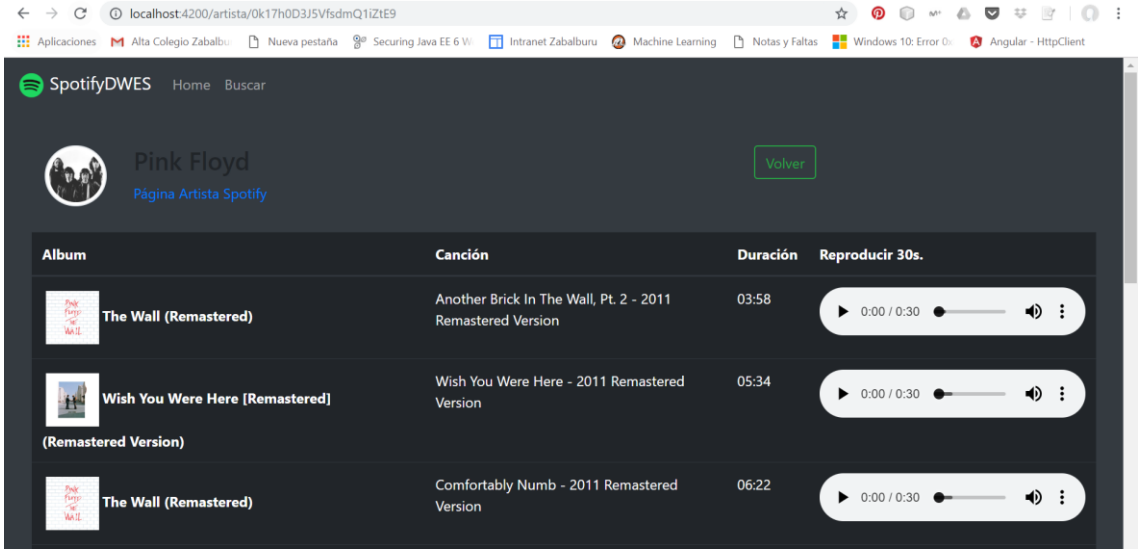
```
<div class="row m-3">
  <div class="col-1 text-center">
…
</div>
<div class="row m-3">
  <table class="table table-dark">
    <thead>
      <tr>
        <th scope="col">Album</th>
        <th scope="col">Canción</th>
        <th scope="col">Duración</th>
        <th scope="col">Reproducir 30s.</th>
      </tr>
    </thead>
```

```
    <tbody>
      <tr *ngFor="let cancion of canciones">
        <th scope="row">
<img    [src]='cancion.album.images[1]    |    imageURL'    class="m-1"
[style.width]="'60px'">{{ cancion.album.name }}</th>
        <td>{{ cancion.name}}</td>
        <td>{{ cancion.duration_ms | ms}}</td>
        <td><audio controls [src]='cancion.preview_url'></td>
      </tr>
      </tbody>
      </table>
</div>
```

Hemos definido un **pipe** para convertir los milisegundos a **mm:ss**:

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


@Pipe({
  name: 'ms'
})
export class MsPipe implements PipeTransform {


  transform(value: any, args?: any): any {
    if (!value){
      return '';
    }
    let minutos = Math.floor(value / 60000);
    let s = Math.floor((value - (minutos *60000))/1000);
    return ((minutos<10)?'0':'') + minutos + ':' + ((s<10)?'0':'') + s;
  }
}
```

Y hemos empleado la etiqueta **audio** de HTML5 para reproducir la vista previa de la canción. La salida:



El siguiente paso es emplear los **widgets de Spotify**.

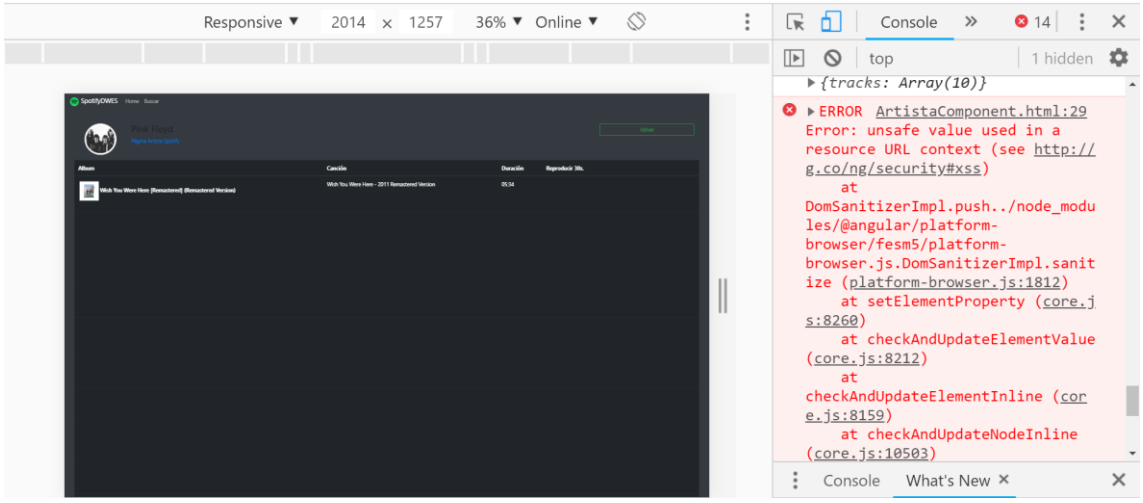
Si miramos en la documentación veremos que tenemos en **Docs → Widgets → Add a Widget** la información para añadir un **widget** (elegimos la opción **compact**). El código es el siguiente:

```
<iframe
src="https://open.spotify.com/embed/album/1DFixLWuPkv3KT3TnV35m3"
width="300"  height="80"  frameborder="0"  allowtransparency="true"
allow="encrypted-media"></iframe>
```

Donde el elemento en negrita es el **SID** del álbum que queremos reproducir. Como lo que queremos repoducir es una canción necesitamos el **SID** de la canción y la **URL** es con **track** en lugar de con álbum. Sustituimos la etiqueta **audio** con el **widget**:

```
<iframe  src="https://open.spotify.com/embed/track/{{  cancion.id  }}"
width="300"  height="80"  frameborder="0"  allowtransparency="true"
allow="encrypted-media"></iframe>
```

Si ejecutamos:



Como vemos aparece un error **unsafe value used in a resource URL context**. El problema es que no se permiten expresiones de plantilla dentro de un **iFrame** por cuestiones de seguridad.

La cuestión es que en una plantilla se podría inyectar código potencialmente peligroso (por ejemplo mediante una etiqueta script o a través de un iFrame). En la mayoría de los casos, Angular se encarga de *sanar* (sanitize) estos elementos (por ejemplo si intentamos asignarlos a una propiedad innderHTML) pero en otros Angular bloquea estas opciones por motivos de seguridad (por ejemplo en la propiedad src de un iFrame).En

El proceso de limpieza se define en diferentes contextos en función de dónde se ejecute

- HTML
- Script
- Style
- URL
- ResourceURL (es el caso de los iFrames)

Si Angular nos bloquea en alguno de estos casos tenemos que emplear el método correspondiente del servicio **DomSanitizer** (bypassSecurityTrustHtml, bypassSecurityTrustScript, bypassSecurityTrustStyle, bypassSecurityTrustURL, bypassSecurityTrustResourceURL) para permitir el elemento

NOTA: No es que **nosotros vayamos a inyectar el código**. Pero supongamos que el valor de la URL se lo pedimos al usuario en un cuadro de texto que luego asignamos a la propiedad URL. En este caso, el usuario podría escribir una URL falsa que incluyera un script.

Para poder limpiar otras posibles URLs (esto también nos pasaría con los widgets de YouTube), vamos a emplear un **pipe**:

```
ng g pipe pipes/Sanitize --spec=false
CREATE src/app/pipes/sanitize.pipe.ts (205 bytes)
UPDATE src/app/app.module.ts (1275 bytes)
```

Otra cuestión es que a un filtro **no podemos pasarle una cadena con una expresión interna**. Así que la opción pasa por crear un filtro que tenga como parámetro la parte **fija de la url** y aplicárselo a la canción:

El código:

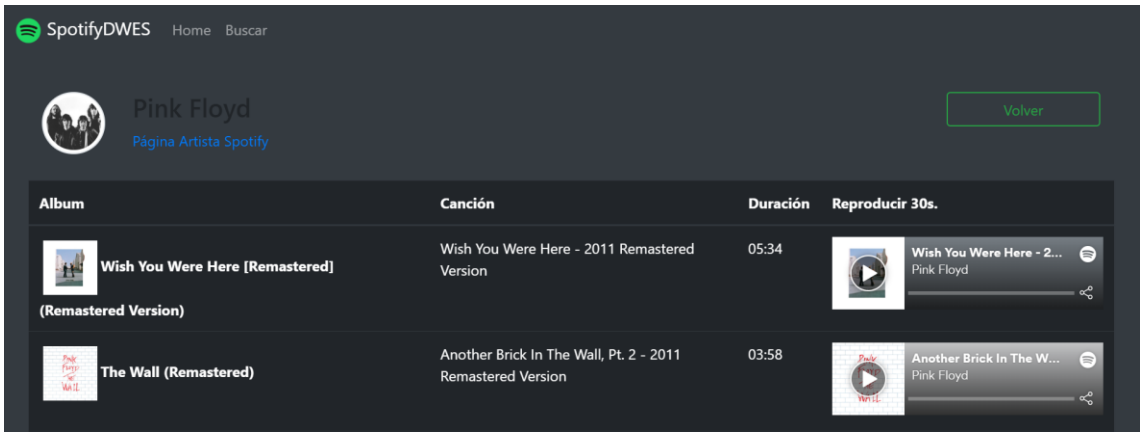
```
import { Pipe, PipeTransform } from '@angular/core';
import { DomSanitizer } from '@angular/platform-browser';


@Pipe({
  name: 'sanitize'
})
export class SanitizePipe implements PipeTransform {
  constructor(private sanitize:DomSanitizer){


  }
```

```
  transform(value: any, url: any): any {
    return this.sanitize.bypassSecurityTrustResourceUrl(url+value);
  }

}
```

Añadimos el **pipe** al componente:

```
<iframe [src]="cancion.id | sanitize:'https://open.spotify.com/embed/track/'"
 width="300" height="80" frameborder="0" allowtransparency="true"
allow="encrypted-media"></iframe>
```

La salida

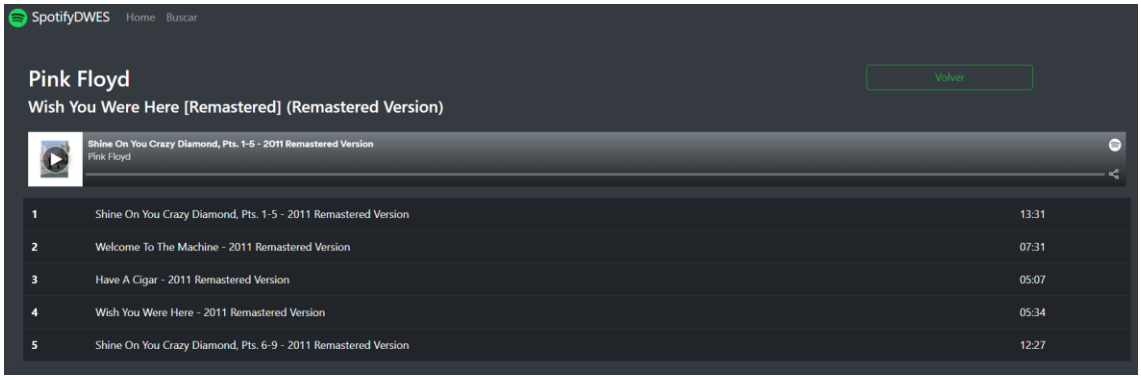


Actividad

Modificar la aplicación para añadir una página para el **álbum** en la que se muestre el álbum de manera similar a la página del artista.

A los álbumes se llegará a través de enlaces en el campo **Álbum** de la tabla de canciones del artista.

La página tendrá un aspecto similar a la siguiente:



Al hacer clic en alguna de las canciones, se actualizará el reproductor para poder reproducirla. Al entrar en la página, el reproductor mostrará la primera canción del álbum.

El botón **volver** retornará al **primer artista** del álbum.

Two-Way Data Binding & Forms Module

Como hemos visto tenemos dos tipos de binding:

- **Input binding** : que se da cuando el componente **recibe** información de un componente externo
- **Output binding** : que se da cuando el componente **emite eventos** al exterior

El **two-way data binding** permite que los **cambios en el modelo se reflejen en la vista y viceversa**. Es decir, que si el usuario modifica la información de un campo, dicho cambio se refleja automáticamente en el componente y, si es el componente el que se modifica, el cambio se refleja automáticamente en la vista.

Básicamente es unir una **propiedad de entrada** con un **evento change** sobre la misma. Aunque es un proceso que podríamos realizar manualmente, Angular nos proporciona la **directiva ngModel** para este proceso.

Dado que se trata de un doble binding la forma de indicarlo es uniendo los indicadores del bindin de entrada **[]** y los de salida **()** quedando **[(ngModel)]**.

Veamos un ejemplo básico. Vamos a crear un componente **NuevoEmpleado** que, en principio, sólo va a tener un campo **Empleado**.

```
ng g c nuevo-empleado
```

```
CREATE src/app/nuevo-empleado/nuevo-empleado.component.html (33 bytes)

CREATE   src/app/nuevo-empleado/nuevo-empleado.component.spec.ts   (678
bytes)

CREATE src/app/nuevo-empleado/nuevo-empleado.component.ts (300 bytes)

CREATE src/app/nuevo-empleado/nuevo-empleado.component.css (0 bytes)

UPDATE src/app/app.module.ts (1028 bytes)
```

En el componente vamos a poner una **propiedad** de tipo **Empleado**:

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

import { Empleado } from '../modelo/empleado';

@Component({

selector: 'app-nuevo-empleado',

templateUrl: './nuevo-empleado.component.html',

styleUrls: ['./nuevo-empleado.component.css']

})
```

```
export class NuevoEmpleadoComponent implements OnInit {
private empleado:Empleado;
constructor() {
this.empleado = new Empleado(1,'Juan','López','Contabilidad','');
  }

}
```

Y vamos a definir un método para limpiar el empleado:

```
reset(){
this.empleado = new Empleado(0,'','','');
  }
```

Por su parte, en la plantilla:

```
Nombre: <input [(ngModel)]='empleado.nombre'><br>
<input type="button" value="Limpiar" (click)='reset()'>
<h3>{{empleado.nombre}}</h3>
```

Además es necesario importar **FormsModule** en **app-module.ts**:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
…
import { NuevoEmpleadoComponent } from './nuevo-empleado/nuevo-empleado.component';
@NgModule({
declarations: [
    …
NuevoEmpleadoComponent
  ],
imports: [
BrowserModule,
FormsModule
  ],
providers: [],
bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

Angular y Ionic

Ionic es un framework que pemite crear aplicaciones para dispositivos móviles empleando Angular. Las aplicaciones generadas son compatibles con Android, IOS y Windows Phone.

¿Cómo se consigue? Todos los móviles disponen de un componente interno capaz de renderizar páginas web con javascript. Así que Ionic lo que hace es "traducir" los componentes Html a componentes móviles.

Para convertir la aplicación Javascript generada en una aplicación móvil nativa (y para poder acceder a recursos móviles como el GPS, la cámara de fotos…), Ionic emplea Apache Cordova.

Vamos a crear una aplicación Ionic. Para ello vamos a https://ionicframework.com/ y seleccionamos la opción **Get Started** donde **creamos una nueva cuenta** o nos **logueamos** con la cuenta si ya la tenemos.

```
npm install -g ionic  // Como Administrador
```

El siguiente paso es crear una aplicación Ionic. Hay tres plantillas iniciales:

- **blank**: Una aplicación en blanco
- **tabs**: Una aplicación basada en una serie de pestañas accesibles desde un menú inferior
- **side**: Una aplicación con menú lateral

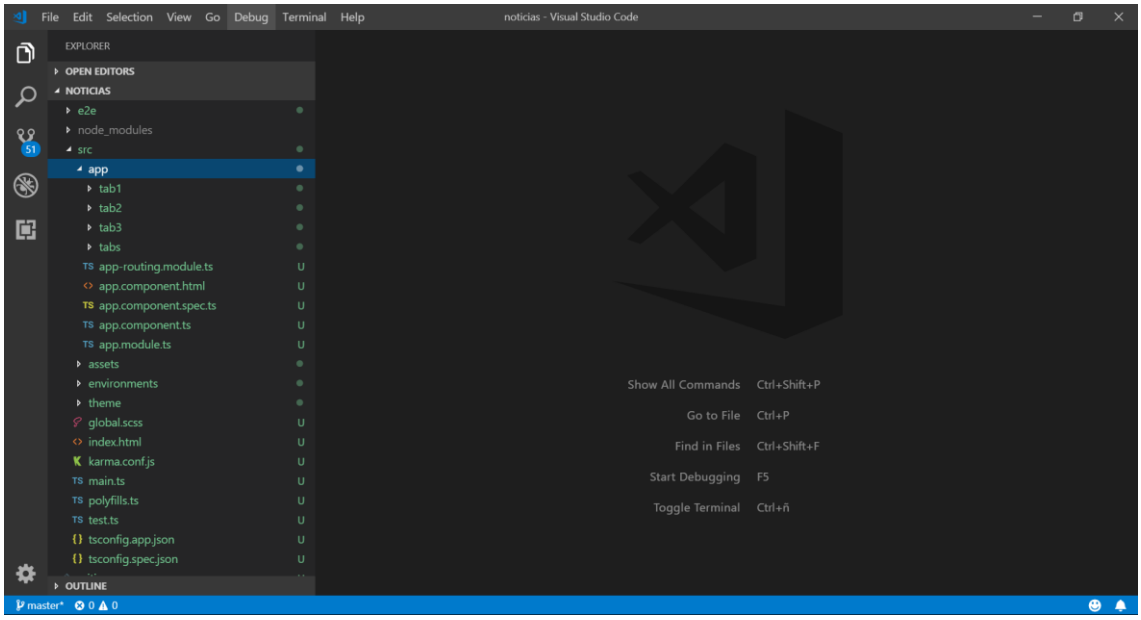
En nuestro caso vamos a emplear **tabs**:

```
ionic start noticias tabs
```

Y dejamos que cree la aplicación.

- Decimos que sí a todo
- Introducimos el email y el password de Ionic
- Seleccionamos Create a new app on Ionic AppFlow
- Elegimos la opción Ionic PRO como repositorio (no lo vamos a usar)

Una vez finalizado el proceso, abrimos el Visual Studio y nos vamos a la capeta del proyecto:

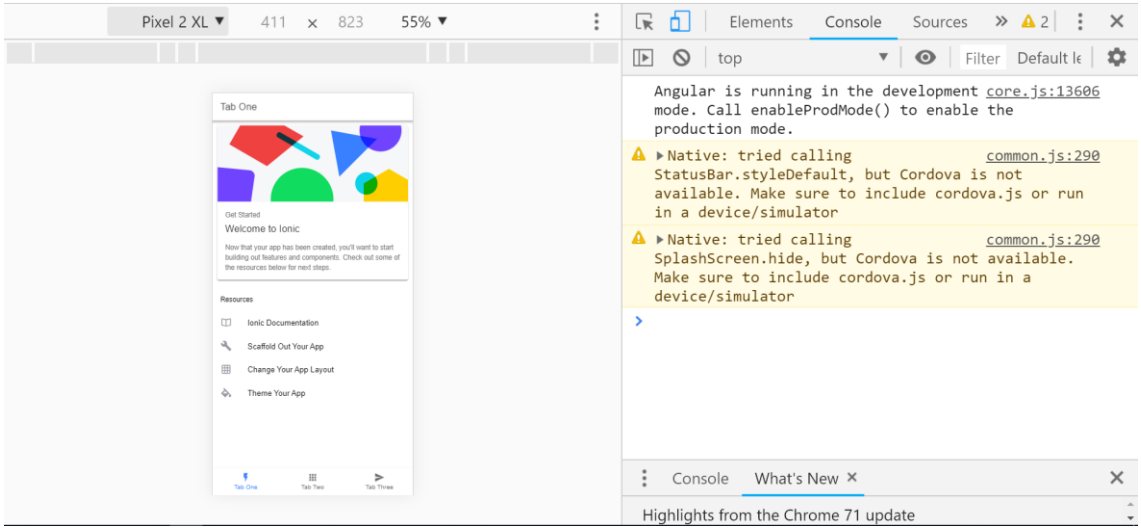


Como podemos ver una aplicación Ionic es un tipo de aplicación Angular, así que todo lo visto hasta el momento es válido.

Para ejecutar la aplicación:

ionic serve -c

Y podremos acceder a ella desde el navegador



Además si nos descargamos **Ionic DevApp** desde las diferentes **stores** y lo ejecutamos en el móvil, podremos acceder a las aplicaciones y ejecutarlas desde allí (siempre que estemos en la misma red local).

Nuestra aplicación va a tener **dos pestañas**:

- La inicial que nos mostrará las **últimas noticias**
- Una segunda, llamada **Categorías** que nos permitirá categoriza las noticias (que básicamente será una lista de categorías)

Emplearemos la **versión para desarrolladores** de la página **newsapi.org**.

Si nos fijamos en la pestaña **Tabs** veremos los siguientes elementos:

- **tabs.module.ts** : Todos los componentes se cargan en módulos angular diferentes (no es obligatorio, pero es la forma en la que ahora lo crea el asistente de Ionic). Aquí simplemente se cargan los módulos necesarios junto con **TabsPageRoutingModule** que es donde se definen las rutas de los tabs de la aplicación
- **tabs.router.module.ts** : Aquí se definen las **rutas** asociadas a los tabs, es decir, los **componentes** que van a ser mostrados cuando se seleccione un **tab**. Como se ve hay un componente principal que es **tabs.page.ts** y luego tenemos las rutas a los diferentes módulos donde está definido cada **tab**. Por ejemplo, para el **tab1** se indica que se cargue el módulo (**página** en terminología Ionic) **Tab1PageModule**:

```
children: [
      {
        path: 'tab1',
        children: [
          {
            path: '',
            loadChildren: '../tab1/tab1.module#Tab1PageModule'
          }
        ]
      },
```

El usar un módulo diferente del raíz y el método **loadChildren** permite lo que se llama **lazy loading**. Es decir que el módulo (y sus componentes) sólo se carguen cuando se acceda a ellos lo que redunda en una caga más rápida de la aplicación. Para indicar el módulo se especifica la **ruta**, el carácter **#** y el nombre del módulo.

- **tabs.page.ts**: El componente que muestra los tabs
- **tabs.page.html**: El HTML de cada tab empleando los componentes de Ionic. Esta asociado al anterior a través del atributo **tab**.

```
<ion-tabs>
  <ion-tab-bar slot="bottom">
    <ion-tab-button tab="tab1">
      <ion-icon name="flash"></ion-icon>
      <ion-label>Tab One</ion-label>
    </ion-tab-button>
…
  </ion-tab-bar>
</ion-tabs>
```

De este modo, cuando el usuario selecciona el primer tab, se carga el componente **Tab1PageModule**. En este módulo es donde definiremos las rutas / componentes a cargar cuando el usuario acceda al tab:

```
import { IonicModule } from '@ionic/angular';
import { RouterModule } from '@angular/router';
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { Tab1Page } from './tab1.page';

@NgModule({
  imports: [
    IonicModule,
    CommonModule,
    FormsModule,
    RouterModule.forChild([{ path: '', component: Tab1Page }])
  ],
  declarations: [Tab1Page]
})
export class Tab1PageModule {}
```

Como vemos, solo hay una ruta (la ruta por defecto) que nos lleva al componente **Tab1Page**.

La misma idea está detrás de las otras páginas de la aplicación.

Este módulo de rutado, a su vez, se carga desde **AppRoutingModule** (cargado desde **app.modules.ts**):

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

const routes: Routes = [
  { path: '', loadChildren: './tabs/tabs.module#TabsPageModule' }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
```

```
})
export class AppRoutingModule {}
```

Dado que solo vamos a necesitar **dos tabs**, vamos a eliminar la carpeta **tab3**. Además, eliminamos el código relacionado de **tabs.page.html** (aprovechamos para cambiar el nombre y los iconos de los tabs). Para los iconos ver https://ionicframework.com/docs/ionicons/

```
<ion-tabs>


  <ion-tab-bar slot="bottom">
    <ion-tab-button tab="tab1">
      <ion-icon name="globe"></ion-icon>
      <ion-label>Últimas Noticias</ion-label>
    </ion-tab-button>


    <ion-tab-button tab="tab2">
      <ion-icon name="bookmarks"></ion-icon>
      <ion-label>Categorias<aside></aside></ion-label>
    </ion-tab-button>


    </ion-tab-bar>


</ion-tabs>
```

También eliminamos el código correspondiente de **tabs.router.module.ts**.

Implementando el servicio de Categorias

Vamos a emplear **Ionic CLI** que es muy similar al CLI de Angular y nos permite crear páginas, servicios, tabs…

```
ionic generate service servicios/Categorias
```

```
> ng generate service servicios/Categorias
CREATE src/app/servicios/categorias.service.spec.ts (353 bytes)
CREATE src/app/servicios/categorias.service.ts (139 bytes)
[OK] Generated service!
```

Como vemos, realmente lo que se crea es un servicio Angular.

De momento vamos a crear una lista de categorías con algunas básicas:

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

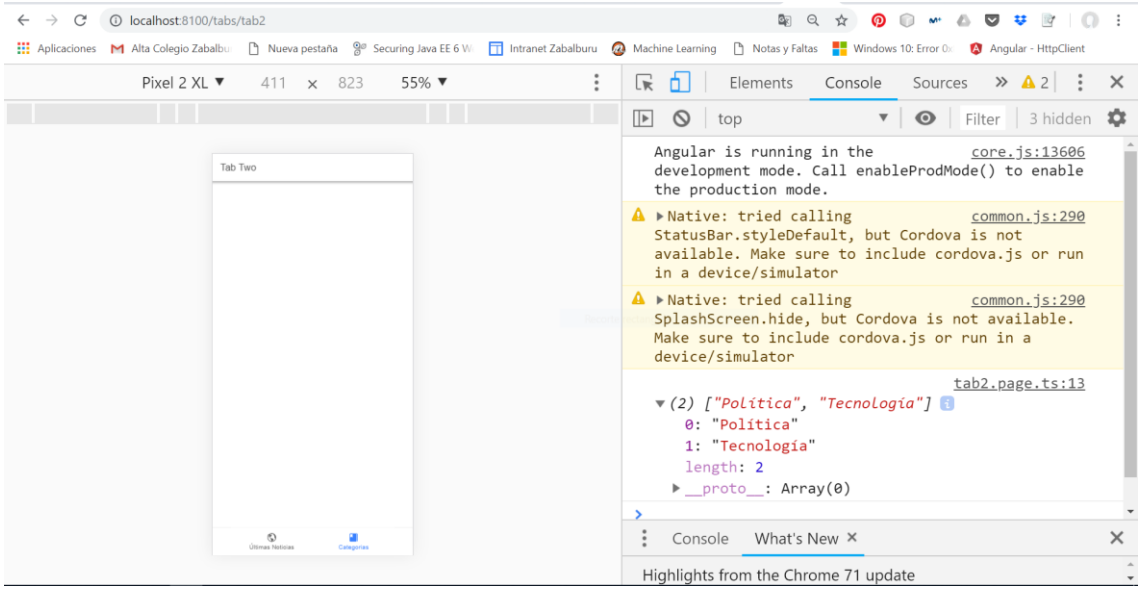
```
@Injectable({
  providedIn: 'root'
})
export class CategoriasService {
  private categorias:String[] = [];
  constructor() {
    this.categorias.push("Política");
    this.categorias.push("Tecnología");
   }
  getCategorias():String[] {
    return this.categorias;
  }
  private buscarCategoria(categoria:string):number {
    let i:number;
    for(i=0;i<this.categorias.length && categoria != this.categorias[i];i++);
    return i;
  }
  nuevaCategoria(categoria:string){
    if (this.buscarCategoria(categoria) == this.categorias.length){
      this.categorias.push(categoria);
    }
  }
  borrarCategoria(categoria:string){
    let i = this.buscarCategoria(categoria);
    if (i < this.categorias.length){
      this.categorias.splice(i,1);
    }
  }
}
```

Ahora inyectamos el servicio en el componente **tab2.page:**

```
import { Component } from '@angular/core';
import { CategoriasService } from '../servicios/categorias.service';

@Component({
```

```
  selector: 'app-tab2',
  templateUrl: 'tab2.page.html',
  styleUrls: ['tab2.page.scss']
})
export class Tab2Page {
  private categorias:String[];
  constructor(private service:CategoriasService){
    this.categorias = service.getCategorias();
    console.log(this.categorias);
  }
}
```

La salida:



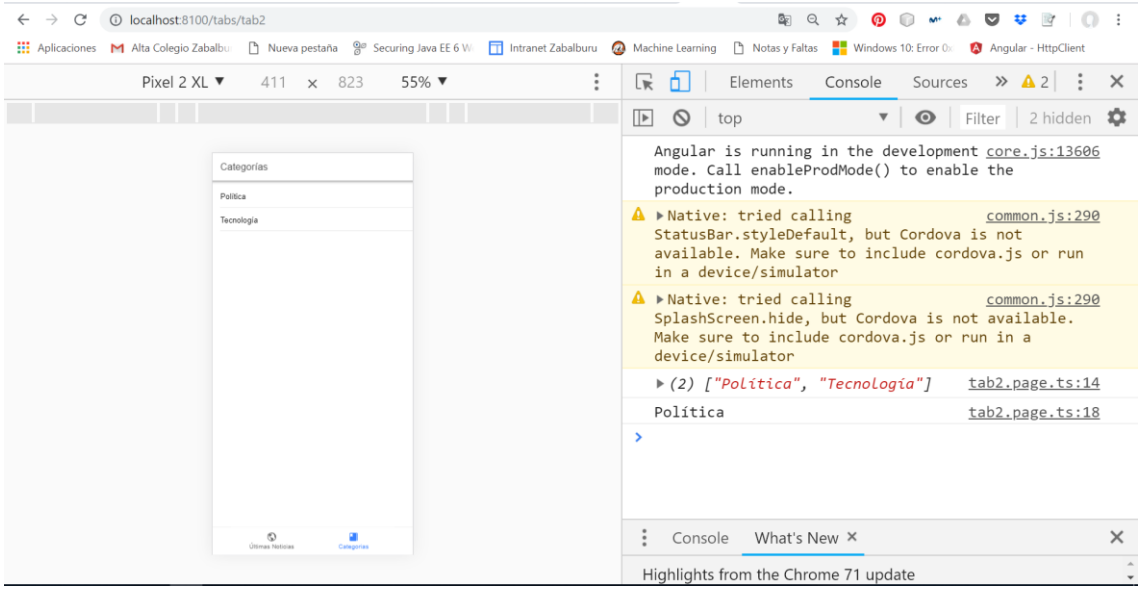
Ahora vamos a mostrar la información en una lista. En la **documentación de los componentes de Ionic** (https://beta.ionicframework.com/docs/api) , buscamos **List** y cogemos el código de **Basic List** y lo adaptamos a las categorías:

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Categorías
    </ion-title>
  </ion-toolbar>
</ion-header>


<ion-content>
  <ion-list>
    <ion-item *ngFor="let categoria of categorias" (click)="itemSelected(categoria)">
      <ion-label>{{ categoria }}</ion-label>
    </ion-item>
  </ion-list>
</ion-content>
```

Y modificamos tab2.page.ts para añadir el manejador de eventos:

```
itemSelected(categoria:String){
    console.log(categoria);
  }
```

La salida, haciendo clic en una categoría:

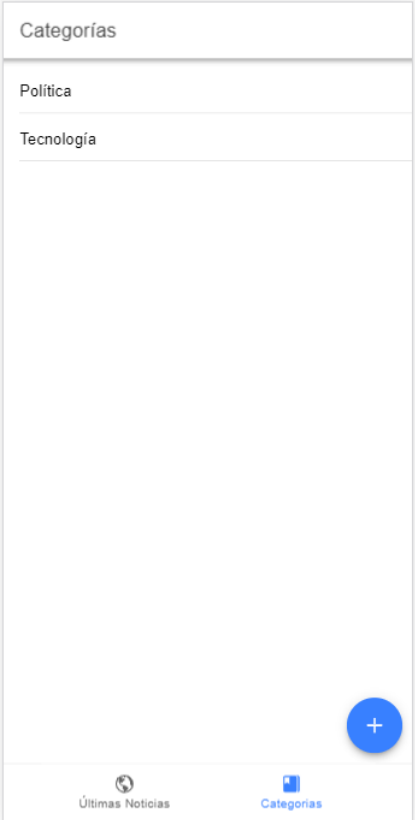


Ahora vamos a pone la opción de **Añadir nuevas categorías**. Para ello, vamos a emplear un botón flotante (FAB – Floating Action Button). Vamos a la documentación y añadimos el código necesario a la página:

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Categorías
    </ion-title>
  </ion-toolbar>
</ion-header>


<ion-content>
  <ion-list>
    <ion-item *ngFor="let categoria of categorias" (click)="itemSelected(categoria)">
      <ion-label>{{ categoria }}</ion-label>
    </ion-item>
  </ion-list>
  <ion-fab vertical="bottom" horizontal="end" slot="fixed">
    <ion-fab-button>
      <ion-icon name="add"></ion-icon>
    </ion-fab-button>
  </ion-fab>
```

```
</ion-content>
```

La salida:



El siguiente paso es pedir la categoría al usuario cuando haga clic en el botón. Para ello, empleamos el componente **Alert prompt** con el que podemos pedir al usuario datos en un componente **modal**.

Si miramos la documentación veremos que este componente se invoca por código y necesitamos un objeto **AlertController** que debe ser **inyectado en el constructor**. Con ese objeto ejecutamos el método **create** con los siguientes parámetros (entre otros):

- **header**: El título del cuadro de diálogo
- **message**: El mensaje a mostrar al usuario
- **inputs**: Una matriz de entradas. Cada entrada es un objeto con, entre otras, las siguientes propiedades:
    - **name**: El nombre del control
    - **type**: El tipo ('text', 'radio','chekbox','date'…). **NOTA**: No se pueden mezclar diferentes tipos de controles (es decir no se pueden emplear cuadros de texto junto con botones de radio por ejemplo). Hay que tener en cuenta que los tipos **date, url, password…** son todos de tipo **text** así que sí se podrían mezclar.
    - **value**: El valor inicial
    - **placeholder**: El placeholder

- **buttons**: Una matriz de botones. Entre otras propiedades:
    - **text**: El texto a mostrar
    - **handler**: El manejador del botón

Añadimos el código a la clase **tab2.page.ts**:

```
import { Component } from '@angular/core';
import { CategoriasService } from '../servicios/categorias.service';
import { AlertController } from '@ionic/angular';

@Component({
  selector: 'app-tab2',
  templateUrl: 'tab2.page.html',
  styleUrls: ['tab2.page.scss']
})
export class Tab2Page {

  private categorias: String[];
  constructor(private service: CategoriasService,
    private alertController: AlertController) {
    this.categorias = service.getCategorias();
    console.log(this.categorias);
  }

  itemSelected(categoria: String) {
    console.log(categoria);

  }

  async nuevaCategoria() {
    const alert = await this.alertController.create({
      header: 'Categorías',
      message: 'Introduzca la categoría',
      inputs: [
        {
```

```
          name: 'categoria',
          type: 'text',
          placeholder: 'Nueva Categoria'
        }
      ],
      buttons: [
        {
          text: 'Cancelar',
          role: 'cancel',
          cssClass: 'secondary',
          handler: () => {
            console.log('Cancelado');
          }
        }, {
          text: 'Aceptar',
          handler: (datos) => {
            if(datos.categoria.length !== 0){
              this.service.nuevaCategoria(datos.categoria);
            }
          }
        }
      ]
    });
    await alert.present();
  }
}
```

NOTA: Cuando un método trabaja con una Promesa (como es el caso aunque no lo veamos) necesitamos emplear la llamada **then** (o **catch**) para indicar el código a ejecutar cuando ciha promesa finalice. Si etiquetamos el método como **async** podemos asignar la promesa a una **constante** y, después, emplear el método **await** para ejecutar el código .

Sin emplear **aync/await** el código hubiera quedado así:

```
nuevaCategoria(){
    this.alertController.create({
      header: 'Categorías',
      …
```

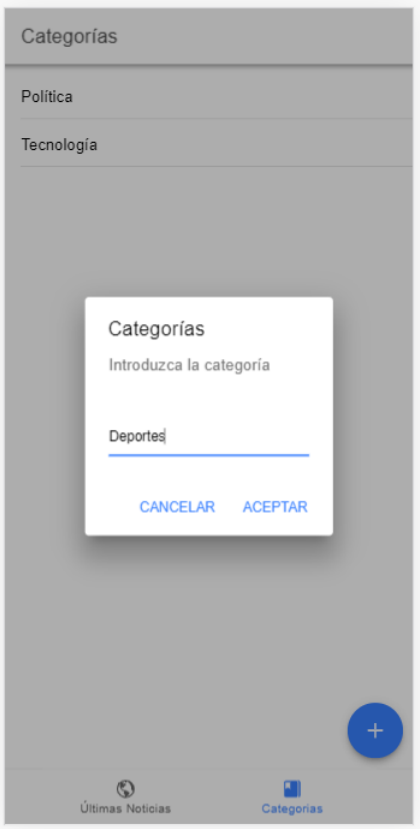
```
    }).then( (datos) => {
      datos.present();
    })
  }
```

NOTA2: Como se ve, los datos del formulario los recibimos encapsulados en un **objeto** al que podemos acceder desde los manejadores. Ese objeto tiene una **propiedad por cada campo** de la alerta.

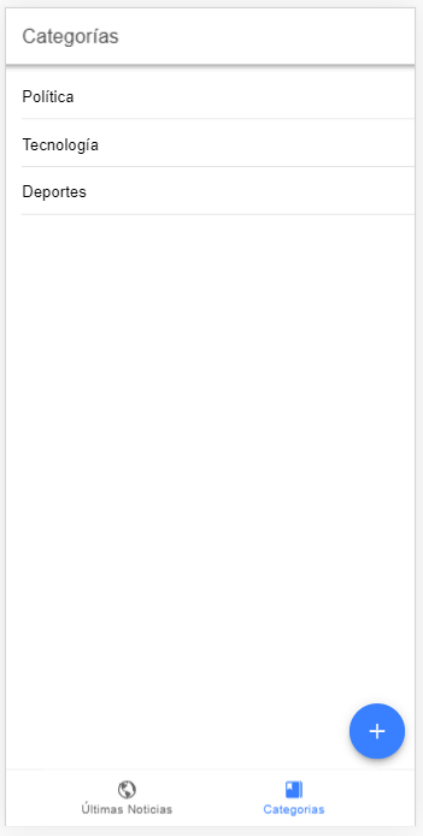
Hacemos la llamada en el botón:

```
<ion-fab-button (click) = "nuevaCategoria()">
  <ion-icon name="add"></ion-icon>
</ion-fab-button>
```

Y probamos el código:



Tras pulsar en Aceptar:



Ahora vamos a modificar la **página uno** para mostrar las últimas noticias. Para ello vamos a emplear la página https://newsapi.org/ en la que nos registramos para obtener una clave.

A la hora de enviar la clave en las peticiones al servidor emplearemos uno de los mecanismos recomendados y utilizaremos la cabecera X-Api-Key.

Creamos el servicio de noticias

```
ng g service servicios/noticias --spec=false
```

```
CREATE src/app/servicios/noticias.service.ts (137 bytes)
```

Y ahora definimos el servicio (previamente hay que cargar **HttpClientModule** en **app.modules.ts**):

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http'
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class NoticiasService {
  private API_KEY='…';
  constructor(private http:HttpClient) {
```

```
  }

  public getNovedades():Observable<any>{
    return this.http.get('https://newsapi.org/v2/top-headlines', {
        headers: {'X-Api-Key':this.API_KEY},
        params: {'language':'es'}
    });
  }
}
```

Ahora modificamos **tab1.page.ts** para hacer uso del servicio

```
import { Component } from '@angular/core';
import { NoticiasService } from '../servicios/noticias.service';

@Component({
  selector: 'app-tab1',
  templateUrl: 'tab1.page.html',
  styleUrls: ['tab1.page.scss']
})
export class Tab1Page {
  private noticias:any[];
  constructor(private service:NoticiasService){
    service.getNovedades()
      .subscribe((datos) => {
        console.log(datos.articles);
        this.noticias = datos.articles
      });
  }
}
```

Ahora modificamos la plantilla para mostrar las noticias como tarjetas (https://beta.ionicframework.com/docs/components#card) :

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Últimas Noticias
    </ion-title>
```

```
  </ion-toolbar>
</ion-header>

<ion-content>
    <ion-card *ngFor="let noticia of noticias">
        <ion-img [src]="noticia.urlToImage"></ion-img>
        <ion-card-header>
            <ion-card-title>{{ noticia.title }}</ion-card-title>
        </ion-card-header>
    </ion-card>
</ion-content>
```

La salida:

Dado que vamos a tener que mostrar noticias en más sitios parece adecuado poner las noticias en un componente adicional:

```
ng g component componentes/noticia --spec=false
CREATE src/app/componentes/noticia/noticia.component.html (26 bytes)
CREATE src/app/componentes/noticia/noticia.component.ts (274 bytes)
CREATE src/app/componentes/noticia/noticia.component.scss (0 bytes)
UPDATE src/app/app.module.ts (933 bytes)
```

NOTA: El componente por defecto se añade a **app.module.ts**. Dado que queremos reutilizar el componente en otros módulos, vamos a crear un **módulo** para almacenar los componentes compartidos:

```
ng g module modulos/Shared
```

```
CREATE src/app/modulos/shared/shared.module.ts (190 bytes)
```

Y, en él, **exportamos** el componente:

```
import { IonicModule } from '@ionic/angular';
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { NoticiaComponent } from 'src/app/componentes/noticia/noticia.component';

@NgModule({
  declarations: [NoticiaComponent],
  imports: [
    CommonModule,
    IonicModule
  ],
  exports:[
    NoticiaComponent
  ]
})
export class SharedModule { }
```

NOTA: Necesitamos **IonicModule** porque nuestro componente emplea componentes de ionic.

Por supuesto, tenemos que **eliminar las referencias** al componente en **app.modules.ts**.

El componente tendrá una propiedad de **entrada** con la noticia (**noticia.component.ts**):

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

@Component({
  selector: 'app-noticia',
  templateUrl: './noticia.component.html',
```

```
  styleUrls: ['./noticia.component.scss']
})
export class NoticiaComponent implements OnInit {
  @Input() private noticia:any;
  constructor() { }

  ngOnInit() {
  }

}
```

Y la plantilla (**noticia.component.html**):

```
<ion-card>
    <ion-img [src]="noticia.urlToImage"></ion-img>
    <ion-card-header>
        <ion-card-title>{{ noticia.title }}</ion-card-title>
    </ion-card-header>
</ion-card>
```

Para pode usar el componente en el primer tab necesitamos incluir el módulo en **tab1.module.ts**:

```
import { IonicModule } from '@ionic/angular';
import { RouterModule } from '@angular/router';
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { Tab1Page } from './tab1.page';
import { SharedModule } from '../modulos/shared/shared.module';
import { NoticiaComponent } from '../componentes/noticia/noticia.component';

@NgModule({
  imports: [
    IonicModule,
    CommonModule,
    FormsModule,
```

```
    SharedModule,
    RouterModule.forChild([{ path: '', component: Tab1Page }])
  ],
  entryComponents: [
    NoticiaComponent
  ],
  declarations: [Tab1Page]
})
export class Tab1PageModule {}
```

Y ahora ya podemos modificar **tab1.page.html** para hacer uso del nuevo componente:

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Últimas Noticias
    </ion-title>
  </ion-toolbar>
</ion-header>


<ion-content>
    <app-noticia *ngFor="let noticia of noticias" [noticia]="noticia">
    </app-noticia>
</ion-content>
```

El resultado debe ser el mismo que antes.

Podemos ir un paso más allá y crear un componente **ListaNoticias** que muestre la lista de las noticias que se reciban en una propiedad **noticias**:

```
ng g c componentes/LIstaNoticias --spec=false
CREATE                        src/app/componentes/lista-noticias/lista-noticias.component.html (33 bytes)
CREATE   src/app/componentes/lista-noticias/lista-noticias.component.ts (301 bytes)
CREATE                        src/app/componentes/lista-noticias/lista-noticias.component.scss (0 bytes)
UPDATE src/app/app.module.ts (960 bytes)
```

Eliminamos las referencias de **app.module.ts** y las añadimos a **SharedModule**:

```
import { IonicModule } from '@ionic/angular';
import { NgModule } from '@angular/core';
```

```
import { CommonModule } from '@angular/common';
import { NoticiaComponent } from 'src/app/componentes/noticia/noticia.component';
import { ListaNoticiasComponent } from 'src/app/componentes/lista-noticias/lista-noticias.component';

@NgModule({
  declarations: [NoticiaComponent, ListaNoticiasComponent],
  imports: [
    CommonModule,
    IonicModule
  ],
  exports:[
    NoticiaComponent,
    ListaNoticiasComponent
  ]
})
export class SharedModule { }
```

Y, por supuesto, la añadimos como componente de entrada en **tab1.module.ts**:

```
import { IonicModule } from '@ionic/angular';
import { RouterModule } from '@angular/router';
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { Tab1Page } from './tab1.page';
import { SharedModule } from '../modulos/shared/shared.module';
import { NoticiaComponent } from '../componentes/noticia/noticia.component';
import { ListaNoticiasComponent } from '../componentes/lista-noticias/lista-noticias.component';

@NgModule({
  imports: [
    IonicModule,
    CommonModule,
    FormsModule,
```

```
    SharedModule,
    RouterModule.forChild([{ path: '', component: Tab1Page }])
  ],
  entryComponents: [
    NoticiaComponent,
    ListaNoticiasComponent
  ],
  declarations: [Tab1Page]
})
export class Tab1PageModule {}
```

Indicamos que el componente tiene una propiedad de entrada en **lista-noticias.component.ts**:

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

@Component({
  selector: 'app-lista-noticias',
  templateUrl: './lista-noticias.component.html',
  styleUrls: ['./lista-noticias.component.scss']
})
export class ListaNoticiasComponent implements OnInit {
  @Input() private noticias:any[];
  constructor() { }

  ngOnInit() {
  }

}
```

Y definimos su plantilla:

```
<app-noticia *ngFor="let noticia of noticias" [noticia]="noticia">
</app-noticia>
```

Por último, modificamos **tab1.page.html**:

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Últimas Noticias
```

```
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content>
    <app-lista-noticias [noticias]='noticias'></app-lista-noticias>
</ion-content>
```

El resultado es el mismo del caso anterior.

Mensaje de Carga de datos

Dependiendo de la conexión a Intenet puede pasa un pequeño tiempo hasta que el usuario puede ver las noticias. Podemos emplear el componente **Loading** de Ionic para mostrar una indicación visual de espera (https://beta.ionicframework.com/docs/api/loading) .

Básicamente necesitamos un servicio **LoadingController** que nos permita mostrar el mensaje deseado (opcionalmente se puede poner un tiempo, css personalizado, etc). El proceso debe hacerse en un método etiquetado como **async** para que se ejecute de manera asíncrona.

La idea es hacerlo cuando se cargan las últimas noticias. Por tanto, en la página **tab1.page.ts**

```
import { Component } from '@angular/core';
import { NoticiasService } from '../servicios/noticias.service';
import { LoadingController } from '@ionic/angular';

@Component({
  selector: 'app-tab1',
  templateUrl: 'tab1.page.html',
  styleUrls: ['tab1.page.scss']
})
export class Tab1Page {
  private noticias:any[];
  private load:any;

  constructor(private service:NoticiasService,
    private loadingController: LoadingController){
    this.mostrarLoading();
    service.getNovedades()
      .subscribe((datos) => {
        console.log(datos.articles);
        this.noticias = datos.articles;
```

```
        this.load.dismiss();
      });
  }
  async mostrarLoading() {
    this.load = await this.loadingController.create({
      message: 'Cargando noticias'
    });
    return await this.load.present();
  }
}
```

Detalle de la Noticia

El siguiente paso va a ser navegar a una nueva página donde aparece la información detallada de la noticia. Una página es un componente angular pero la vamos a crear con ionic:

```
ionic generate page paginas/noticiaDetalle --spec=false
> ng generate page paginas/noticiaDetalle --spec=false
CREATE  src/app/paginas/noticia-detalle/noticia-detalle.module.ts  (584 bytes)
CREATE  src/app/paginas/noticia-detalle/noticia-detalle.page.html  (141 bytes)
CREATE   src/app/paginas/noticia-detalle/noticia-detalle.page.ts   (291 bytes)
CREATE   src/app/paginas/noticia-detalle/noticia-detalle.page.scss   (0 bytes)
UPDATE src/app/app-routing.module.ts (431 bytes)
```

Como vemos crea un modulo y una página. Además modifica el fichero de rutas para añadir una ruta a la página:

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

const routes: Routes = [
  { path: '', loadChildren: './tabs/tabs.module#TabsPageModule' },
  { path: 'noticiaDetalle', loadChildren: './paginas/noticia-detalle/noticia-detalle.module#NoticiaDetallePageModule' }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
```

```
})
export class AppRoutingModule {}
```

Por tanto, la ruta a la página es **noticiaDetalle**.

Para viajar a un componente empleamos el servicio **NavController** para realizar la navegación. Vamos a añadir un enlace a la noticia:

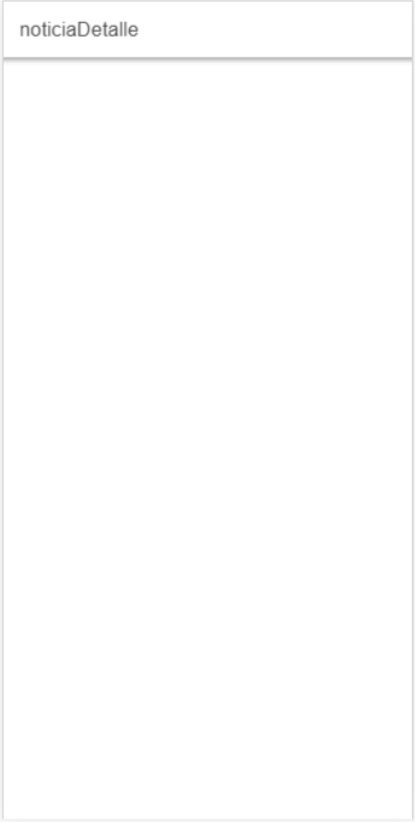
```
<ion-card>
    <ion-img  [src]="noticia.urlToImage"  (click)='IrADetalle()'></ion-img>
    <ion-card-header>
        <ion-card-title     (click)='IrADetalle()'>{{     noticia.title }}</ion-card-title>
    </ion-card-header>
    <ion-card-content>
        <p>{{ noticia.description }}</p>
    </ion-card-content>
</ion-card>
```

Y el código:

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

@Component({
  selector: 'app-noticia',
  templateUrl: './noticia.component.html',
  styleUrls: ['./noticia.component.scss']
})
export class NoticiaComponent implements OnInit {
  constructor(private navCtrl: NavController) { }

  ngOnInit() {
  }

  IrADetalle(){
    this.navCtrl.navigateForward('noticiaDetalle');
  }
```

```
}
```

Si ahora ejecutamos la aplicación y hacemos clic en cualquier sitio nos llevará a la página recién creada:



NOTA: También podemos emplear la navegación estándar de Angular:

```
import { Component, OnInit, Input } from '@angular/core';
import { NavController } from '@ionic/angular';
import { Router } from '@angular/router';

@Component({
  selector: 'app-noticia',
  templateUrl: './noticia.component.html',
  styleUrls: ['./noticia.component.scss']
})
export class NoticiaComponent implements OnInit {
  constructor(private router:Router) { }

  ngOnInit() {
```

```
  }

  IrADetalle(){
    this.router.navigate(['noticiaDetalle']);
  }

}
```

La cuestión ahora es que necesitamos **pasar la noticia** al nuevo componente y, dado que estamos navegando, no podemos pasar directamente un objeto complejo (podríamos pasar parámetros sueltos pero eso no nos soluciona el problema).

Una solución pasa por almacenar esa información en el servicio:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http'
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class NoticiasService {
  private API_KEY='…';
  private actual:any;
  constructor(private http:HttpClient) {

  }

  public setActual(noticia){
    this.actual = noticia;
  }

  public getActual(){
    return this.actual;
  }

  public getNovedades():Observable<any>{
    return this.http.get('https://newsapi.org/v2/top-headlines', {
```

```
        headers: {'X-Api-Key':this.API_KEY},
        params: {'language':'es'}
    });
  }
}
```

Ahora, al pasar a NoticiasDetalle desde NoticiaComponent almacenamos la noticia en el servicio:

```
import { Component, OnInit, Input } from '@angular/core';
import { Router } from '@angular/router';
import { NoticiasService } from 'src/app/servicios/noticias.service';

@Component({
  selector: 'app-noticia',
  templateUrl: './noticia.component.html',
  styleUrls: ['./noticia.component.scss']
})
export class NoticiaComponent implements OnInit {
  @Input() private noticia:any;
  constructor(private service:NoticiasService, private router:Router)
{ }

  ngOnInit() {
  }

  IrADetalle(){
    this.service.setActual(this.noticia);
    this.router.navigate(['noticiaDetalle']);
  }
}
```

Ahora modificamos **noticia-detalle.page.ts**:

```
import { Component, OnInit } from '@angular/core';
import { NavParams } from '@ionic/angular';
import { NoticiasService } from 'src/app/servicios/noticias.service';

@Component({
```

```
  selector: 'app-noticia-detalle',
  templateUrl: './noticia-detalle.page.html',
  styleUrls: ['./noticia-detalle.page.scss'],
})
export class NoticiaDetallePage implements OnInit {
  private noticia:any;
  constructor(service:NoticiasService) {
    this.noticia = service.getActual();
  }

  ngOnInit() {
  }

}
```

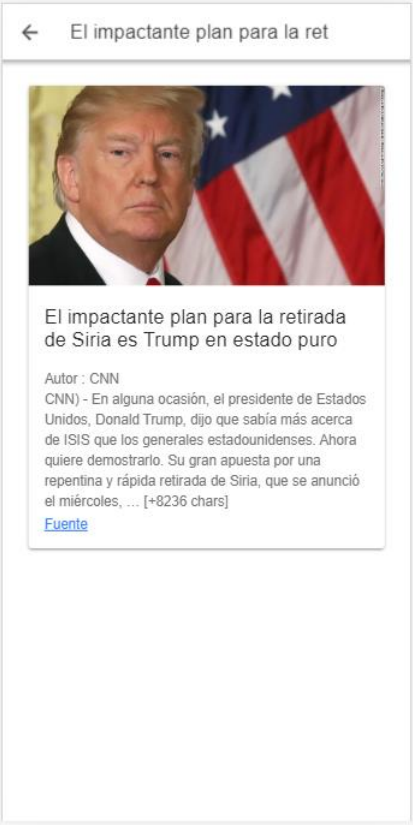
Y la plantilla:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-back-button></ion-back-button>
    </ion-buttons>

    <ion-title>{{noticia.title.substring(0,30)}}</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
    <ion-card>
        <ion-img [src]="noticia.urlToImage"></ion-img>
        <ion-card-header>
            <ion-card-title>{{ noticia.title }}</ion-card-title>
        </ion-card-header>
        <ion-card-content>
            <h4>Autor : {{ noticia.author }}</h4>
            <p>{{ noticia.content }}</p>
```

```
            <p><a [href]="noticia.url" target="_blank">Fuente</a></p>
        </ion-card-content>
    </ion-card>
</ion-content>
```

Ponemos un botón de retorno en la cabecera (hay que ponerlo en un componente de tipo **Buttons**).

La salida:



NOTA: Al usar la versión de desarrollo del API sólo nos muestra los 26 primeros caracteres.

Añadiendo una página para las noticias por categoría

En este caso vamos a crear una página para las noticias de una determinada categoría. En primer lugar modificamos el servicio para buscar todas las noticias de la categoría:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http'
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class NoticiasService {
```

```
  private API_KEY='…';
  private actual:any;

  constructor(private http:HttpClient) {
  }

  public getNovedades():Observable<any>{
    return this.http.get('https://newsapi.org/v2/top-headlines', {
        headers: {'X-Api-Key':this.API_KEY},
        params: {'language':'es'}
    });
  }

  public getPorCategoria(categoria:string):Observable<any>{
    return this.http.get('https://newsapi.org/v2/everything', {
        headers: {'X-Api-Key':this.API_KEY},
        params: {'language':'es',
                 'q':categoria}
    });
  }

  public setActual(noticia){
    this.actual = noticia;
  }

  public getActual(){
    return this.actual;
  }
}
```

Creamos la página de noticasCategoria:

```
ionic generate page paginas/noticiasCategoria --spec=false
> ng generate page paginas/noticiasCategoria --spec=false
CREATE  src/app/paginas/noticias-categoria/noticias-categoria.module.ts
(599 bytes)
```

```
CREATE  src/app/paginas/noticias-categoria/noticias-categoria.page.html (144 bytes)

CREATE   src/app/paginas/noticias-categoria/noticias-categoria.page.ts (303 bytes)

CREATE  src/app/paginas/noticias-categoria/noticias-categoria.page.scss (0 bytes)

UPDATE src/app/app-routing.module.ts (566 bytes)

[OK] Generated page!
```

Si comprobamos la ruta en **app-routing.module.ts** comprobaremos que es **noticias-categoria**. Vamos a modificar dicha ruta para que reciba la categoría en la misma (**/noticias-*categoria/política***)

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

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


const routes: Routes = [

  { path: '', loadChildren: './tabs/tabs.module#TabsPageModule' },

  {   path:   'noticia-detalle',   loadChildren:   './paginas/noticia-detalle/noticia-detalle.module#NoticiaDetallePageModule' },

  {      path:      'noticias-categoria/:categoria',      loadChildren: './paginas/noticias-categoria/noticias-categoria.module#NoticiasCategoriaPageModule' }

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule {}
```

El siguiente paso es navegar a dicha ruta al hacer clic en **tab2.page.ts**:

```
itemSelected(categoria: String) {

    console.log(categoria);

    this.router.navigate(['/noticias-categoria',categoria]);

  }
```

Y modificar el código de **noticias-categorias.page.ts** para que localice las noticias:

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

import { ActivatedRoute } from '@angular/router';

import { NoticiasService } from 'src/app/servicios/noticias.service';
```

```
@Component({
  selector: 'app-noticias-categoria',
  templateUrl: './noticias-categoria.page.html',
  styleUrls: ['./noticias-categoria.page.scss'],
})
export class NoticiasCategoriaPage implements OnInit {
  private categoria:string;
  private noticias:any[] = [1,2,3];
  private load:any;
  constructor(ruta:ActivatedRoute,   service:NoticiasService,   private
loadingController:LoadingController) {
    this.mostrarLoading();
    ruta.params.subscribe(
      (params) => {
        this.categoria = params.categoria;
        service.getPorCategoria(params.categoria)
        .subscribe( (datos:any) => {
          this.noticias = datos.articles;
          this.load.dismiss();
        });
      }
    )
   }

  ngOnInit() {
  }
  async mostrarLoading() {
    this.load = await this.loadingController.create({
      message: 'Cargando noticias'
    });
    return await this.load.present();
  }
}
```

Para poder emplear los componentes **NoticiasComponent y ListaNoticiasComponent** tenemos que incluirlos, junto con el módulo **Shared**, en el fichero **noticias.categorias.module.ts**:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { Routes, RouterModule } from '@angular/router';

import { IonicModule } from '@ionic/angular';
import { NoticiasCategoriaPage } from './noticias-categoria.page';

import { SharedModule } from '../../modulos/shared/shared.module';
import { NoticiaComponent } from '../../componentes/noticia/noticia.component';
import { ListaNoticiasComponent } from '../../componentes/lista-noticias/lista-noticias.component';

const routes: Routes = [
  {
    path: '',
    component: NoticiasCategoriaPage
  }
];

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    SharedModule,
    RouterModule.forChild(routes)
  ],
  declarations: [NoticiasCategoriaPage],
  entryComponents: [
    NoticiaComponent,
    ListaNoticiasComponent
  ],
})
```

```
export class NoticiasCategoriaPageModule {}
```

Y ya, por último, emplear la lista de noticias en **noticias-categorias.page.html**:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-back-button></ion-back-button>
    </ion-buttons>
    <ion-title>{{categoria}}</ion-title>
  </ion-toolbar>
</ion-header>


<ion-content>
  <app-lista-noticias [noticias]='noticias'></app-lista-noticias>
</ion-content>
```

La salida:

Eliminar categorías

Para eliminar categorías vamos a emplear las **listas deslizantes** de Ionic que muestran una opción al deslizar el elemento de la lista a derecha o izquierda. La documentación está en https://beta.ionicframework.com/docs/api/list.

El HTML en **tab2.page.html**:

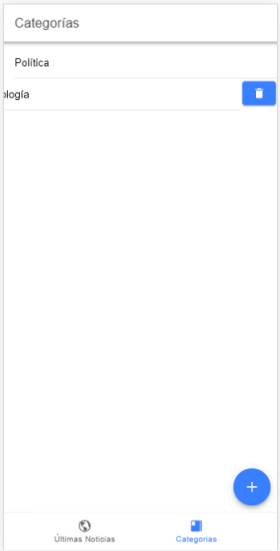
```
<ion-content>
  <ion-list>
    <ion-item-sliding *ngFor="let categoria of categorias"  #sldItem
(click)="itemSelected(categoria)">
      <ion-item>
        <ion-label>{{ categoria }}</ion-label>
      </ion-item>
      <ion-item-options side="end">
        <ion-button      color="primary"      (click)="borrar(sldItem,
categoria)">
          <ion-icon name="trash"></ion-icon>
        </ion-button>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>
  <ion-fab vertical="bottom" horizontal="end" slot="fixed">
    <ion-fab-button (click)="nuevaCategoria()">
      <ion-icon name="add"></ion-icon>
    </ion-fab-button>
  </ion-fab>
</ion-content>
```

NOTA: El hecho de pasar el **slidingItem** como parámetro al método es debido a que la Beta actual de Ionic tiene un problema con las listas deslizantes basadas en contenido dinámico (***ngFor**) que hace que sólo **funcione una vez la eliminación de un elemento**. Una solución pasa por llamar al método **close** del elemento al eliminarlo.

Añadimos a **tab2.page.ts** el siguiente código:

```
borrar(sldItem:IonItemSliding,categoria:string){
    sldItem.close();
    this.service.borrarCategoria(categoria);
    console.log(categoria);
  }
```

La salida:



Al pulsar sobre el botón de borrar se eliminará la categoría de la lista.

Local Storage / Session Storage / Storage

Cuando necesitamos almacenar información en una aplicación Angular podemos emplear los mecanimos provistos por los navegadores web para almacenar información persistente mediante JavaScript (desde HTML 5). Disponemos de dos tipos de almacenamiento:

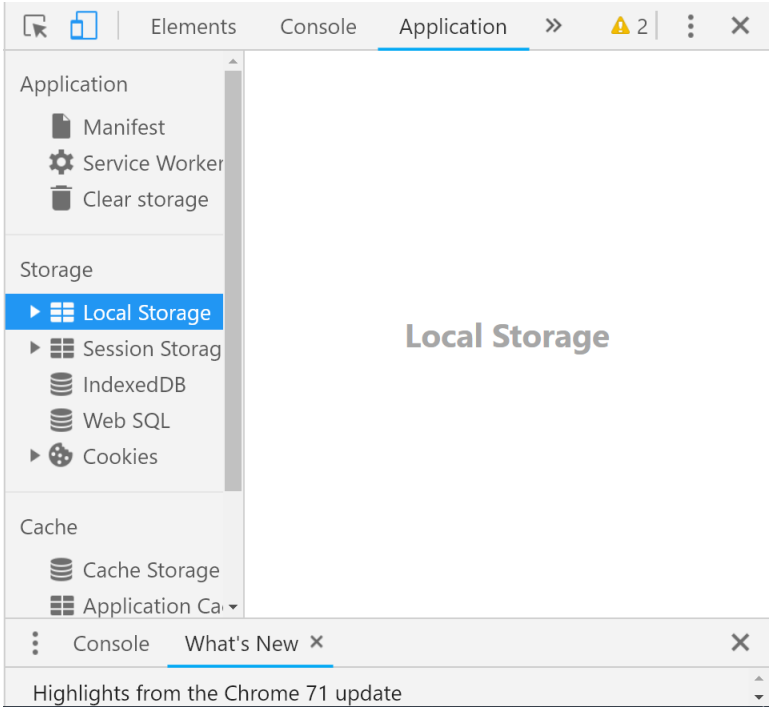
- **LocalStorage** : se emplea para almacenar información de manera **persistente**, lo qe implica que, si reiniciamos el navegador, la información seguirá estando disponible dado que se define en base al protocolo, IP origen y puerto lo que implica que la información es **accesible a todos los scripts cargados de páginas cargadas desde el mismo origen**.
- **SessionStorage** : almacena información de manera temporal. Se define en base a la ventana o pestaña en la que se realiza el almacenamiento y sólo existe mientras la misma exista.

Dado que Ionic se basa en Javascript puede emplear ambos tipos de almacenamiento, aunque también incluye, en su API, el objeto **Storage** que puede emplear alguno de ellos u otros mecanismos (en función del dispositivo) para almacenar dicha información.

Dado que la idea es trabajar sobre Angular y que en nuestro proyecto necesitamos almacenar información de manera **persistente** vamos a utilizar el **LocalStorage**.

NOTA: Los navegadores web disponen de un espacio entre 5MB y 20MB reservado para este tipo de almacenamiento por servidor por lo que su utilidad es limitada.

En Chrome podemos ver el LocalStorage y el SessionStorage en modo depuración en la pestaña **Aplicación**:


Elements
Console
Application
2
Application
Manifest
Service Worker
Clear storage
Storage
Local Storage
Session Storag
IndexedDB
Web SQL
Cookies
Cache
Cache Storage
Application Ca
Local Storage
Console
What's New
Highlights from the Chrome 71 update

Rutas Hijas

Vamos a ver cómo se puede hacer rutas hijas. Vamos a crear un componente principal con un par de páginas.

Creamos un nuevo proyecto Angular llamado RutasYForms:

```
ng new RutasYForms
```

Una vez creado, vamos a crear un componente nuevo para el home de la aplicación:

```
ng g c componentes/home --spec=false -is
```

Creamos el fichero **app.routes.ts** que va a ser el fichero de rutas principal de la aplicación (podemos emplear el snippet ng-routes para generar el esqueleto de las mismas):

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

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

const routes: Routes = [
    { path: 'home', component: HomeComponent },
    { path: '**', redirectTo: 'home' }
];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
})
export class FeatureRoutingModule {}
```

Y añadimos el modulo de rutas a **app.module**:

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

import { AppComponent } from './app.component';
import { HomeComponent } from './componentes/home/home.component';

import {FeatureRoutingModule} from './app.routes';
```

```
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent
  ],
  imports: [
    BrowserModule,
    FeatureRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Modificamos el fichero **index.html** para usar **Bootstrap**:

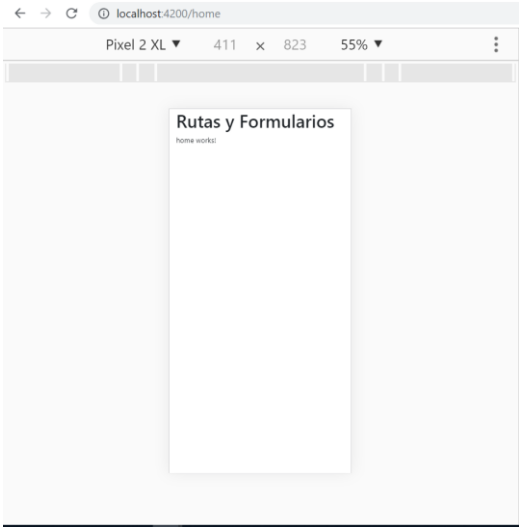
```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.2.1/css/bootstrap.min.css" integrity="sha384-GJzZqFGwb1QTTN6wy59ffF1BuGJpLSa9DkKMp0DgiMDm4iYMj70gZWKYbI706tWS" crossorigin="anonymous">

    <title>Rutas y Forms</title>
    <base href="/">
  </head>
  <body>
    <app-root></app-root>

    <!-- Optional JavaScript -->
```

```
    <!-- jQuery first, then Popper.js, then Bootstrap JS -->

    <script      src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
integrity="sha384-
q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>

    <script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.6/umd/poppe
r.min.js"                                                integrity="sha384-
wHAiFfRlMFy6i5SRaxvfOCifBUQy1xHdJ/yoi7FRNXMRBu5WHdZYu1hA6ZOblgut"
crossorigin="anonymous"></script>

    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.2.1/js/bootstrap.m
in.js"                                                   integrity="sha384-
B0UglyR+jN6CkvvICOB2joaf5I4l3gm9GU6Hc1og6Ls7i6U/mkkaduKaBhlAXv9k"
crossorigin="anonymous"></script>

  </body>

</html>
```

Y modificamos el fichero **app.component.html** para inclui **router-outlet**:

```
<div class="container">

  <h1>Rutas y Formularios</h1>

  <router-outlet></router-outlet>

</div>
```

La salida:



Vamos ahora a crear un componente para gestionar productos y otro para categorías:

```
ng g c componentes/productos --spec=false -is

ng g c componentes/categorias --spec=false -is
```

Además, para navegar entre los componentes vamos a crear un nuevo componente **NavBar**:

```
ng g c componentes/navbar --spec=false -is
```

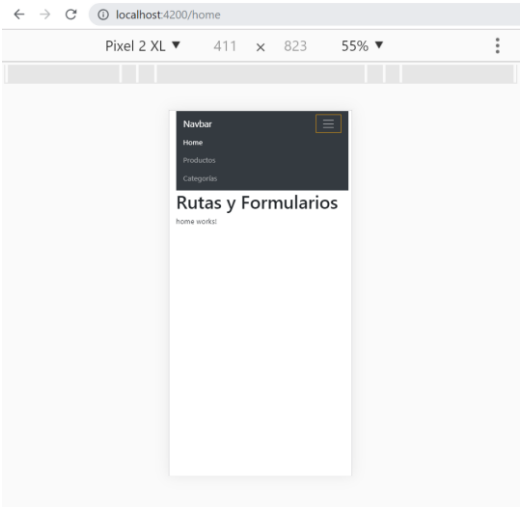
En este componente empleamos la barra de navegación de Bootstrap:

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <a class="navbar-brand" href="#">Navbar</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 active">
        <a class="nav-link" href="#">Home <span class="sr-only">(current)</span></a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Productos</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Categorías</a>
      </li>
    </ul>
  </div>
</nav>
```

Y añadimos la barra de navegación a app-component:

```
<div class="container">
  <app-navbar></app-navbar>
  <h1>Rutas y Formularios</h1>
  <router-outlet></router-outlet>
</div>
```

La salida:



Ahora modificamos app.routes para que nos envíe a los componentes Productos y Categorias:

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

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


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

import { ProductosComponent } from
'./componentes/productos/productos.component';

import { CategoriasComponent } from
'./componentes/categorias/categorias.component';


const routes: Routes = [

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

    { path: 'productos', component: ProductosComponent },

    { path: 'categorias', component: CategoriasComponent },

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

];


@NgModule({

    imports: [RouterModule.forRoot(routes)],

    exports: [RouterModule]

})

export class FeatureRoutingModule {}
```

Y modificamos navbar.component.html para permitir acceder a dichas rutas:

```
<li class="nav-item">

        <a class="nav-link" routerLink="home"
```

```
            routerLinkActive="active">Home</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" routerLink="productos"
            routerLinkActive="active">Productos</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" [routerLink]="['categorias']"
            routerLinkActive="active"> Categorías</a>
      </li>
```

Ahora podemos ir a todos los componentes.

El siguiente paso es crear un submenú con las opciones (empezaremos con las de productos). Por ejemplo, vamos a suponer que queremos poder ver todos los productos, buscar un producto o añadir nuevos productos.

Empezamos definiendo los tres nuevos componentes:

```
ng g c componentes/productos/todos --spec=false -is
ng g c componentes/productos/buscar --spec=false -is
ng g c componentes/productos/nuevo --spec=false -is
```

Añadimos al ficheo productos.component.html una serie de botones que van a actuar como submenú y aprovechamos para añadir la directiva router-outlet:

```
<div class="btn-group" role="group" aria-label="Basic example">
    <button type="button" class="btn btn-secondary" [routerLink]="['todos']" routerLinkActive="active">Ver Todos</button>
    <button type="button" class="btn btn-secondary" [routerLink]="['buscar']" routerLinkActive="active">Buscar</button>
    <button type="button" class="btn btn-secondary" [routerLink]="['nuevo']" routerLinkActive="active">Nuevo Producto</button>
</div>
<router-outlet></router-outlet>
```

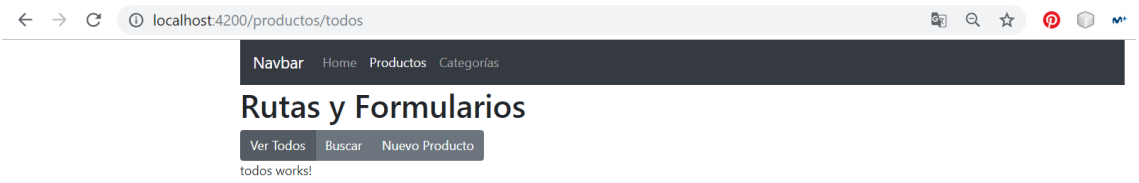
¿Cómo especificamos las rutas hijas?. Añadiendo el atributo **children** a la ruta padre en **app.routes.ts**:

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

import { HomeComponent } from './componentes/home/home.component';
import { ProductosComponent } from './componentes/productos/productos.component';
```

```
import { CategoriasComponent } from
'./componentes/categorias/categorias.component';

import { TodosComponent } from
'./componentes/productos/todos/todos.component';

import { BuscarComponent } from
'./componentes/productos/buscar/buscar.component';

import { NuevoComponent } from
'./componentes/productos/nuevo/nuevo.component';


const routes: Routes = [

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

    { path: 'productos', component: ProductosComponent, children : [

        { path: 'todos', component: TodosComponent},

        { path: 'buscar', component: BuscarComponent },

        { path: 'nuevo', component: NuevoComponent },

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

        ]

     },

    { path: 'categorias', component: CategoriasComponent },

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

];


@NgModule({

    imports: [RouterModule.forRoot(routes)],

    exports: [RouterModule]

})

export class FeatureRoutingModule {}
```

Esas rutas se concatenan con la ruta padre y pasan a ser **/productos/todos**, **/productos/buscar** etc.

La salida:



¿Cómo podemos gestionar mejor las rutas hijas? Si hay una estructura compleja de rutas podemos gestionarla mejor guardando las rutas en diferentes archivos.

Creamos el fichero de rutas **producto.routes.ts** en la carpeta del componente producto:

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

import { TodosComponent } from './todos/todos.component';
import { BuscarComponent } from './buscar/buscar.component';
import { NuevoComponent } from './nuevo/nuevo.component';

export const PRODUCTO_RUTAS: Routes = [
    { path: 'todos', component: TodosComponent},
        { path: 'buscar', component: BuscarComponent },
        { path: 'nuevo', component: NuevoComponent },
        { path: '**', redirectTo: 'todos' }

];

@NgModule({
    imports: [RouterModule.forChild(PRODUCTO_RUTAS)],
    exports: [RouterModule]
})
export class FeatureRoutingModule {}
```

Y luego utilizamos las rutas exportadas en el padre (**app.routes.ts**):

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

import { HomeComponent } from './componentes/home/home.component';
import { ProductosComponent } from
'./componentes/productos/productos.component';
import { CategoriasComponent } from
'./componentes/categorias/categorias.component';
import { PRODUCTO_RUTAS } from
'./componentes/productos/productos.routes';
```

```
const routes: Routes = [
    { path: 'home', component: HomeComponent },
    { path: 'productos', component: ProductosComponent, children: PRODUCTO_RUTAS },
    { path: 'categorias', component: CategoriasComponent },
    { path: '**', redirectTo: 'home' }
];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
})
export class FeatureRoutingModule {}
```

El resultado es el mismo

Formularios en Angular

Hay dos formas de gestionar formularios en Angular:

- Con **plantillas**
- Mediante **código** (formularios **reactivos**)

El primer paso es importar el módulo de formularios en nuestra aplicación. En **app.modules.ts**:

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

import { AppComponent } from './app.component';
import { HomeComponent } from './componentes/home/home.component';

import {FeatureRoutingModule} from './app.routes';
import { ProductosComponent } from './componentes/productos/productos.component';
import { CategoriasComponent } from './componentes/categorias/categorias.component';
import { NavbarComponent } from './componentes/navbar/navbar.component';
import { TodosComponent } from './componentes/productos/todos/todos.component';
import { BuscarComponent } from './componentes/productos/buscar/buscar.component';
```

```
import { NuevoComponent } from
'./componentes/productos/nuevo/nuevo.component';
import { FormsModule } from "@angular/forms";


@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ProductosComponent,
    CategoriasComponent,
    NavbarComponent,
    TodosComponent,
    BuscarComponent,
    NuevoComponent
  ],
  imports: [
    BrowserModule,
    FeatureRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

En nuestro caso vamos a emplear plantillas para trabajar con los formularios. Vamos a definir un formulario para los productos en el componente **Nuevo Producto**. A la hora de trabajar con elementos de un formulario disponemos de la directiva **ngModel** que permite una relación entre los controles de los formularios y los campos de un componente en lo que se denomina **doble binding**. De este modo, si asociamos el control a una propiedad, cualquier cambio en el control se actualiza en la propiedad y viceversa.

Para pode emplear el **ngModel** es imprescindible que los controles tengan especificado el atributo **name**.

El primer paso que vamos a dar es definir el modelo para los objetos Producto:

```
ng g class modelo/producto
```

Y lo modificamos:

```
export class Producto {
    constructor(public id:number,
```

```
        public idCategoria:number,
        public nombre:string,
        public fecha:string,
        public precio:string){
    }
}
```

NOTA: De momento nos olvidamos del campo idCategoria.

NOTA: Los campos fecha y precio los declaramos como **string** dado que, en el caso del precio, queremos que el usuario pueda introducir los datos en castellano y, en el caso de la fecha, vamos a emplear el control Date que requiere la fecha en formato yyyy-MM-dd.

```
export class Producto {
    constructor(public id:number,
        public idCategoria:number,
        public nombre:string,
        public fecha:string,
        public precio:string){
    }
}
```

Queremos enlazar el formulario con un producto, así que lo definimos en **nuevo.component.ts**:

```
import { Component, OnInit } from '@angular/core';
import { Producto } from 'src/app/modelo/producto';

@Component({
  selector: 'app-nuevo',
  templateUrl: './nuevo.component.html',
  styles: []
})
export class NuevoComponent implements OnInit {
  public producto:Producto;
  constructor() {
    this.producto = new Producto(0,0,'', new Date().toISOString().slice(0,10) ,"0,00");
  }
}
```

Y creamos el documento HTML para el formulario:

```
<div class=row class="m-3">
  <form>
    <div class="form-group">
      <label for="nombre">Nombre</label>
      <input type="text" required #nombre class="form-control" [(ngModel)]="producto.nombre" name="nombre" placeholder="Nombre del Producto">
    </div>
    <div class="form-group">
      <label for="precio">Precio</label>
      <input type="text" class="form-control" [(ngModel)]="producto.precio" name="precio" placeholder="Precio del Producto">
    </div>
    <div class="form-group">
      <label for="fecha">Fecha</label>
      <input type="date" class="form-control" [(ngModel)]="producto.fecha" name="fecha" placeholder="Fecha de Compra">
    </div>
    <input type="submit" class="btn btn-primary" value="Guardar">
  </form>
Producto : {{ this.producto | json }}<br>
Clases Nombre : {{ nombre.className}}
</div>
```

En el formulario:

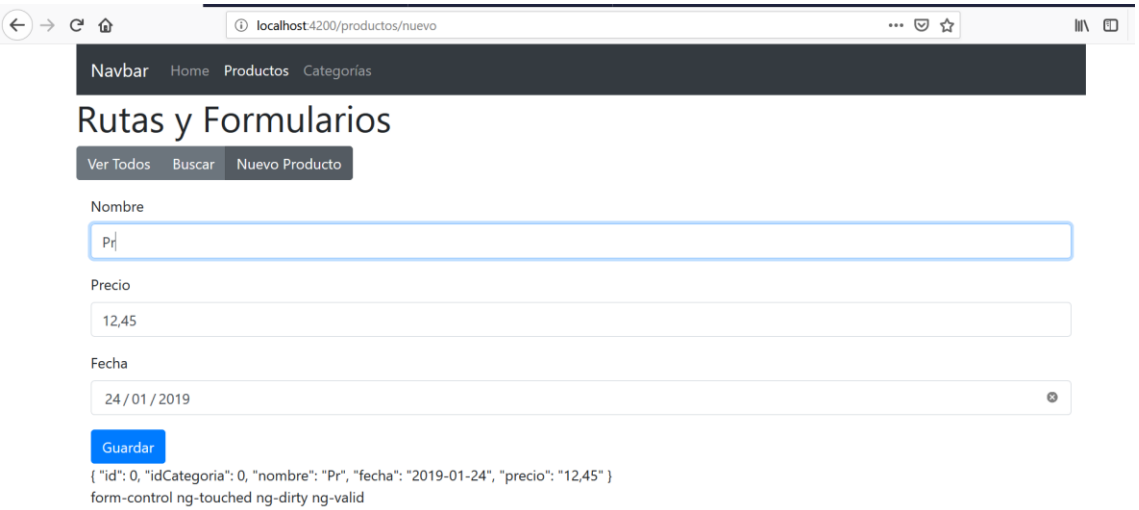
- Empleamos **[(ngModel)]** para asociar los controles con las propiedades del componente mediante un **doble enlazado**. El control almacenará e valor de la propiedad (de ahí los corchetes) y, cada vez que cambie el control, se informará a la propiedad mediante un evento (de ahí los paréntesis). Para que sea más fácil de recorda el orden al símbolo se le llama **banana () in a box []**
- Se ha añadido una variable de plantilla en el campo **nombre** para poder referenciarla más tarde
- Se ha añadido la validación HTML5 **required** para indicar que el control no puede estar vacío
- Para ver el funcionamiento se muestra el producto como un objeto Json y las clases del control nombre

Si ejecutamos el formulario:

Si nos fijamos en las clases del formulario aparecen las siguientes que gestiona Angular:

- ng-untouched: Indica que el control no ha sido tocado (no hemos accedido a él). Se convierte en ng-touched al acceder (y dejar) el control
- ng-pristine: Indica que el control no ha sido modificado. Se convierte en ng-dirty cuando lo ha sido
- ng-invalid: Indica que el control no es válido (lleva el atributo **required** y no tiene información)

Segn vamos escribiendo se va actualizando la información del producto se actualizan las clases el objeto producto:



Como se puede ver la infomación se va actualizando según se va escribiendo y las clases pasan a ng-touched, ng-dirty y ng-valid.

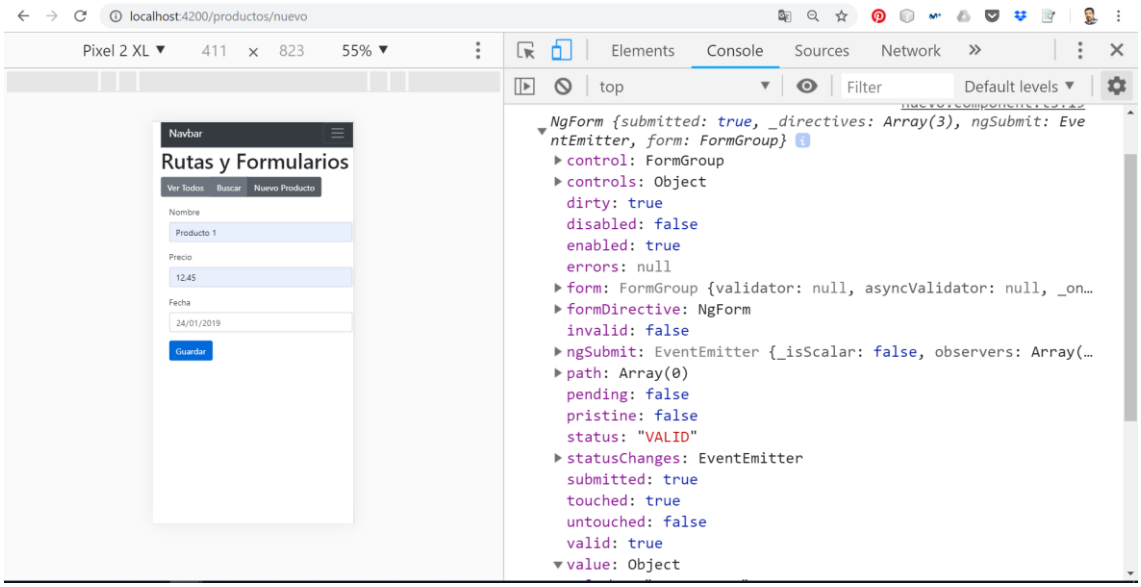
¿Cómo enviamos la infomación al componente para trabajar con ella? Para ello tenemos el evento **ngSubmit**. En este evento podemos (si es necesario) enviar el formulario entero (que será un objeto de tipo NgForm):

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

Mostramos el contenido en el método:

```
enviar(formulario:NgForm){
  console.log(formulario);
  console.log(this.producto);
}
```

La salida:

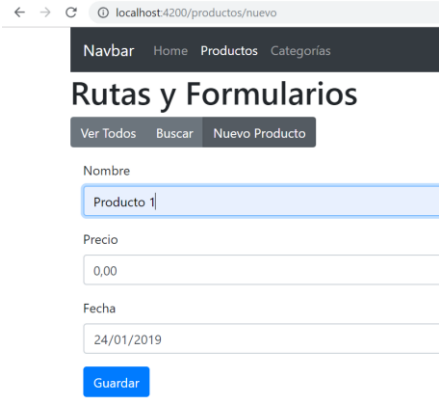


En la salida podemos ver todas las propiedades del formulario (entre ellas la propiedad value que contiene los valores de todos los controles del mismo). También tenemos propiedades para saber si el formulario ha sido modificado (dirty), si es válido (valid), etc.

Por ejemplo, podemos deshabilitar el botón de enviar hasta que el formulario sea válido:

```
<input type="submit" class="btn btn-primary" value="Guardar" [disabled]="!formulario.valid">
```

Si ejecutamos ahora:



También podemos hacer algo similar para mostrar un mensaje al usuario:

```
<input type="text" required #nombre="ngModel" class="form-control" [(ngModel)]="producto.nombre" name="nombre" placeholder="Nombre del Producto">
```

```
      <div  [hidden]="nombre.valid  ||  !nombre.touched"  class="mt-2
alert alert-danger">El nombre es obligatorio</div>
```

En primer lugar empleamos una variable de plantilla para poder hacer referencia al control. En el caso de los formularios con **ngModel** es preciso asignar dicha variable a **ngModel** para poder acceder a ella.

Una vez tenemos acceso al control, mostramos el mensaje de error lo ocultamos si **el nombre es válido O si no se ha accedido al campo nombre**. Esto quiere decir que el mensaje **sólo aparecerá** cuando el usuario haya pasado por el control y lo haya dejado en blanco.

Si quitamos la comprobación **!touched** el mensaje aparecería siempre.

Validación HTML

Angular admite todos los atributos de validación HTML5 que son los siguientes:

- **minlength** número: Especifica el número mínimo de caracteres en un control
- **maxlength** número: Número máximo de caracteres
- **min** valor: El valor mínimo que debe tener el control. Se aplica a valores numéricos y fechas
- **max** valor: Similar pero con el valor máximo
- **required**: indica que el control debe tener algún valor
- **pattern** expr: El control debe cumplir con la expresión regular indicada.

Con estos valores y las clases de Angular podemos modificar el aspecto o mostrar mensajes de error al usuario. Por ejemplo, para el campo nombre que es obkligatorio.

Pongamos una nueva validación al control nombre de modo que sea obligatorio un mínimo de 5 caracteres.

¿Cómo podemos mostrar diferentes mensajes en función del tipo de error?.

Todos los controles tienen una propiedad **errors** que dispone de una propiedad con el tipo de error donde **required** es un valor booleano, **minlength, maxlenght** son objetos con las propiedades **requiredLength** y **actualLength**) y **pattern** es otro objeto con las propiedades **requiredPattern** y **actualValue**.

En caso de que no haya errores **no se genera el objeto errors**.

NOTA: Las validaciones **min** y **max** se emplean en los controles **number** y **date** para especificar los valores válidos **cuando se introducen los valores con los controles estándar**. Es decir, el control number no permitirá con las flechas pasar de esos valores (de manera similar debería comportarse el control fecha). Pero eso **no evita que el usuario pueda introducir manualmente** valores erróneos que no hacen que Angular etiquete el valor como inválido.

En la práctica, si queremos controlar esos valores deberíamos hacerlo por código.

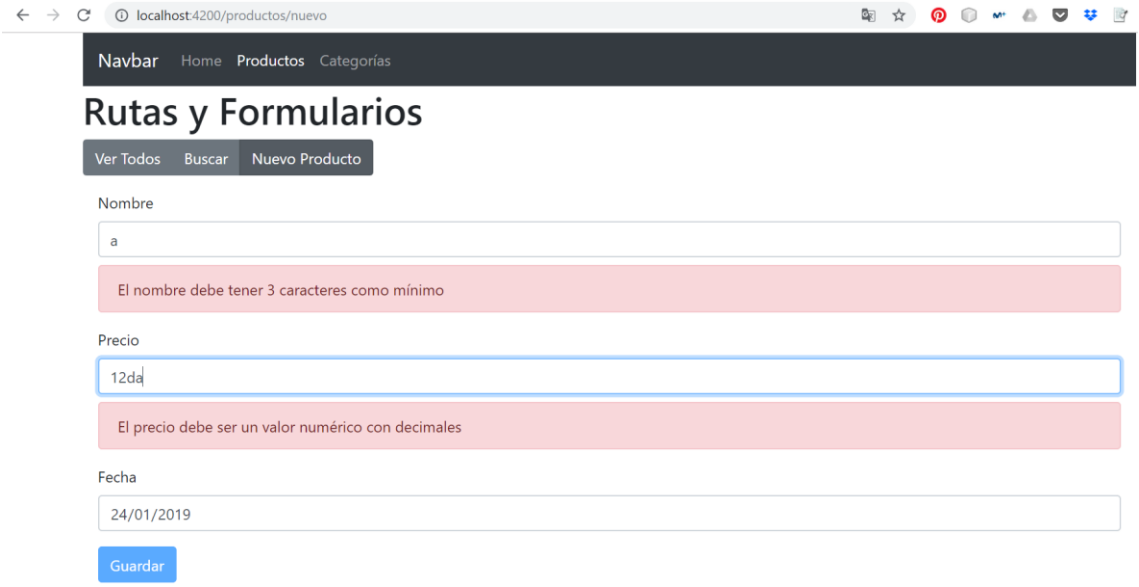
Además del nombre vamos a validar que el precio sea un número en castellano.

Si queremos mostrar mensajes personalizados **tendremos que usar el objeto error**.

```
<div class=row class="m-3">
  <form #formulario="ngForm" (ngSubmit)="enviar(formulario)">
    <div class="form-group">
```

```
      <label for="nombre">Nombre</label>

      <input type="text" required minlength=3 #nombre="ngModel" class="form-control" [(ngModel)]="producto.nombre" name="nombre" placeholder="Nombre del Producto">

      <div [hidden]="!nombre.errors?.required || !nombre.touched" class="mt-2 alert alert-danger">El nombre es obligatorio</div>

      <div *ngIf="nombre.dirty && nombre.errors?.minlength " class="mt-2 alert alert-danger">El nombre debe tener {{ nombre.errors.minlength.requiredLength}} caracteres como mínimo</div>

    </div>

    <div class="form-group">

      <label for="precio">Precio</label>

      <input type="text" pattern="^[-+]?\d+(\,\d+)?$" #precio="ngModel" class="form-control" [(ngModel)]="producto.precio" name="precio" placeholder="Precio del Producto">

      <div *ngIf="precio.dirty && precio.errors?.pattern" class="mt-2 alert alert-danger">El precio debe ser un valo numérico con decimales</div>

    </div>

      <div class="form-group">

      <label for="fecha">Fecha</label>

      <input type="date" max='2019-01-29' class="form-control" [(ngModel)]="producto.fecha" name="fecha" placeholder="Fecha de Compra">

    </div>

    <input type="submit" class="btn btn-primary" value="Guardar" [disabled]="!formulario.valid">

  </form>


</div>
```

La salida con errores:



Otros controles

En el formulario podemos añadir el resto de tipos de controles. Vamos a añadir un control **select** basado en categorías. Para ello vamos a crear la clase Categoria:

```
ng g class modelo/categoría
```

Y la clase:

```
export class Categoria {
    public id:number;
    public nombre:string;
    constructor(id:number,nombre:string){
        this.id = id;
        this.nombre = nombre;
    }
}
```

Modificamos la clase nuevo.component.ts para añadir las categorías (debería ir en un servicio):

```
import { Categoria } from "./categoria";

export class NuevoComponent {

    public categorias:Categoria[] = [
        new Categoria(1,"Categoría 1"),
        new Categoria(2,"Categoría 2"),
        new Categoria(2,"Categoría 3"),
        new Categoria(4,"Categoría 4")
    ]
```

```
    …
}
```

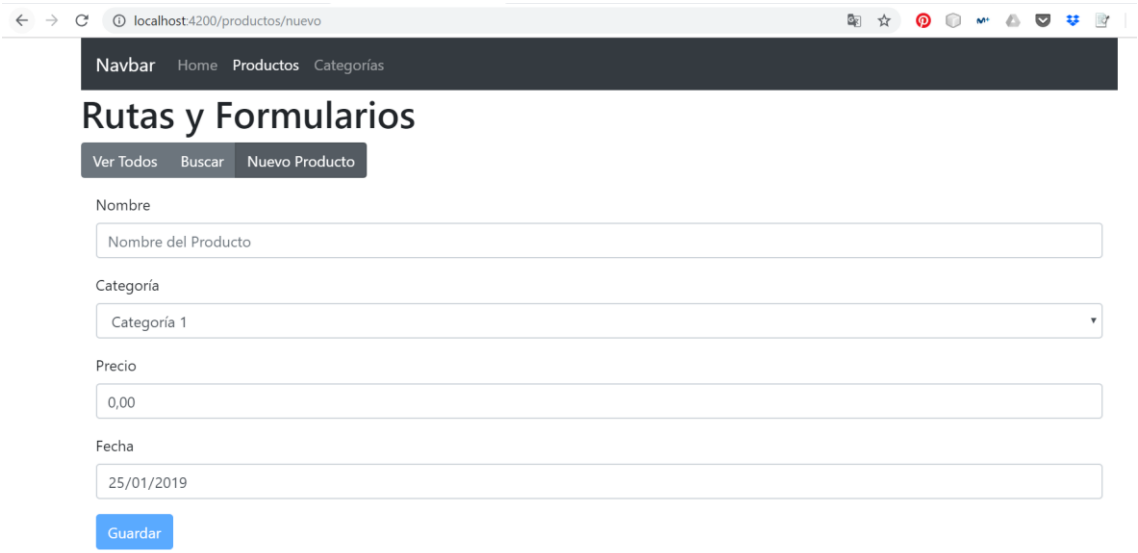
Añadimos una categoría al producto en **nuevo.component.ts**:

```
this.producto = new Producto(0,1,'', new Date().toISOString().slice(0,10) ,"0,00");
```

Y añadimos la select:

```
    <div class="form-group">
      <label for="categoria">Categoría</label>
      <select class="custom-control custom-select"
            [(ngModel)]="producto.idCategoria" name="categoria">
        <option *ngFor="let cat of categorias"
                [value]="cat.id">{{cat.nombre}}</option>
      </select>
    </div>
```

La salida:



Ahora añadimos un checkbox para indicar si queremos modificar el producto si existe.

Creamos una propiedad en el formulario llamada modificar y añadimos el control:

```
  private modificar:boolean = false;
```

El formulario:

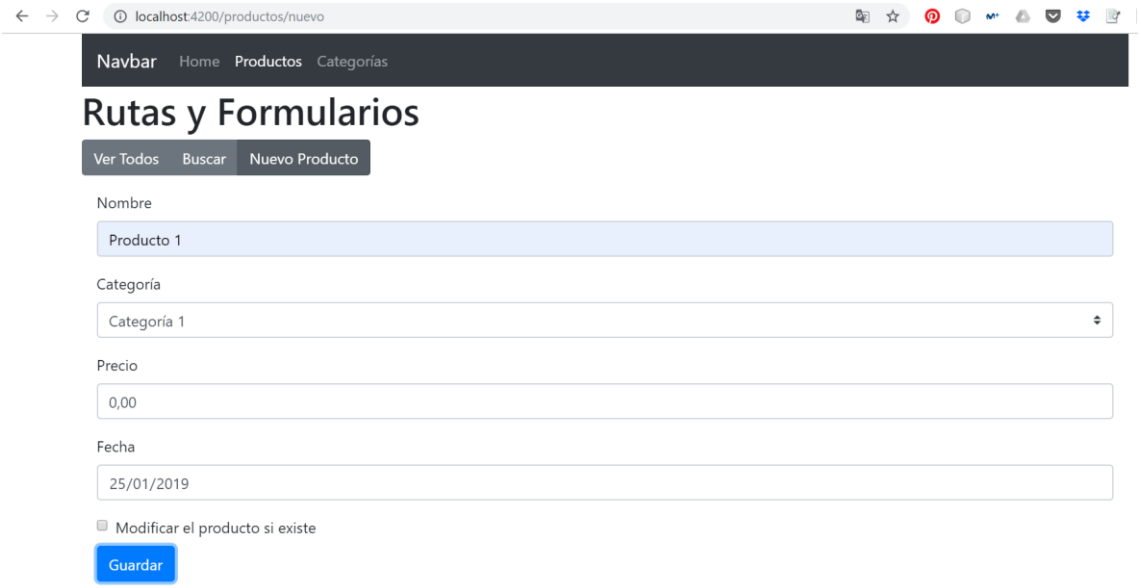
```
<div class="form-check">
        <input type="checkbox" class="form-check-input" name="modificar" [(ngModel)]="modificar">
        <label class="form-check-label" for="modificar">Modificar el producto si existe</label>
```

```
</div>
```

Si se marca la casilla, se almacenará un valor **true**, en caso contrario un valor **false**.

NOTA: Si **no se inicializa** la propiedad y se envía el formulario sin cambiarla estaría en estado **undefined**.

La salida:



En cuanto a los **botones de radio** el valor a almacenar en la propiedad es el especificado en los atributos **value** de cada cada opción. Vamos a cambiar el ejemplo para que nos indique qué acción tomar.

La popiedad:

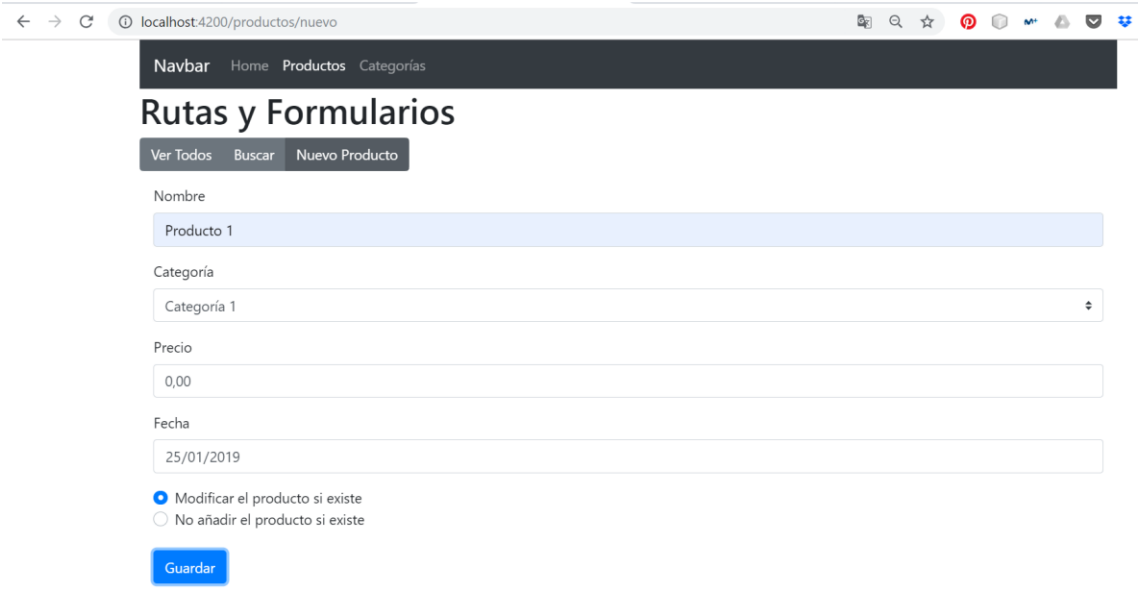
```
private modificar:string = "Modificar";
```

Y el formulario:

```
<div class="form-group">

        <div class="custom-control custom-radio">

            <input type="radio" name="modificar" id="m2" [(ngModel)]="modificar" value="Modificar" class="custom-control-input">

            <label class="custom-control-label" for="m2">Modificar el producto si existe</label>

        </div>

        <div class="custom-control custom-radio">

            <input type="radio" name="modificar" id="m1" [(ngModel)]="modificar" value="NoAñadir" class="custom-control-input">

            <label class="custom-control-label" for="m1">No añadir el producto si existe</label>

        </div>
```

```
</div>
```

La salida:



Formularios reactivos

En este caso el formulario se define por código. ¿Cómo definiríamos el mismo fomulario por código?. Vamos a crear un nuevo componente para hacer la prueba:

```
ng g c componentes/productos/nuevoReactive/nuevo --spec=false -is
```

Para poder usar este tipo de fomularios necesitamos importar **ReactiveForms** en **app.modules.ts**:

```
import { BrowserModule } from '@angular/platform-browser';
…
import { FormsModule, ReactiveFormsModule } from "@angular/forms";


@NgModule({
  declarations: [
    AppComponent,
      …
  ],
  imports: [
    BrowserModule,
    FeatureRoutingModule,
    FormsModule,
    ReactiveFormsModule
```

```
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Creamos el componente:

```
ng g c componentes/productos/nuevoReactive/nuevoReactive --spec=false
-is
```

Definimos la plantilla como un formulario HTML normal especificando el atributo **formGroup** del formulario que definiremos posteriormente por código. Además empleamos el atributo **formControlName** para asociar cada control con HTML con el creado por código:

```
<div class=row class="m-3">
  <form [formGroup]="formulario" (ngSubmit)="enviar()">
    <div class="form-group">
      <label for="nombre">Nombre</label>
      <input type="text" formControlName="nombre" class="form-control"
placeholder="Nombre del Producto">
    </div>
<div class="form-group">
      <label for="categoria">Categoría</label>
      <select class="custom-control custom-select"
formControlName="categoria">
        <option *ngFor="let cat of categorias"
[value]="cat.id">{{cat.nombre}}</option>
      </select>
    </div>
    <div class="form-group">
      <label for="precio">Precio</label>
      <input type="text" class="form-control" formControlName="precio"
placeholder="Precio del Producto">
    </div>
    <div class="form-group">
      <label for="fecha">Fecha</label>
      <input type="date" class="form-control" formControlName="fecha"
placeholder="Fecha de Compra">
    </div>
    <div class="form-group">
```

```
      <div class="custom-control custom-radio">
        <input type="radio" formControlName="modificar" id="m2" value="Modificar" class="custom-control-input">
        <label class="custom-control-label" for="m2">Modificar el producto si existe</label>
      </div>
      <div class="custom-control custom-radio">
        <input type="radio" formControlName="modificar" id="m1" value="NoAñadir" class="custom-control-input">
        <label class="custom-control-label" for="m1">No añadir el producto si existe</label>
      </div>
    </div>
    <input type="submit" class="btn btn-primary mt-2" value="Guardar">
  </form>
</div>
```

Y creamos el formulario en el constructor de la clase:

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';
import { Categoria } from 'src/app/modelo/categoria';

@Component({
  selector: 'app-nuevo',
  templateUrl: './nuevo.component.html',
  styles: []
})
export class NuevoReactiveComponent implements OnInit {
  formulario:FormGroup;
  public categorias:Categoria[] = [
    new Categoria(1,"Categoría 1"),
    new Categoria(2,"Categoría 2"),
    new Categoria(2,"Categoría 3"),
    new Categoria(4,"Categoría 4")
]

  constructor() {
    this.formulario = new FormGroup({
      'nombre': new FormControl(),
```

```
      'categoria': new FormControl(1),
      'precio': new FormControl('0,00'),
      'fecha': new FormControl(new Date().toISOString().substring(0,10)),
      'modificar': new FormControl('Modificar')
    });
  }

  ngOnInit() {
  }

  enviar(){
    console.log(this.formulario);
  }
}
```

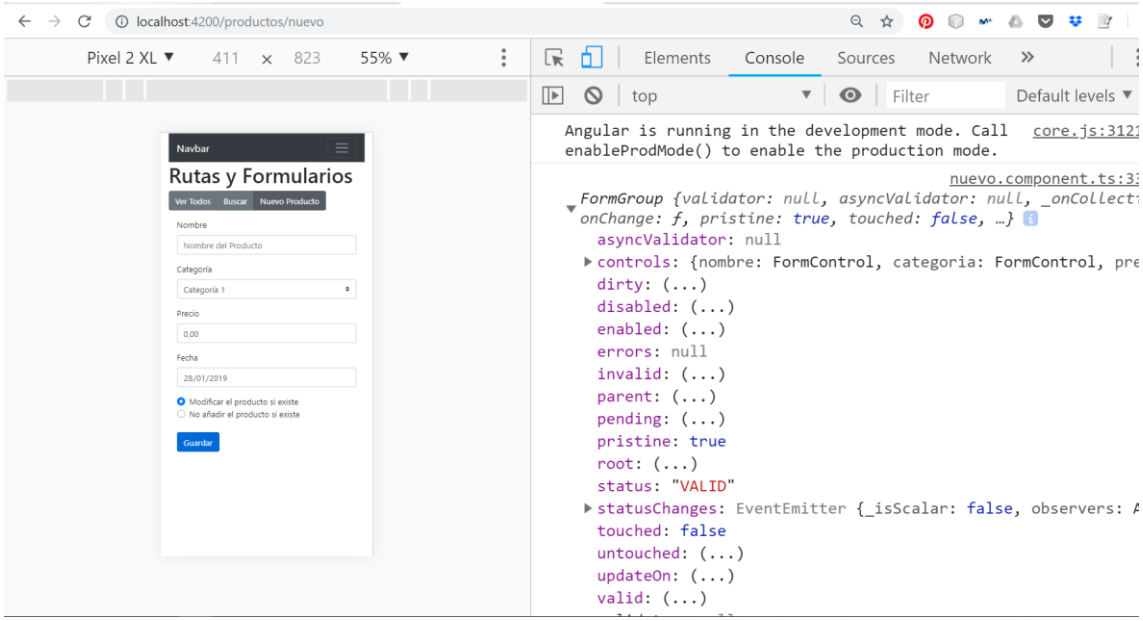
Como se puede ver, hemos creado los controles inicializándolos con valores por defecto.

El último paso es modificar el fichero de rutas para que emplee este componente. En **producto.routes.ts**:

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

import { TodosComponent } from './todos/todos.component';
import { BuscarComponent } from './buscar/buscar.component';
import { NuevoReactiveComponent } from './nuevoReactive/nuevo/nuevoReactive.component';

export const PRODUCTO_RUTAS: Routes = [
    { path: 'todos', component: TodosComponent},
        { path: 'buscar', component: BuscarComponent },
        { path: 'nuevo', component: NuevoReactiveComponent },
        { path: '**', redirectTo: 'todos' }

];

@NgModule({
    imports: [RouterModule.forChild(PRODUCTO_RUTAS)],
    exports: [RouterModule]
```

```
})
export class FeatureRoutingModule {}
```

La salida:



Para añadir **validaciones** necesitamos el objeto **Validator** y asociar el tipo de validación a cada componente:

```
this.formulario = new FormGroup({
      'nombre': new FormControl('', [Validators.required, Validators.minLength(3)]),
      'categoria': new FormControl(1),
      'precio': new FormControl('0,00', [Validators.required ,
        Validators.pattern("^[-+]?\\d+(\\,\\d+)?$")]),
      'fecha': new FormControl(new Date().toISOString().substring(0,10), Validators.required),
      'modificar': new FormControl('Modificar')
    });
```

NOTA: En los patrones hay que **escapar** el carácter **** con otro ****.

¿Cómo mostramos mensajes de validación en el HTML? Podemos acceder al control a través del formulario con la expresión **formulario.controls[*'nombre'*]**. De este modo, una validación similar al ejemplo anterior quedaría así:

```
<div class=row class="m-3">
  <form [formGroup]="formulario" (ngSubmit)="enviar()">
    <div class="form-group">
      <label for="nombre">Nombre</label>
```

```
      <input    type="text"    formControlName="nombre"    id="nombre"
class="form-control" placeholder="Nombre del Producto">

    </div>

    <div *ngIf="formulario.controls['nombre'].errors?.required && (

        formulario.controls['nombre'].dirty                          ||
formulario.controls['nombre'].touched)"   class="mt-2   alert   alert-
danger">El nombre es obligatorio</div>

      <div        *ngIf="formulario.controls['nombre'].dirty        &&
formulario.controls['nombre'].errors?.minlength  "  class="mt-2  alert
alert-danger">El         nombre         debe         tener         {{
formulario.controls['nombre'].errors.minlength.requiredLength}}
caracteres como mínimo</div>

    <div class="form-group">

      <label for="categoria">Categoría</label>

      <select           class="custom-control           custom-select"
formControlName="categoria">

        <option       *ngFor="let       cat       of       categorias"
[value]="cat.id">{{cat.nombre}}</option>

      </select>

    </div>

    <div class="form-group">

      <label for="precio">Precio</label>

      <input type="text" class="form-control" formControlName="precio"
placeholder="Precio del Producto">

    </div>

    <div         *ngIf="formulario.controls['precio'].dirty         &&
formulario.controls['precio'].errors?.pattern"    class="mt-2    alert
alert-danger">El precio debe ser un valo numérico con decimales</div>

    <div class="form-group">

      <label for="fecha">Fecha</label>

      <input  type="date"  class="form-control"  formControlName="fecha"
placeholder="Fecha de Compra">

    </div>

    <div class="form-group">

      <div class="custom-control custom-radio">

        <input    type="radio"    formControlName="modificar"    id="m2"
value="Modificar" class="custom-control-input">

        <label   class="custom-control-label"   for="m2">Modificar   el
producto si existe</label>

      </div>
```

```
      <div class="custom-control custom-radio">
        <input type="radio" formControlName="modificar" id="m1" value="NoAñadir" class="custom-control-input">
        <label class="custom-control-label" for="m1">No añadir el producto si existe</label>
      </div>
    </div>
    <input type="submit" class="btn btn-primary mt-2" value="Guardar" [disabled]="!formulario.valid">
  </form>
</div>
```

Evidentemente, la salida es igual al caso anterior.

Inicialización de un formulario con un objeto

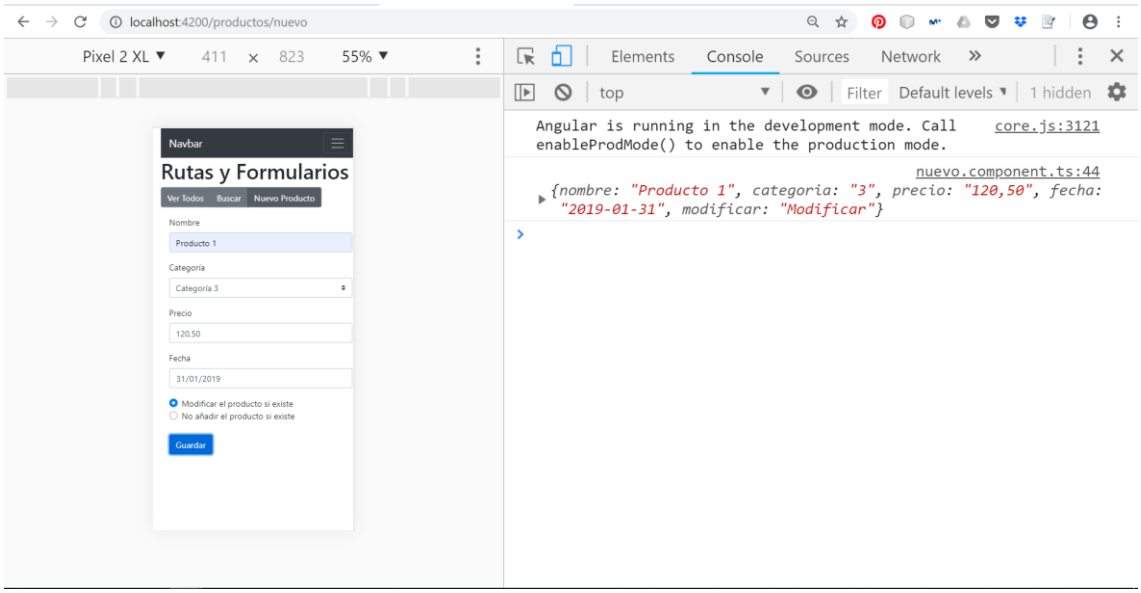
Una ventaja de emplear los formularios reactivos es que es muy simple asociar los formularios con objetos con la misma estructura para inicializarlos o para recuperar su valor.

En nuestro caso vamos a hacer un ejemplo simple trabajando con un objeto **Producto** con los mismos campos que los controles del formulario:

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
import { Categoria } from 'src/app/modelo/categoria';

@Component({
  selector: 'app-nuevo',
  templateUrl: './nuevo.component.html',
  styles: []
})
export class NuevoReactiveComponent implements OnInit {
  formulario:FormGroup;
  public categorias:Categoria[] = [
    new Categoria(1,"Categoría 1"),
    new Categoria(2,"Categoría 2"),
    new Categoria(3,"Categoría 3"),
    new Categoria(4,"Categoría 4")
]

  producto = {
```

```
    'nombre':'',
    'categoria':1,
    'precio':'0,00',
    'fecha':new Date().toISOString().substring(0,10),
    'modificar':'Modificar'
  };

  constructor() {
    this.formulario = new FormGroup({
      'nombre':        new        FormControl('',[Validators.required,
Validators.minLength(3)]),
      'categoria': new FormControl(1),
      'precio': new FormControl('', [Validators.required ,
        Validators.pattern("^[-+]?\\d+(\\,\\d+)?$")]),
      'fecha': new FormControl('', Validators.required),
      'modificar': new FormControl('Modificar')
    });
    this.formulario.setValue(this.producto);
  }

  ngOnInit() {
  }

  enviar(){
    this.producto =this.formulario.getRawValue();
    console.log(this.producto);
  }
}
```

La salida:


localhost:4200/productos/nuevo
Pixel 2 XL
411 x 823
55%
Elements
Console
Sources
Network
top
Filter
Default levels
1 hidden
Navbar
Rutas y Formularios
Ver Todos
Buscar
Nuevo Producto
Nombre
Producto 1
Categoría
Categoría 3
Precio
120,50
Fecha
31/01/2019
Modificar el producto si existe
No añadir el producto si existe
Guardar
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
core.js:3121
nuevo.component.ts:44
{nombre: "Producto 1", categoria: "3", precio: "120,50", fecha: "2019-01-31", modificar: "Modificar"}

Compartiendo el enlace (uso del plugin SocialShare)

Para poder compartir la noticia tenemos que incluir el plugin de SocialShare:

```
npm install --save @ionic-native/social-sharing@beta
```

Para poder usarlo, necesitamos instalar previamente Cordova y el plugin correspondiente:

```
npm -i -g cordova
ionic cordova plugin add cordova-plugin-x-socialsharing
```

Modificamos **app.module.ts** para incluir el módulo:

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



import { IonicModule, IonicRouteStrategy } from '@ionic/angular';
import { SplashScreen } from '@ionic-native/splash-screen/ngx';
import { StatusBar } from '@ionic-native/status-bar/ngx';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HttpClientModule } from '@angular/common/http';

import { SocialSharing } from '@ionic-native/social-sharing/ngx'



@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(), AppRoutingModule,
HttpClientModule],
  providers: [
    StatusBar,
    SplashScreen,
    SocialSharing,
    { provide: RouteReuseStrategy, useClass: IonicRouteStrategy }
  ],
  bootstrap: [AppComponent]
```

```
})
export class AppModule {}
```

Ahora añadimos botones **FAB** (ver la documentación) a la página junto con botones: