

TEMA II

LAS BASES DE ANGULAR



2.3 Primeros pasos en Angular

Para comprender el funcionamiento de Ionic, es necesario saber ciertas cosas de angular:

- Estructura del proyecto de Angular
- Componentes de Angular
- Módulos en Angular
- Navegación mediante el RouterModule
- Módulos en Angular
- LazyLoad
- Servicios

Estos conceptos son usados en casi toda aplicación de ionic hoy en día.

Creación y ejecución de un proyecto

Para crear un proyecto, utilizaremos la herramienta angular cli(Command Line Interface) que instalamos. Es una herramienta incluida en angular que nos permite crear el esqueleto de una aplicación y configura algunas herramientas básicas

Para crear un proyecto angular: **ng new "Nombre"** (no queremos que añada rutas, lo haremos después nosotros)

```
C:\Users\Administrador>ng new Ejangular-uno
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use? CSS
CREATE Ejangular-uno/angular.json (3622 bytes)
CREATE Ejangular-uno/package.json (1256 bytes)
CREATE Ejangular-uno/README.md (1030 bytes)
CREATE Ejangular-uno/tsconfig.json (458 bytes)
CREATE Ejangular-uno/tslint.json (3185 bytes)
CREATE Ejangular-uno/.editorconfig (274 bytes)
CREATE Ejangular-uno/.gitignore (631 bytes)
CREATE Ejangular-uno/.browserslistrc (853 bytes)
CREATE Ejangular-uno/karma.conf.js (1025 bytes)
CREATE Ejangular-uno/tsconfig.app.json (287 bytes)
CREATE Ejangular-uno/tsconfig.spec.json (333 bytes)
CREATE Ejangular-uno/src/favicon.ico (948 bytes)
CREATE Ejangular-uno/src/index.html (298 bytes)
CREATE Ejangular-uno/src/main.ts (372 bytes)
CREATE Ejangular-uno/src/polyfills.ts (2835 bytes)
```

Todos los paquetes que instala, al igual que pasa con Ionic no van a producción.

Para ejecutar el proyecto (desde la carpeta creada) **ng serve -o**

Levanta un servidor local en un puerto, por defecto 4200, con el parámetro -o indicamos que además queremos abrir la aplicación en el navegador

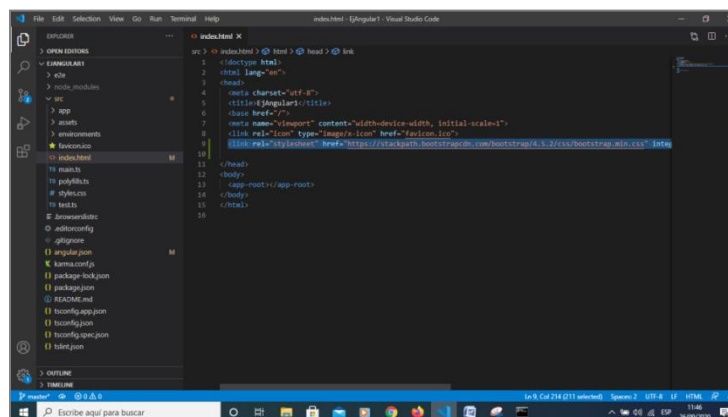
```
C:\Users\Administrador\Ejangular-uno>ng serve -o
? Would you like to share anonymous usage data about this project with the Angular Team
at
Google under Google's Privacy Policy at https://policies.google.com/privacy? For more
details and how to change this setting, see http://angular.io/analytics. No
Compiling @angular/core : es2015 as esm2015
Compiling @angular/common : es2015 as esm2015
Compiling @angular/platform-browser : es2015 as esm2015
Compiling @angular/platform-browser-dynamic : es2015 as esm2015

chunk (main) main.js, main.js.map (main) 57 kB [initial] [rendered]
chunk (polyfills) polyfills.js, polyfills.js.map (polyfills) 141 kB [initial] [rendered] chunk {run
time} runtime.js, runtime.js.map (runtime) 6.15 kB [entry] [rendered]
chunk (styles) styles.js, styles.js.map (styles) 12.5 kB [initial] [rendered]
chunk (vendor) vendor.js, vendor.js.map (vendor) 2.37 MB [initial] [rendered]
Date: 2020-09-24T18:22:40.489Z - Hash: 5be5a488aa14a9783e3c - Time: 54265ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://loc
alhost:4200/ **
: Compiled successfully.
```

Puede ser que al ejecutar en ubuntu tengamos que aumentar el límite de observadores de archivos, para ello desde el terminal ejecutamos

echo fs.inotify.max_user_watches = 524288 | sudo tee -a /etc/sysctl.conf && sudo sysctl -p

El código de nuestro primer proyecto



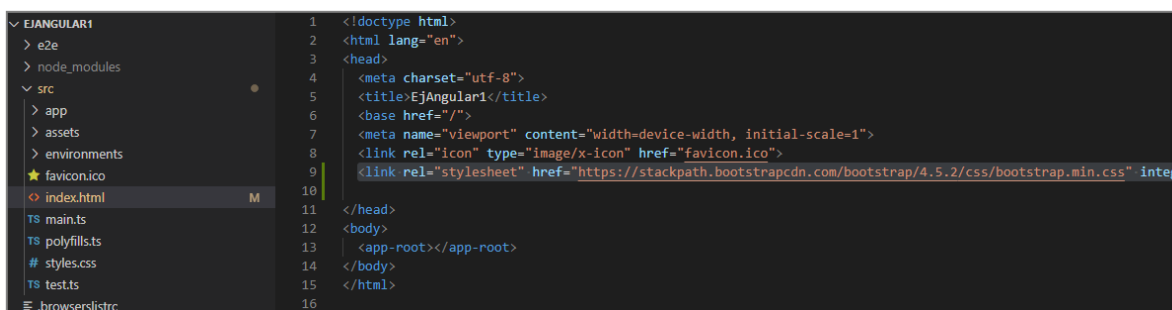
Ejemplo 2: Hagamos algunos cambios sencillos

- Añadimos bootstrap, desde la web getbootstrap.com

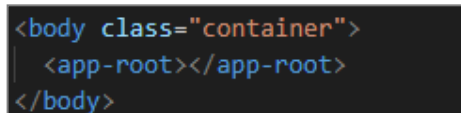


Copiamos el link de estilos solamente.

En el archivo index de nuestra aplicación añadimos el link para aplicar bootstrap a nuestros proyectos.

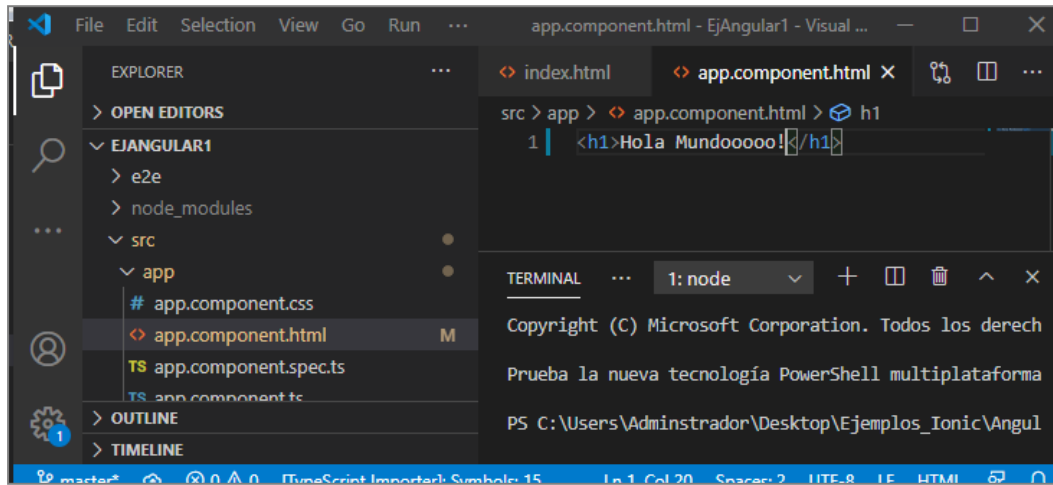


Aplico el bootstrap al body del archivo index.html



Al refrescar el navegador se aprecia el cambio de estilos

- Hagamos aparecer el clásico Hola Mundo



En el archivo app.components.html

Componentes básicos

Los componentes son bloques básicos de construcción de las páginas web en angular, pueden ser una página entera o un elemento independiente que nos permite reutilizar código.

Son una combinación de un archivo html (parte visual) y un archivo ts (su funcionalidad). Se definen como clases (escritas en TypeScript) y tienen una función específica: menú de navegación, contact, barrar lateral....

Los componentes se pueden crear manualmente o través de un comando

Si nos fijamos en la carpeta app, vemos que por defecto tenemos ya un componente creado, la app. Para cada componente tenemos una serie de archivos:

- .ts: se describe la clase

```
src > app > TS app.components.ts > ...
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-root',
5    templateUrl: './app.component.html',
6    styleUrls: ['./app.component.css']
7  })
8  export class AppComponent {
9    title = 'Ejemplo3';
10 }
11
```

La clase se llama AppComponent, y la etiqueta @Component es su "decorador", le indica a angular que es lo que podrá ser inyectado en html.

- .HTML : será lo que se muestre en html, la etiqueta app-root es el selector de mi componente, también aparece este archivo como templateUrl de mi @Component
- .css: los estilos del componente, la ruta de este archivo también aparece en @Componente
- El otro archivo. spect.ts lo veremos más adelante, es para hacer pruebas unitarias.

Volcar el valor de una propiedad en la vista

Si en un componente, en la clase de su definición, tenemos un dato, podemos mostrarlo en la vista del componente, para que aparezca su valor al visualizar el componente en la página.

Nuestro componente tendrá esta definición en el TypeScript:

```
export class HomePage {  
  nombre_propiedad:string = 'Esto es un test';  
}
```

Después de crear la propiedad en el componente, podemos mostrar su valor en la vista. Esto se hace desde la vista con la sintaxis de las dobles llaves.

```
{{nombre_propiedad}}
```

Asignar un valor del componente a un atributo

Ahora tenemos una propiedad definida de esta manera:

```
autor:string = 'Nicolás Molina';
```

Si queremos mostrar ese valor como un atributo de un componente, usamos la sintaxis de los corchetes. Solo que ahora colocamos los corchetes en el nombre del atributo cuyo valor tenemos que traer de una propiedad.

```
<input type="text" [value]="autor">
```

Nota: En este ejemplo solamente estamos asignando el valor, lo que haya en la propiedad "autor" se mostrará como value del input. Sin embargo, si modificamos el texto del input nosotros mismos, el nuevo valor no estará viajando al componente. Dicho de manera técnica, el binding es de una única dirección. Angular es capaz ahora de desplegar un sistema de binding mucho más optimizado, luego lo veremos.

Definir un evento sobre un elemento

La manera de definir un evento sobre un elemento de la vista, o simplemente invocar una función (método de la clase del componente) cuando ocurra cualquier cosa, es mediante la sintaxis de los paréntesis.

Entre paréntesis colocaremos el tipo de evento y luego como valor el nombre de la función a ejecutar.

```
<button ion-button (click)="cambiarAutor()"> Cambiar Autor  
</button>
```

Tenemos un botón y su evento "click" asignado a la función cambiarAutor(). Esa función se define como un método en el código TypeScript del componente.

```
export class HomePage {  
  autor:string = 'Ana M.a López';  
  
  cambiarAutor() {  
    this.autor = 'Colaborador en salesianas';  
  }  
}
```

Al hacer clic se invocará el método `cambiarAutor()`, que modificará la propiedad "autor". Eso provocará un cambio en la página, allí donde se esté usando esa propiedad, actualizando el valor en la vista.

Realizar un doble binding sobre una propiedad editable

Veamos cómo conseguir un binding a dos vías. Esto se hace por medio de un mecanismo explícito, en el que usamos la directiva `ngModel` y la sintaxis "banana in a box" (paréntesis metidos dentro de corchetes).

```
<input type="text" [(ngModel)]="autor">
```

Estamos usando dos características de las comentadas antes. Los corchetes sirven para asignar un valor en un atributo y los paréntesis para ejecutar una función. Lo que se está asignando y ejecutando es `ngModel`, la directiva que nos permite hacer esta operación.

Para que esto funcione es necesario importar en el módulo del componente

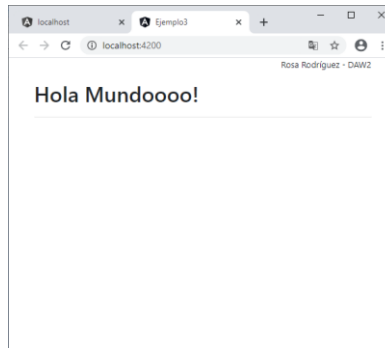
```
import { FormsModule } from '@angular/forms';
```

También hay que añadirlo a los imports

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

Ejemplo_3: Cambiemos algunas cosas de nuestro "Hola Mundo"

Añadimos en nuestra clase AppComponent algunas propiedades para obtener el siguiente resultado



1. Añadimos dos propiedades a la clase (en el archivo .ts)
2. Podemos utilizarlas directamente en nuestro .html con {{ }}
3. Aplicamos estilos en el archivo .css

```
src > app > TS app.component.ts > AppComponent
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-root',
5    templateUrl: './app.component.html',
6    styleUrls: ['./app.component.css']
7  })
8  export class AppComponent {
9    title = 'HolaMundo';
10   identificador="Rosa Rodríguez";
11   curso="DAW2";
12 }
13
```

```
src > app > # app.component.css > p
1  p {
2    text-align: right;
3    font-size: small;
4  }
```

```
src > app > app.component.html > p
1  <p>{{identificador}} - {{curso}}</p>
2
3  <h2>Hola Mundoooo!</h2>
4  <hr>
5
6
```


Componentes

Un componente en Angular es un bloque de código re-utilizable, que consta básicamente de 3 archivos: un CSS, un HTML (también conocido como plantilla o template) y un TypeScript (en adelante, TS). La carpeta app con la que viene Angular por defecto es un componente, aunque un tanto especial.

De cada componente exportamos una clase, que luego se podrá importar a otros componentes. Una aplicación de angular está compuesta por componentes. Los componentes son pequeñas partes lógicas de la aplicación, que van a estar representando a un “trozo de la pantalla”.

Para añadir un componente: **ng g c ruta/nombre**

Ejemplo_4: "Añadimos tres componentes: home, about, contact".

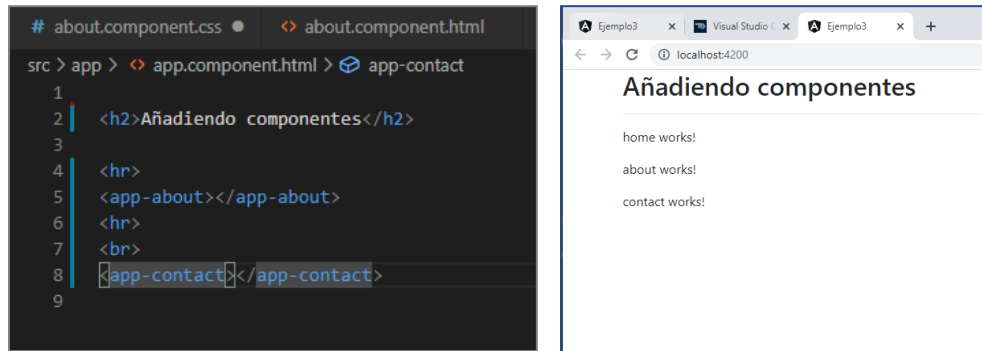
Desde la terminal del Code ejecutamos **ng g c pages/about**

Genera un componente llamado about y crea la carpeta pages dentro de app para almacenar dicho componente.

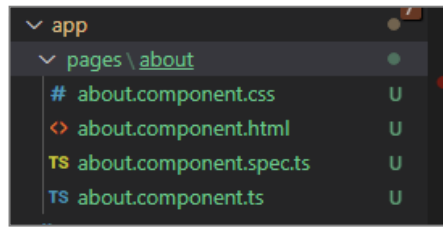
Generamos los otros componentes contact, home

```
PS C:\Users\Administrador\Desktop\Ejemplos_Ionic\Angular\EjAngular1> ng g c pages/contact
CREATE src/app/pages/contact/contact.component.html (22 bytes)
CREATE src/app/pages/contact/contact.component.spec.ts (633 bytes)
CREATE src/app/pages/contact/contact.component.ts (279 bytes)
CREATE src/app/pages/contact/contact.component.css (0 bytes)
UPDATE src/app/app.module.ts (490 bytes)
PS C:\Users\Administrador\Desktop\Ejemplos_Ionic\Angular\EjAngular1> []
```

Para utilizarlos, sólo tengo que incluir la etiqueta en el archivo html de mi appComponent puesto que mis componentes los voy a incluir en este otro componente.

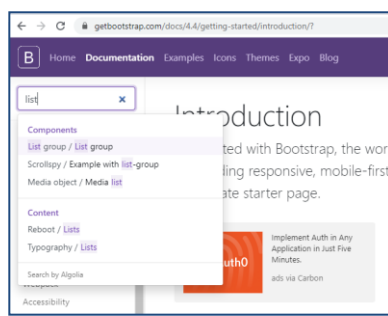


Asociado al componente `AppComponent` tenemos el archivo `app.module.ts`, en este archivo Angular declara todos los componentes que hemos definido.



Vamos a aplicar estilos de bootstrap para que nuestros componentes se vean en una lista más atractiva

En la web de bootstrap 4, busco el código para list group

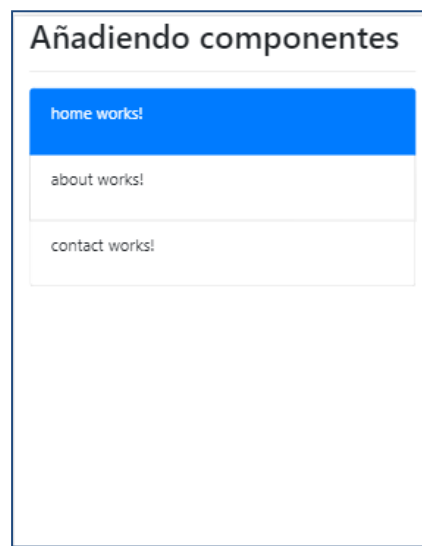


Cambio el código de mi `appcomponent.html` para pegar código bootstrap

```
src > app > <> app.component.html > ...  
1 <h2>Añadiendo componentes</h2>  
2 <hr>  
3 <ul class="list-group">  
4   <li class="list-group-item active"><app-home></app-home></li>  
5   <li class="list-group-item"><app-about></app-about></li>  
6   <li class="list-group-item"><app-contact></app-contact></li>  
7 </ul>  
8  
9
```

No se pueden ejecutar acciones pq es una lista de "texto", no enlaces o botones

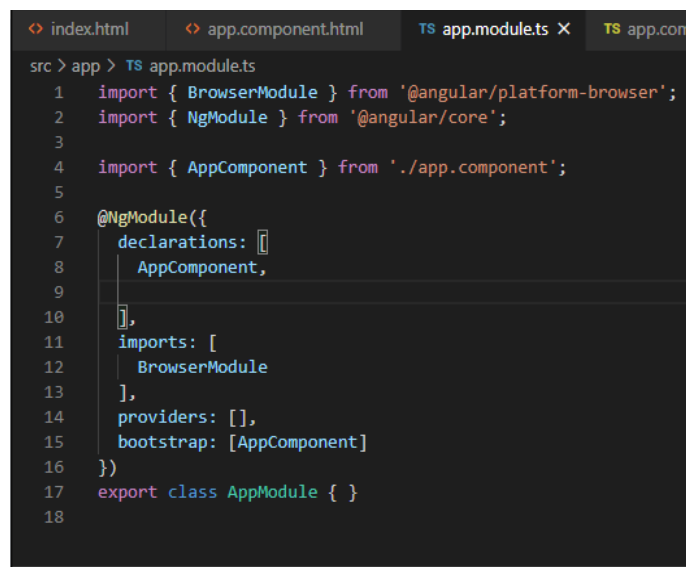
El aspecto de tu aplicación será



Módulos

Los módulos son **contenedores para almacenar los componentes y servicios** de una aplicación. En Angular cada programa se puede ver como un árbol de módulos jerárquico. A partir de un módulo raíz se enlazan otros módulos en un proceso llamado importación. Los módulos se declaran como clases de TypeScript. Estas clases, habitualmente vacías, son "decoradas" con una función especial `@NgModule()` que recibe un objeto como argumento. En las propiedades de ese objeto es donde se configura el módulo.

El módulo AppModule original es generado por el CLI en el fichero `app.module.ts`.

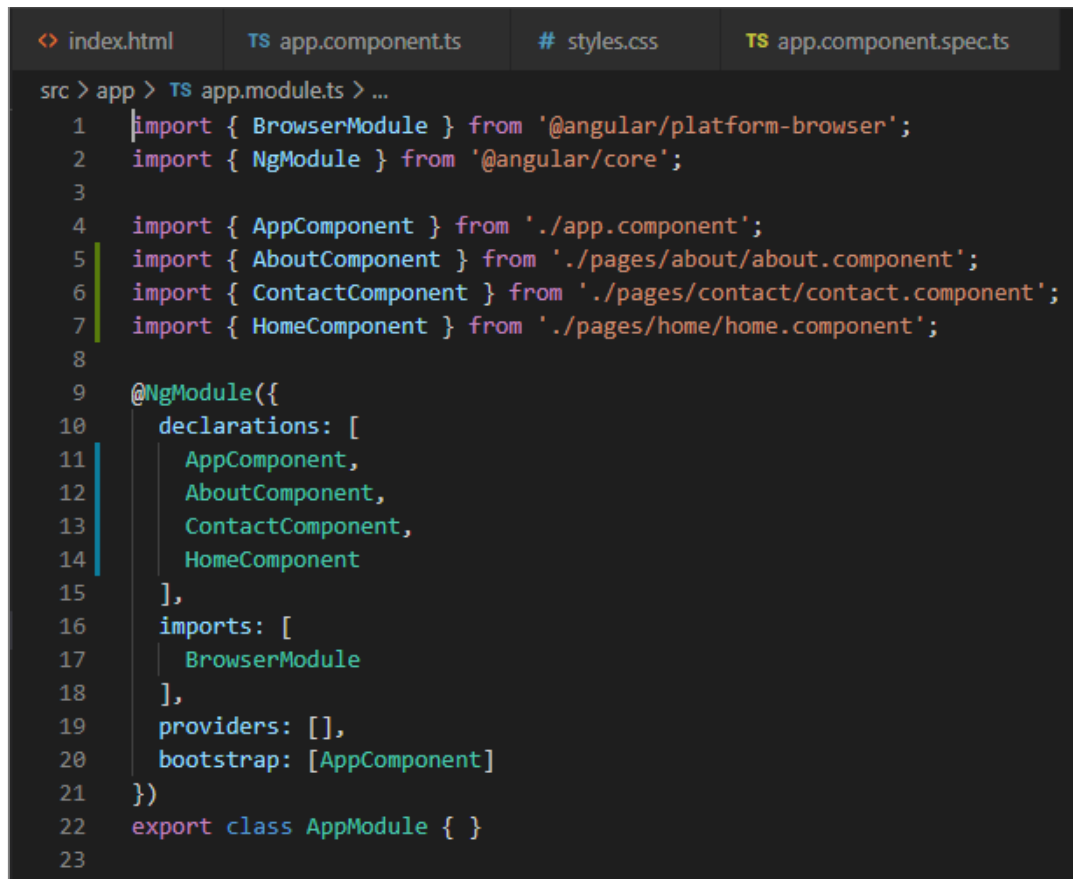


```
src > app > TS app.module.ts
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4  import { AppComponent } from './app.component';
5
6  @NgModule({
7    declarations: [
8      AppComponent,
9    ],
10   imports: [
11     BrowserModule
12   ],
13   providers: [],
14   bootstrap: [AppComponent]
15 })
16 export class AppModule { }
17
18
```

Si abrimos el del otro ejemplo, deberíamos tener declarados los otros tres componentes que añadimos: `about`, `home`, `contact`

El módulo App también es el **módulo raíz** porque de él surgen las demás ramas que forman una aplicación.

La asignación de los nodos hijos se realiza en la propiedad **imports**, indicamos que nuestro nuevo módulo será importado por el raíz. Así iremos construyendo el árbol de módulos



```
src > app > TS app.module.ts > ...
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4  import { AppComponent } from './app.component';
5  import { AboutComponent } from './pages/about/about.component';
6  import { ContactComponent } from './pages/contact/contact.component';
7  import { HomeComponent } from './pages/home/home.component';
8
9  @NgModule({
10   declarations: [
11     AppComponent,
12     AboutComponent,
13     ContactComponent,
14     HomeComponent
15   ],
16   imports: [
17     BrowserModule
18   ],
19   providers: [],
20   bootstrap: [AppComponent]
21 })
22 export class AppModule { }
23
```

El módulo raíz, al igual que el componente raíz, es especial. Su nombre oficial es App, aunque la documentación se refiere a él como raíz o root.

Crear un módulo

Para crear un módulo usaremos otro comando del *cli*

ng generate module.

En una ventana del terminal escribe: `ng g m "nombre"`

El resultado es la creación del fichero nombre.module.ts con la declaración y decoración del módulo. Este módulo nos sirve de **contenedor para guardar componentes** y otros servicios esenciales para nuestra aplicación.

En nuestro caso como la carpeta pages ya existe `ng g m pages`

Importar un módulo

La importación la realiza automáticamente angular, aunque conviene comprobar que la línea de código se ha añadido en el archivo app.module. Si no es así debemos añadirla nosotros

```
import {nombre} from 'ruta relativa';
```

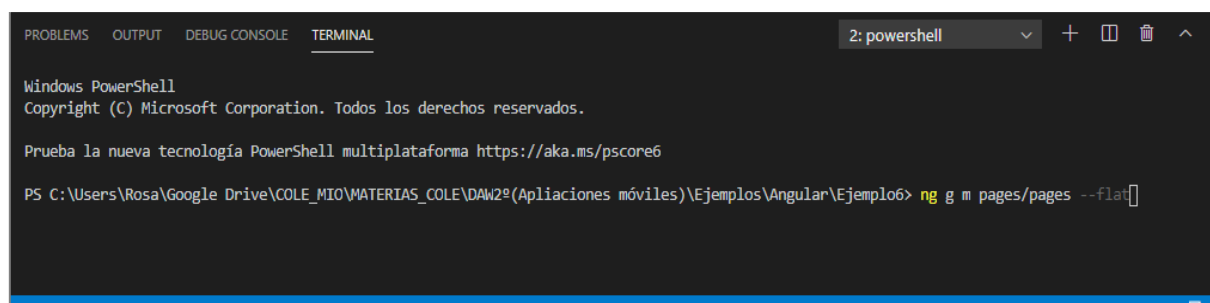
Si abrimos el archivo app.module.ts, observaremos que en el decorador aparecen todos los componentes que hemos creado. Si en nuestra aplicación tuviéramos muchos, serían muchas las líneas que aparecerían ahí.

```
import { AppComponent } from './app.component';
import { HomeComponent } from './pages/home/home.component';
import { ContactComponent } from './pages/contact/contact.component';
import { AboutComponent } from './pages/about/about.component';
import { AppRoutingModule } from './app-routing.module';
import { MenuComponent } from './mycomponents/menu/menu.component';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ContactComponent,
    AboutComponent,
    MenuComponent,
```

Vamos a agruparlas en otro módulo que crearemos para tal fin.

En la carpeta pages, vamos a crear un módulo para estas declaraciones



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\Rosa\Google Drive\COLE_MIO\MATERIAS_COLE\DAW2º(Aplicaciones móviles)\Ejemplos\Angular\Ejemplo6> ng g m pages/pages --flat
```

Borramos las importaciones y declaraciones de los componentes páginas en mi app.module.

En nuestro nuevo módulo PagesModule, añadimos las declaraciones de los tres componentes. Al añadir la declaración, el framework añade automáticamente las instrucciones de importación. Aun así, es conveniente asegurarse de que han sido añadidas.

```
import { HomeComponent } from './home/home.component';
import { AboutComponent } from '../../src/app/pages/about/about.component';
import { ContactComponent } from './contact/contact.component';

@NgModule({
  declarations: [
    HomeComponent,
    AboutComponent,
    ContactComponent
  ],
  imports: [
    CommonModule
  ]
})
export class PagesModule { }
```

Estos componentes pueden utilizarse internamente en los lugares dónde utilice este módulo, para que podamos usar los componentes desde cualquier punto de mi aplicación debemos exportarlos

```
8
9  @NgModule({
10    declarations: [
11      HomeComponent,
12      AboutComponent,
13      ContactComponent
14    ],
15    imports: [
16      CommonModule
17    ],
18    exports: [
19      HomeComponent,
20      AboutComponent,
21      ContactComponent
22    ]
23  })
```

Por último, necesito decirle a mi AppComponent que usará este módulo, para lo que añado el PagesModule en los imports del AppModule

De esta manera, al crear nuevas páginas no tendré que tocar el app.module.

Recorrer un array de datos con ngFor

Una de las directivas más útiles y recurrentes en Angular es ngFor, que permite recorrer una estructura de Array.

En nuestro componente creamos un array, o accedemos a él mediante un origen de datos y luego lo tendremos disponible en el template para recorrer.

```
export class HomePage {  
  test:string = "Bienvenido a Ionic";  
  
  arraySitios:any[] = [  
    {  
      nombre: 'DesarrolloWeb.com',  
      url: 'https://desarrolloweb.com'  
    },  
    {  
      nombre: 'EscuelaIT',  
      url: 'https://escuela.it'  
    },  
  ]  
  constructor(public navCtrl: NavController) {}  
}
```

Ahora en la vista podremos recorrer este array de la siguiente forma:

```
<ion-list>  
  <ion-item *ngFor="let sitio of arraySitios">  
    <a href="{{sitio.url}}">{{sitio.nombre}}</a>  
  </ion-item>  
</ion-list>
```


Rutas

Es raro que una aplicación web exponga toda su funcionalidad e información en una única vista. Lo normal es que se carguen varias páginas en diferentes direcciones.

Antes de las aplicaciones SPA, la única opción era que el servidor procesase dicha ruta y enviase el contenido a visualizar en el navegador. A cada ruta le corresponde un html o un script que lo genera.

Esto significa mucho trabajo para el servidor y poca responsabilidad para los navegadores. Actualmente, se intenta distribuir esa carga y que sea el navegador el que prepare la vista ejecutando instrucciones.

Por tanto, desde el front se procesan las rutas para determinar cuál será la vista (componente, página) que se deba mostrar en cada dirección.

El concepto SPA que sigue angular se basa en que solo tenemos una página html, index.html, toda la acción se desarrolla dentro de esa página. En Angular lo común es que el index sólo tenga un componente en su BODY y realmente toda la acción se desarrollará en ese componente. Todas las "páginas" (pantallas o vistas) del sitio web se mostrarán sobre ese index, intercambiando el componente que se esté visualizando en cada momento.

Para facilitar la navegación por un sitio donde realmente sólo hay un index, existe lo que llamamos el sistema de routing, encargado de reconocer cuál es la ruta que el usuario quiere mostrar, presentando la pantalla correcta en cada momento.

El sistema de routing de angular es bastante sofisticado, envuelve a muchas clases, interfaces, configuraciones...se basa en los siguientes elementos:

- **RouterModule:** módulo del sistema para las rutas, contiene el código de angular para que podamos enrutar, expone un par de métodos de configuración

```
.forRoot(routes:Routes)  
.forChild(routes:Routes)
```

Ambos reciben una estructura que mantiene un array de rutas y las instrucciones a ejecutar cuando dichas rutas se activen. Las rutas pueden ser estáticas o usar comodines. Las acciones pueden ser de elección de componente para la vista, pasar el trabajo a otro módulo o redirigir al usuario a otra ruta.

- **Rutas de mi aplicación:** declaramos nuestras rutas en un array, el tipo de datos del array y su formato vienen determinados por angular.

miarray : Routes []

Routes es un tipo de datos de angular, que corresponde a un array de objetos Route

```
type Routes = Route[];
```

{ path: 'url', component: nombre de la clase del componente }

La declaración completa

```
myarray: Routes[  
  {path: '...', component },  
  .....  
]
```

Route es una **interfaz**, no es una clase

```
interface Route {  
  path?: string  
  pathMatch?: string  
  matcher?: UrlMatcher  
  component?: Type<any>  
  redirectTo?: string  
  outlet?: string  
  canActivate?: any[]  
  canActivateChild?: any[]  
  canDeactivate?: any[]  
  canLoad?: any[]  
  data?: Data  
  resolve?: ResolveData  
  children?: Routes  
  loadChildren?: LoadChildren  
  runGuardsAndResolvers?: RunGuardsAndResolvers  
}
```

- **Enlaces:** nuestros enlaces HTML en los que incluimos la directiva (una directiva es una extensión de HTML propia de angular) adecuada para indicar que estamos enrutando.

Veamos cómo crear un manejador de rutas para mi aplicación

Ejemplo_5: "Sobre el Ejemplo_4 generamos rutas para que home, about, contact se redirijan a sus páginas correspondientes"

1. Al crear el proyecto, indicamos que queremos routing, angular genera un módulo llamado `appRoutingModule`

2. Escribimos nuestras rutas en el módulo creado

Utilizamos el tipo de dato `Routes`, se encuentra en `@angular/router`, lo importamos para poderlo usar y declarar nuestras rutas.

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

Para definir una ruta tenemos que declarar el path (lo que se verá en nuestra URL) y el componente que cargaremos cuándo nos escriban la URL indicada en el path

```
const routes: Routes = [  
  {  
    path: 'home',  
    component: HomeComponent  
  },  
]
```

Si no hemos importado nuestro componente, nos dirá que no puede cargarlo

```
import {HomeComponent} from './pages/home/home.component';  
import { ContactComponent } from './pages/contact/contact.component';  
import { AboutComponent } from './pages/about/about.component';  
|
```

Para cada ruta, asociada a cada página o componente tenemos que definir esos dos parámetros, se suele añadir una ruta más, para indicar que cuándo el usuario nos escribe un URL que pueda contener cualquier cosa se dirija al Home

```
const routes: Routes = [  
  {  
    path: 'home',  
    component: HomeComponent  
  },  
  {  
    path: 'contact',  
    component: ContactComponent  
  },  
  {  
    path: 'about',  
    component: AboutComponent  
  },  
  {  
    path: '**',  
    redirectTo: 'home'  
  }  
];
```

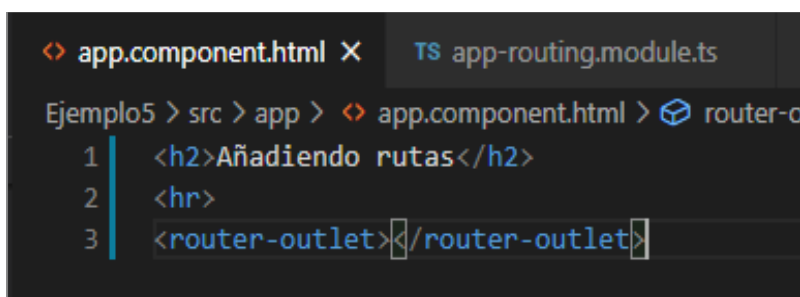
Angular sabe que tiene que usar este módulo porque en su decorador importa la constante que he definido y exporta este módulo para que el módulo principal de la aplicación pueda a su vez importarlo. También lo importa de forma automática en el app.module

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

4. Indico dónde quiero que se muestren mis componentes. Tengo que enlazar las rutas que he creado con el html de mi componente raíz (app).

<router-outlet>

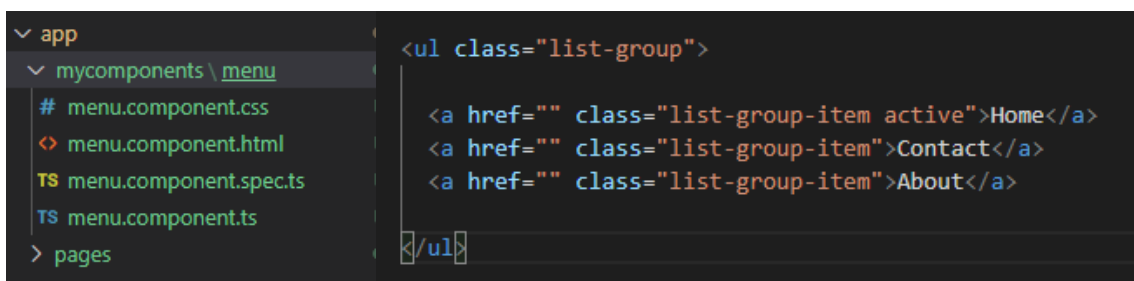
Este elemento de angular inyectará dinámicamente el componente que corresponda según la ruta activa en mi URL, que estará "redirigida" por mi array de rutas.



```
<> app.component.html X TS app-routing.module.ts
Ejemplo5 > src > app > <> app.component.html > router-outlet
1 <h2>Añadiendo rutas</h2>
2 <hr>
3 <router-outlet></router-outlet>
```

Si probamos nuestro ejemplo con href, veremos que desde la URL podemos redirigir a cada componente. Hay que observar que la página se refresca, lo que supone una llamada al servidor para cargarla de nuevo.

Vamos a aplicar estas rutas para poder navegar en mi app no sólo a través de la URL



```
app
└─ mycomponents \ menu
   # menu.component.css
   <> menu.component.html
   TS menu.component.spec.ts
   TS menu.component.ts
  > pages
  <ul class="list-group">
    <a href="" class="list-group-item active">Home</a>
    <a href="" class="list-group-item">Contact</a>
    <a href="" class="list-group-item">About</a>
  </ul>
```

Si probamos el ejemplo, observamos que cualquiera de los enlaces nos redirige a Home porque href="" y esa URL encaja en nuestro patrón de "cualquier cosa que no sea home,contact,about".

Además, es una etiqueta `<a href>` lo que implica llamada a servidor para cargar la página. Veamos cómo solucionar esto

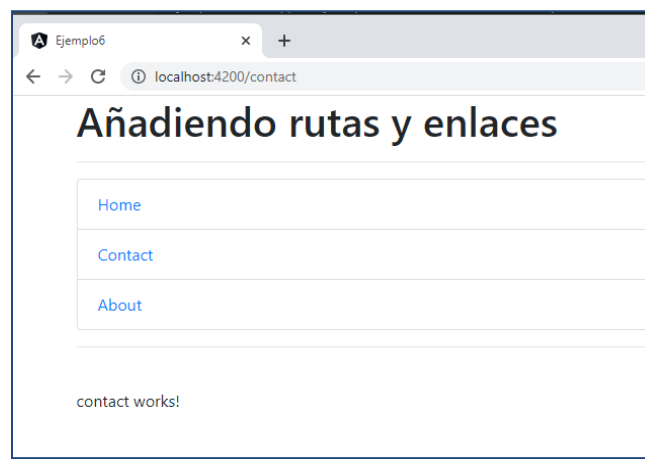
5. Creo el menú para que funcione en front

Al utilizar el RouterModule, disponemos de una directiva `routerLink` para no tener que usar `href`

```
<ul class="list-group">
  <a routerLink="/home" class="list-group-item">Home</a>
  <a routerLink="/contact" class="list-group-item">Contact</a>
  <a routerLink="/about" class="list-group-item">About</a>
</ul>
```

Las directivas de angular se emplean como si fuesen un atributo de cualquier elemento y durante la compilación genera el código estándar necesario para que lo entiendan los navegadores. En concreto esta directiva del RouterModule, se usa en sustitución del atributo estándar `href`, instruye al navegador para que no solicite la ruta al servidor, sino que el propio código local de JavaScript se encargará de procesarla.

El resultado de nuestro ejemplo, con los enlaces funcionando sin recargar página y por tanto sin llamar al servidor



Podemos automatizar un poco más el proceso, si observamos el componente menú, estamos repitiendo código porque escribimos nuestras opciones de la lista tantas veces como cantidad de ellas tenemos. Esto no parece muy dinámico y desde luego no es la mejor forma de "programar"

6. Automatizamos un poquito más la creación de la lista de enlaces

En el componente menú declaramos una propiedad que será un array de objetos con las opciones de la lista, de manera que luego podremos acceder a esa propiedad para recorrer el array y que se generen dinámicamente los enlaces de esta. Puedo definir una interfaz con el tipo de los ítems del menú.

```
//
export class MenuComponent implements OnInit {

  rutas=[
    {
      name: 'Home',
      path: '/home'
    },
    {
      name: 'Contact',
      path: '/contact'
    },
    {
      name: 'About',
      path: '/about'
    }
  ]

  constructor() { }
```

rutas es un array objetos, en el que cada name será la opción que aparezca en mi lista y cada path el valor asociado a routerLink

Recorremos el array

```
<ul class="list-group">
  <a routerLink="{{ruta.path}}" class="list-group-item" *ngFor="let ruta of rutas">
    {{ruta.name}}
  </a>
</ul>
```

Para asignar ruta.name hay otra posibilidad que se utiliza más

```
<a [routerLink]='ruta.path' class="list-group-item" *ngFor="let ruta of rutas">
  {{ruta.name}}
</a>
```

Aun así, esta no es la mejor solución porque todas las páginas que cree nuevas tendré que añadirlas al PagesModule y además cuándo ponga en producción mi aplicación, todos los archivos estarán compactados en un único archivo.

Veremos luego cómo hacer para cargar los componentes de manera "perezosa", que es cómo realmente se trabaja en Ionic.

Rutas parametrizadas

Ahora veremos como trabajar en Angular si una ruta tiene uno o más parámetros. Un ejemplo de ruta con parámetros podría ser pasar el número de artículo en la ruta:

```
http://localhost:4200/articulos/323
```

Estamos pasando a la ruta 'articulos' el parámetro '323'. Como vimos antes es obligatorio definir exactamente las rutas que puede procesar la aplicación Angular. Veamos como las declaramos, como pasamos los parámetros y como los recuperamos.

1. Definir la ruta

path: 'ruta/:parametro'

```
// src/app/app.module.ts
const routes: Route[] = [
  { path: "home", component: HomeComponent },
  { path: "posts", component: PostsComponent },
  { path: "posts/:id", component: PostComponent },
  { path: "**", redirectTo: "home" }
];
```


2. Enlazar con nuestra aplicación

Para navegar hacia el nuevo componente desde nuestra aplicación haremos uso de la directiva `routerLink` y agregaremos valores de parámetros

```
<p>Mis post</p>
<ul>
  <li><a [routerLink]="['/Post',1]"> POST 1</a></li>
  <li><a [routerLink]="['/Post',2]"> POST 2</a></li>
  <li><a [routerLink]="['/Post',3]"> POST 3</a></li>
</ul>
```

Si utilizo otra notación

`routerLink="/Post/1"`

`routerLink="{{item.path}}/{{item.id}}"`

`[routerLink]="[item.path, item.id]"`

3. Obtener el valor del parámetro

Como hablamos al principio, es posible que queramos obtener cierta información utilizando el valor del parámetro `id`. Hay dos maneras diferentes de obtener el valor del parámetro.

- La primera es a través del *snapshot* de la ruta. El snapshot de la ruta nos provee de un objeto llamado [paramMap](#) que expone los métodos `get`, `getAll` y `has` para interactuar con los parámetros de la ruta actual.

Para acceder al snapshot de la ruta es necesario inyectar en el componente la clase [ActivatedRoute](#) como se muestra a continuación:

```
constructor(private router:ActivatedRoute) { }

ngOnInit(): void {
  this.id=this.router.snapshot.paramMap.get("id");
}
```

Esta alternativa para obtener el valor del identificador tiene un problema, si la navegación se produce dentro del propio Componente, éste no detecta el cambio en el paramMap por lo que no realiza la navegación.

b) Suscribirse a los cambios en paramMap

Para solucionar el inconveniente antes mencionado debemos reemplazar el uso del snapshot de la ruta por una suscripción al parámetro paramMap del ActivatedRoute inyectado. De esta forma estaremos observando los cambios que suceden sobre los parámetros de la ruta y reaccionando correctamente ante cada uno de ellos.

El componente pasaría verse de la siguiente forma:

```
// src/app/post/post.component.ts
import { Component, OnInit } from "@angular/core";
import { ActivatedRoute, ParamMap } from "@angular/router";

@Component({
  selector: "app-post",
  templateUrl: "./post.component.html",
  styleUrls: ["./post.component.css"]
})
export class PostComponent implements OnInit {
  id: string;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.route.paramMap.subscribe((params: ParamMap) => {
      this.id = params.get('id');
    });
  }
}
```

Lo más importante a recordar a la hora de decidir entre cuáles de los métodos utilizar es saber en dónde sucede el cambio del parámetro.

Si el valor del parámetro es modificado dentro del mismo componente que lo utiliza entonces debemos suscribirnos al paramMap.

En caso de que este no sea alterado dentro del componente podremos utilizar el snapshot de la ruta sin inconvenientes.

Método ngOnInit

Angular, como otros frameworks Javascript puede ejecutar código cuando el componente carga por primera vez, es el método ngOnInit.

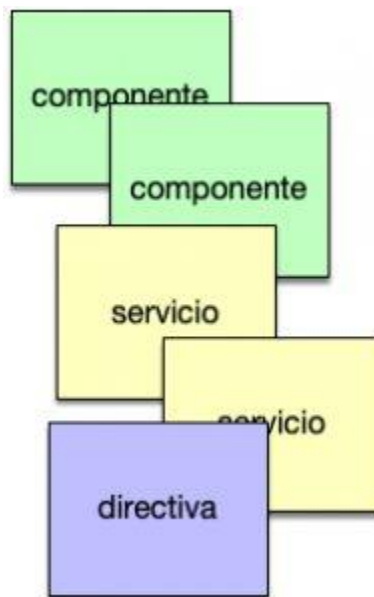
Para que un componente de Angular pueda ejecutar el ngOnInit tienes que importar OnInit de Angular y además tienes que hacer que el componente implemente el ngOnInit:

typescript también tiene un constructor de clase, en este caso el constructor se ejecuta antes que el ngOnInit().

Normalmente se usa el constructor para inicializar variables, y el ngOnInit para inicializar o ejecutar tareas que tienen que ver con Angular. Todo esto lo podemos poner directamente en el constructor y funcionaría de la misma manera, pero no está de más tener más separado el código para que sea más mantenible.

LazyLoad

El concepto de **Angular Lazy Loading Modules** es un concepto importante a nivel de programación en el mundo de las arquitecturas SPA. Cuando nosotros construimos una aplicación en Angular esta aplicación está compuesta por un montón de componentes, servicios, directivas etc.



Esto no es muy problemático para una aplicación de tamaño pequeño o mediano, pero según la aplicación crece hay que cargar en el navegador mucho código de JavaScript de golpe y los tiempos de carga se pueden ver afectados.

Para solventar este problema el primer enfoque es organizar nuestros componentes y servicios en módulos de tal forma que estos se puedan cargar de forma "vaga" en un futuro y no en el momento de arranque de la aplicación.

Es decir, al arrancar la aplicación cargamos aquellos componentes que son necesarios en un principio y el resto se cargan según se necesitan.

Ejemplo7: "Crear tres componentes que serán cargados usando lazy load"

Creamos un proyecto nuevo con la opción de routing para que se genere automáticamente el AppRoutingModule

ng new Ejemplo7 --routing

Creamos tres módulos con sus correspondientes componentes y rutas (Inicio, Blog, Contacto)

ng g m inicio --routing

No añadido el parámetro --flat porque como no he creado el componente todavía sí quiero que me cree la carpeta inicio

ng g c inicio

Repito el proceso para los otros dos módulos y sus componentes

En cada módulo de mis componentes, incluyo sus rutas

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { InicioModule } from './inicio.module';
import { InicioComponent } from './inicio.component';

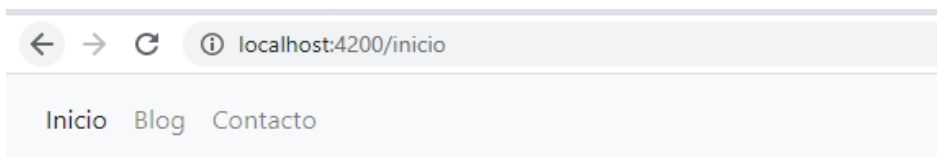
const routes: Routes = [
  {
    path: '',
    component: InicioComponent
  }
];










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

En el AppRoutingModuleModule indico las rutas de estos tres componentes

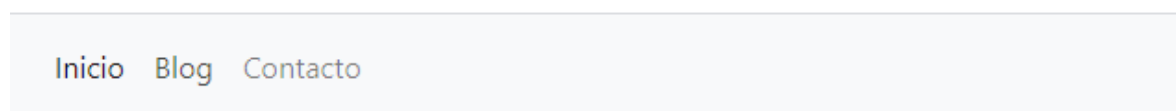
```
const routes: Routes = [  
  {  
    path: 'inicio',  
    loadChildren: () => import('./inicio/inicio.module').then(m=>m.InicioModule)  
  },  
  {  
    path: 'blog',  
    loadChildren: ()=>import('./blog/blog.module').then(m=>m.BlogModule)  
  },  
  {  
    path: 'contacto',  
    loadChildren: ()=>import('./contacto/contacto.module').then(m=>ContactoModule)  
  }  
];
```

Para probar nuestro ejemplo, pondremos tres enlaces en nuestro AppComponent y ejecutaremos la aplicación. Podremos observar que los componentes se irán cargando según pulsemos los enlaces. En los archivos vendor y main no aparecerá de inicio ninguna referencia a dichos componentes, puesto que sólo se cargará lo que no carguemos con lazy load (en este ejemplo todos). En muchas aplicaciones sólo se carga de inicio el componente principal con aquello que mostraremos de inicio.













Name	Status	Type	Initiator	Size	Time
 websocket	101	websoc...	sockjs.js:1684	0 B	Pending
 localhost	304	docum...	Other	210 B	325 ms
 bootstrap.min.css	200	stylesh...	(index)	(disk ca...	93 ms
 runtime.js	304	script	(index)	211 B	101 ms
 polyfills.js	304	script	(index)	212 B	91 ms
 styles.js	304	script	(index)	211 B	118 ms
 vendor.js	304	script	(index)	213 B	189 ms
 main.js	304	script	(index)	211 B	207 ms
 info?t=1602493576213	200	xhr	zone-evergree...	367 B	17 ms

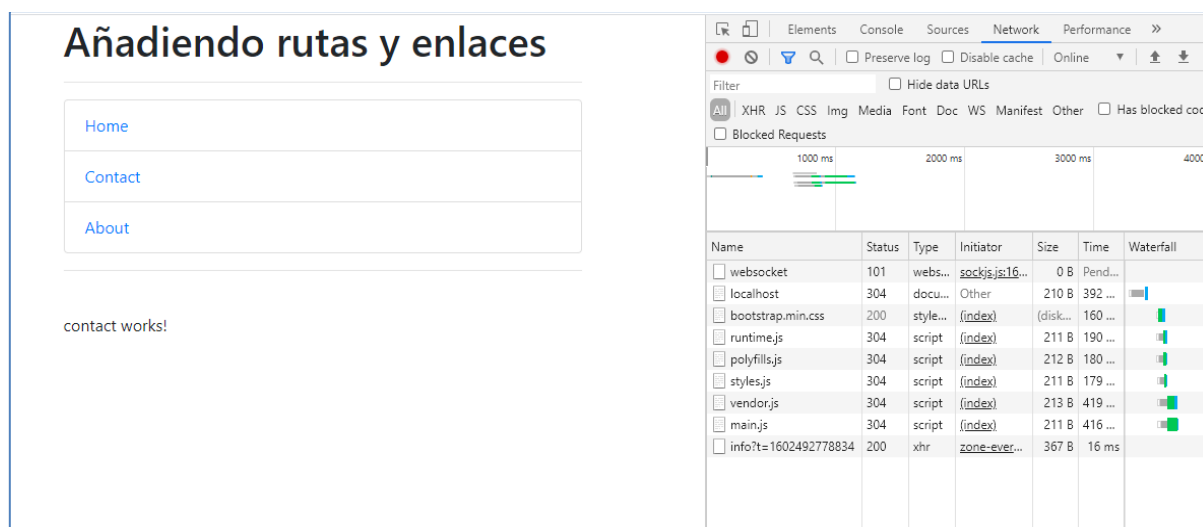
Si elegimos una opción del menú



blog works!

Name	Status	Type	Initiator	Size	Time
 localhost	304	document	Other	210 B	39 ms
 bootstrap.min.css	200	stylesheet	(index)	(disk cache)	80 ms
 runtime.js	304	script	(index)	211 B	83 ms
 polyfills.js	304	script	(index)	212 B	128 ms
 styles.js	304	script	(index)	211 B	118 ms
 vendor.js	304	script	(index)	213 B	227 ms
 main.js	304	script	(index)	211 B	246 ms
 info?t=1602493271088	200	xhr	zone-evergreen.js:2845	368 B	55 ms
 websocket	101	websocket	sockjs.js:1684	0 B	Pending
 blog-blog-module.js	304	script	bootstrap:149	211 B	23 ms

En el Ejemplo6, toda la aplicación se carga de inicio, los componentes están compactados en los archivos vendor, main sobre todo. Al navegar por los enlaces, podremos apreciar que no se carga nada más



Aquellos componentes que queramos cargar al iniciar la aplicación, los "enrutaremos" en el AppRoutingModuleModule tal y como vimos en el Ejemplo6, al cargar la aplicación en el navegador se cargan en el main.js. Aquellos componentes que queremos cargar según sean solicitadas sus URL, los "enrutaremos" tal y como hemos visto en el Ejemplo7. En angular hay una tercera posibilidad para cuándo los módulos son muy pesados, se pueden ir cargando de forma asíncrona mientras la aplicación sigue "funcionando", de esta manera cuándo se solicite su URL el tiempo de carga se reduce.

Cada módulo puede contener varios componentes, no siempre es buena práctica crear un módulo para cada componente tal y como hemos hecho en el Ejemplo7, en la vida real, es frecuente tener varios componentes "agrupados" en un mismo módulo.

Los servicios

Una pieza fundamental en la mayoría de las webs y app es la capa de los datos. El acceso a datos se puede hacer desde los propios componentes, pero no es lo más adecuado.

La organización del código, manteniendo piezas pequeñas de responsabilidad reducida, es siempre muy positiva. Además, es muy frecuente que dos o más componentes tengan que acceder a los mismos datos y hacer operaciones similares con ellos, lo que podría obligarnos a repetir código. Para solucionar estas situaciones tenemos los **servicios**.

Un servicio es un proveedor de datos, que mantiene lógica de acceso a ellos y operativa relacionada con el negocio y las cosas que se hacen con los datos dentro de una aplicación. Los servicios serán consumidos por los componentes, que delegarán en ellos la responsabilidad de acceder a la información y la realización de operaciones con los datos.

Este acceso a los datos puede ser de cualquier tipo: datos estáticos, datos de una API o aplicación backend, etc. Si en algún momento hay que cambiar el lugar de almacenamiento de los datos, será muy importante tener una estructura con el acceso a los datos separados.

Crear un servicio

Usaremos el comando `ng` de AngularCLI

ng g s nombre

La coetilla "Service", al final del nombre, te la agrega Angular CLI, así como también nombra al archivo generado con la finalización "-service", para dejar bien claro que es un servicio.

Lo normal será colocar el servicio en el módulo adecuado, según los componentes que se provean de él. Para ello indicas la ruta al generar el servicio (como en el caso de los módulos o componentes). Si varios componentes de distintos módulos acceden a los mismos servicios, se suele crear un módulo aparte para estos servicios.

Ejemplo8: "Generar un componente mensajes y su correspondiente servicio".

1. Creamos el proyecto con el parámetro

```
ng new Ejemplo8
```

2. Vamos a tener una carpeta messages donde irán nuestro módulo, nuestro componente y nuestro servicio

```
ng g m messages
```

No añadido el parámetro --flat pq no he creado el componente y por tanto quiero que cree la carpeta

```
ng g c messages
```

```
ng g s messages/messages
```

3. Hago los cambios necesarios en el app-module y en messages-module

Con las versiones anteriores de angular, era necesario importar el servicio en el app-module. Ahora un servicio en angular incluye

```
3  @Injectable({  
4    providedIn: 'root'  
5  })
```

Lo que hace esa instrucción es que el servicio se "autoimporte" y no haya que importarlo en el AppModule (esto es nuevo en angular, en versiones anteriores era necesario). En general, lo dejaremos así para que los servicios se puedan utilizar de forma global en toda la aplicación.

Si queremos especificar nosotros que un servicio se usará en un determinado componente de un módulo, añadiremos el servicio en la parte providers de dicho módulo. Todos los componentes de ese módulo podrán acceder a ese servicio

```
@NgModule({
  declarations: [MessagesComponent],
  imports: [
    CommonModule
  ],
  exports: [MessagesComponent],
  providers: [MessagesService]
})
export class MessagesModule { }
```

Esto significa que este módulo ha declarado el MessagesService como proveedor, todos los componentes de este módulo podrán acceder a este servicio.

Podemos declarar un servicio como proveedor en un componente, eso significa que este servicio está "asignado" sólo a dicho componente y no a todos los que estén en su módulo.

Para nuestro ejemplo, lo dejamos como proveedor global

4. Para utilizar el servicio en nuestro componente, tenemos que pasarle al constructor del componente una variable del tipo del servicio

```
10 export class MessagesComponent implements OnInit {
11
12   constructor(private ms: MessagesService) { }
13
14   ngOnInit(): void {
15   }
16
17 }
18
```

Estamos indicando a TypeScript y Angular que vamos a usar un objeto ms que es de la clase MessagesService. A partir de entonces, dentro del componente existirá ese objeto, proporcionando todos los datos y funcionalidad definida en el servicio.

Para que angular realice este proceso, es imprescindible declarar ámbito de visibilidad de la variable de tipo servicio.

- private: podrás usar el servicio sólo desde el código de la clase del componente
- public: podrás usar el servicio desde el template directamente

Cuando TypeScript detecta el modificador de visibilidad "public" o "private" en el parámetro enviado al constructor, declara una propiedad en la clase y le asigna el valor recibido en el constructor. Por tanto, esa declaración sería equivalente a

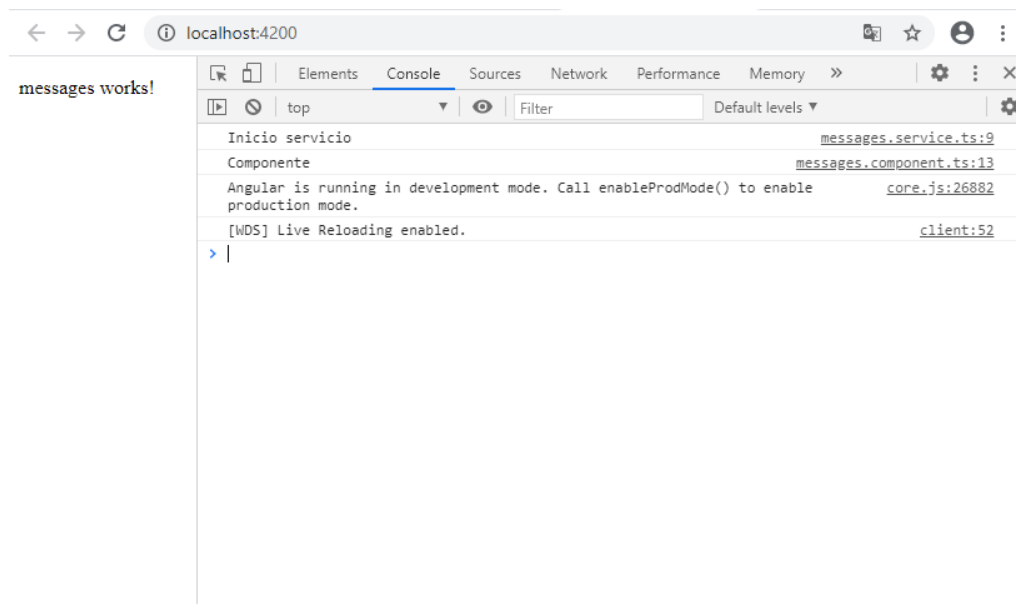
```
export class MessagesComponent {  
  ms: MessagesService;  
  constructor(ms: MessagesService) {  
    this.ms = ms;  
  }  
}
```

Código POO clásico

Comprobemos que el servicio se carga: añadimos mensajes a nuestra consola

```
//  
export class MessagesComponent implements OnInit {  
  constructor(private ms: MessagesService) {  
    console.log("Componente");  
  }  
  ngOnInit(): void {  
  }  
}  
  
6 export class MessagesService {  
7  
8   constructor() {  
9     console.log("Inicio servicio");  
10  }  
11 }  
12
```

Al inspeccionar nuestra aplicación



5. Añadimos código al servicio

Vamos a añadir una propiedad que contenga alguna url válida, desde el componente accederemos a dicha propiedad. Podremos usar el servicio tanto desde la clase como desde el componente. El código que implementamos para ello varía en función de si el objeto declarado en el constructor de mi componente es privado o público.

```
<p>messages works!</p>
<a href={{enlace}}>IR</a>
```

Añadimos al servicio una variable que contiene la URL de la web del colegio

```
1 import { Injectable } from '@angular/core';
2
3 @Injectable({
4   providedIn: 'root'
5 })
6 export class MessagesService {
7
8   private accesoWeb='http://www.salesianasnsp.es';
9   constructor() {
10     console.log("Inicio servicio");
11   }
12   getAcceso() {
13     return this.accesoWeb;
14   }
15 }
16
```

En nuestro componente habíamos declarado el **objeto ms como privado**, lo que implica que el código en la clase del componente queda

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

  enlace: string;
  constructor(private ms: MessagesService) {
    //console.log("Componente");
    this.enlace=this.ms.getAcceso();
  }

  ngOnInit(): void {
  }
}
```

Si hubiéramos declarado el objeto como público, la clase componente quedaría

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

  //enlace: string;
  constructor(public ms: MessagesService) {
    //console.log("Componente");
    // this.enlace=this.ms.getAcceso();
  }

  ngOnInit(): void {

  }

}
```

El template del componente

```
<p>messages works!</p>
<a href={{this.ms.getAcceso()}}>IR</a>
```

Servicios y data externa

Las llamadas a la API, que utilizan el módulo HttpClient, son asíncronas por naturaleza, ya que deben esperar a que la respuesta provenga de los servidores remotos sin bloquear la aplicación la API y también la respuesta HTTP, por lo que debe ejecutarse en segundo plano antes de que los datos puedan estar listos para ser consumidos.

Con Ionic 5 / Angular puede hacer uso de las API de JavaScript modernas: Promesas y Observables que garantizan abstracciones de alto nivel para manejar la naturaleza asíncrona de las operaciones de obtención de datos y consumo de API o cualquier otra operación que tarde en finalizar.

Promesa: El objeto Promise representa la eventual finalización (o falla) de una operación asincrónica y su valor resultante

Una promesa puede ser:

- pendiente: el estado de espera inicial antes de un eventual cumplimiento o rechazo.
- cumplida: la operación se ha completado con éxito con un valor.
- la operación ha fallado rechaza con un error.

Se colocan las acciones asíncronas, ya sea cuando la promesa se ha resuelto con éxito o ha fallado, dentro de los **métodos**. **then (() => {})** y **.catch (() => {})**.

Observable: Al igual que las promesas, los observables son abstracciones que nos ayudan a lidiar con las operaciones asíncronas, excepto que las manejan de una manera diferente y brindan más funciones, por lo que se vuelven preferibles a las promesas entre la comunidad JavaScript / Angular.

A diferencia de las promesas, que solo pueden manejar eventos individuales, los observables pueden pasar más de un evento. Un Observable se puede representar como un flujo de eventos que se pueden manejar con la misma API y se pueden cancelar.

En la URL **<https://jsonplaceholder.typicode.com/>** podemos encontrar varios servicios para practicar (los datos almacenados están en formato json).

Nuestra aplicación angular consumirá los datos ahí almacenados, para ello necesitamos un servicio dónde implementar el código de acceso a dichos datos.

Nuestro servicio realizará una petición http a dicha URL, para ello necesitamos un módulo de angular HttpClientModule

```
import {HttpClientModule} from '@angular/common/http'
```

Este módulo tiene todo lo necesario para que nuestro servicio angular realice peticiones http al servicio publicado en esa URL.

En el constructor de nuestro servicio pasamos como parámetro una variable del tipo HttpClient. Dicha clase HttpClient, proporciona un método get al que se le pasa la url que queremos acceder. Implementamos los métodos que usarán ese método para acceder al servicio publicado en la URL o bien a la ruta estática donde tengamos el json.

```
Private http: HttpClient
```

```
http.get("url")
```

Este método nos retorna un array por lo que al recibirlo en el componente no podremos usar *ngFor ni tipificar las variables. Para que el componente transforme esos datos y espere a que el array esté completo usamos el pipe async. Este método cargará los datos una vez y no detectará cambios en los mismos en el origen de los datos. Esta no será una opción válida si sabemos que esos cambios podrían producirse, lo que será una situación más normal que la contraria.

El código sería la línea comentada y en el html

```
<ul class="list-group">
  <li class="list-group-item" *ngFor="let m of ms | async">
    <h4>{{m.title}}</h4>
    <p>{{m.body}}</p>
  </li>
  <hr>
</ul>
```

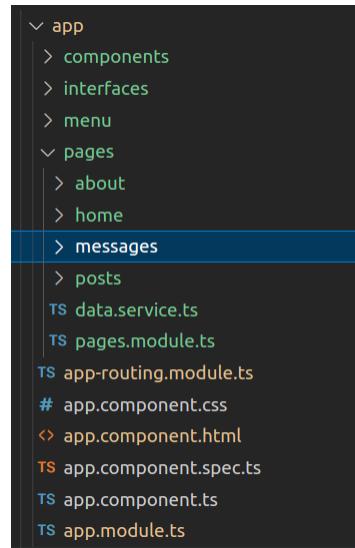
A partir de ahí, implementamos en nuestros componentes el uso de esos datos a través de nuestro servicio.

Esta no es la mejor manera de trabajar, estamos haciendo uso del tipo any para no lidiar con lo que el servicio retorna y por otro lado si los datos los cogemos de una API, es probable que haya cambios y no los veríamos. En estos casos, debemos utilizar el método **subscribe**

El método subscribe puede aplicarse al método get en el propio servicio, pero es muy posible que debido a la asincronía antes de terminar de cargar todos los datos el componente intente acceder a ellos y no los tendrá. Para solventar este problema, retornamos los datos con el método get del servicio y usamos subscribe en el componente. Aun así, podría ocurrir que antes de que el método subscribe termine la carga el html quiera pintar esos datos, para “esperar” a que le lleguen usamos *ngFor para pintar arrays y *ngIf para determinar, en el caso de que sea un solo datos, si ya podemos pintar esa variable.

Ejemplo_9: Nuestra aplicación cargará esos datos desde la URL a un servicio, estos datos serán consumidos por nuestros componentes

- Componente menú con las opciones Inicio, Mensajes, Post, AcercaDe.



- Las opciones serán componentes de un mismo módulo Pages. En este módulo llevaremos un servicio que nos proporcionará tanto los posts como los mensajes.
- Tendremos dos componentes para cabecera y pie.

```
export class MensajesService {  
  listaMensajes:IMensaje[];  
  constructor(private http: HttpClient) {  
    this.listaMensajes=[];  
  }  
  getMensajes(): IMensaje[] {  
    this.http.get<IMensaje[]>("../assets/data/mensajes.json").subscribe(  
      l=>{ l.forEach(e=>this.listaMensajes.push(e))}  
    )  
    return this.listaMensajes;  
  }  
}
```

Utilizaremos los componentes header y footer que te proporciona ya implementados



1. Creamos un servicio llamado data en pages
2. Importamos el HttpClientModule en nuestro PageModule
3. Añadimos providers en PageModule
4. Implementamos el servicio y definimos las interfaces que se corresponden con los datos que nos retorna ese API
5. Para utilizar esta información en mi componente, declaro una variable en el constructor del mismo tipo del servicio, con esta variable puedo acceder al método getData del servicio que a su vez me retorna la información publicada en esa URL. **Debo tener en cuenta el formato de esos datos.**
6. De la misma manera implementamos el componente post

```
<table class="table table-striped">
  <thead>
    <tr>
      <th scope="col">Id</th>
      <th scope="col">Title</th>
      <th scope="col">body</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let p of list_post | async">
      <td>{{p.id}}</td>
      <td>{{p.title}}</td>
      <td>{{p.body}}</td>
    </tr>
  </tbody>
</table>
```

Comunicación entre componentes

Hemos insistido en que una aplicación Angular es un árbol de componentes, que colaboran entre sí para resolver la lógica de negocio. Veamos como los componentes pueden intercambiar información a través de lo que se llama propiedades decoradas.

En el apartado anterior, hemos visto que un servicio puede proporcionar datos que a su vez pueden ser compartidos por componentes, no es la única posibilidad. Atendiendo a la estructura de árbol mencionada arriba, la comunicación puede realizarse desde los componentes padre a los hijos y viceversa

Enviar información desde un componente padre a un hijo

Podemos enviar propiedades desde un componente padre a un componente hijo, esto supone una división más en nuestra estructura y define que ciertas propiedades puedan asignarse desde el template del padre.

Para ello las propiedades del componente deben estar "decoradas" con @Input, así Angular es capaz de saber que serán inicializadas y/o modificadas desde fuera del componente.

Ejemplo_10: Partiendo del ejemplo 9, crearemos una componente hijo en el que irá el html necesario para mostrar cada mensaje, de esa manera tenemos la lógica que muestra cada componente de forma independiente y quizás podríamos usarla en varios sitios.

Crearemos un componente llamado mensaje, en la carpeta mensajes

```
ng g c pages/mensajes/mensaje
```

Nos llevamos al template del componente que hemos creado el código del

```
app > pages > mensajes > mensaje > <> mensaje.component.html > li.list-group-item
<li class="list-group-item" ">
  <h4 >{{m.title}}</h4> <p>{{m.body}}</p>
</li>
```

No incluimos la instrucción `*ngFor` puesto que este componente es para mostrar cada uno de los mensajes, la lógica del `*ngFor` debe ir, por tanto, en el componente mensajes que deberá pasar información de cada mensaje al componente que acabamos de crear. Veamos como lo hace

En el template de mensajes

```
app > pages > mensajes > <> mensajes.component.html > ul.list-group
<ul class="list-group" >
  |
  <app-mensaje *ngFor="let m of ms | async" ></app-mensaje>
</ul>
```

Para pasar la información desde el padre al hijo, importamos `Input` del `angular/core` y declaramos una propiedad decorada `@Input()` con el mismo nombre que tiene la propiedad del padre que queremos ver en el hijo

```
app > pages > mensajes > mensaje > TS mensaje.component.ts > MensajeComponent
import { Component, OnInit, Input } from '@angular/core';

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

  @Input() m;
  constructor() { }

  ngOnInit(): void {
  }
}
```

Observemos que dicha propiedad m tiene el mismo nombre que la variable del *ngFor en el template del padre, en el que debemos indicar que pasamos ese valor al hijo

```
<ul class="list-group" >  
  <app-mensaje *ngFor="let m of ms | async" [m]="m"></app-mensaje>  
</ul>
```

[m]="m" :

- la primera m es el nombre de la variable declarada @Input() m
- la segunda m es el nombre de la variable dentro del *ngFor

Emitir eventos desde el componente hijo al padre

La comunicación del hijo hacia el padre se lleva a cabo mediante eventos personalizados, generados con propiedades @Output:

El componente hijo será el encargado de avisar al padre de un suceso, y al hacerlo podrá comunicar datos que el padre deba conocer relacionados con dicho evento.

El componente padre será capaz de capturar el evento emitido por el hijo y recuperar la información que se le envía.

Veamos que partes de Angular intervienen en este proceso

Class EventEmitter

es la clase de Angular que implementa objetos capaces de emitir un evento. Pertenece al core de angular por lo que hay que importarla de esa librería

Decorador @Output

La propiedad del tipo EventEmitter necesarios para lanzar el evento, debe ser decorada con output. Esto le dice al framework que va a existir una vía de comunicación desde el hijo al padre. También está dentro del core.

Ejemplo_11:

Cada vez que nos hagan click en un mensaje, mostraremos una alerta con su id