

Angular

1	Introducción a Angular.....	4
1.1	Preparación del entorno de trabajo	4
1.1.1	El gestor de paquetes	4
1.1.2	El paquete de soporte de línea de comandos para Angular	5
1.1.3	El entorno de desarrollo	6
1.2	Hola mundo con Angular	6
1.2.1	Creación de la aplicación	6
1.2.2	Estructura básica de la aplicación	9
1.2.3	Cambios sobre la aplicación por defecto	11
1.2.3.1	El archivo src/index.htm	11
1.2.3.2	El archivo src/main.ts.....	11
1.2.3.3	La carpeta app	12
1.2.3.4	El archivo src/app/app.component.ts.....	12
1.2.3.5	El archivo src/app/app.component.html.....	13
1.2.3.6	El archivo src/app/app.component.css	14
1.2.3.7	El archivo src/styles.css	16
2	Binding.....	19
2.1	Introducción.....	19
2.2	Tipos de binding	19
2.3	Interpolation	19
2.4	Property binding	20
2.5	Event binding.....	21
2.6	Binding de dos direcciones	22
3	Formularios.....	24
3.1	Introducción.....	24
3.2	Doble binding (ligadura de dos direcciones).....	24
3.3	Elementos de formulario.....	24
3.3.1	Cajas de texto	25
3.3.2	Botones de selección única	26
3.3.3	Botones de selección múltiple.....	27
3.3.4	Desplegables	28
3.4	Declaración de tipos.....	30
4	Comunicación entre componentes	32

4.1	Introducción.....	32
4.2	Comunicar información a un componente con @Input().....	32
4.2.1	Decorador @Input().....	32
4.3	Recibir información de un componente con @Output().....	33
4.3.1	Decorador @Output()	34
4.3.2	Desarrollo del evento	36
5	Servicios	39
5.1	Introducción.....	39
5.2	Servicio como intermediario de servicios REST	39
5.2.1	Ejemplo más elaborado.....	43
5.3	Servicio como intermediario de comunicación entre procesos.....	44
6	Comunicación con el protocolo HTTP.....	46
6.1	Introducción.....	46
6.2	El método get	46
6.3	Integración con un componente Angular.....	47
6.4	Especificación de tipos en los observables	49
7	Enrutamiento.....	53
7.1	Introducción.....	53
7.2	Contenedor de componentes.....	53
7.3	Rutas	54
7.4	Empleo de las rutas en las plantillas.....	55
7.4.1	Rutas sin parámetros.....	55
7.4.2	Ruta por defecto.....	57
7.4.3	Ruta de error 404	57
7.5	Rutas con parámetros.....	58
7.6	Acceso a rutas desde TypeScript.....	63

1 Introducción a Angular

1.1 Preparación del entorno de trabajo

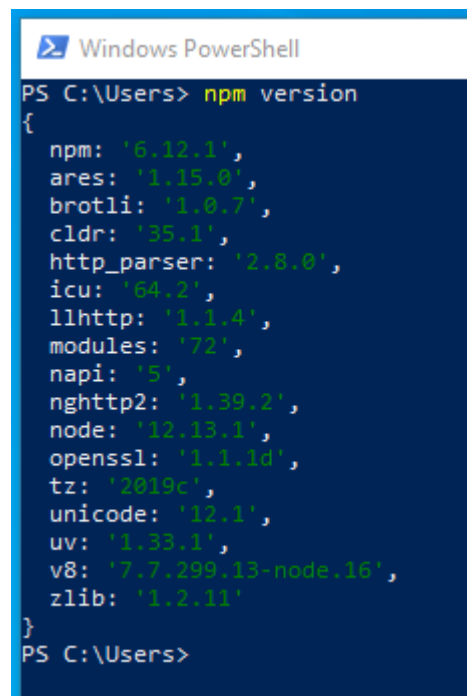
1.1.1 El gestor de paquetes

Para trabajar con Angular necesitamos instalar el gestor de paquetes **npm**. Esta herramienta requiere de **Node.js**, así que realmente lo que instalamos es esta segunda herramienta, y con ello tendremos instalado **npm**.

La dirección de la descarga es

<https://nodejs.org/es/download/>

Comprueba que la herramienta se ha instalado correctamente desde la consola (la consola debe ser abierta después de instalar Node.js):



```
Windows PowerShell
PS C:\Users> npm version
{
  npm: '6.12.1',
  ares: '1.15.0',
  brotli: '1.0.7',
  cldr: '35.1',
  http_parser: '2.8.0',
  icu: '64.2',
  llhttp: '1.1.4',
  modules: '72',
  napi: '5',
  nghttp2: '1.39.2',
  node: '12.13.1',
  openssl: '1.1.1d',
  tz: '2019c',
  unicode: '12.1',
  uv: '1.33.1',
  v8: '7.7.299.13-node.16',
  zlib: '1.2.11'
}
PS C:\Users>
```

Actualización de NPM

Su ya está instalado NPM y queremos actualizarlo ejecutamos los siguientes comandos:

```
$ npm --version
$ npm install npm@latest -g
$ npm --version
```

1.1.2 El paquete de soporte de línea de comandos para Angular

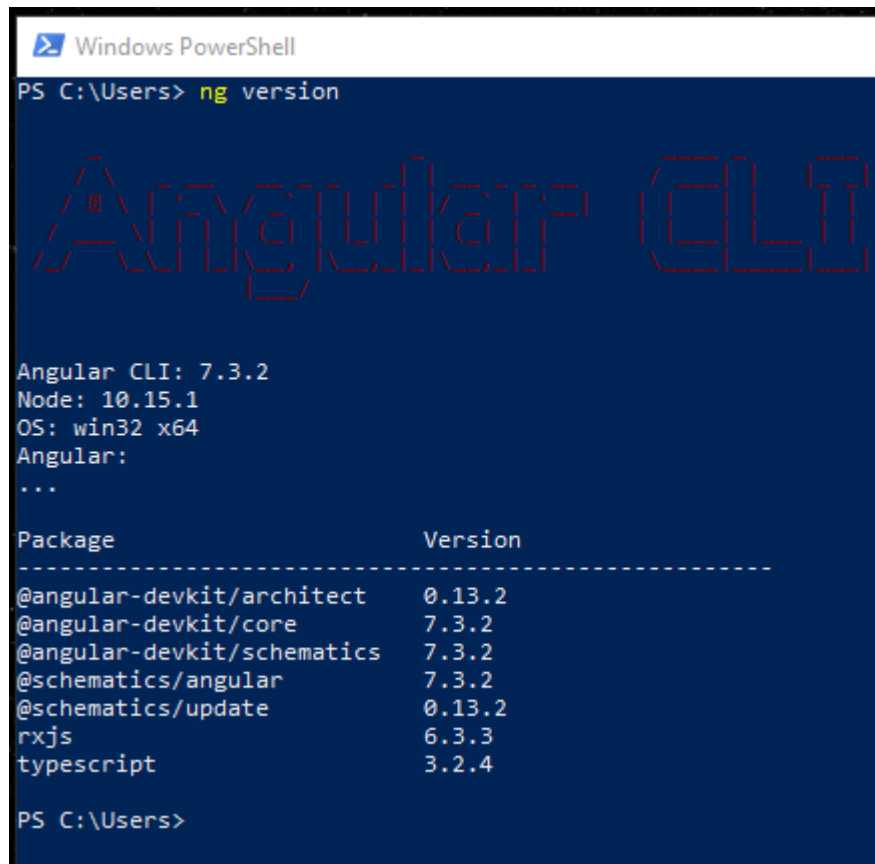
Para trabajar con Angular necesitamos un intérprete de comandos propio de Angular llamado **Angular CLI** (Command Line Interpreter), que instalamos empleando el comando correspondiente npm desde una consola cmd o Powershell.

La instalación del paquete se realiza a nivel global porque se usará en muchos proyectos.

```
PS C:\Users> npm install -g @angular/cli
C:\Users\adolfo\AppData\Roaming\npm\ng -> C:\Users\adolfo\AppData\Roaming\npm\node_modules\@angular\cli
> @angular/cli@8.3.20 postinstall C:\Users\adolfo\AppData\Roaming\npm\node_modules\@angular\cli
> node ./bin/postinstall/script.js

? Would you like to share anonymous usage data 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
+ @angular/cli@8.3.20
added 250 packages from 186 contributors in 24.17s
PS C:\Users>
```

Comprueba la versión instalada.



```
Windows PowerShell
PS C:\Users> ng version

Angular CLI
Angular CLI: 7.3.2
Node: 10.15.1
OS: win32 x64
Angular:
...

Package      Version
-----
@angular-devkit/architect 0.13.2
@angular-devkit/core      7.3.2
@angular-devkit/schematics 7.3.2
@schematics/angular       7.3.2
@schematics/update        0.13.2
rxjs                 6.3.3
typescript           3.2.4

PS C:\Users>
```

En este momento puedes empezar a trabajar con Angular.

Actualización de Angular CLI

Si está instalado el interprete de Angular y se quiere actualizar hay que realizar el siguiente proceso, en el que consultamos los paquetes instalados a nivel global, a continuación desinstalamos el intérprete y finalmente volvemos a instalarlo:

```
$ npm ls --depth=0 -g
$ npm uninstall -g @angular/cli
$ npm install -g @angular/cli
$ npm ls --depth=0 -g
```

1.1.3 El entorno de desarrollo

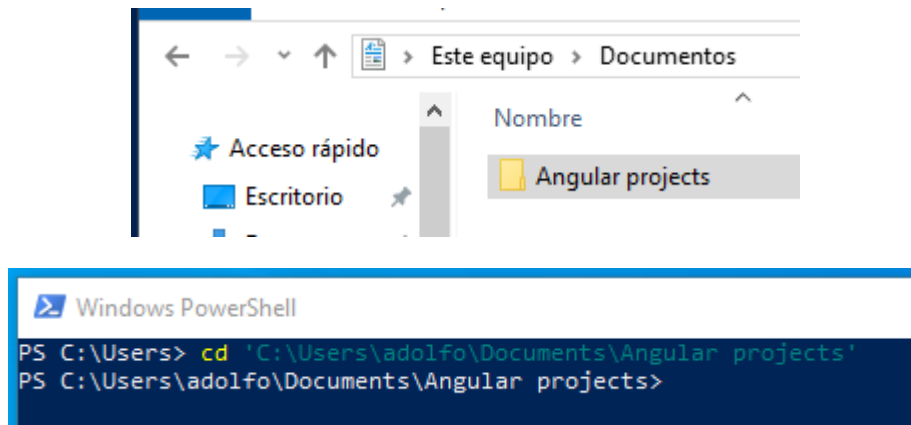
Emplearemos como entorno de desarrollo **Visual Studio Code**.

1.2 Hola mundo con Angular

1.2.1 Creación de la aplicación

Para desarrollar con Angular, primero se crea la aplicación, y a continuación se van añadiendo los componentes que esta aplicación necesita.

Creamos una carpeta de trabajo para Angular, por ejemplo, *Angular projects*. Nos situamos en esta carpeta desde la consola y creamos nuestra primera aplicación Angular.



Nuestra primera aplicación recibe el nombre *hola-mundo*.

```

Administrador: Windows PowerShell

PS C:\Users\adolfo\Documents\Angular projects> ng new hola-mundo
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use? CSS
CREATE hola-mundo/angular.json (3625 bytes)
CREATE hola-mundo/package.json (1296 bytes)
CREATE hola-mundo/README.md (1027 bytes)
CREATE hola-mundo/tsconfig.json (543 bytes)
CREATE hola-mundo/tslint.json (1953 bytes)
CREATE hola-mundo/.editorconfig (246 bytes)
CREATE hola-mundo/.gitignore (631 bytes)
CREATE hola-mundo/browserslist (429 bytes)
CREATE hola-mundo/karma.conf.js (1022 bytes)
CREATE hola-mundo/tsconfig.app.json (270 bytes)
CREATE hola-mundo/tsconfig.spec.json (270 bytes)
CREATE hola-mundo/src/favicon.ico (948 bytes)
CREATE hola-mundo/src/index.html (295 bytes)
CREATE hola-mundo/src/main.ts (372 bytes)
CREATE hola-mundo/src/polyfills.ts (2838 bytes)
CREATE hola-mundo/src/styles.css (80 bytes)
CREATE hola-mundo/src/test.ts (642 bytes)
CREATE hola-mundo/src/assets/.gitkeep (0 bytes)
CREATE hola-mundo/src/environments/environment.prod.ts (51 bytes)
CREATE hola-mundo/src/environments/environment.ts (662 bytes)
CREATE hola-mundo/src/app/app.module.ts (314 bytes)
CREATE hola-mundo/src/app/app.component.html (25498 bytes)
CREATE hola-mundo/src/app/app.component.spec.ts (993 bytes)
CREATE hola-mundo/src/app/app.component.ts (214 bytes)
CREATE hola-mundo/src/app/app.component.css (0 bytes)
CREATE hola-mundo/e2e/protractor.conf.js (808 bytes)
CREATE hola-mundo/e2e/tsconfig.json (214 bytes)
CREATE hola-mundo/e2e/src/app.e2e-spec.ts (643 bytes)
CREATE hola-mundo/e2e/src/app.po.ts (262 bytes)
npm WARN      core-js@2.6.11: core-js@<3 is no longer maintained
f issues. Please, upgrade your dependencies to the actual version of co
npm WARN      fsevents@1.2.9: One of your dependencies needs to u
pt 2) No more fetching binaries from AWS, smaller package size

```

Después de descargar y crear los recursos necesarios, tendremos el esqueleto para nuestra aplicación listo. Como vemos, se crea una carpeta con el nombre indicado, en cuyo interior está localizado nuestro proyecto.

```

Administrador: Windows PowerShell

PS C:\Users\adolfo\Documents\Angular projects> dir

Directorio: C:\Users\adolfo\Documents\Angular projects

Mode                LastWriteTime         Length Name
----                -
d-----          10/12/2019   13:48                hola-mundo

PS C:\Users\adolfo\Documents\Angular projects>

```

Ya tenemos una aplicación funcional. Si entramos en la carpeta podemos arrancar un servidor propio de Angular y ver en el navegador la aplicación creada por defecto que tenemos que modificar.

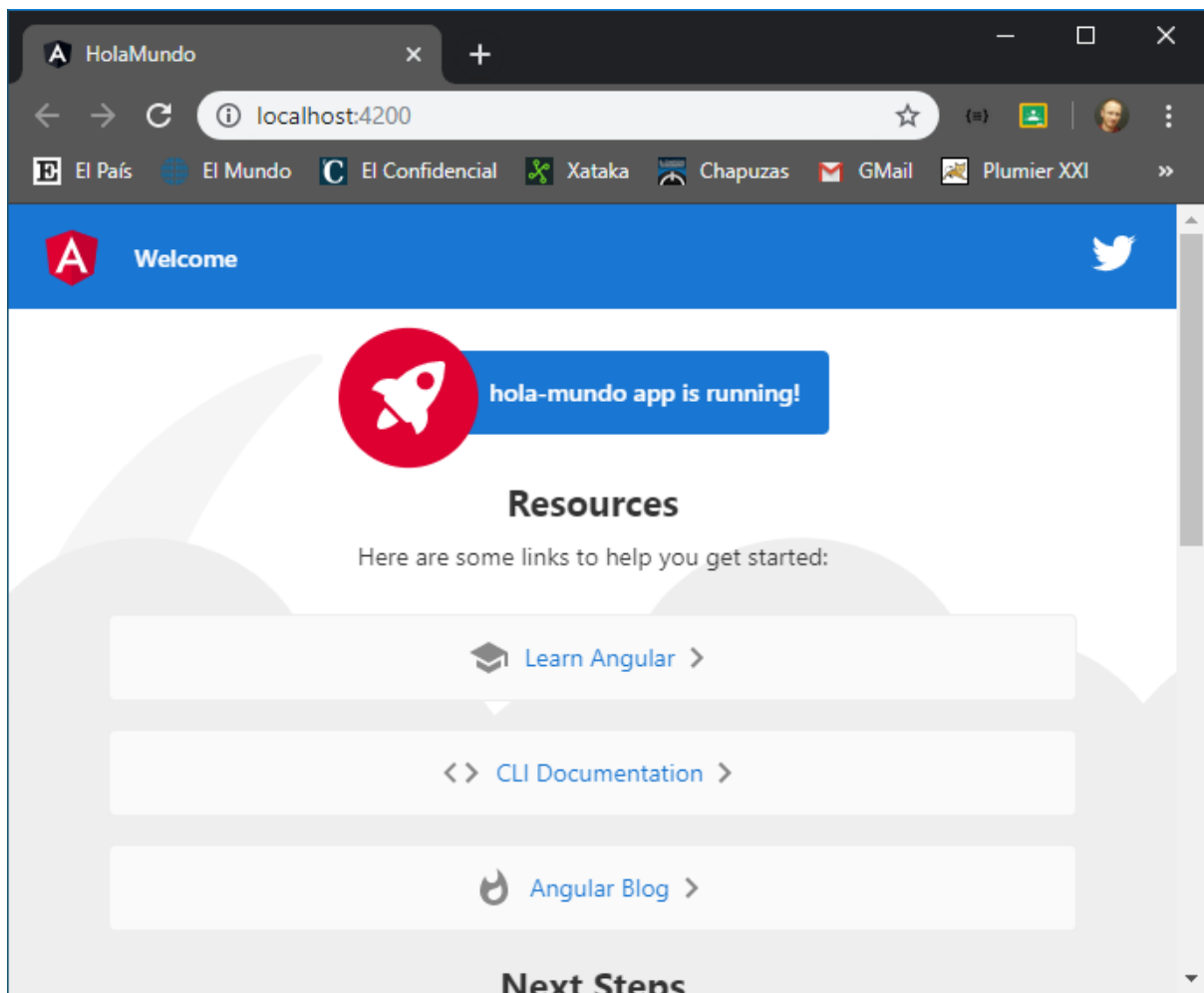
```
ng serve --open
PS C:\Users\adolfo\Documents\Angular projects> cd hola-mundo
PS C:\Users\adolfo\Documents\Angular projects\hola-mundo> ng serve --open
```

La opción **--open** indica que se también se abra un navegador cargando la aplicación actual. Se arranca el servidor que por defecto escucha en el puerto 4200.

```
ng serve --open
PS C:\Users\adolfo\Documents\Angular projects> cd hola-mundo
PS C:\Users\adolfo\Documents\Angular projects\hola-mundo> ng serve --open
10% building 3/3 modules 0 active @wds: Project is running at http://localhost:4200/webpa
  @wds: webpack output is served from /
  @wds: 404s will fallback to //index.html

chunk {main} main.js, main.js.map (main) 47.8 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 264 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.15 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 9.73 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 3.81 MB [initial] [rendered]
Date: 2019-12-10T13:00:33.112Z - Hash: 45d1d5c9898282d854ef - Time: 7268ms
** Angular Live Development Server is listening on localhost:4200, open your browser on ht
  @wdm: Compiled successfully.
```

Y en el navegador vemos:

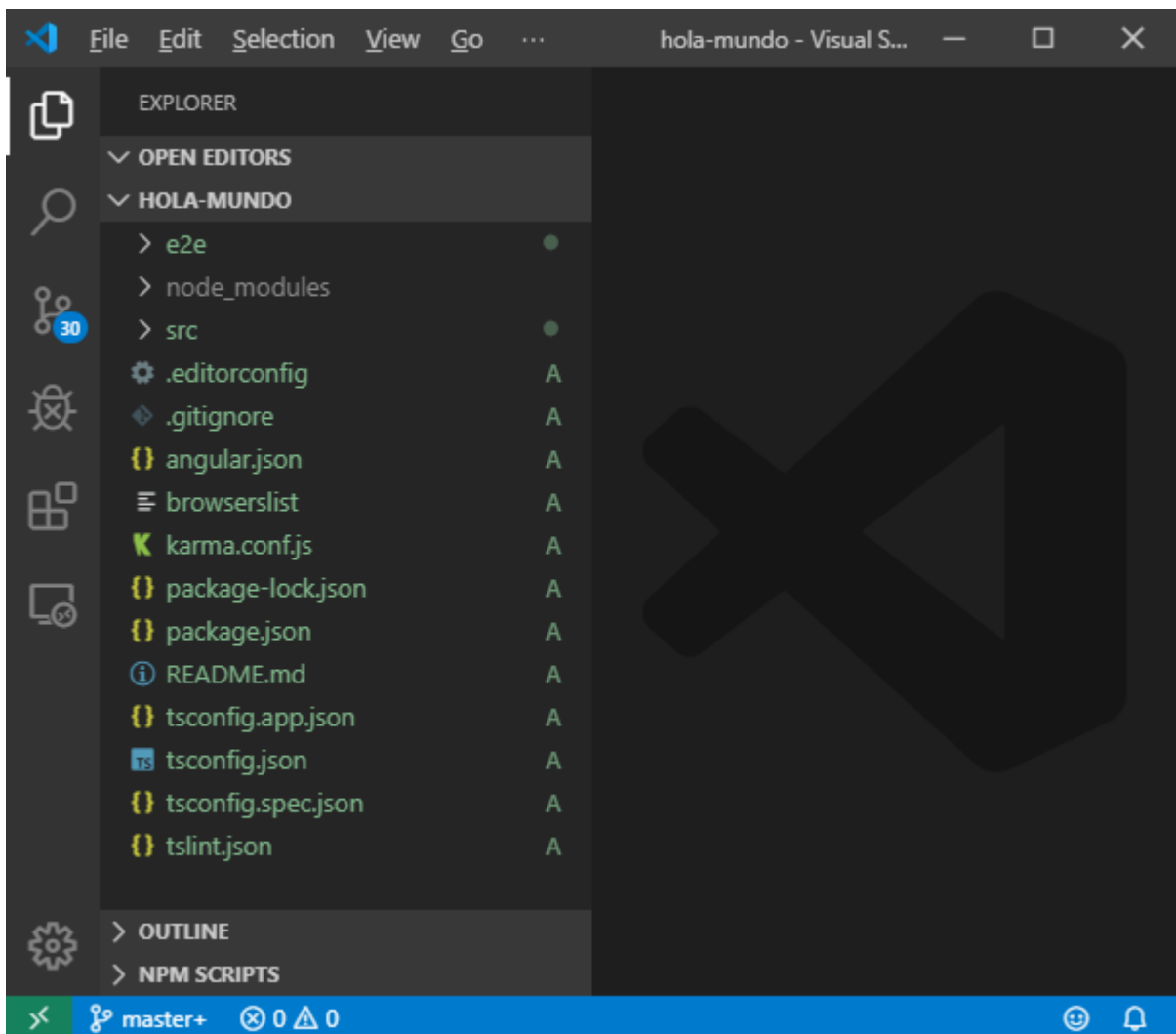


Hay que modificar la aplicación por defecto. Cada vez que guardemos cambios el contenido del navegador se actualizará automáticamente.

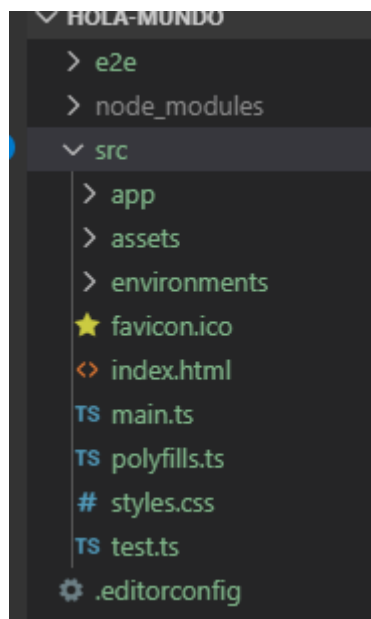
Trabajaremos con la carpeta a través de nuestro entorno de desarrollo, en este caso *Visual Studio Code*. Recuerda que debes abrir la carpeta.

1.2.2 Estructura básica de la aplicación

Al abrir la carpeta encontrarás esta estructura. La aplicación se programa en la carpeta **src**, así que realmente ésta es la carpeta que nos hace falta abrir, y en el futuro lo haremos así.



En la carpeta `src` encontramos la siguiente estructura de archivos y carpetas:



1.2.3 Cambios sobre la aplicación por defecto

1.2.3.1 El archivo src/index.htm

Las aplicaciones Angular son aplicaciones **SPA** (Single Page Application), es decir, la aplicación está compuesta de una única página web en la que van apareciendo y desapareciendo elementos. Por ello, este archivo es el único archivo HTML completo que corresponde a esa única página, el único que tiene cabecera y cuerpo.

Debemos arreglar el <title>.

```
<meta charset="utf-8">
<title>Hola mundo</title>
<base href="/">
```

Guardamos los cambios y el navegador reacciona al cambio.

Debemos observar que el cuerpo contiene una única etiqueta, <app-root>. El estándar HTML no permite etiquetas con guiones, por lo tanto esta etiqueta es una etiqueta personalizada que corresponde a la aplicación que estamos desarrollando.

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

Cambiaremos el nombre de la etiqueta a algo más descriptivo. En la empresa habrá que preguntar si debemos cambiar el nombre.

```
<body>
  <app-hola-mundo></app-hola-mundo>
</body>
```

Al guardar los cambios la aplicación deja de funcionar.

1.2.3.2 El archivo src/main.ts

Este archivo contiene dos líneas de interés en este momento.

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

Esta línea importa una clase situada en el archivo indicado.

```
platformBrowserDynamic().bootstrapModule(AppModule)
```

Esta línea indica que al arrancar la aplicación debe cargar la clase *AppModule*.

Esta clase estará asociada a la etiqueta <app-hola-mundo>, y actualmente falla porque sigue asociada a <app-root>.

1.2.3.3 La carpeta app

Angular desarrolla lo que llamamos componentes. En general, cada componente está formado por tres archivos, un html, un css y un TypeScript (equivalente a JavaScript).

La etiqueta `<app-hola-mundo>` es un componente Angular. En esta carpeta encontramos los tres archivos asociados al componente, que son los archivos cuyo nombre empieza por *app.component*.

1.2.3.4 El archivo src/app/app.component.ts

Aquí encontramos el TypeScript (equivalente a JavaScript) asociado al componente.

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

@Component({
  selector: 'app-hola-mundo',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'hola-mundo';
}
```

Podemos distinguir tres partes:

1. La primera parte importa las clases necesarias para desarrollar el componente, actualmente solo importa la clase *Component*, declarada en el núcleo de Angular.
2. El decorador *@Component*, cuya función es indicar que este componente está asociado a la etiqueta *app-root*, formado por la plantilla de html indicado y formado por el archivo de estilos indicado.
3. La clase *AppComponent*, que contiene el código o lo que se denomina lógica de negocio.

Hemos de asociar la etiqueta `<app-hola-mundo>` a este componente, modificando la propiedad *selector* al valor

```
selector: 'app-hola-mundo',
```

y la aplicación vuelve a funcionar.

La clase contiene una propiedad, *title*, que veremos en el punto siguiente.

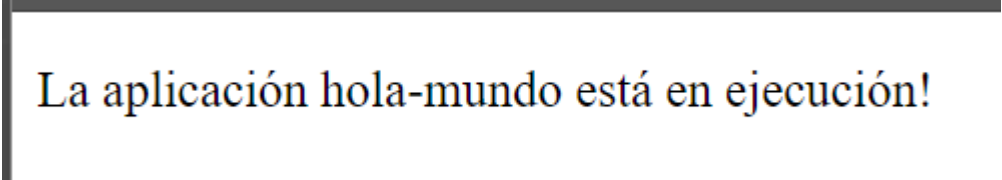
1.2.3.5 El archivo src/app/app.component.html

Este archivo contiene el HTML asociado al componente.

El contenido creado hay que borrarlo totalmente, y poner el correcto. Sustituimos todo el contenido por la línea

```
1 <p>La aplicación {{ title }} está en ejecución!</p>
```

En el navegador aparece



La aplicación hola-mundo está en ejecución!

El elemento dobles-llaves se denomina binding de solo lectura, lo que significa que Angular leerá la propiedad con el mismo nombre en la clase asociada y lo mostrará en ese lugar. Solo actúa en una dirección, de la clase hacia la etiqueta.

Modificamos el TypeScript del componente.

```
export class AppComponent {  
  title = 'Hola mundo';  
  message = 'Esta es nuestra primera aplicación Angular';  
  informacion = 'Angular es un proyecto soportado por Google';  
}
```

Modificamos el HTML del componente.

```
<div class="wrapper">  
  <h1>Angular</h1>  
  <h2>La aplicación "{{ title }}" está en ejecución!</h2>  
  <p>{{ message }}</p>  
  <p>{{ informacion }}</p>  
</div>
```

Y en el navegador observamos como la página web es completada con el valor almacenado en los bindings.

Angular

La aplicación Hola mundo está en ejecución!

Esta es nuestra primera aplicación Angular

Angular es un proyecto soportado por Google

1.2.3.6 El archivo `src/app/app.component.css`

Este archivo gestiona el diseño css del componente que estamos desarrollando, y solo de este componente.

Añade el siguiente contenido:

```

.wrapper * {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}

.wrapper {
  --angular-red: ■ rgb(202, 7, 7);
  border: solid 5px var(--angular-red);
  border-radius: 15px;
  margin: 25px;
  padding: 25px;
}

.wrapper > h1 {
  text-transform: uppercase;
  font-weight: bolder;
  text-align: center;
  color: var(--angular-red);
  margin-top: 15px;
}

.wrapper > h2 {
  text-align: center;
  color: var(--angular-red);
  margin-top: 15px;
  margin: 15px;
}

.wrapper > p {
  font-family: Arial, Helvetica, sans-serif;
  margin-top: 15px;
}

```

Con lo que nuestra página adquiere el siguiente aspecto:

ANGULAR

La aplicación "Hola mundo" está en ejecución!

Esta es nuestra primera aplicación Angular

Angular es un proyecto soportado por Google

1.2.3.7 El archivo `src/styles.css`

Aquí se desarrolla el CSS de la aplicación en su conjunto, considerando que solo conocemos el contenido de nuestro `index.html`. Por ello, debemos abstraernos de los futuros componentes que desarrollemos.

Ya que conocemos que el cuerpo es el siguiente:

```
<body>  
| <app-hola-mundo></app-hola-mundo>  
</body>
```

El contenido de este archivo será


```
:root {
  --fondo-color: #132f42;
}

* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}

body {
  height: 100vh;
  background-color: var(--fondo-color);
  display: flex;
  justify-content: center;
  align-items: center;
  padding: 100px;
  overflow: hidden;
}

app-hola-mundo {
  transform: rotate(-30deg);
  transform-origin: center;
}
```

Por lo que el aspecto final de la aplicación es:

ANGULAR

**La aplicación "Hola mundo" está
en ejecución!**

Esta es nuestra primera aplicación Angular

Angular es un proyecto soportado por Google

2 Binding

2.1 Introducción

Los bindings (ligaduras) o databindings son una técnica con la que se asocia una fuente de datos con una pantalla o, dicho de otro modo, una herramienta que permite asociar el valor de una propiedad de la lógica de negocio con un elemento gráfico en una plantilla del interfaz gráfico.

2.2 Tipos de binding

Angular ofrece los siguientes tipos:

- Interpolation (interpolación).
- Property binding (ligadura de propiedades).
- Event binding (ligadura de eventos).
- Binding de dos direcciones.

2.3 Interpolation

Con este tipo de binding, Angular asocia el valor de una propiedad en el código TypeScript (TS de ahora en adelante) con una zona delimitada por dobles llaves `{{}}` en el HTML.

En Angular, el HTML lo llamaremos template (plantilla).

Este tipo de binding es de una sola dirección, va del código TS al template. Por lo tanto, cada vez que cambie el valor de una propiedad en el TS de forma automática cambia su valor en el interfaz.

Para crear un componente ejecutamos el comando:

```
$ ng generate component interpolation-binding
```

Esta línea de comandos es equivalente a:

```
$ ng generate component InterpolationBindingComponent
```

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

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

  private nombre: string = 'Sonia Castillo';
  private edad: number = 25;
  private saludo: string = 'Buenos días!';

  constructor() { }

  ngOnInit() {
  }

}
```

En esta imagen aparecen declaradas en la clase tres propiedades, que son asociadas mediante un binding de interpolación a tres elementos de la plantilla delimitados por dobles-llaves que encierran los identificadores de dichas propiedades.

```
<p>{{saludo}}</p>
<p>Soy {{nombre}} y tengo {{edad}} años.</p>
```

En el navegador aparece:

Bienvenidos a tipos-binding!

Buenos días!

Soy Sonia Castillo y tengo 25 años.

2.4 Property binding

Este tipo de binding permite enviar información de la lógica de negocio TS a elementos HTML del template. Los elementos HTML pueden ser propiedades de etiquetas, propiedades de componentes Angular y directivas.

El símbolo utilizado es el corchete. Este tipo de binding es de una sola dirección, va del TS al template,

En el caso de propiedades de etiquetas, los corchetes rodean el nombre de la propiedad y en el igual se asigna el valor de la propiedad TS del que debe obtenerse el valor.

En estas imágenes aparecen declaradas en la clase tres propiedades, dos de las cuáles son asociadas mediante un property binding a dos propiedades de la etiqueta de enlace de la plantilla delimitados por corchetes, en cuyos valores asignados encierran los identificadores de dichas propiedades.

```
<a [href]="url" [target]="objetivo">{{texto}}</a>
```

```
//  
export class PropertyBindingComponent implements OnInit {  
  
  private url: string = 'https://angular.io/';  
  private texto: string = 'Sitio web oficial del framework Angular';  
  private objetivo = '_blank';  
  
  constructor() { }
```

En el navegador aparece:

Bienvenidos a tipos-binding!

[Sitio web oficial del framework Angular](https://angular.io/)

2.5 Event binding

Este tipo de ligadura nos permite la comunicación del código TS con los eventos que ocurren a elementos HTML.

Hay que emplear eventos propios de Angular. El nombre del evento va encerrado por paréntesis y se le asigna el método que debe ejecutarse en el código.

En las imágenes aparecen dos botones en los que se programa el evento click. Al pulsar un botón aparece la letra correspondiente al botón, ya que se pasa como argumento al método asociado al evento.

```
<p>Has pulsado el botón {{letraBoton}}</p>
<br>
<input type="button" value="A" (click)="registrarLetra('A')"/>
<br>
<input type="button" value="B" (click)="registrarLetra('B')"/>
```

```
export class EventBindingComponent implements OnInit {
  private letraBoton: string = 'Ninguna';

  private registrarLetra(letra: string) {
    this.letraBoton = letra;
  }
}
```

Bienvenidos a tipos-binding!

Has pulsado el botón Ninguna



2.6 Binding de dos direcciones

El binding de dos direcciones combina la lectura y la escritura de valores, es decir, si el valor cambia en el código TS también cambiará en el template (el código HTML), y a la vez si cambia en el template también cambia en el código TS.

El operador que se utiliza es corchetes-paréntesis `[]`, que de manera informal se denomina “banana in a box”. Su empleo requiere usar la directiva `ngModel`, a la que se le asigna el nombre de la propiedad asociada.

En el siguiente código tenemos una caja de texto asociada a la propiedad del componente nombre. Cuando cambia el contenido de la caja de texto, cambia el contenido de la propiedad en el componente.

```
<label>Cuál es tu nombre: </label>
<input type="text" [(ngModel)]="nombre">

<br>
<p>Mi nombre es {{nombre}}</p>
```

```
//
export class TwoWaysBindingComponent implements OnInit {

  private nombre: string = 'tu nombre';

  constructor() { }

  ngOnInit() {
  }

}
```

Para que funcione este binding es necesario importar el módulo FormsModule en el módulo que acoge a este componente:

```
],
imports: [
  BrowserModule,
  FormsModule
],
providers: []
```

En el navegador observaremos:

Bienvenidos a tipos-binding!

Cuál es tu nombre:

Mi nombre es Manuel

3 Formularios

3.1 Introducción

En la actualidad en Angular se emplean dos tecnologías para el desarrollo de formularios:

- Formularios.
- Formularios reactivos.

Normalmente, la elaboración de un formulario supondrá la creación de un componente con propiedades que son un reflejo de los elementos del formulario. La combinación de las etiquetas propias de los formularios con el mecanismo de binding ayudará a que el desarrollo de los formularios sea bastante sencillo.

3.2 Doble binding (ligadura de dos direcciones)

El doble binding permite que una etiqueta de un formulario y una propiedad de una clase estén ligadas en operaciones de lectura y escritura.

Para poder emplear el doble binding, es necesario emplear en el proyecto el módulo FormsModule, que hay que importar manualmente.

Con este módulo importado, hay que emplear el elemento Angular [(ngModel)] para asociar la etiqueta y la propiedad, como veremos en los siguientes apartados.

3.3 Elementos de formulario

Para trabajar con estos contenidos vamos a crear un proyecto con un componente denominado DatosPersonalesComponent asociado a la etiqueta datos-personales.

El resultado final de este proyecto mostrará el siguiente contenido:

Datos personales

Nombre:

Apellidos:

Género:

☐ Hombre

☒ Mujer

Aficiones

☒ Música

☐ Deportes

☒ Cine

Situación laboral:

Estudios:

```
{
  "nombre": "Sonia",
  "apellidos": "Castillo",
  "genero": "mujer",
  "aficionMusica": true,
  "aficionCine": true,
  "situacionLaboral": "1",
  "estudios": "Desarrollo de aplicaciones web"
}
```

3.3.1 Cajas de texto

Para emplear una caja de texto en la clase declaramos una propiedad de tipo cadena de texto, que asociamos a una etiqueta input de tipo texto mediante una operación binding de dos direcciones.

```
private datos: Object = {
  nombre: undefined,
```

Vamos a trabajar con objetos, en este caso con “Object literal”, y para ello declaramos una propiedad de tipo objeto con la propiedad destinada a almacenar el nombre. Más adelante hablaremos de cómo declarar los tipos al trabajar con objetos.

Añadimos en el componente la propiedad nombre de tipo cadena de texto. En la plantilla (template) añadimos la correspondiente caja de texto.

```

<h1>Datos personales</h1>

<div>
  <label for="nombre">Nombre: </label>
  <input type="text" name="nombre" placeholder="Nombre" [(ngModel)]="datos.nombre">
</div>

```

Ampliamos el código para trabajar además con los apellidos.

```

private datos: Object = {
  nombre: undefined,
  apellidos: undefined,

```

```

<div>
  <label for="apellidos">Apellidos: </label>
  <input type="text" name="apellidos" placeholder="Apellidos" [(ngModel)]="datos.apellidos">
</div>

```

Campo adicional para comprobar que el binding funciona correctamente

Añadimos en el formulario una etiqueta que incluye una interpolación, en la que visualmente podemos comprobar que en tiempo real se producen las ligaduras.

```

<div>
  <pre>
    {{datos | json}}
  </pre>
</div>

```

La tubería (pipe) json nos muestra el contenido de la variable en dicho formato.

3.3.2 Botones de selección única

Para emplear los botones de selección única (botones de radio) en la clase declaramos una propiedad de tipo del identificador correspondiente, que asociamos a las correspondientes etiqueta input de tipo radio mediante una operación binding de dos direcciones.

Añadimos en el componente la propiedad genero de tipo cadena de texto. En la plantilla (template) añadimos las etiquetas para seleccionar hombre o mujer.

```
private datos: Object = {  
  nombre: undefined,  
  apellidos: undefined,  
  genero: undefined,
```

El template para manipular el género es

```
<div>  
  <label for="genero">Género: </label>  
  <br>  
  <input type="radio" name="genero" value="hombre" [(ngModel)]="datos.genero">  
  <label>Hombre</label><br>  
  <input type="radio" name="genero" value="mujer" [(ngModel)]="datos.genero">  
  <label>Mujer</label><br>  
</div>
```

3.3.3 Botones de selección múltiple

Para emplear los botones de selección múltiple (botones checkbox) en la clase declaramos una propiedad de tipo del identificador correspondiente, que asociamos a las correspondientes etiqueta input de tipo checkbox mediante una operación binding de dos direcciones.

Añadimos en el componente tres propiedades para poder solicitar si gustan tres aficiones, y serán de tipo booleano.

```
private datos: Object = {  
  nombre: undefined,  
  apellidos: undefined,  
  genero: undefined,  
  aficionMusica: undefined,  
  aficionDeportes: undefined,  
  aficionCine: undefined,
```

En la plantilla (template) añadimos el correspondiente código:

```

<div>
  <label>Aficiones</label>
  <br>
  <input type="checkbox" name="aficion-musica" [(ngModel)]="datos.aficionMusica">
  <label>Música</label>
  <br>
  <input type="checkbox" name="aficion-deportes" [(ngModel)]="datos.aficionDeportes">
  <label>Deportes</label>
  <br>
  <input type="checkbox" name="aficion-cine" [(ngModel)]="datos.aficionCine">
  <label>Cine</label>
  <br>
</div>

```

3.3.4 Desplegables

Para emplear los desplegable (etiqueta select) en la clase declaramos una propiedad de tipo del identificador correspondiente, que asociamos a las correspondientes etiqueta de tipo select mediante una operación binding de dos direcciones.

Añadimos en el componente la propiedad situacionLaboral de tipo numérico.

```

private datos: Object = {
  nombre: undefined,
  apellidos: undefined,
  genero: undefined,
  aficionMusica: undefined,
  aficionDeportes: undefined,
  aficionCine: undefined,
  situacionLaboral: undefined,

```

Vamos a obtener la información para desarrollar el desplegable de un arreglo de objetos preparado para ello.

```
private situacionesLaborales: Object[] = [  
  {id: 1, descripcion: 'Estudiante'},  
  {id: 2, descripcion: 'Asalariado'},  
  {id: 3, descripcion: 'Autónomo'},  
  {id: 4, descripcion: 'Funcionario'},  
  {id: 5, descripcion: 'Desempleado'},  
  {id: 6, descripcion: 'Jubilado'},  
];
```

La directiva estructural *ngFor

Angular proporciona una herramienta que facilita el trabajo con arreglos, la directiva estructural ***ngFor**, que trabaja de modo similar a un for-each, por lo que tenemos que indicarle el arreglo a recorrer y la variable en la que se va a almacenar el elemento actual.

En la plantilla (template) añadimos el correspondiente desplegable.

```
<div>  
  <label>Situación laboral:</label>  
  <select name="situacion-laboral" [(ngModel)]="datos.situacionLaboral">  
    <option *ngFor="let situacion of situacionesLaborales" [value]="situacion.id">  
      {{situacion.descripcion}}  
    </option>  
  </select>  
</div>
```

Podemos observar que ngModel se sitúa en la etiqueta select. La directiva *ngFor se sitúa en el elemento que debe ser repetido de acuerdo al contenido del arreglo. Hay que emplear una ligadura de propiedad para indicar de dónde obtener el value, y una interpolación para indicar de donde obtener el texto a mostrar en la opción.

La directiva estructural *ngIf

La directiva ***ngIf** permite establecer que una etiqueta y todo su contenido se muestre si se cumple o no una condición evaluada a verdadero o falso.

Para ver como funciona, añadimos una última propiedad llamada estudios, que solo debe ser completada en caso de que en el desplegable el usuario seleccione la opción *Estudiante*.

```
private datos: Object = {
  nombre: undefined,
  apellidos: undefined,
  genero: undefined,
  aficionMusica: undefined,
  aficionDeportes: undefined,
  aficionCine: undefined,
  situacionLaboral: undefined,
  estudios: undefined
}
```

Empleando esta directiva en la plantilla evaluamos si la situación laboral es la de estudiante, y en caso afirmativo se mostrará la etiqueta, y en caso negativo no. Hay que recordar que el contenido de un value es siempre una cadena de texto y programar la condición de acuerdo a ello.

```
<div *ngIf="datos.situacionLaboral === '1'">
  <label>Estudios: </label>
  <input type="text" [(ngModel)]="datos.estudios">
</div>
```

3.4 Declaración de tipos

Para emplear todas las características que ofrece TypeScript debemos indicar los tipos de datos de cada elemento. En este ejemplo optamos por emplear dos interfaces que nos permitirán indicarle a TypeScript cuál es la comprobación de tipos correcta.

El primer interfaz es para indicar los tipos de situación laboral, y lo creamos con la orden de la línea de comandos:

```
\app> ng generate interface SituacionLaboral
CREATE src/app/situacion-laboral.ts (38 bytes)
```

Aquí indicamos los tipos de cada propiedad:

```
export interface SituacionLaboral {
  id: number;
  descripcion: string;
}
```

El arreglo anterior debe modificarse al siguiente contenido, que si no indica ningún error es que estamos asignando valores del tipo correcto:

```
private situacionesLaborales: SituacionLaboral[] = [
  {id: 1, descripcion: 'Estudiante'},
  {id: 2, descripcion: 'Asalariado'},
  {id: 3, descripcion: 'Autónomo'},
];
```

Desarrollamos otro interfaz para indicar los tipos de datos correctos para las propiedades del objeto datos.

```
\app> ng generate interface DatosPersonales
CREATE src/app/datos-personales.ts (37 bytes)
PS C:\Users\adelfo\Documents\Angular-projects\202
```

Cuyo contenido es

```
export interface DatosPersonales {
  nombre: string,
  apellidos: string,
  genero: string,
  aficionMusica: boolean,
  aficionDeportes: boolean,
  aficionCine: boolean,
  situacionLaboral: number,
  estudios: string
}
```

Y finalmente indicamos el tipo correcto para la propiedad datos

```
private datos: DatosPersonales = {
  nombre: undefined,
  apellidos: undefined,
  genero: undefined,
  aficionMusica: undefined
```

4 Comunicación entre componentes

4.1 Introducción

En este apartado vamos a explicar cómo realizar la comunicación entre elementos adyacentes, que en este caso significa elementos que están situados en la misma página de diseño. Así que abordaremos los siguientes elementos:

- Comunicar información a un componente que está siendo usado.
- Recibir información de un componente que está siendo usado.

Es importante entender que cuando un programador emplea un componente, conoce las características y los atributos de dicho componente, pero el programador de dicho componente no conoce nada del lugar donde se usará dicho componente.

4.2 Comunicar información a un componente con @Input()

El paso de información a un componente que está siendo usado desde el punto de vista de quien lo usa es muy simple. Solo hay que asignar información a una propiedad del componente. Esta propiedad debe ser preparada en el componente para tal efecto. Por ejemplo, si he desarrollado un componente asociado a la etiqueta “mi-etiqueta” y en ésta se ha preparado la propiedad “mensaje”, emplearíamos la siguiente notación:

```
<mi-etiqueta mensaje="Hola, probando la comunicación entre etiquetas">
</mi-etiqueta>
```

4.2.1 Decorador @Input()

El decorador `@Input()` se emplea para crear este atributo personalizado que permite a quien use el componente el paso de información relevante. Lo empleamos así:

```
//
export class MiEtiquetaComponent implements OnInit {
  @Input('mensaje') private mensajeRecibido: string;
```

Entre los paréntesis debemos poner la denominación del atributo HTML que deseamos emplear, en este caso “mensaje”. Vemos que aparece el término “private”, lo que nos da una pista de que es una propiedad de la clase. A continuación, aparece el nombre de la propiedad de la clase asociada, ya que no es obligatorio que coincida con el nombre del atributo HTML. De este modo, por un sistema de binding lo escrito en el atributo mensaje se asigna automáticamente a la propiedad mensajeRecibido.

Esto es todo, ya hemos desarrollado todo el proceso.

Comprobación

Vamos a añadir algo de código para verificar que se ha producido el paso de información. Añadimos este código al template (plantilla):

```
<div>
  <label>Información suministrada en la propiedad "mensaje": </label>
  <br>
  <span>{{mensajeRecibido}}</span>
</div>
```

En el navegador debe aparecer:

```
Información suministrada en la propiedad "mensaje":
Hola, probando la comunicación entre etiquetas
```

Si añadimos más etiquetas de este componente:

```
<mi-etiqueta mensaje="Hola, probando la comunicación entre etiquetas">
</mi-etiqueta>
<mi-etiqueta mensaje="Adios, todo ha ido bien">
</mi-etiqueta>
```

Vemos que cada etiqueta toma el valor correspondiente:

```
Información suministrada en la propiedad "mensaje":
Hola, probando la comunicación entre etiquetas
Información suministrada en la propiedad "mensaje":
Adios, todo ha ido bien
```

4.3 Recibir información de un componente con @Output()

El paso de información desde un componente a quien lo usa es algo elaborado. Hay que preparar lo que podríamos llamar evento o un observable, ya que esta información estará disponible cuando lo indique el componente, y quien lo usa no conocerá el momento es que este hecho se produzca.

Así, hay que programar un evento del componente. Este evento debe ser preparado en el componente para tal efecto. Siguiendo con nuestro ejemplo, emplearíamos la siguiente notación para indicar que queremos saber cuándo el componente nos envía un mensaje:

```
<mi-etiqueta mensaje="Hola" (onmensaje)="procesarMensaje($event)">
</mi-etiqueta>
```

Vemos que empleamos una interpolación de evento.

4.3.1 Decorador @Output()

El decorador **@Output()** se emplea para crear este evento personalizado que permite a quien use el componente recibir información relevante. Lo empleamos así:

```
export class MiEtiquetaComponent implements OnInit {  
  @Output('onmensaje') private emisorMensaje = new EventEmitter<string>();
```

Entre los paréntesis debemos poner la denominación del evento HTML que deseamos emplear, en este caso “onmensaje”. Vemos que aparece el término “private”, lo que nos da una pista de que es una propiedad de la clase. A continuación, aparece el nombre de la propiedad de la clase asociada, ya que no es obligatorio que coincida con el nombre del evento HTML. De este modo, por un sistema de binding de eventos se asocia el evento a la propiedad. La propiedad almacena un objeto de la clase “EventEmitter”, y en la construcción se parametriza con el operador diamante (< >) para indicar que el evento imite una cadena de texto, que será el mensaje.

Para generar eventos empleamos el método “emit” de la clase EventEmitter, pasando como argumento lo que queremos enviar, en este caso un mensaje de texto.

```
this.emisorMensaje.emit("Hola, saludos desde el componente <mi-etiqueta>");
```

Esta sentencia será ejecutada cuando corresponda.

Comprobación

Vamos a añadir algo de código para verificar que se ha producido el paso de información.

Decidimos que el evento se lance cinco segundos después de la creación del componente:

```
ngOnInit() {  
  setTimeout(  
    () => this.emisorMensaje.emit("Hola, saludos desde el componente <mi-etiqueta>"),  
    5000  
  )  
}
```

Hay que terminar de programar el evento. Para la primera prueba cambiamos el código anterior:

```
<mi-etiqueta mensaje="Hola" (onmensaje)="mensajeRecibido = $event">
</mi-etiqueta>

<br>
<span>{{mensajeRecibido}}</span>
```

En el navegador debe aparecer después de cinco segundos:

```
Información suministrada en la propiedad "mensaje":
Hola

Hola, saludos desde el componente <mi-etiqueta>
```

En esta primera prueba hemos eliminado la función y directamente hemos hecho una asignación a una variable. Angular nos permite crear la propia variable aquí, no es necesario declararla previamente en el fichero TypeScript. La variable \$event la crea automáticamente Angular, y en ella se recibe la información enviada a través del evento, en este caso el mensaje. Luego empleamos una interpolación para mostrar el mensaje.

Segunda comprobación

Modificamos el código para que sea más parecido al tratamiento de eventos al que estamos habituados, que es empleando una función (en realidad un método).

```
<mi-etiqueta mensaje="Hola" (onmensaje)="procesarMensaje($event)">
</mi-etiqueta>

<br>
<span>{{mensajeRecibido}}</span>
```

Ahora no funciona ya que no existen ni el método “procesarMensaje” ni la propiedad “mensajeRecibido”. Los incluimos en el código de este componente, que en mi caso es directamente la etiqueta app-root.

```
export class AppComponent {
  private mensajeRecibido: string;

  private procesarMensaje(mensaje) {
    this.mensajeRecibido = mensaje;
  }
}
```

En el navegador debe aparecer después de cinco segundos:

```
Información suministrada en la propiedad "mensaje":
Hola

Hola, saludos desde el componente <mi-etiqueta>
```

4.3.2 Desarrollo del evento

El método emit de la clase EventEmitter puede emitir como evento cualquier tipo de elemento, y en este ejemplo hemos emitido una cadena de texto. Vamos a desarrollar algo más parecido al típico evento. Además, usaremos interfaces no porque sea necesario, sino para practicar su empleo.

El interface

El comando para generar la interface es:

```
$ ng generate interface mi-etiqueta.event.interface
```

```
mi-etiqueta > ts mi-etiqueta-event-interface.ts > MiEtiquetaEventInterface
import { MiEtiquetaComponent } from "../mi-etiqueta.component";

export default interface MiEtiquetaEventInterface {
  getMessage(): string;
  getTimestamp(): number;
  getTarget(): MiEtiquetaComponent;
}
```

Esta interface se parece bastante al evento que estamos acostumbrados a ver en JavaScript puro. Si convertimos los métodos en propiedades el símil será mayor.

La implementación de la interface

```
$ ng generate class mi-etiqueta.event.class
```

```

import MiEtiquetaEventInterface from "../mi-etiqueta-event-interface";
import { MiEtiquetaComponent } from "../mi-etiqueta.component";

export default class MiEtiquetaEvent implements MiEtiquetaEventInterface {
  private timestamp: number;

  constructor(private target: MiEtiquetaComponent, private message: string) {
    this.timestamp = new Date().getTime();
    console.log(this.timestamp);
  }

  getMessage(): string {
    return this.message;
  }

  getTimestamp(): number {
    return this.timestamp;
  }

  getTarget(): MiEtiquetaComponent {
    return this.target;
  }
}

```

Creación del evento y emisión

El emisor se configura para emitir la interface. Se crea un evento que cumple la interface desarrollada.

```

export class MiEtiquetaComponent {

  @Input() mensaje: string = '';
  @Output('onmensaje') emisorMensaje = new EventEmitter<MiEtiquetaEventInterface>();

  constructor() {
    setTimeout(
      () => this.fireEvent(),
      10000
    )
  }

  private fireEvent() {
    const event: MiEtiquetaEventInterface = new MiEtiquetaEvent(this, this.mensaje);
    this.emisorMensaje.emit(event);
  }
}

```

Recepción del evento

La etiqueta que está interesada en el evento responde con su trabajo cuando recibe este evento.

```
export class AppComponent {  
  title = 'original';  
  mensajeRecibido: string = '';  
  
  procesarMensaje(e: MiEtiquetaEventInterface) {  
    this.mensajeRecibido = e.getMessage();  
    alert(this.mensajeRecibido);  
    console.log(`Target: ${e.getTarget()}`);  
    console.log(`Timestamp: ${e.getTimestamp()}`);  
  }  
}
```

5 Servicios

5.1 Introducción

Un servicio Angular, como su nombre indica, es un elemento que ofrece un servicio a otros elementos.

Cuando se programan, se definen como elementos inyectables, lo que quiere decir que Angular creará una única instancia de forma automática y desde el constructor de los componentes que quieran usar dicho servicio hay que emplear la inyección de dependencias.

Intermediario con servicios REST

En general, usaremos servicios como intermediarios para consumir servicios REST de la web y ofrecer a través de ellos los servicios REST a otros elementos de nuestro proyecto. Es decir, no se va a permitir que los componentes se comuniquen directamente con los servicios REST, siempre deberán usar como intermediario un servicio Angular.

Intermediario de comunicación entre componentes

Un segundo uso de los servicios es servir de sistema de comunicación entre los componentes de la aplicación Angular.

El sistema tradicional que emplea los decoradores `@Input()` y `@Output()` es incómodo de emplear si los componentes no son cercanos. Como alternativa, se puede emplear un servicio Angular que sea el nexo de unión de las comunicaciones de los distintos componentes de la aplicación, y que además puede provocar que el servicio sea quien gestione el estado de la aplicación (aunque para gestionar el estado de la aplicación hay que usar las tecnologías desarrolladas para ello, por ejemplo, Redux).

5.2 Servicio como intermediario de servicios REST

Como hemos indicado, no se va a permitir que los componentes se comuniquen directamente con los servicios REST. Siempre debe existir un servicio Angular que gestione esta tarea.

Ejemplo

Creemos un proyecto sobre productos tecnológicos.

En este punto no vamos a emplear un servicio REST que sea la fuente de la información. Como se trata de un ejemplo, la información estará codificada directamente en código (hard coded).

La fuente de información es un array de productos tecnológicos. Crea en el directorio raíz (app) un fichero llamado "datos.ts" con el siguiente contenido:

```
export const datos: Array<Object> = [
  {id: 1, nombre: 'Ratón', precio: '9'},
  {id: 2, nombre: 'Teclado', precio: '12'},
  {id: 3, nombre: 'Monitor', precio: '95'},
  {id: 4, nombre: 'Caja', precio: '30'},
  {id: 5, nombre: 'Tarjeta gráfica', precio: '120'},
  {id: 6, nombre: 'Placa base', precio: '65'},
  {id: 7, nombre: 'Módulo de memoria', precio: '45'},
  {id: 8, nombre: 'Procesador', precio: '135'},
  {id: 9, nombre: 'Fuente de alimentación', precio: '40'},
];
```

Hemos dicho que no se permite que los componentes accedan directamente a la fuente de información, deben hacerlo por medio de un servicio. Creamos un servicio para acceder a los productos:

```
s\src\app> ng generate service productos
```

Vemos que un servicio es una clase en el que Angular añade el sufijo Service.

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

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

  constructor() { }
}
```

Observamos que hace uso del decorador @Injectable, así que el servicio participa del sistema de inyección de dependencias.

En un caso real, el servicio se comunicaría empleando el API Angular para comunicaciones HTTP con el servicio REST para obtener la información. Aquí simplemente se importa la información del fichero.

Dicha información se debe ofrecer a los clientes del servicio. Se puede ofrecer por medio de propiedades públicas o por medio de métodos públicos. Aquí emplearemos un método.


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

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

  constructor() { }

  public recuperarProductos(): Array<Object> {
    return datos;
  }
}
```

Añadimos al proyecto un componente que sea cliente del servicio. Crea un componente que muestre una lista de los productos:

```
\src\app> ng generate component productos-lista
```

Para indicar que queremos usar el servicio empleamos inyección de dependencias en el constructor. De este modo podemos usar el elemento que nos interese del servicio.

En este caso, declaramos una propiedad productos a la que asignamos los productos recuperados por el servicio. Recordamos que no se deben poner en el constructor tareas largas, las tareas de inicialización largas se ponen en el método ngOnInit.

Por el mismo motivo, en este ejemplo no es necesario la directiva estructurar `*^ngIf`, pero en un caso real, como los datos tardarán en estar disponibles, es necesario para indicar que solo deben mostrarse los datos cuando estos estén disponibles.

```

import { Component, OnInit } from '@angular/core';
import { ProductosService } from '../productos.service';

@Component({
  selector: 'productos-lista',
  templateUrl: './productos-lista.component.html',
  styleUrls: ['./productos-lista.component.css']
})
export class ProductosListaComponent implements OnInit {
  private productos: Array<Object>;

  constructor(private productosServicio: ProductosService) { }

  ngOnInit() {
    this.productos = this.productosServicio.recuperarProductos();
  }
}

```

La plantilla para observar los resultados es:

```

<ol *ngIf="productos">
  <li *ngFor="let producto of productos">
    {{producto.nombre + ' - ' + producto.precio + ' euros'}}
  </li>
</ol>

```

En pantalla aparece:

1. Ratón - 9 euros
2. Teclado - 12 euros
3. Monitor - 95 euros
4. Caja - 30 euros
5. Tarjeta gráfica - 120 euros
6. Placa base - 65 euros
7. Módulo de memoria - 45 euros
8. Procesador - 135 euros
9. Fuente de alimentación - 40 euros

5.2.1 Ejemplo más elaborado

```
export default class Product {
  constructor(private _uuid: string, private _nombre: string, private _precio: number) {}

  public get uuid(): string {
    return this._uuid;
  }

  public get nombre(): string {
    return this._nombre;
  }

  public get precio(): number {
    return this._precio;
  }
}
```

```
import Product from "../models/product.class";

export interface ProductosRespositoryInterface {
  retrieveProducts(): Array<Product>;
  retrieveProductByUuid(uuid: string): Product;
  createProduct(uuid: string, name: string, price: number): void;
  updateProduct(uuid: string, name: string, price: number): void;
  deleteProduct(uuid: string): void;
}
```

```
import { Injectable } from "@angular/core";
import Product from "../models/product.class";
import { ProductosRespositoryInterface } from "../productos-respository-interface";

@Injectable({
  providedIn: 'root'
})
export class ProductosRepositoryMock implements ProductosRespositoryInterface {
  private static _datos: Array<Product> = [
    new Product('2534-5487', 'Ratón', 18.80),
    new Product('4759-3428', 'Teclado', 25.50),
    new Product('1425-6824', 'Monitor', 195.90),
    new Product('6531-8765', 'Caja', 40.25),
    new Product('3625-7485', 'Tarjeta gráfica', 495.95),
    new Product('1214-7895', 'Placa base', 87.50),
    new Product('7432-6241', 'Módulo de memoria', 74.90),
    new Product('5836-9812', 'Procesador', 189.95),
    new Product('5973-6284', 'Fuente de alimentación', 58.75),
  ];

  retrieveProducts(): Array<Product> {
    return ProductosRepositoryMock._datos;
  }

  retrieveProductByUuid(uuid: string): Product {
    throw new Error("Method not implemented.");
  }

  createProduct(uuid: string, name: string, price: number): void {
    throw new Error("Method not implemented.");
  }

  updateProduct(uuid: string, name: string, price: number): void {
    throw new Error("Method not implemented.");
  }

  deleteProduct(uuid: string): void {
    throw new Error("Method not implemented.");
  }
}
```

```

import { Injectable } from '@angular/core';
import Product from '../models/product.class';
import { ProductosRepositoryMock } from '../repositories/productos-repository-mock';

@Injectable({
  providedIn: 'root'
})
export class ProductosService {
  constructor(private repository: ProductosRepositoryMock) { }

  public retrieveProducts(): Array<Product> {
    return this.repository.retrieveProducts();
  }
}

```

5.3 Servicio como intermediario de comunicación entre procesos

Un uso común de los servicios es facilitar la comunicación entre componentes, incluso convirtiéndose en el elemento que gestiona el estado de la aplicación (hay tecnologías para gestionar el estado de la aplicación, como Redux, y deberíamos emplearlas ya que han sido desarrolladas para ello).

El servicio centraliza la información con la que trabaja la aplicación, y los componentes deben pedir información al servicio o solicitar cambios en la información que ella gestiona. Con este esquema, los componentes no necesitan conocer la existencia de otros componentes, solo conocen el servicio y se comunican con él.

Siguiendo este esquema, no es necesario emplear el mecanismo de comunicación basado en @Input() y @Output() que en ocasiones puede ser complicado de implementar.

Ejemplo

Ampliamos el proyecto con nueva funcionalidad.

El servicio incluye un nuevo método para añadir productos.

```

public anhadirProducto(producto: Object): void {
  datos.push(producto);
}

```

Creamos un nuevo componente con un producto fijo para facilitar el desarrollo.

```

src\app> ng generate component producto-formulario

```

El producto estará predefinido en el TypeScript:

```

    })
    export class ProductoFormularioComponent implements OnInit {
      private producto: Object = {
        id: 10,
        nombre: 'Disco m.2 NVMe',
        precio: '85'
      }

      constructor(private productosServicio: ProductosService) { }

      ngOnInit() {
        this.productosServicio.anhadirProducto(this.producto);
      }
    }
  }

```

No es necesario desarrollar la plantilla para comprobar que la comunicación funciona correctamente. Simplemente incorporamos la nueva etiqueta a la aplicación.

En este momento ni el formulario ni la lista conocen de la existencia del otro. Sin embargo, el formulario le comunica al servicio que añada un nuevo producto, y este producto es mostrado en la lista.

1. Ratón - 9 euros
2. Teclado - 12 euros
3. Monitor - 95 euros
4. Caja - 30 euros
5. Tarjeta gráfica - 120 euros
6. Placa base - 65 euros
7. Módulo de memoria - 45 euros
8. Procesador - 135 euros
9. Fuente de alimentación - 40 euros
10. Disco m.2 NVMe - 85 euros

producto-formulario works!

6 Comunicación con el protocolo HTTP

6.1 Introducción

Angular ofrece una librería para realizar las operaciones básicas del protocolo HTTP. Podremos emplear los cuatro verbos básicos de este protocolo: get, post, put y delete.

6.2 El método get

El método get del protocolo HTTP permite realizar consultas REST.

El método devuelve un objeto de tipo Observable, por lo que es necesario consumirlo mediante la librería RxJS. Puede recibir varios argumentos, pero como mínimo necesita el endpoint (url) que identifica el recurso al que accedemos.

En el caso del siguiente endpoint:

`https://restcountries.eu/rest/v2/all`

consumimos el servicio aplicando el siguiente proceso.

Declaramos en el fichero de módulos `app.modules.ts` si no se ha hecho anteriormente que vamos a usar el módulo de comunicaciones HTTP. Hay que indicar que deseamos importarlo.

```
TS app.module.ts > AppModule
> import { HttpClientModule } from '@angular/common/http'; ...

@NgModule({
  declarations: [ ... ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [],
})
```

A continuación, hay que emplear *inyección de dependencias* para acceder a la instancia que se ha creado con la importación del módulo. En el constructor del servicio implicado declaramos la inyección de dependencias:

```
export class PaísesService {
  constructor(private http: HttpClient) { }
```

Se declara un método que encapsule el proceso de la comunicación AJAX consumiendo el servicio REST. La sintaxis puede ser más o menos específica, y en este ejemplo hemos diseñado esta solución:

```

public recuperarPaíses$(): Observable<Pais[]> {
  const url = 'https://restcountries.eu/rest/v2/all';

  interface PaisEnIngles {
    alpha3Code: string;
    name: string;
  }

  const convertirEnObjeto = ({ alpha3Code, name }) => new Pais(alpha3Code, name);

  return this.http
    .get<PaisEnIngles[]>(url)
    .pipe<Pais[]>(
      map<PaisEnIngles[], Pais[]>(
        paises => paises.map<Pais>(convertirEnObjeto)
      )
    )
}

```

Esta solución declara todos los tipos de datos involucrados, pero no es necesario ser tan específico. Para ser específico se emplea el operador diamante.

El método en sí devuelve un observable de tipo arreglo de países.

El nombre del método contiene un signo de dólar (\$) al final, indicando que devuelve un observable (también llamado stream de datos).

La llamada get devuelva valores que no deseamos. Hemos declarado un interfaz indicando que elementos son los que queremos de todo lo recibido mediante la llamada get, en este caso el código y el nombre.

El código RxJS es el más difícil de entender. La línea del get indica que esa operación devuelve un array de países con la notación en inglés. La línea del pipe indica que al final obtendremos un array de elementos de la clase Pais. El operador map interior es el que realiza el cambio de países en inglés a países en español, y para ello comenzamos teniendo un array de países en inglés que mapeamos al array de paises.

Notemos que tenemos dos map, pero son operadores distintos. El segundo map es el operador de mapeo de la clase array. Hemos decidido ponerle nombre a la función flecha para ser más descriptivos con el proposito.

6.3 Integración con un componente Angular

La integración con un componente se hace considerando que el servicio nos proporciona un observable (stream de datos), por ello, hay que subscribirse al stream para que en el momento en que los datos estén disponibles procesarlos adecuadamente.

Primero hay que emplear *inyección de dependencias* para tener acceso al servicio y por medio de el a las herramientas que proporciona.

```
constructor(private paisesServicio: PaisesService) { }
```

Lo que no se ve en esta imagen es que Angular al iniciar la aplicación creo una instancia de dicha clase, y ahora la está asignando a una propiedad de este componente con el sistema de inyección de dependencias.

En este punto podemos usar el servicio. Hay varias formas de usarlo:

Primera forma

Esta forma es más tradicional. El componente se subscribe al servicio y cuando los datos estén disponibles se asignan a una propiedad accesible por la vista.

```
export class Paises2Component implements OnInit {  
  public paises: Pais[];
```

Hay que subscribirse al servicio para inicializar la variable. Hay que recordar que no se hace en el constructor ya que nunca hay que realizar tareas asíncronas en el constructor. Se hace en el método *ngOnInit*.

```
ngOnInit() {  
  this.paisesServicio.recuperarPaises$()  
    .subscribe(países => this.países = países);  
}
```

En la vista, empleamos la directiva **ngIf* para indicar que solo se use la etiqueta si hay datos, y la directiva **ngFor* para recorrer el arreglo.

```
países2 > <países2.component.html> <ol  
  <li *ngIf="países">  
    <li *ngFor="let país of países">{{país.nombre}}</li>  
  </ol>
```

Segunda forma

Actualmente se propone una forma más abstracta que implica que automáticamente Angular realiza varias tareas.

En el componente hay que declarar una variable que permita desde la vista el acceso al observable. No se hace nada más.


```

    })
    export class PaísesComponent implements OnInit {
        private países$ = this.paísesServicio.recuperarPaíses$();
    }

```

En la vista se implementa el sistema de subscripción y la extracción de datos.

```

países / > países.component.html / > div
<div *ngIf="países$ | async as países">
    <ol>
        <li *ngFor="let país of países">{{país.nombre}}</li>
    </ol>
</div>

```

La directiva **ngIf* realiza la subscripción al emplear el término **async**, y cuando los valores estén disponibles se almacenarán en la variable declarada mediante **as**.

Al estar disponibles los datos se hace lo que proceda, en este caso una lista con un bucle Angular.

6.4 Especificación de tipos en los observables

Indicar los tipos en una operación con observables a través de la tubería puede ser complicado. Analicemos el siguiente código (al final del apartado aparece todo el código junto):

El método `retrieveBooks$` devuelve un observable de un array de libros, lo que provoca que el editor VSCode compruebe que el código del método cumpla este requisito.

```

public retrieveBooks$(): Observable<Book[]> {
    const url = 'http://localhost/controlador.php?operacion=obte

```

TypeScript obliga a describir para el método `get` el tipo devuelto por este método. Solo estamos obligados a describir el contenido que nos interesa, ignorando el resto. Esta operación se realiza empleando el operador diamante entre el nombre del método y sus parámetros.

```

return this.http
    .get<{libros: { clave: string, titulo: string }[]}>(url)
    .pipe(map((libros: { clave: string, titulo: string }[]) => libros.map(libro => new Book(libro.clave, libro.titulo))),

```

El formato de la tubería es más complejo. Empleando el operador diamante, hay que describir el tipo devuelto por cada una de las operaciones incluidas en la tubería. En este caso la tubería incluye tres operaciones, el primer mapeo devuelve un array de objetos, el segundo mapeo devuelve un array de libros, y el retardo devuelve también un array de libros.

```

return this.http
    .get<{libros: { clave: string, titulo: string }[]}>(url)
    .pipe<{ clave: string, titulo: string }[], Book[], Book[]>(
        map(extractBooks),
        map(convertToBooks),
        delay(1000),
    )

```

Pero para que el analizador no de problemas las funciones incluidas en los mapeos deben realizar un tratamiento correcto de tipos, es decir, los tipos indicados en la tubería se correspondan con los que devuelvan estas funciones:

```

const extractBooks =
    (response: {libros: { clave: string, titulo: string }[]}):
        { clave: string, titulo: string }[] =>
        response.libros;

const convertToBook =
    (book: { clave: string, titulo: string }): Book =>
        new Book(parseInt(book.clave), book.titulo);

const convertToBooks =
    (books: { clave: string, titulo: string }[]): Book[] =>
        books.map(convertToBook);

```

A continuación se muestra todo junto, y en el siguiente apartado se realizará una mejora para no repetir tantos tipos:

```

services > books.service.ts > booksService > retrieveBooks$
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { delay, map } from 'rxjs/operators';
import { Book } from '../models/book';

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

  constructor(private http: HttpClient) { }

  public retrieveBooks$(): Observable<Book[]> {
    const url = 'http://localhost/controlador.php?operacion=obtener_libros';

    const extractBooks =
      (response: {libros: { clave: string, titulo: string }[][]}):
      { clave: string, titulo: string }[] =>
        response.libros;

    const convertToBook =
      (book: { clave: string, titulo: string }): Book =>
        new Book(parseInt(book.clave), book.titulo);

    const convertToBooks =
      (books: { clave: string, titulo: string }[][]): Book[] =>
        books.map(convertToBook);

    return this.http
      .get<{libros: { clave: string, titulo: string }[][]}>(url)
      .pipe<{ clave: string, titulo: string }[], Book[], Book[]>(
        map(extractBooks),
        map(convertToBooks),
        delay(1000),
      )
  }
}

```

Mejora para eliminar tipos repetidos y simplificar el código

La mejora consiste en definir dos interfaces describiendo los tipos que estamos repitiendo continuamente, lo que luego genera código más entendible.

```
public retrieveBooks$(): Observable<Book[]> {
  const url = 'http://localhost/controlador.php?operacion=obtener_libros';

  interface BookInterface {
    clave: string, titulo: string
  }

  interface ResponseInterface {
    libros: BookInterface[]
  }

  const extractBooks = (response: ResponseInterface): BookInterface[] =>
    response.libros;

  const convertToBook = (book: BookInterface): Book =>
    new Book(parseInt(book.clave), book.titulo);

  const convertToBooks = (books: BookInterface[]): Book[] =>
    books.map(convertToBook);

  return this.http
    .get<ResponseInterface>(url)
    .pipe<BookInterface[], Book[], Book[]>(
      map(extractBooks),
      map(convertToBooks),
      delay(1000),
    )
}
```

7 Enrutamiento

7.1 Introducción

El sistema de enrutamiento en Angular emplea enlaces. Esto es curioso si consideramos que Angular genera aplicaciones SPA (aplicaciones de una única página), por lo que los enlaces no pueden dirigir a otra página. En Angular, un enlace quita un componente de lo que podríamos llamar un contenedor de componentes y los sustituye por el componente al que apunta el enlace.

Las rutas de Angular siguen el formato moderno en el que no se indica la tecnología empleada y se identifica a los parámetros de la ruta.

Por comparar, un ejemplo es:

- Ruta tradicional: <https://www.paises.edu/paises.php?codigo=es>
- Ruta moderna: <https://www.paises.edu/paises/es>

Ejemplo

Para conocer cómo funciona el sistema de enrutamiento vamos a desarrollar un proyecto en el que incluiremos cuatro componentes que serán los componentes a los que se accederá con cuatro rutas asociadas a cada uno de ellos.

Creemos los componentes asociados a las etiquetas:

- <ficha-inicio>
- <ficha-productos>
- <ficha-servicios>
- <ficha-contacto>

El contenido por defecto es suficiente para el objetivo de la explicación.

Si se consulta el fichero `app.modules.ts` observamos como los cuatro componentes aparecen declarados.

7.2 Contenedor de componentes

El contenedor de componentes es una etiqueta de Angular preparada para funcionar con el sistema de enrutamiento. Esta etiqueta contendrá el componente apuntado por la ruta actualmente activa, la cual es visible en la barra de direcciones.

El contenedor de componentes se corresponde con la etiqueta `<router-outlet>`

Ejemplo

Incluimos la etiqueta `<router-outlet>` como único elemento de la aplicación (en el fichero `app.component.html`).

```
<router-outlet></router-outlet>
```

7.3 Rutas

Para poder trabajar con el sistema de enrutamiento debemos desarrollar un módulo de gestión de rutas. En este módulo asociaremos rutas a componentes, es decir, indicaremos que a una posible cadena de texto que representa una ruta en la barra de direcciones le corresponde un determinado componente de nuestra aplicación.

Ejemplo

Creemos el módulo con el comando Angular correspondiente:

```
s\src\app> ng generate module enrutamiento --module=app --flat
```

El módulo recibe el nombre “enrutamiento”.

La opción `module` es necesaria para indicar qué módulo es padre de este módulo, en este caso el módulo “app”, el módulo por defecto de la aplicación. A partir de este momento, como la aplicación contiene dos módulos, cada vez que se cree un elemento hay que indicar a qué módulo está asociado, ya que todo elemento debe estar asociado a un módulo y por defecto no sabe a cuál de ellos sería.

La opción `flat` es para indicar que no cree un directorio para almacenar los ficheros relacionados con el enrutamiento, por lo tanto se crearan en la carpeta actual.

Si se consulta el fichero `app.modules.ts` observamos como el fichero de enrutamiento aparece declarado.

Editamos el fichero de enrutamiento para declarar las rutas que necesitamos. Debemos incluir el contenido que indicará lo siguiente:

- Un elemento de tipo `Routes`, que es el arreglo de posibles rutas, donde cada ruta es un objeto con dos propiedades, la ruta y el componente asociado a dicha ruta. En puntos posteriores veremos más propiedades.
- Un elemento que importa las rutas. El método `forRoot` indica que las rutas son relativas al ámbito de toda la aplicación.
- Un elemento que exporta el módulo de enrutamiento de Angular.

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterModule, Routes } from '@angular/router';
import { FichaInicioComponent } from '../ficha-inicio/ficha-inicio.component';
import { FichaProductosComponent } from '../ficha-productos/ficha-productos.component';
import { FichaServiciosComponent } from '../ficha-servicios/ficha-servicios.component';
import { FichaContactoComponent } from '../ficha-contacto/ficha-contacto.component';

const rutas: Routes = [
  { path: 'inicio', component: FichaInicioComponent },
  { path: 'productos', component: FichaProductosComponent },
  { path: 'servicios', component: FichaServiciosComponent },
  { path: 'contacto', component: FichaContactoComponent },
];

@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    RouterModule.forRoot(rutas)
  ],
  exports: [
    RouterModule
  ]
})
export class EnrutamientoModule { }

```

7.4 Empleo de las rutas en las plantillas

7.4.1 Rutas sin parámetros

Para emplear las rutas que hemos definido, debemos incluirlas en una etiqueta acompañando al atributo **routerLink**, que es un atributo propio de Angular. Cuando se seleccione dicha etiqueta, en la etiqueta <router-outlet> se cargará el componente asociado a la ruta reemplazando al que hubiera anteriormente.

Ejemplo

Desarrollamos un menú en la aplicación para emplear las cuatro rutas declaradas. En el fichero de la aplicación incluimos el siguiente contenido:

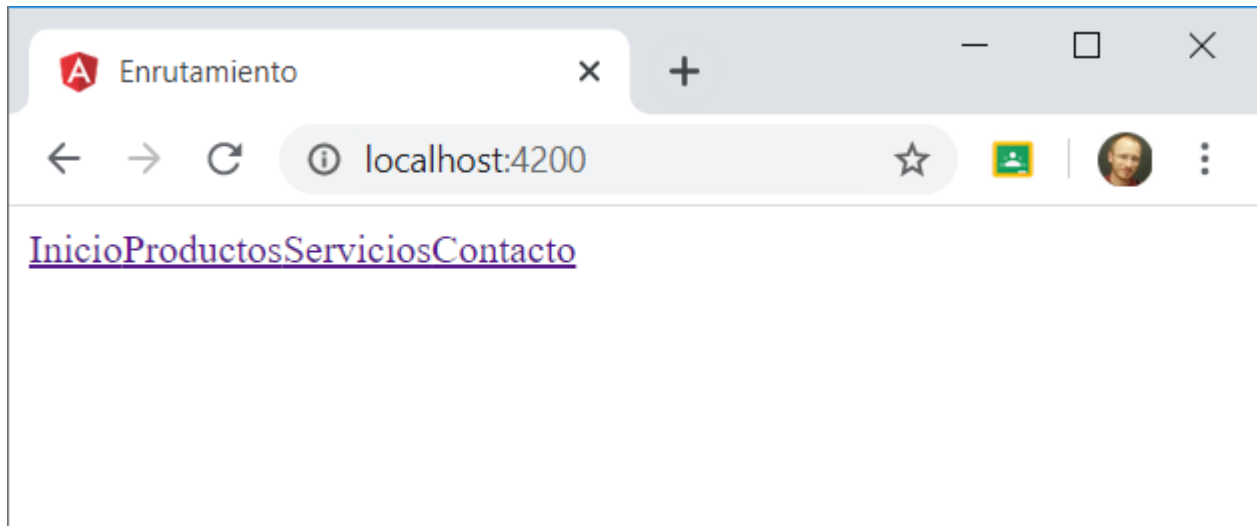
```

<nav>
  <a routerLink="/inicio">Inicio</a>
  <a routerLink="/productos">Productos</a>
  <a routerLink="/servicios">Servicios</a>
  <a routerLink="/contacto">Contacto</a>
</nav>

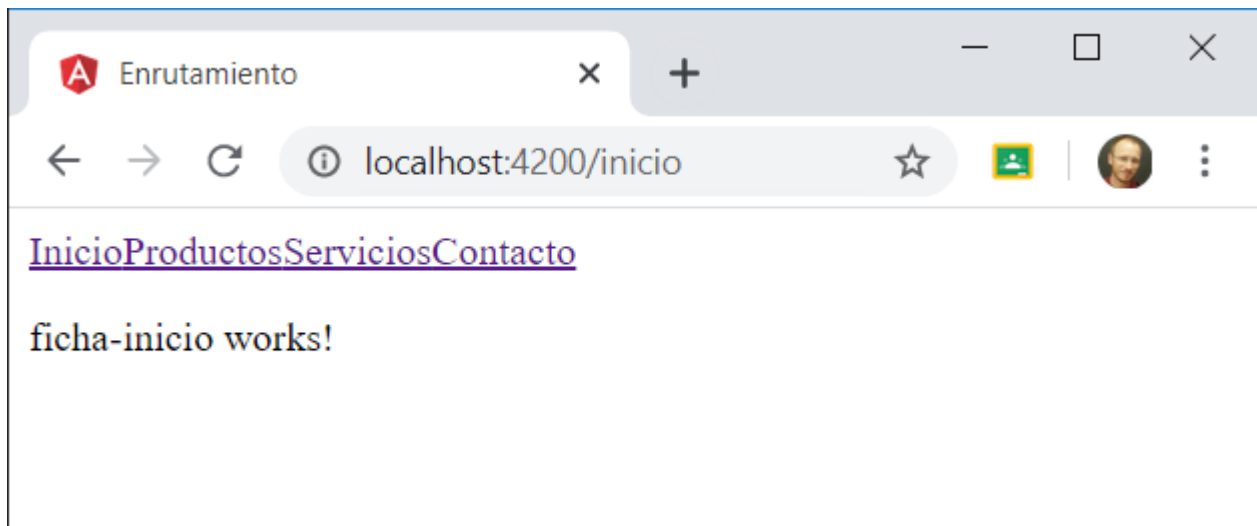
<router-outlet></router-outlet>

```

Al poner en marcha la aplicación debemos observar la página:



Si pulsamos el enlace Inicio debe cambiar la ruta de la barra de direcciones y mostrarse el componente FichaInicioComponent.



Lo mismo debe ocurrir con los otros enlaces.

7.4.2 Ruta por defecto

La ruta por defecto redirigirá la ruta vacía a otra ruta que es la que se desea ver en esa situación. Hay que añadir una nueva ruta en el arreglo de rutas con la configuración adecuada.

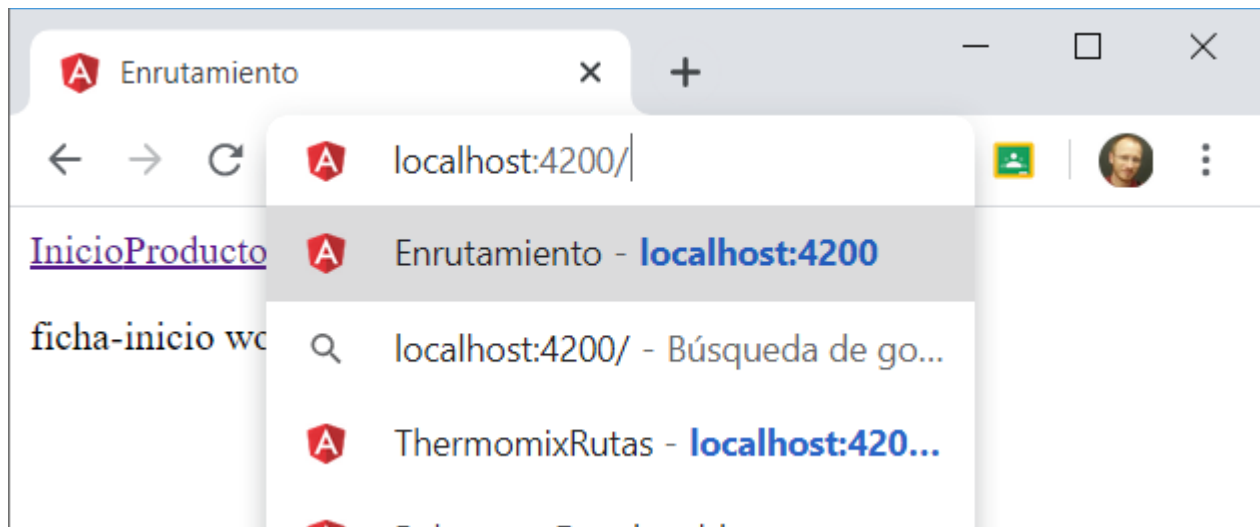
Ejemplo

Modificamos el arreglo de rutas añadiendo la ruta por defecto:

```
const rutas: Routes = [  
  { path: '', redirectTo: '/inicio', pathMatch: 'full'},  
  { path: 'inicio', component: FichaInicioComponent},  
  { path: 'productos', component: FichaProductosComponent}
```

Redirigimos a la ruta “/inicio”. La propiedad pathMatch indica el grado de correspondencia entre ruta de la barra de direcciones y la ruta del arreglo, y es este caso indicamos que debe ser completa.

Si probamos la ruta por defecto, al pulsar “Intro” debe cargarse la página inicio.



7.4.3 Ruta de error 404

Debemos prever que se use una ruta que no exista.

Declaramos una ruta en el arreglo de rutas para el caso de que se intente usar una ruta que no se corresponda con ninguna de las declaradas. Esta ruta debe situarse en la última posición del arreglo.

Ejemplo

Añadimos un nuevo componente que representa un error de acceso a un recurso que no existe. La misma idea se puede aplicar a otro tipo de errores o problemas. Creamos el

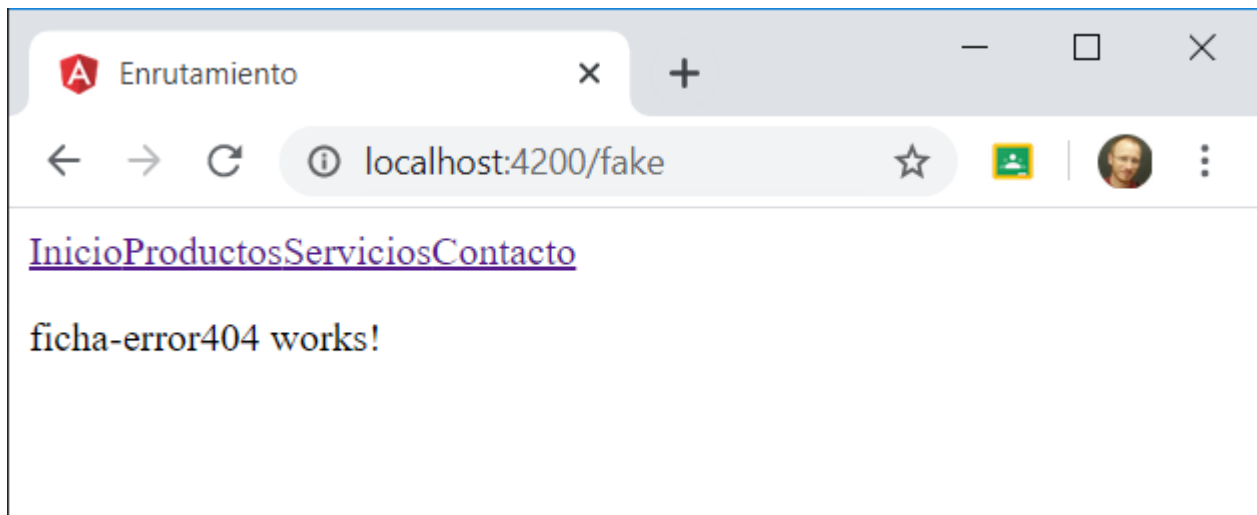
componente `FichaError404Component`. Hay que indicar el módulo al que está relacionado ya que actualmente hay dos módulos.

```
s\src\app> ng generate component ficha-error404 --module=app
```

Añadimos como último elemento del arreglo la ruta de error:

```
{ path: 'contacto', component: FichaContactoComponent },  
{ path: '**', component: FichaError404Component }  
];
```

El sistema de enrutamiento busca la primera ruta que case con la ruta actual. Si intentamos emplear una ruta que no está en el arreglo de rutas, la única que casará será la ruta `"**"`, lo que carga el componente de error.



7.5 Rutas con parámetros

Angular permite suministrar parámetros a las rutas para poder producir nuevos comportamientos. Nos estamos refiriendo a los elementos que en una ruta HTTP tradicional van después de la `?`. Como hemos indicado anteriormente, Angular emplea lo que hemos denominado rutas modernas.

Veremos que los parámetros se añaden en el arreglo de rutas precediéndolos por dos puntos (`:`). En TypeScript hay que conocer el procedimiento para extraer el parámetro y emplearlo de forma adecuada.

Ejemplo

Crea un proyecto Angular denominado Cientificos.

Crea el sistema de enrutamiento como hemos indicado anteriormente. Las rutas a incluir en el desarrollo son:

```
const rutas: Routes = [  
  { path: '', redirectTo: '/cientificos-lista', pathMatch: 'full' },  
  { path: 'cientificos-lista', component: CientificosListaComponent },  
  { path: 'cientifico/:id', component: CientificoComponent },  
];
```

La tercera ruta corresponde a la ruta parametrizada, y el elemento id es el parámetro.

Deben de aparecer errores ya que en tu código aún no existen los componentes necesarios.

Inicialmente, declaramos un menú para la aplicación con el siguiente contenido:

```
<nav>  
  <a routerLink="/cientificos-lista">Lista de científicos</a>  
</nav>  
  
<router-outlet></router-outlet>
```

Crea una clase llamada Cientificos con el siguiente código:

```

export class Cientifico {
  constructor(
    public id: string,
    public nombre: string,
    public areaEstudio: string,
    public nacionalidad: string
  ) {}
}

export class Cientificos {
  private static científicos = [
    new Cientifico('mc', 'Marie Curie', 'Física', 'polaca'),
    new Cientifico('if', 'Ian Fleming', 'Biología', 'inglesa'),
    new Cientifico('ae', 'Albert Einstein', 'Física', 'alemana'),
    new Cientifico('gg', 'Galileo Galilei', 'Física', 'italiana'),
    new Cientifico('in', 'Isaac Newton', 'Física', 'inglesa'),
    new Cientifico('lp', 'Louis Pasteur', 'Biología', 'francesa'),
    new Cientifico('al', 'Ada Lovelace', 'Matemática', 'inglesa'),
    new Cientifico('sr', 'Santiago Ramón y Cajal', 'Biología', 'española'),
    new Cientifico('cd', 'Charles Darwin', 'Biología', 'inglesa'),
    new Cientifico('nt', 'Nicola Tesla', 'Física', 'italiana'),
  ];

  public static recuperarCientificos(): Array<Cientifico> {
    return Cientificos.científicos;
  }

  public static recuperarCientifico(id: string): Cientifico {
    return Cientificos.científicos.find(cientifico => científico.id === id);
  }
}

```

Crea un componente llamado “científicos-lista” para dicha etiqueta. Recuerda indicar el módulo donde debe ser declarado este componente. Este componente recuperará todos los científicos y los mostrará en una lista. La plantilla es:

```

<ul>
  <li *ngFor="let científico of científicos">
    <a routerLink="/cientifico/{{científico.id}}">
      {{científico.nombre}}
    </a>
  </li>
</ul>

```

Observa como la propiedad routerLink recibe como argumento en la ruta el identificador deseado.

El código TypeScript correspondiente es:

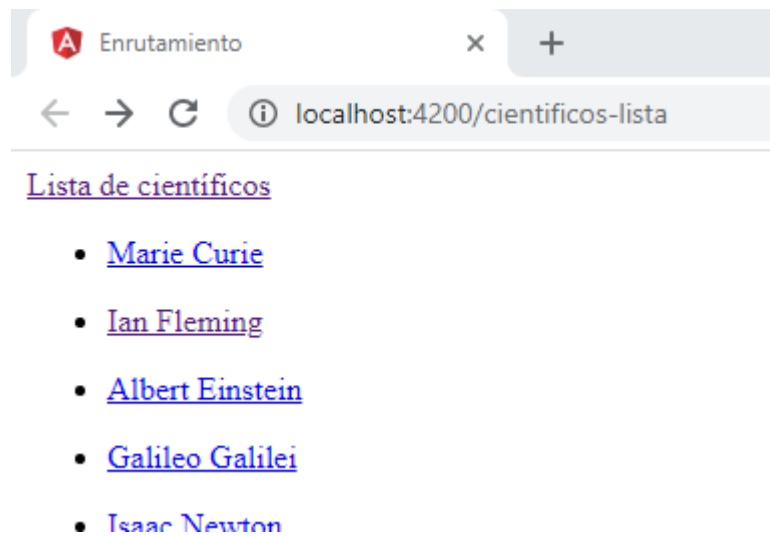
```
import { Component, OnInit } from '@angular/core';
import { Cientificos, Cientifico } from '../cientificos';

@Component({
  selector: 'app-cientificos-lista',
  templateUrl: './cientificos-lista.component.html',
  styles: []
})
export class CientificosListaComponent implements OnInit {
  private cientificos: Array<Cientifico> = Cientificos.recuperarCientificos();

  constructor( ) { }

  ngOnInit() {
  }
}
```

El aspecto de la aplicación en este momento es el siguiente:



Al abrir la aplicación sin indicar un recurso en concreto nos reenvía al componente “cientificos-lista”, como hemos indicado en el arreglo de rutas.

Creamos el componente Cientifico, que mostrará el detalle de un científico. La plantilla debe incluir este código:

```

<div class="envoltura">
  <div>
    <label>Nombre: </label>
    <span>{{cientifico.nombre}}</span>
  </div>
  <div>
    <label>Nacionalidad: </label>
    <span>{{cientifico.nacionalidad}}</span>
  </div>
  <div>
    <label>Área de estudio: </label>
    <span>{{cientifico.areaEstudio}}</span>
  </div>
</div>

```

El código TypeScript es más interesante:

```

import { Component, OnInit } from '@angular/core';
import { Cientifico, Cientificos } from 'src/cientificos';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'cientifico',
  templateUrl: './cientifico.component.html',
  styles: []
})
export class CientificoComponent implements OnInit {
  private científico: Cientifico;

  constructor(
    private ruta: ActivatedRoute
  ) { }

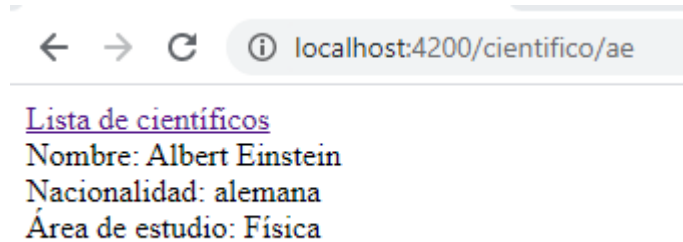
  ngOnInit() {
    const científicoId: string = this.ruta.snapshot.paramMap.get('id');
    this.científico = Cientificos.recuperarCientifico(científicoId);
  }
}

```

Primero nos fijamos en el constructor, donde inyectamos (inyección de dependencias) la ruta activa, es decir, el elemento que permite a este componente recuperar información sobre la ruta que actualmente está cargada en el navegador.

A continuación, en el método de inicialización se usa dicha ruta para extraer el identificador, lo cual permite recuperar la información del científico actual.

Podemos verlo si seleccionamos un científico:



Al seleccionar un científico, en la barra de direcciones podemos ver el routerLink correspondiente, y de él se extrae el identificador para a continuación obtener toda la información del científico.

7.6 Acceso a rutas desde TypeScript

Angular permite el acceso a una ruta en tiempo de ejecución desde TypeScript. Ello nos permite construir la cadena de la ruta de forma personalizada y acceder a ella sin tener que ser incorporada a un enlace.

El módulo de enrutamiento incorpora los elementos para realizar esta tarea.

Ejemplo

Ampliamos nuestro proyecto. El nuevo menú es:

```
<nav>
  <a routerLink="/cientificos-lista">Lista de científicos</a>
  <a routerLink="/cientificos-tabla">Tabla de científicos</a>
</nav>

<router-outlet></router-outlet>
```

Añadimos la correspondiente ruta al arreglo de enrutamiento:

```
const rutas: Routes = [
  { path: '', redirectTo: '/cientificos-lista', pathMatch: 'full' },
  { path: 'cientificos-lista', component: CientificosListaComponent },
  { path: 'cientificos-tabla', component: CientificosTablaComponent },
  { path: 'cientifico/:id', component: CientificoComponent },
];
```

Crea un componente llamado “científicos-tabla” para dicha etiqueta. Recuerda indicar el módulo donde debe ser declarado este componente. Este componente recuperará todos los científicos y los mostrará en una tabla. La plantilla es:

```
<table border="1">
  <tr *ngFor="let científico of científicos">
    <td (click)="mostrarCientifico(científico.id)">
      {{científico.nombre}}
    </td>
  </tr>
</table>
```

Observa que no aparece la propiedad routerLink. El motivo es que desde TypeScript se creará la ruta correspondiente cuando el usuario cliquee sobre un científico. Este código se implementará en el método mostrarCientifico que como vemos recibe como argumento el identificador de científico.

El código TypeScript correspondiente es:

```
import { Component, OnInit } from '@angular/core';
import { Cientifico, Cientificos } from 'src/cientificos';
import { Router } from '@angular/router';

@Component({
  selector: 'cientificos-tabla',
  templateUrl: './cientificos-tabla.component.html',
  styleUrls: ['./cientificos-tabla.component.css']
})
export class CientificosTablaComponent implements OnInit {
  private científicos: Array<Cientifico> = Cientificos.recuperarCientificos();

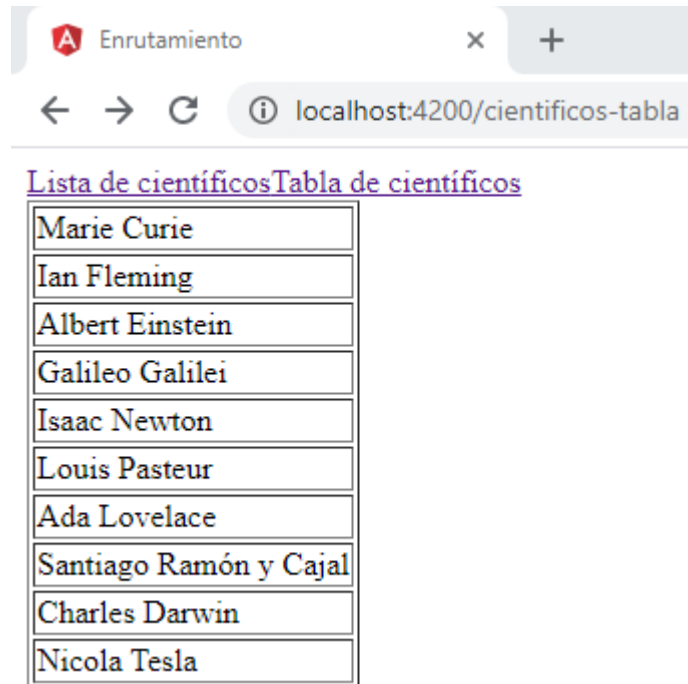
  constructor(private enrutador: Router) { }

  ngOnInit() {
  }

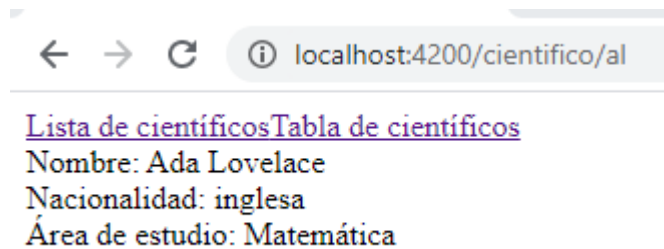
  private mostrarCientifico(id: string): void {
    this.enrutador.navigate(['/cientifico/', id]);
  }
}
```

El método mostrarCientifico emplea un objeto de la clase enrutador para dirigirse a una determinada ruta, cuya cadena de conexión se construye concatenando los elementos incluidos en el arreglo que recibe como argumento el método navigate, que es quien ordena que se busque la ruta y se cargue el componente asociado.

El aspecto de la aplicación en este momento es el siguiente:



Al clicar en un científico aparece:



En la barra de direcciones podemos ver la ruta calculada en el callback del evento, además del hecho de que se ha cargado el componente asociado a dicha ruta.