



Table of Contents

1. Introducción	1
2. Características	3
3. Single Page Applications (SPAs)	4
4. Lenguaje	7
5. Requisitos básicos	8
6. Arquitectura	9
6.1. Módulos	9
6.2. Componentes	9
6.3. Plantillas	9
6.4. Metadatos	10
6.5. Data Binding	10
6.6. Directivas	10
6.7. Servicios	10
6.8. Inyección de dependencias	10
7. Angular CLI	11
7.1. Instalación	12
7.2. Creación de contenido	13
7.3. Ejecución	14
7.4. Producción	15
7.5. Generar Elementos de Angular	16
7.6. Testing	18
7.7. Añadir librerías externas	19
7.8. Lab: Añadir librerías externas (bootstrap)	20
7.9. Actualizar proyecto y librerías	23
7.10. Hot Module Replacement	24
8. Componentes	25
8.1. Template	26
8.2. Class	27
8.3. Decorator	28
8.4. Ciclo de vida	29
9. Data Binding	30
9.1. String Interpolation	31
9.2. Lab: String Interpolation	32
9.3. Property Binding	36
9.4. Lab: Property Binding	37
9.5. Event Binding	40
9.6. Lab: Event Binding	42
9.7. Two-Way Data Binding	46

9.8. Lab: Two-Way Data Binding	48
9.9. Lab: ¿Cómo funciona internamente el 2-Way Data Binding?	53
10. Variables de plantilla (Referencias)	56
10.1. Lab: Variables de plantilla (Referencias)	57
11. Decorador @Input()	63
11.1. Lab: Decorador @Input()	67
12. Decorador @Output()	74
12.1. Lab: Decorador @Output()	75
13. Directivas	80
13.1. Directivas de atributo	81
13.2. Crear una directiva de atributo	84
13.3. Lab: Crear una directiva de atributo	85
13.4. Directivas estructurales	92
13.5. @if	93
13.6. @if con @else	94
13.7. Lab: ngIf	95
13.8. @for	97
13.9. Lab: ngFor	99
13.10. @switch	105
13.11. Lab: ngSwitch	106
14. Pipes	111
14.1. Pipe uppercase	113
14.2. Pipe lowercase	114
14.3. Pipe titlecase	115
14.4. Pipe currency	116
14.5. Pipe date	117
14.6. Pipe slice	118
14.7. Pipe json	119
14.8. Lab: Pipes	120
14.9. Crear un pipe	124
14.10. Lab: Crear custom pipes	126
14.11. Pipes puros e impuros	134
14.12. Lab: Pipes puros e impuros	135
14.13. Pipe async	145
15. Formularios	146
15.1. Formularios de plantilla (FormsModule)	147
15.2. Lab: Formularios (FormsModule)	149
15.3. Formularios reactivos (ReactiveFormsModule)	164
15.4. Lab: Formularios reactivos (ReactiveFormsModule)	166
15.5. Crear una validación personalizada	180
15.6. Validación de campos cruzados	181

15.7. Lab: Crear validaciones	182
16. Servicios e inyección de dependencias.....	191
16.1. Lab: Servicios e inyección de dependencias	194
16.2. Comunicar componentes mediante servicios	203
16.3. Lab: Comunicar componentes mediante servicios	205
17. Observables (RxJS).....	210
17.1. Suscripciones	212
17.2. Operadores	215
17.3. Lab: Observables	217
18. Peticiones Http	238
18.1. HttpClient	239
18.2. Lab: HttpClient	241
19. Angular Router	256
19.1. Definición de rutas.....	257
19.2. Componente router-outlet	258
19.3. Directiva routerLink	259
19.4. Navegación por código	260
19.5. Ruta comodín	261
19.6. Rutas con parámetros	262
19.7. Rutas con query params	263
19.8. Rutas anidadas	264
19.9. Guards.....	265
19.10. Lab: Angular Router	266

Chapter 1. Introducción

El 20 de Octubre de 2010 nace AngularJS, un framework JavaScript de código abierto respaldado por Google, ayuda con la construcción de las Single Page Applications o aplicaciones de una sola página. El patrón Single Page Applications define que podemos construir o desarrollar aplicaciones web en una única página html, teniendo todo el ciclo de vida seleccionado en dicha página, y variando los componentes y controles con códigos JavaScript y las librerías o frameworks como AngularJS.

Aparte, también es adecuado seguir el patrón Modelo Vista Controlador (MVC), que muchos otros frameworks de desarrollo lo programaban en el lado servidor, pero que con AngularJS se hace factible desarrollar en el lado cliente.

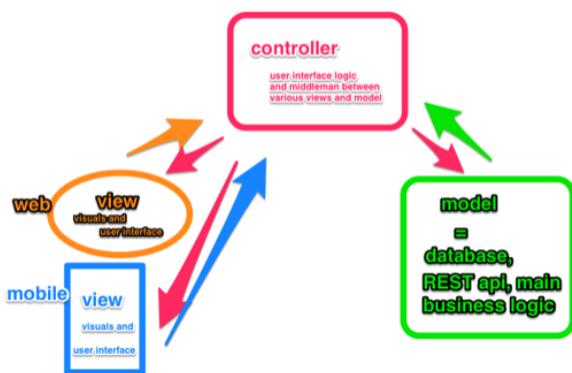


Figure 1. Model View Controller flow

Angular 2 fue anunciado en la *ng-Europe* conference 2014, causando un revuelo entre los desarrolladores ya que fue rediseñado por completo, trayendo bastantes mejoras. Finalmente el 14 de septiembre de 2016 lanzaron su primera versión estable.

Una aplicación en Angular 2 funciona en dispositivos móviles y de escritorio gracias a que es un Framework cross-platform, se maneja un desarrollo basado en modelo vista controlador (MVC) y la ejecución se lleva al lado del cliente (client-side) haciendo que dependa en gran medida del navegador del usuario.



Figure 2. Uso de Angular, fuente: <https://libscore.com/#angular>

Las últimas grandes actualizaciones en Angular han sido en las versiones 14 a la 18 en las que entre otras cosas, han empezado a migrar los módulos a los componentes standalone y han cambiado la sintaxis de las plantillas de las directivas ngIf, ngFor...

Chapter 2. Características

- **Desarrollo Móvil:** El desarrollo de aplicaciones de escritorio es mucho más fácil cuando primero se manejan los problemas de rendimiento en el desarrollo móvil.
- **Modularidad:** Para desarrollar una nueva funcionalidad esta se empaqueta en un módulo, produciendo un núcleo más ligero y más rápido.
- **Compatibilidad:** Es compatible con los navegadores más modernos y recientes.

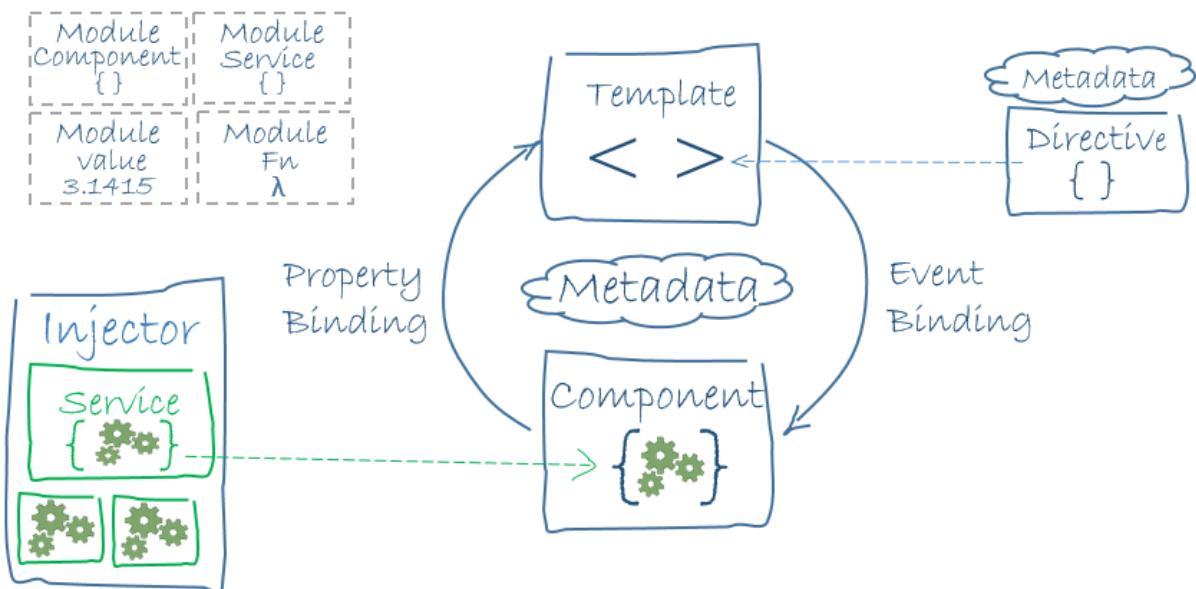


Figure 3. Overview

Chapter 3. Single Page Applications (SPAs)

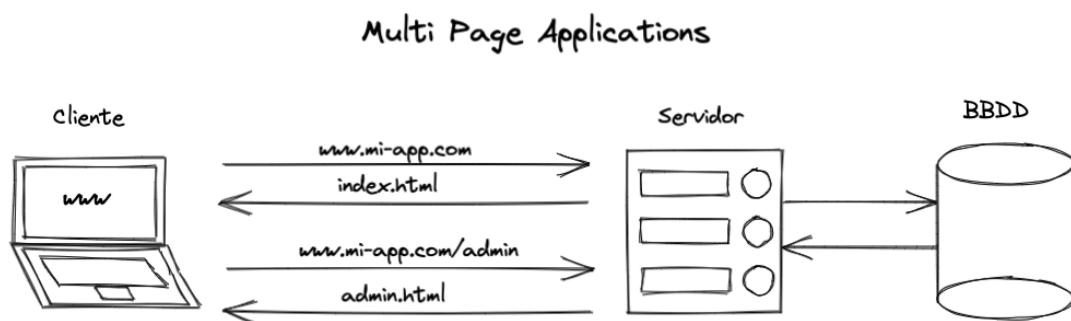
Con Angular estaremos creando aplicaciones que se conocen como Single Page Applications o las siglas SPAs. Estas aplicaciones son aplicaciones que solo tienen una única página de HTML para todo el proyecto.

Pero ¿cómo es posible esto? ¿Y si mi aplicación tiene que tener más de una página?

Bueno, vamos a retroceder un poco en el tiempo.

Antiguamente, antes de que empezasen a aparecer los primeros frameworks y librerías para el frontend enfocados a SPAs como es el caso de Angular, React o Vue, las aplicaciones tenían múltiples páginas.

Es decir, cada vez que cambiáramos de ruta en nuestra aplicación se realizaba una petición al servidor para descargarnos la página de HTML asociada a dicha ruta, por ejemplo, al entrar en la ruta inicial, ibamos a descargarnos el index.html. Mientras que si entrabamos a una url terminada en /admin, ibamos a descargarnos un admin.html.



Estas páginas pueden ser archivos estáticos o archivos que se generan dinámicamente en el servidor a partir de una serie de datos provenientes de la BBDD o cualquier otro sitio y que se inyectan dentro de una plantilla de HTML con alguna librería como Pug, Jade, JSP, Erb...

Este tipo de funcionamiento era viable antiguamente cuando estas páginas digamos que eran poco interactivas, más bien se usaban para mostrar información.

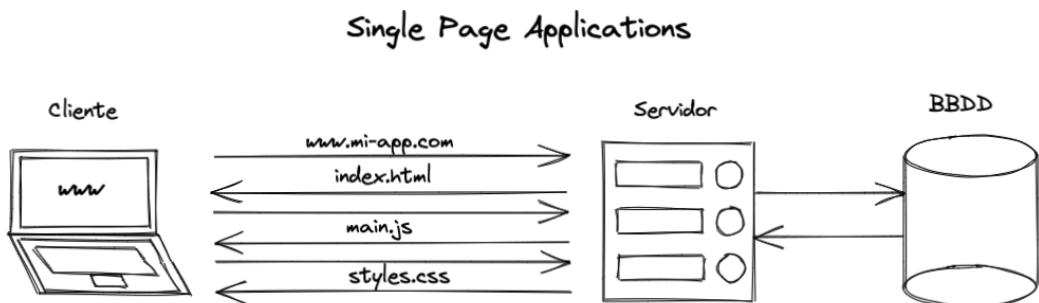
Pero con el paso del tiempo, la tecnología va evolucionando y estas aplicaciones empiezan a tener mucha más funcionalidad, por lo que se empiezan a añadir bastantes archivos de JS para añadirlas.

También se empieza a poner el foco en que estas sean muy bonitas y fáciles de usar para mejorar la experiencia de los usuarios y que estos nos elijan frente a nuestros competidores, por lo que los archivos de CSS empiezan a crecer también.

Y todo esto provoca que los servidores se empiecen a saturar con muchas peticiones, ya que donde antes se descargaban una página de HTML, un JS y un CSS, ahora probablemente se tengan que

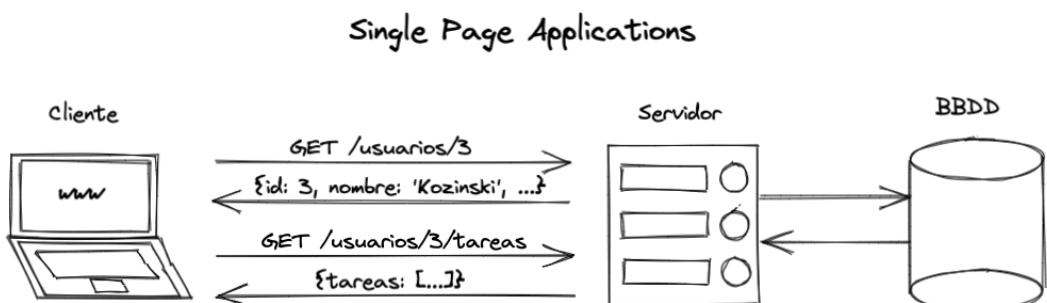
descargar varios de estos últimos. También tienen que gestionar el orden en el que todos estos archivos se van descargando, lo que puede darnos verdaderos quebraderos de cabeza.

Viendo que el servidor empieza a encargarse de demasiadas tareas, surge la idea de las Single Page Apps. Aplicaciones que solo van a tener una página de HTML, unos pocos archivos de JS y otros pocos de CSS, pero que todo esto se va a descargar de una vez cuando se entre a las páginas.



De esta forma conseguimos quitarle carga al servidor, ya que ahora solo va a recibir unas pocas peticiones por cliente para descargarse estos archivos.

A partir de aquí se encargará de gestionar las peticiones que le lleguen para devolver los datos necesarios para pintar en estas aplicaciones. Normalmente, datos en formato JSON quepesan mucho menos que una página de HTML con estos datos incluidos más sus JS y sus CSS.



Por tanto con este tipo de aplicaciones, le quitamos bastante carga a los servidores para darsela al cliente, ya que al descargarse todos los archivos indicados anteriormente al entrar en la aplicación, ya tiene toda la estructura, estilos y funcionalidad de la aplicación, faltandole únicamente los datos necesarios para pintar las vistas.

Pero si solo tenemos una página, ¿qué pasa si necesito tener distintas rutas/urls?

No hay problema, las podemos tener, solo que ahora en lugar de pedir otra página de HTML al servidor al cambiar la ruta, lo que se va a cambiar es el contenido de la página HTML que descargamos al principio. Vamos a ir pintando y eliminando de nuestro index.html los distintos componentes que habremos creado para construir la aplicación.

Por ejemplo, ahora podremos tener un componente **Home** que se pintará al entrar en la ruta inicial de la aplicación, y otro componente **Admin** que será el que se va a pintar cuando entremos a la ruta **/admin** de la aplicación.

En el caso de angular, es **@angular/router** en encargado de gestionar este tipo de comportamiento.

Chapter 4. Lenguaje

La lenta evolución de JavaScript parece haber salido de su hibernación. En el último año disfrutamos ya de las mejoras de ES6 (ES2015) y empezamos a probar ES7 (2016).



Figure 4. TypeScript vs ES

Angular 2 recomienda usar TypeScript un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft, considerado como un superconjunto de JavaScript actualizado, que esencialmente añade tipado estático y objetos basados en clases.

TypeScript no es ni mucho menos obligatorio. Se puede desarrollar en ES5 y ES6 sin problemas. Pero el conjunto de herramientas, la recomendación de la plataforma y la gran ventaja de la programación orientada a objetos, hace de TypeScript la mejor elección.

Chapter 5. Requisitos básicos

Para poder empezar con Angular dos vamos a necesitar Node v5.x.x y su manejador de paquetes npm v3.x.x. Desde la página de npm tenemos un tutorial sobre cómo [descargar e instalar todo](#).

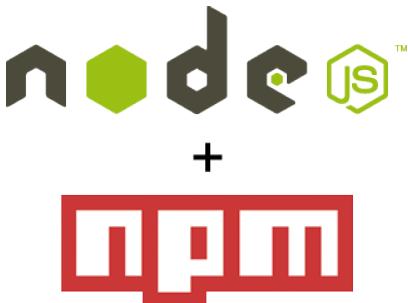


Figure 5. Node y npm

Editores

Podemos usar cualquier editor de texto, aunque es recomendable alguno que nos de soporte al lenguaje que utilicemos

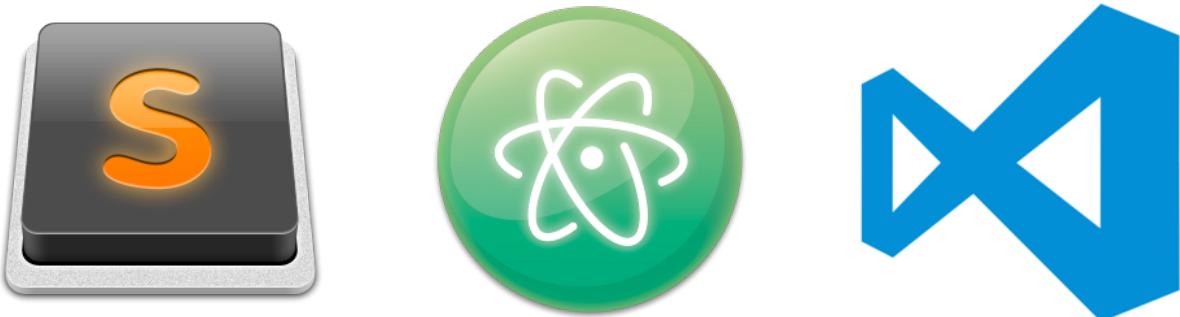


Figure 6. por orden, Sublime, Atom y VSCode

Aunque actualmente el mejor IDE es WebStorm, de pago y alrededor de 100€ al año.



Figure 7. WebStorm

Para el curso podemos usar [Sublime](#) con su extensión oficial de Microsoft para [TypeScript](#) o [VSCode](#) instalando varias extensiones para programar con Angular 2.

Chapter 6. Arquitectura

Angular 2 es un framework para construir aplicaciones cliente usando HTML y JavaScript, o cualquier lenguaje que se pueda compilar a JavaScript como TypeScript. El framework se compone de un conjunto de librerías (algunas pertenecientes al núcleo y otras opcionales).

La arquitectura de Angular 2 se compone de 8 bloques que veremos a continuación.

6.1. Módulos

Las aplicaciones en Angular 2 son modulares y estos modulos se conocen como **NgModules**. Todas las aplicaciones de Angular tienen un módulo, el módulo raíz que se llama **AppModule** por convención.

Un módulo es una clase con el decorador **@NgModule**. Los decoradores son funciones que modifican las clases JavaScript. Y Angular tiene muchos decoradores que añaden metadatos a las clases.

Las propiedades más importantes de NgModules son:

- **declarations**: aquí se añaden las clases de vista que pertenecen al modulo, como los componentes, las directivas y los pipes.
- **exports**: aquí se añaden las declaraciones que deben de ser visibles y utilizables por componentes de otros módulos.
- **imports**: aquí se añaden los módulos necesarios para usar en los componentes declarados en este módulo.
- **providers**: aquí se añaden los servicios necesarios, y estos se vuelven accesibles en toda la aplicación.
- **bootstrap**: aquí se pone cual es la vista principal de la aplicación.



Los módulos los están quitando desde las últimas versiones para pasar a utilizar los **standalone components**, donde cada componente sería un propio módulo.

6.2. Componentes

Un componente controla una parte de lo que se muestra en la pantalla. El componente interactúa con la vista a través de propiedades y métodos, en los que se define la lógica.

6.3. Plantillas

Las plantillas son fragmentos HTML que indican a Angular como representar un componente. También pueden tener etiquetas especiales de Angular 2.

6.4. Metadatos

Los metadatos contienen información que Angular necesita para procesar una clase. Por ejemplo, en un componente, estos metadatos indican que archivo contiene el la plantilla, donde se encuentran los estilos, cual es el nombre de la etiqueta que hay que usar para poder mostrar el componente...

6.5. Data Binding

El data binding es el enlace de información que hay entre la plantilla y el componente que nos permite usar en la plantilla los datos que hay en el componente. Es importante para comunicar estos elementos. Hay cuatro formas en las que se produce el data binding:

- **string interpolation**
- **property binding**
- **event binding**
- **two-way data binding**

6.6. Directivas

Las directivas se encargan de transformar el DOM, hacen que las plantillas de Angular sean dinámicas. Son como componentes pero con menos funcionalidad.

Hay dos tipos de directivas:

- **Directivas estructurales**
- **Directivas de atributos**

6.7. Servicios

Un servicio puede ser cualquier cosa, valores, funciones... Son clases que se encargan de algo en concreto. Los componentes son grandes consumidores de servicios.

Los componentes deben de estar lo más limpios posibles, y para ello usan los servicios, por ejemplo, para mostrar mensajes en consola, realizar peticiones al servidor... El componente solo se tiene que encargar de pedir los datos a los servicios para luego usarlos en las plantillas. Cuantos mas servicios tengamos, mejor estará organizado nuestro código.

6.8. Inyección de dependencias

La inyección de dependencias se usa para proporcionar una nueva instancia de una clase en un elemento como un componente o un servicio. La mayoría de las dependencias son servicios.

Chapter 7. Angular CLI



Figure 8. Angular cli

Angular CLI es el intérprete de línea de comandos de Angular 2, facilita el inicio de proyectos y la creación del esqueleto, o scaffolding, de la mayoría de los componentes de una aplicación Angular 2.

Angular CLI no es una herramienta de terceros, sino que nos la ofrece el propio equipo de Angular. En resumen, nos facilita mucho el proceso de inicio de cualquier aplicación con Angular, ya que en pocos minutos te ofrece el esqueleto de archivos y carpetas que vas a necesitar, junto con una cantidad de herramientas ya configuradas. Además, durante la etapa de desarrollo nos ofrecerá muchas ayudas, generando el "scaffolding" de muchos de los componentes de una aplicación. Durante la etapa de producción o testing también nos ayudará, permitiendo preparar los archivos que deben ser subidos al servidor, transpilar las fuentes, etc.

7.1. Instalación

Para poder instalarlo necesitamos tener Node y NPM instalados.

La manera mas sencilla de empezar es descargarlos e instalar la distribución para nuestro sistema operativo de [Node](#), que viene ya con su gestor de paquetes NPM.

Para instalar Angular cli, desde la terminal hay que lanzar el siguiente comando:

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

Una vez instalado, desde la línea de comandos podremos usar el comando **ng**.

Para comprobar que todo ha ido bien, ejecutamos:

```
$ ng v
```

7.2. Creación de contenido

Creamos un directorio para nuestra aplicación y ejecutamos

```
$ ng new hola-angular
```

Después de unos segundos tendremos nuestro proyecto creado y listo para ejecutarse.

Angular cli nos crea un proyecto bastante mas grande que si lo hacemos nosotros a mano, además añade test, pero en general es el mismo proyecto de Angular 2.

7.3. Ejecución

Para lanzar y probar tu aplicación necesitas otro comando de Angular-CLI. Este comando se ocupa entre otras cosas de todo el proceso necesario para transformar el código TypeScript en JavaScript reconocible por el navegador. También crea un mini servidor estático y además refresca el navegador a cada cambio los fuentes.

```
$ ng serve  
$ ng s
```

7.4. Producción

Para la fase de producción necesitaremos los archivos del proyecto (componentes, servicios, directivas...) transpilados a JavaScript, minificados para que el proyecto ocupe el menor espacio posible, se encarga de eliminar el código que es inaccesible (aquellos que no se llegan a ejecutar nunca en la aplicación).

Para generar estos archivos, y así preparar la aplicación para subir a producción, lanzaremos el siguiente comando:

```
$ ng build
```

7.5. Generar Elementos de Angular

Podemos generar funcionalidades como componentes, servicios, directivas... en nuestro proyecto usando los comandos que nos proporciona Angular-CLI.

Crear un componente:

```
$ ng generate component nombre-componente  
$ ng g c nombre-componente
```

Crear una directiva

```
$ ng generate directive nombre-directiva  
$ ng g d nombre-directiva
```

Crear un pipe

```
$ ng generate pipe nombre-pipe  
$ ng g p nombre-pipe
```

Crear un servicio:

```
$ ng generate service nombre-servicio  
$ ng g s nombre-servicio
```

Crear una clase

```
$ ng generate class nombre-clase  
$ ng g cl nombre-clase
```

Crear un guard

```
$ ng generate guard nombre-guard  
$ ng g g nombre-guard
```

Crear un interface

```
$ ng generate interface nombre-interface  
$ ng g i nombre-interface
```

Crear un enum

```
$ ng generate enum nombre-enum  
$ ng g e nombre-enum
```

Crear un módulo

```
$ ng generate module nombre-module  
$ ng g m nombre-module
```

A esos comandos les podemos añadir las siguientes opciones:

- **-it**: no genera archivo de plantilla.
- **-is**: no genera archivo de estilos.
- **--flat**: genera los archivos en la carpeta actual en lugar de en una nueva carpeta.
- **--skip-tests**: no genera los archivos de testing (**.spec**).

7.6. Testing

Angular CLI nos permite ejecutar los archivos de testing a través de comandos.

Si queremos ejecutar los **test unitarios**, es decir, todos aquellos que se crean por defecto con la extensión **.spec.ts**, entonces vamos a lanzar el comando:

```
$ ng test
```

Si queremos ejecutar los **test e2e**, es decir, aquellos que se han definido dentro de la carpeta **e2e** o que tienen la extensión **.e2e-spec.ts**, entonces lanzaremos el comando:

```
$ ng e2e
```

7.7. Añadir librerías externas

Para añadir una librería externa a Angular como puede ser *Bootstrap* o *Angular Material*, había que usar *NPM* o *Yarn* para instalarlas y después añadir los archivos fuente instalados en el archivo `angular.json`.

A partir de Angular 6 se puede usar un nuevo comando de Angular CLI que nos permite **añadir las librerías externas** (aquellas que están soportadas) instalándolas y añadiendo las referencias necesarias al archivo mencionado antes.

```
$ ng add nombre-libreria
```

7.8. Lab: Añadir librerías externas (bootstrap)

En este laboratorio vamos a ver como añadir la librería de bootstrap en un proyecto de Angular.

Empezaremos creando el proyecto de Angular con el siguiente comando, aceptando los valores por defecto a las preguntas que se nos hacen:

```
$ ng new angular-cli-add-librerias-externas-bootstrap-lab  
? Do you want to enforce stricter type checking and stricter bundle budgets in the workspace?  
  This setting helps improve maintainability and catch bugs ahead of time.  
  For more information, see https://angular.io/strict No  
? Would you like to add Angular routing? No  
? Which stylesheet format would you like to use? CSS
```

Una vez creado el proyecto, vamos a proceder a instalar los paquetes de NPM de bootstrap para poder utilizarlo en el proyecto.

```
$ npm install bootstrap
```

Ahora vamos a añadir algunos elementos de bootstrap en nuestro componente **App** para ver si se aplican los estilos de este framework o no.

/angular-cli-add-librerias-externas-bootstrap-lab/src/app/app.component.html

```
<h1>Proyecto hecho con Bootstrap</h1>  
<button type="button" class="btn btn-primary">Un botón primario de Bootstrap</button>
```

Ahora toca levantar el servidor para poder abrir la aplicación en <http://localhost:4200/>.

```
$ ng s
```

Al entrar a la URL anterior vemos que no se están aplicando los estilos de bootstrap. Esto se debe a que por el momento solo hemos instalado la dependencia pero todavía no la hemos importado en nuestro proyecto.

En este caso, para añadirlo y que se empiece a aplicar en el proyecto vamos a ir al archivo **angular.json**.

Dentro de este archivo nos encontramos un array **styles** (están dentro de la clave **build**) dentro del cual tenemos que poner la ruta a aquellos archivos de estilos que queremos que se añadan durante el proceso de construcción en la aplicación.

Así que añadiremos las rutas a los archivos de distribución minificados del paquete de bootstrap que tenemos en la carpeta de node_modules.

```
{  
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",  
  "version": 1,  
  "newProjectRoot": "projects",  
  "projects": {  
    "angular-cli-add-librerias-externas-bootstrap-lab": {  
      ...  
      "architect": {  
        "build": {  
          "builder": "@angular-devkit/build-angular:browser",  
          "options": {  
            "outputPath": "dist/angular-cli-add-librerias-externas-bootstrap-lab",  
            "index": "src/index.html",  
            "main": "src/main.ts",  
            "polyfills": "src/polyfills.ts",  
            "tsConfig": "tsconfig.app.json",  
            "aot": true,  
            "assets": [  
              "src/favicon.ico",  
              "src/assets"  
            ],  
            "styles": [  
              "node_modules/bootstrap/dist/css/bootstrap.min.css",  
              "src/styles.css"  
            ],  
            "scripts": []  
          },  
          ...  
        }  
      }  
    }  
  }  
}
```

Una vez guardado el archivo anterior con los cambios aplicados, vamos a reiniciar el servidor de desarrollo para que coja estos últimos cambios, y al volver a entrar en la aplicación, ya deberíamos de ver los estilos de bootstrap aplicados sobre los elementos que habíamos añadido.

Esta era la primera forma de poder añadir este tipo de librerías o frameworks externos en nuestros proyectos de Angular.

A partir de la versión 6 de Angular, se ha añadido un comando nuevo **ng add** que permite añadir algunas de estas librerías automáticamente a los proyectos de Angular, y al hacerlo así se configura todo automáticamente sin necesidad de que nosotros tengamos que modificar e instalar las dependencias manualmente.

Para probar esta otra forma, vamos a deshacer lo que hemos hecho. Tendremos que quitar la dependencia de **bootstrap** del **package.json** y deshacer los cambios que hemos realizado sobre el archivo **angular.json**.

Una vez deshechos los cambios, vamos a añadir a nuestro proyecto la librería de **ng-bootstrap**, pero esta vez, usando el comando **ng add** para que todos los cambios que hemos realizado

anteriormente se hagan de forma automática y se quede todo configurado solo con el siguiente comando:

```
$ ng add @ng-bootstrap/ng-bootstrap
```

Una vez ejecutado el anterior comando, ya deberíamos de volver a tener los estilos de bootstrap en nuestra aplicación.

Podemos mirar de nuevo el archivo **angular.json** para comprobar que se ha añadido la ruta a los estilos de bootstrap. También podremos ver que en el archivo **package.json** ahora tenemos **bootstrap** de nuevo en la sección de las dependencias.

7.9. Actualizar proyecto y librerías

Además del comando anterior, se ha añadido otro más que se encarga de analizar el `package.json` en busca de paquetes que no estén actualizados a la última versión disponible. Una vez lanzado y analizado dicho archivo, nos va a mostrar una lista con aquellos paquetes que deberíamos de actualizar y nos mostrará los comandos para ello.

```
$ ng update
```

```
----
```

A la hora de actualizar de versión las aplicaciones de Angular, nos puede venir bien echar un ojo a la página <https://update.angular.io/> en la que al llenar la información que se nos pide sobre el proyecto, nos muestra una serie de pasos que debemos de seguir para que la actualización salga bien.

7.10. Hot Module Replacement

En la versión 11 de Angular se ha añadido una nueva opción a la hora de levantar el servidor de desarrollo. El **Hot Module Replacement** o **HMR**.

Esta funcionalidad viene dada por Webpack (el module bundler que utiliza Angular internamente) y nos permite actualizar el proyecto de angular (modificar archivos de los elementos del proyecto) sin tener que recompilar y refrescar el proyecto entero con lo que perderíamos los cambios realizados hasta ese momento.

Al activar esta opción solo se refrescará aquella parte de la aplicación correspondiente a los archivos que hemos modificado dentro del proyecto.

```
$ ng serve --hmr
```

Chapter 8. Componentes

Los componentes son los bloques de construcción de Angular 2 que representan regiones de la pantalla. Las aplicaciones en Angular 2 se desarrollan en base a componentes. En lugar de tener un árbol de etiquetas como tenemos en las páginas web, ahora tendremos un árbol de componentes que cuelgan de un componente padre. De un componente pueden colgar uno o más componentes.

Los componentes a parte de encapsular contenido (como hacen las etiquetas), también encapsulan alguna funcionalidad. Podriamos decir que los componentes son piezas de negocio.

Un componente consta de las siguientes tres partes fundamentales:

- Un template
- Una clase
- Un decorator

A la hora de crear un componente, tendremos que importarlo en aquellos componentes donde se vaya a utilizar, dentro de la propiedad **imports** que tiene el decorador.

8.1. Template

El template o plantilla representa la vista (capa V del MVC) que se escribe con HTML.

```
<h1>Mi componente</h1>
```

Se puede definir el template en una linea en lugar de definirlo en un archivo externo. Solo hay que indicarselo en el *decorator* del componente. También lo podemos hacer con los estilos.

```
// Con archivos externos para el template y los estilos
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

// Con el template y los estilos en linea
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  template: `
    <h1>Mi componente</h1>
  `,
  styles: [
    h1 {
      color: blue;
    }
  ]
})
```

8.2. Class

La clase de un componente se corresponde con el controlador. Es donde inicializamos y definimos el estado de los componentes. Aquí también se define el comportamiento de los componentes en forma de funciones, las cuales se asignan a los eventos del template.

```
export class AppComponent {  
  title = 'app works!';  
  
  constructor() {}  
  
  onClick() {  
    // ...  
  }  
}
```

8.3. Decorator

Un **decorator** es una herramienta que sirve para extender una función con mediante otra función, pero sin tocar la original que se está extendiendo. Angular 2 usa los decoradores para registrar los componentes, añadiéndoles información para que sean reconocidos en otras partes de la aplicación.

```
// Con archivos externos para el template y los estilos
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

En el decorator anterior podemos distinguir las tres propiedades que le está agregando al componente. Estas propiedades describen el componente creado. Las propiedades son:

- **selector**: es el nombre de la etiqueta que se crea cuando se procesa el componente. Para mostrar el componente tenemos que llamar a la etiqueta `<app-root></app-root>` en el lugar del HTML donde queremos mostrarlo. El nombre del selector tiene que ser único en la aplicación.
- **standalone**: con esto indicamos si estamos utilizando un componente standalone o un componente antiguo.
- **imports**: si el componente es standalone, aquí añadimos aquellos elementos que queremos utilizar en el componente que antiguamente se añadían en los módulos (componentes, directivas y pipes).
- **templateUrl**: indica donde se encuentra la plantilla o template del componente.
- **styleUrls**: indica los archivos de estilos que se le van a aplicar a este componente.

8.4. Ciclo de vida

Los componentes tienen un ciclo de vida que maneja Angular 2 usando los **hooks**. Los hooks del ciclo de vida nos permiten interactuar con los componente en momentos claves del ciclo de vida, como por ejemplo justo cuando se ha inicializado un componente, cuando este sufre cambios en alguna de sus propiedades o cuando va a ser eliminado.

Estos hooks son interfaces, las cuales tienen un método llamado como la propia interface precedido de `ng`.

A continuación vienen todos los hooks del ciclo de vida de un componente en el orden en que ocurren:

- **ngOnChanges**: se ejecuta cuando hay un cambio en el valor de alguna propiedad.
- **ngOnInit**: se lanza cuando se han inicializado todas las propiedades del componente, por lo que el componente también se ha inicializado.
- **ngDoCheck**: se llama en cada detección de cambios, justo después del `ngOnchanges` y `ngOnInit`.
- **ngAfterContentInit**: se ejecuta cuando se inserta contenido externo al componente en dicho componente (`<ng-content></ng-content>`).
- **ngAfterContentChecked**: cuando hay un cambio en el contenido insertado con `ng-content`.
- **ngAfterViewInit**: se lanza cuando se ha inicializado la vista del componente y las vistas de sus componente hijos.
- **ngAfterViewChecked**: se lanza despues de checkear las vistas del componente y sus hijos, es decir, justo despues del `ngAfterViewInit`.
- **ngOnDestroy**: se ejecuta justo antes de destruir un componente.

```
import { Component, OnInit } from '@angular/core';
import { Item } from '../item';
import { ItemService } from '../item.service';
// ...
export class ItemListComponent implements OnInit {
  items: Item[] = [];

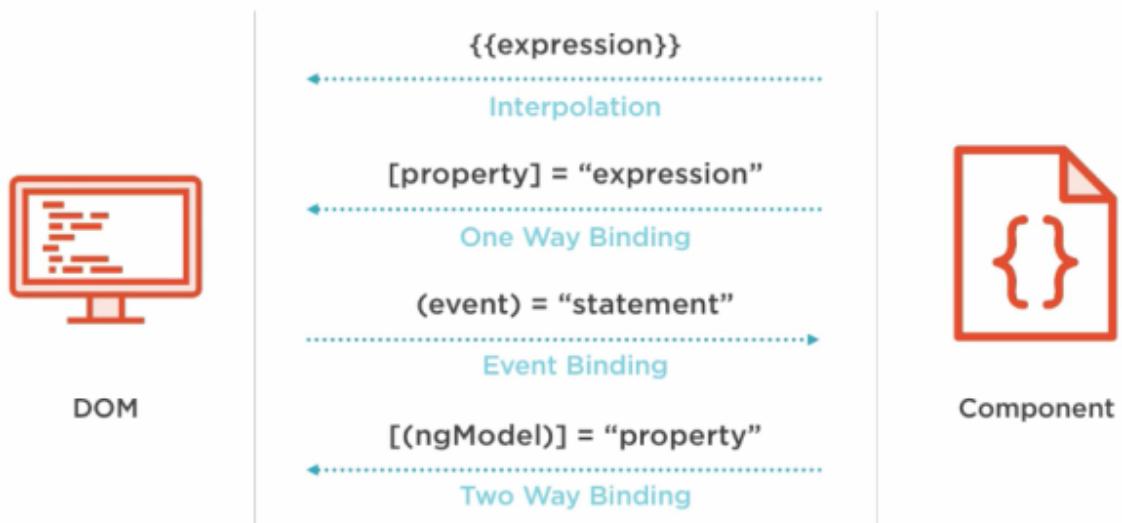
  constructor(private itemService: ItemService) { }

  ngOnInit() {
    this.items = this.itemService.getItems();
  }
}
```

Chapter 9. Data Binding

El **data binding** consiste en la sincronización entre los datos del componente y la vista. En Angular nos encontramos cuatro formas distintas de controlar el flujo en el que se mueven los datos:

- String Interpolation
- Property Binding
- Event Binding
- Two-Way Binding



9.1. String Interpolation

El **String Interpolation** se usa para renderizar el valor de una variable en las plantillas de los componentes.

Los datos que se usan son solo de lectura, es decir, no podemos modificarlos directamente dentro del String Interpolation.

Se usa con **{} nombreVariable {}**. También podemos meter entre las dobles llaves, llamadas a funciones o expresiones de JS/TS como un operador ternario.

/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  nombre: string = 'Sara'
}
```

/src/app/app.component.html

```
<h1>Mis datos</h1>
<p>Mi nombre es {{ nombre }}.</p> <!-- Mi nombre es Sara. -->
```

9.2. Lab: String Interpolation

En este laboratorio vamos a ver varias formas de utilizar el **String Interpolation** para mostrar el valor de una propiedad en nuestros componentes.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-data-binding-string-interpolation-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? (y/N) N
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-data-binding-string-interpolation-lab  
$ ng s
```

Vamos a empezar por declarar tres propiedades en el componente App, cada una de ellas será el título de una serie.

/angular-data-binding-string-interpolation-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  serie1: string = 'Gangland Undercover'
  serie2: string = 'Taboo'
  serie3: string = 'The Walking Dead'
}
```

Vamos a mostrar la primera propiedad en la plantilla del componente App. Por lo que añadiremos **serie1** entre las dobles llaves para usar el String Interpolation y que Angular pueda acceder al valor de esta para mostrarlo.

/angular-data-binding-property-binding-lab/src/app/app.component.html

```
<p>Serie 1: {{serie1}}</p>
```

Una vez guardados los cambios, en <http://localhost:4200> debemos de ver el párrafo que muestra el nombre de la primera serie.

Dentro del String Interpolation podemos utilizar expresiones, por lo que esta vez, la segunda serie se va a mostrar porque, dentro del String Interpolation, llamaremos a una función que va a retornar su valor.

Vamos a empezar declarando una función **getSerie2** dentro del archivo de TypeScript.

/angular-data-binding-string-interpolation-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  serie1: string = 'Gangland Undercover'
  serie2: string = 'Taboo'
  serie3: string = 'The Walking Dead'

  getSerie2(): string {
    return this.serie2;
  }
}
```

Y el siguiente paso será llamar entre las dobles llaves a esta función, en la plantilla del componente App.

/angular-data-binding-property-binding-lab/src/app/app.component.html

```
<p>Serie 1: {{serie1}}</p>
<p>Serie 2: {{getSerie2()}}</p>
```

Para mostrar la tercera serie vamos a utilizar un operador ternario (?), en el que comprobaremos que si **es distinto de un string vacío, null o undefined**, entonces se va a mostrar su valor, y si no existe, entonces se mostrará el texto "No hay serie 3".

/angular-data-binding-property-binding-lab/src/app/app.component.html

```
<p>Serie 1: {{serie1}}</p>
<p>Serie 2: {{getSerie2()}}</p>
<p>Serie 3: {{serie3 ? serie3 : 'No hay serie 3'}}</p>
```

Ahora ya deberían de verse las 3 series que hemos puesto. Podemos probar a dejar serie3 con un string vacío como valor para comprobar que el mensaje que se muestra es el otro.

/angular-data-binding-string-interpolation-lab/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  serie1: string = 'Gangland Undercover'
  serie2: string = 'Taboo'
  // serie3: string = 'The Walking Dead'
  serie3: string = ''

  getSerie2(): string {
    return this.serie2
  }
}
```

Para finalizar, vamos a añadir una última serie, pero esta vez no queremos que coja el valor de una propiedad, sino que queremos concatenar un string **parteSerie4** (una nueva propiedad del componente), con un string hardcodeado en el propio String Interpolation.

Primero, vamos a añadir la propiedad en el archivo de TypeScript.

/angular-data-binding-string-interpolation-lab/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  serie1: string = 'Gangland Undercover'
  serie2: string = 'Taboo'
  serie3: string = 'The Walking Dead'
  // serie3: string = ''
  parteSerie4: string = 'The Last'

  getSerie2(): string {
    return this.serie2
  }
}
```

Y ahora en la plantilla, entre las dobles llaves, vamos a concatenar esta nueva propiedad con el texto " Kingdom".

/angular-data-binding-property-binding-lab/src/app/app.component.html

```
<p>Serie 1: {{serie1}}</p>
<p>Serie 2: {{getSerie2()}}</p>
<p>Serie 3: {{serie3 ? serie3 : 'No hay serie 3'}}</p>
<p>Serie 4: {{parteSerie4 + ' Kingdom'}}</p>
```

Con esto ya debería de verse la cuarte serie en el navegador.

9.3. Property Binding

El **Property Binding** nos va a permitir darle valores a los atributos de las etiquetas HTML y propiedades de los componentes. Se suele usar con las propiedades del DOM, de los componentes y de las directivas.

Estos valores pueden ser datos que pongamos hardcodeados en la plantilla o los valores de las propiedades que hayamos declarado en los archivos de TypeScript. Incluso podemos llamar a funciones que nos devuelvan estos valores.

Se usa con **[propiedad]="expresión"**.

/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  datosInvalidos: boolean = true

  isValid() {
    return this.datosInvalidos
  }

}
```

/src/app/app.component.html

```
<button type="button" [disabled]="datosInvalidos">Enviar datos</button> <!-- Deshabilita el botón si datosInvalidos es true. -->
<button type="button" [disabled]="isValid()">Enviar datos 2</button> <!-- Deshabilita el botón si la función isValid devuelve un valor true. -->
```

Los corchetes solo los vamos a poner cuando el valor que queremos asignarle viene de una propiedad del TS, de una función del componente, o de una expresión de TS que le asignamos directamente al atributo de la etiqueta.



En el ejemplo anterior, si no ponemos los corchetes de disabled, el valor que se le va a asignar es el string "datosInvalidos".

9.4. Lab: Property Binding

En este laboratorio vamos a ver como utilizar el **Property Binding** para mostrar los logos de los top 4 frameworks/librerías de JS para el frontend.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-data-binding-property-binding-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-data-binding-property-binding-lab  
$ ng s
```

Vamos a empezar por declarar cuatro propiedades en el componente App, cada una de ellas con la url a los logos de los frameworks/librerías de React, Angular, Vue y Svelte.

/angular-data-binding-property-binding-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  logoReact = 'https://upload.wikimedia.org/wikipedia/commons/thumb/4/47/React.svg/1200px-React.svg.png';
  logoAngular = 'https://upload.wikimedia.org/wikipedia/commons/thumb/c/cf/Angular_full_color_logo.svg/2048px-Angular_full_color_logo.svg.png';
  logoVue = 'https://upload.wikimedia.org/wikipedia/commons/thumb/9/95/Vue.js_Logo_2.svg/2367px-Vue.js_Logo_2.svg.png';
  logoSvelte = 'https://upload.wikimedia.org/wikipedia/commons/thumb/1/1b/Svelte_Logo.svg/498px-Svelte_Logo.svg.png';
}
```

Ahora en la plantilla vamos a empezar por mostrar el primer logo, para ello asignaremos directamente la propiedad al atributo **src** de una etiqueta **img** utilizando el Property Binding, es decir, poniendo **src** entre corchetes.

/angular-data-binding-property-binding-lab/src/app/app.component.html

```
<img [src]="logoReact" alt="Logo de React" width="200" />
```

Ahora si abrimos el <http://localhost:4200/> debemos de ver el logo de React.

Ahora vamos a mostrar el logo de Angular, pero esta vez, en lugar de asignarle la propiedad directamente al atributo **src**, vamos a hacer que se la devuelva un método del componente.

Vamos a crear una función **getLogoAngular** dentro de la clase de App en la que vamos a retornar la propiedad de logoAngular.

/angular-data-binding-property-binding-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  logoReact = 'https://upload.wikimedia.org/wikipedia/commons/thumb/4/47/React.svg/1200px-React.svg.png';
  logoAngular = 'https://upload.wikimedia.org/wikipedia/commons/thumb/c/cf/Angular_full_color_logo.svg/2048px-Angular_full_color_logo.svg.png';
  logoVue = 'https://upload.wikimedia.org/wikipedia/commons/thumb/9/95/Vue.js_Logo_2.svg/2367px-Vue.js_Logo_2.svg.png';
  logoSvelte = 'https://upload.wikimedia.org/wikipedia/commons/thumb/1/1b/Svelte_Logo.svg/498px-Svelte_Logo.svg.png';

  getLogoAngular(): string {
    return this.logoAngular
  }
}
```

Como ya hemos comentado, estas propiedades siempre que vayan entre corchetes pueden recibir como valor una expresión de JS/TS, por lo que esta vez en lugar de asignarle directamente la propiedad, vamos a asignarle **la ejecución del método que acabamos de añadir**.

/angular-data-binding-property-binding-lab/src/app/app.component.html

```
<img [src]="logoReact" alt="Logo de React" width="200" />
<img [src]="getLogoAngular()" alt="Logo de Angular" width="200" />
```

Con esto ya podemos ver el logo de Angular también.

El siguiente caso va a ser en el que vamos a utilizar un getter de la clase para obtener el valor del logo de Vue.

Un **getter** es una función que se crea con la palabra **get** seguida del nombre del getter (que puede ser cualquiera, excepto el nombre de la propiedad de la clase que va a retornar). Nuestro getter se llamará **get urlLogoVue**

/angular-data-binding-property-binding-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
```

```

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  logoReact = 'https://upload.wikimedia.org/wikipedia/commons/thumb/4/47/React.svg/1200px-React.svg.png';
  logoAngular = 'https://upload.wikimedia.org/wikipedia/commons/thumb/c/cf/Angular_full_color_logo.svg/2048px-
Angular_full_color_logo.svg.png';
  logoVue = 'https://upload.wikimedia.org/wikipedia/commons/thumb/9/95/Vue.js_Logo_2.svg/2367px-Vue.js_Logo_2.svg.png';
  logoSvelte = 'https://upload.wikimedia.org/wikipedia/commons/thumb/1/1b/Svelte_Logo.svg/498px-Svelte_Logo.svg.png';

  getLogoAngular(): string {
    return this.logoAngular
  }

  getUrlLogoVue(): string {
    return this.logoVue
  }
}

```

Ahora en la plantilla le asignaremos como valor del **src**, **urlLogoVue**.



En TypeScript cuando queremos utilizar un getter, no hay que ejecutar la función, sino que lo usaremos como una propiedad normal y corriente. En nuestro caso solo tenemos que poner **urlLogoVue**.

/angular-data-binding-property-binding-lab/src/app/app.component.html

```

<img [src]="logoReact" alt="Logo de React" width="200" />
<img [src]="getLogoAngular()" alt="Logo de Angular" width="200" />
<img [src]="urlLogoVue" alt="Logo de Vue" width="200" />

```

Ahora también podemos ver el logo de Vue en nuestro navegador.

Y para mostrar el último logo, no vamos a utilizar los corchetes, por lo que el valor que le vamos a dar a **src** se va a tomar como un string en lugar de el valor de una propiedad. Por tanto, tenemos que copiar la url del logo y asignarselo directamente a **src**.

/angular-data-binding-property-binding-lab/src/app/app.component.html

```

<img [src]="logoReact" alt="Logo de React" width="200" />
<img [src]="getLogoAngular()" alt="Logo de Angular" width="200" />
<img [src]="urlLogoVue" alt="Logo de Vue" width="200" />


```

Y ya podemos ver el último logo en el navegador.

9.5. Event Binding

El **Event Binding** consiste en poder detectar eventos sobre las etiquetas para así poder ejecutar funciones que vamos a implementar en los archivos de TypeScript de los componentes.

Para detectar un evento, tenemos que poner el nombre del evento entre paréntesis, y después asignarle la función que queremos ejecutar cuando dicho evento se dispare. La sintaxis es **(evento)="expresión"** donde la expresión será una función declarada en el componente.

/src/app/app.component.html

```
<button (click)="saludar()">Saludame</button>
```

La función asignada al evento tenemos que declararla en la parte del TypeScript del mismo componente donde se va a utilizar.

/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  saludar() {
    alert('Hola mundo!');
  }
}
```

Algunos de los eventos que podremos utilizar son:

Table 1. Eventos de ratón y teclado

click	dblclick	mousemove
mouseover	mouseenter	mouseout
mouseleave	mousedown	mouseup
keydown	keyup	keypress

Table 2. Eventos de foco y formularios

focus	blur
-------	------

input	change
-------	--------

Per hay muchos más, todos aquellos que podemos utilizar en JavaScript, y para usarlos aquí, solo hay que poner el nombre del evento sin la palabra **on** como hemos visto en las tablas anteriores.

- "onclick" → "click"
- "onchange" → "change"

En algunos casos vamos a necesitar acceder al objeto event que se emite siempre que se detecta un evento sobre cualquier etiqueta, por ejemplo para poder acceder a dicha etiqueta o alguna propiedad de esta etiqueta donde ha ocurrido dicho evento.

Con el event binding es fácil acceder a este evento, lo único que tenemos que hacer es pasarle (en la plantilla) como parámetro de la función el **\$event**.

/src/app/app.component.html

```
<button (click)="saludar($event)">Saludame</button>
```

Y luego indicar en la declaración de la función que vamos a recibir dicho parámetro.

/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'
})
export class AppComponent {

  saludar(event: any) {
    console.log(event) // Es el objeto que representa el evento que se ha detectado
    console.log(event.target) // Hace referencia a la etiqueta button sobre la que se ha pulsado
    alert('Hola mundo!');
  }
}
```

9.6. Lab: Event Binding

En este laboratorio vamos a ver como detectar el evento click para hacer que nuestro navegador cante la intro de la serie de dibujos de Batman, pero mal cantada.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-data-binding-event-binding-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-data-binding-event-binding-lab  
$ ng s
```

Lo primero que vamos a hacer es crear un botón con el que llamaremos a la función encargada de hacer que el navegador hable.

/angular-data-binding-event-binding-lab/src/app/app.component.html

```
<button type="button">Que suene la intro</button>
```

La idea es que empiece a cantar mal cuando pulsemos sobre el botón por lo que tenemos que detectar un evento, en este caso es el **click**, por lo que vamos a añadirlo entre los paréntesis (event binding) y le vamos a asignar como valor la función **aCantar()**

/angular-data-binding-event-binding-lab/src/app/app.component.html

```
<button type="button" (click)="aCantar()">Que suene la intro</button>
```

El siguiente paso es declarar esta función en el TypeScript del componente App.

/angular-data-binding-event-binding-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

```

export class AppComponent {

  aCantar(): void {
    }

}

```

Empezaremos añadiendo el código que genera el texto que queremos que diga el navegador. En este caso hay que generar un texto que nos devuelva la letra de la intro de Batman.

Para ello tenemos que crear un array de 16 posiciones vacías. Unir todas estas posiciones con el método **join()** pasandole como parámetro un string **NaN** (Not a Number). Y finalmente le vamos a sumar el string " Batman!".

/angular-data-binding-event-binding-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  aCantar(): void {
    const textoIntro = new Array(16).join(NaN.toString()) + ' Batman!'
  }
}

```

El **textoIntro** generado en el paso anterior se lo vamos a pasar al constructor de **SpeechSynthesisUtterance**.

/angular-data-binding-event-binding-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  aCantar(): void {
    const textoIntro = new Array(16).join(NaN.toString()) + ' Batman!'
  }
}

```

```

    styleUrls: './app.component.css'
})
export class AppComponent {

  aCantar(): void {
    const textoIntro = new Array(16).join(NaN.toString()) + ' Batman!'
    const configSpeech = new SpeechSynthesisUtterance(textoIntro)
  }
}

```

Con la instancia generada vamos a cambiar el valor de rate para que hable un poco más lento de lo que por defecto hace.

/angular-data-binding-event-binding-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: './app.component.css'
})
export class AppComponent {

  aCantar(): void {
    const textoIntro = new Array(16).join(NaN.toString()) + ' Batman!'
    const configSpeech = new SpeechSynthesisUtterance(textoIntro)
    configSpeech.rate = 0.8
  }
}

```

Y por último llamaremos a **window.speechSynthesis.speak()** pasandole la configuración anterior y que así nos pueda cantar mal la intro.

/angular-data-binding-event-binding-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],

```

```
templateUrl: './app.component.html',
styleUrl: './app.component.css'
})
export class AppComponent {

aCantar(): void {
  const textoIntro = new Array(16).join(NaN.toString()) + ' Batman!'
  const configSpeech = new SpeechSynthesisUtterance(textoIntro)
  configSpeech.rate = 0.8
  window.speechSynthesis.speak(configSpeech)
}

}
```

Y con esto ya lo tenemos. Si pulsamos el botón que habíamos añadido y ponemos el volumen a tope, podremos escuchar al próximo ganador de "La Voz Tech". O no, parece que todavía tiene que mejorar.

9.7. Two-Way Data Binding

El **Two-Way Data Binding** es la combinación de los dos casos anteriores, el **property binding** y el **event binding** y se refleja en la sintaxis. Recordad que se usaba `[]` para asignar un valor a un atributo y `()` para detectar un evento y ejecutar una función del componente.

Este tipo de binding nos permite sincronizar propiedades que tenemos en el componente (archivo TypeScript) con las que tenemos en la plantilla (archivo HTML), en ambos sentidos. Es decir, que si una propiedad cambia en el componente también va a cambiar en la vista, y viceversa.

La sintaxis es `[(ngModel)]="propiedad"`.

Para poder utilizar este tipo de binding, tenemos que importar el **FormsModule** dentro del componente standalone donde lo vayamos a utilizar para que Angular reconozca la directiva `ngModel`.

`/src/app/app.component.ts`

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FormsModule], // Añadimos el módulo de Forms aquí
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent { }
```

Justo después ya podemos definir los datos que vamos a querer gestionar con el **Two-Way Data Binding**.

`/src/app/app.component.ts`

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
```

```
})
export class AppComponent {
  persona = {
    nombre: 'Robb',
    apellidos: 'Stark',
    edad: 28,
  }
}
```

/src/app/app.component.html

```
<!-- Al modificar el nombre en el primer input envía los datos al componente y cambia el valor, y al cambiar el valor en el componente, cambia el valor en el resto de elementos que lo usan. -->
<input type="text" [(ngModel)]="persona.nombre" />
<input type="text" [(ngModel)]="persona.nombre" disabled />
<p>{{persona.nombre}}</p>
```

9.8. Lab: Two-Way Data Binding

En este laboratorio vamos a ver como utilizar el Two Way Data Binding con un desplegable para poder seleccionar entre las distintas opciones de este.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-data-binding-two-way-data-binding-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-data-binding-two-way-data-binding-lab  
$ ng s
```

Como para este laboratorio vamos a necesitar usar la directiva **ngModel** (el Two Way Data Binding), vamos a empezar por importar el módulo de **FormsModule** dentro de nuestro componente.

/angular-data-binding-two-way-data-binding-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

Ahora vamos a añadir una propiedad **temaSeleccionado** en nuestro componente App. Esta propiedad será la que va a coger el valor de un desplegable, y con la que indicaremos que tema de colores queremos aplicar sobre unos elementos que añadiremos más adelante.

Esta propiedad vamos a inicializarla con un tema claro inicialmente.

/angular-data-binding-two-way-data-binding-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  temaSeleccionado: string = 'lightTheme'
}
```

El siguiente paso es añadir el desplegable en la plantilla del componente. Este desplegable (etiqueta **select**) va a llevar tres opciones para elegir, el tema claro, el tema oscuro y un tema de contraste alto.

/angular-data-binding-two-way-data-binding-lab/src/app/app.component.html

```
<select>
  <option value="darkTheme">Tema oscuro</option>
  <option value="lightTheme">Tema claro</option>
  <option value="highContrastTheme">Tema de alto contraste</option>
</select>
```

Si entramos al <http://localhost:4200/> y nos fijamos en el desplegable, este no tiene elegida la opción del tema claro que habíamos inicializado en el TypeScript. Esto se debe a que no le hemos dicho que el valor del select tiene que obtenerse y cambiarse por la propiedad **temaSeleccionado**.

Esto lo vamos a hacer añadiendo el Two Way Data Binding sobre la etiqueta **select** y asignandole esa propiedad.

/angular-data-binding-two-way-data-binding-lab/src/app/app.component.html

```
<select [(ngModel)]="temaSeleccionado">
  <option value="darkTheme">Tema oscuro</option>
  <option value="lightTheme">Tema claro</option>
  <option value="highContrastTheme">Tema de alto contraste</option>
</select>
```

Ahora ya podemos cambiar entre las distintas opciones del desplegable, y al hacerlo, la propiedad **temaSeleccionado** irá obteniendo como valor aquel que se le ha dado al atributo **value** de la etiqueta **option** que hemos seleccionado.

Para comprobar que todo esto funciona correctamente, vamos a añadir debajo del desplegable una caja con un párrafo.

/angular-data-binding-two-way-data-binding-lab/src/app/app.component.html

```
<select [(ngModel)]="temaSeleccionado">
  <option value="darkTheme">Tema oscuro</option>
  <option value="lightTheme">Tema claro</option>
  <option value="highContrastTheme">Tema de alto contraste</option>
</select>

<div>
  <p>Un texto de ejemplo</p>
</div>
```

La idea es que según vayamos seleccionando un tema u otro, los estilos de estos dos elementos vayan cambiando.

Para ello, vamos a empezar añadiendo en el archivo de CSS los siguientes estilos para cada uno de los temas.

/angular-data-binding-two-way-data-binding-lab/src/app/app.component.css

```
div.lightTheme {
  height: 150px;
  background-color: white;
  color: black;
}

div.darkTheme {
  height: 150px;
  background-color: black;
  color: white;
}

div.highContrastTheme {
  height: 150px;
  background-color: black;
  color: white;
}

div.highContrastTheme > p {
  border: 1px solid yellow;
}
```

Ahora solo nos queda ir añadiéndole una clase u otra al elemento div según vayamos cambiando de tema.

En el archivo de TypeScript vamos a añadir un getter **claseTema()** que va a retornar la clase que hay que aplicar dependiendo del valor de **temaSeleccionado**.

Cuando se haya seleccionado **Tema oscuro**, aplicaremos la clase **darkTheme**, al seleccionar **Tema claro** aplicaremos **lightTheme**, y al seleccionar la última opción, vamos a aplicar la clase **highContrastTheme**.

/angular-data-binding-two-way-data-binding-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  temaSeleccionado: string = 'lightTheme'

  get claseTema(): any {
    if (this.temaSeleccionado === 'darkTheme') {
      return { darkTheme: true }
    } else if (this.temaSeleccionado === 'highContrastTheme') {
      return { highContrastTheme: true }
    } else {
      return { lightTheme: true }
    }
  }
}
```

Para terminar, vamos a utilizar la directiva **ngClass**, sobre el div, a la que le vamos a asignar el valor que devuelva este método que acabamos de añadir.

La directiva ngClass se encarga de aplicar clases de CSS al elemento sobre el que se ha puesto.



El valor que recibe esta directiva es un objeto de JS, en el que las claves son el nombre de las clases de CSS, y el valor de estas es un booleano.

Si el valor es true, la clase de CSS se aplica sobre el elemento, pero si es false, la clase se deja de aplicar.

/angular-data-binding-two-way-data-binding-lab/src/app/app.component.html

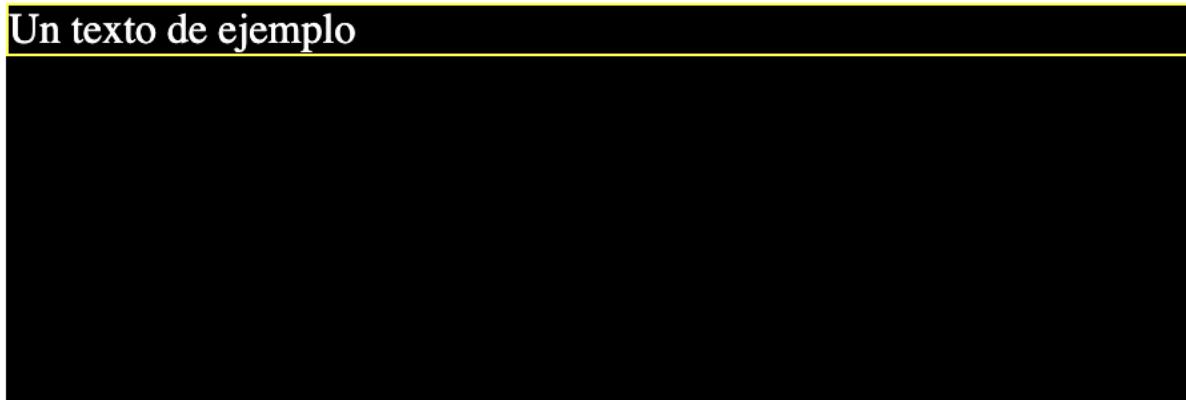
```
<select [(ngModel)]="temaSeleccionado">
  <option value="darkTheme">Tema oscuro</option>
  <option value="lightTheme">Tema claro</option>
  <option value="highContrastTheme">Tema de alto contraste</option>
```

```
</select>

<div [ngClass]="claseTema">
  <p>Un texto de ejemplo</p>
</div>
```

Ahora ya podemos cambiar entre los distintos temas de color y deberíamos de ver algo parecido a lo que se muestra en la siguiente imagen.

Tema de alto contraste ▾



9.9. Lab: ¿Cómo funciona internamente el 2-Way Data Binding?

En este laboratorio vamos a crear un input al cual le vamos a añadir las propiedades necesarias para poder replicar la funcionalidad de la directiva ngModel.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-data-binding-crea-tu-ngmodel-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-data-binding-crea-tu-ngmodel-lab  
$ ng s
```

Lo primero que vamos a hacer es declarar una propiedad de tipo string en el componente App, ya que será el dato que vamos a modificar con nuestro propio **input**.

/angular-data-binding-crea-tu-ngmodel-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  nombre: string = 'Trafalgar D. Law'
}
```

Ahora en el HTML vamos a poner una etiqueta input de tipo "text".

/angular-data-binding-crea-tu-ngmodel-lab/src/app/app.component.html

```
<input type="text">
```

Como vimos, el ngModel se encarga de darle como valor del input el valor asignado a esta directiva.

Pues si nosotros queremos que nuestro input muestre como valor inicial el valor de la propiedad nombre que hemos declarado en el TypeScript, vamos a asignarsela al atributo **value** del input.

Al asignarle esta propiedad al value, tendremos que utilizar el property binding para que coja el valor del nombre, en lugar de que nos muestre "nombre" dentro del input. Por lo tanto hay que meter **value** entre corchetes.

```
/angular-data-binding-crea-tu-ngmodel-lab/src/app/app.component.html
```

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

Ahora deberíamos de poder ver "Trafalgar D. Law" en el input.

El siguiente paso es permitir modificar este valor cuando escribamos dentro del input. Antes de realizar este paso, vamos a mostrar el nombre de nuevo, pero esta vez dentro de un párrafo.

```
/angular-data-binding-crea-tu-ngmodel-lab/src/app/app.component.html
```

```
<input type="text" [value]="nombre">
<p>Nombre: {{nombre}}</p>
```

Para modificar el valor del nombre desde el input, vamos a tener que añadir un evento en esta etiqueta. El evento que vamos a utilizar es el **input**, al cual le vamos a asignar la función de **changeNombre** que tendremos que implementar después. Le pasaremos como parámetro de dicha función el **\$event** para poder sacar del evento el valor que vamos escribiendo en el campo de texto.

El evento **input** se va disparando cada vez que pulsamos una tecla, por lo que los cambios en el nombre los vamos a realizar en tiempo real.

```
/angular-data-binding-crea-tu-ngmodel-lab/src/app/app.component.html
```

```
<input type="text" [value]="nombre" (input)="changeNombre($event)">
<p>Nombre: {{nombre}}</p>
```

Ahora en el TypeScript vamos a implementar esta función.

```
/angular-data-binding-crea-tu-ngmodel-lab/src/app/app.component.ts
```

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

```

export class AppComponent {
  nombre: string = 'Trafalgar D. Law'

  changeNombre(event: any): void {
  }

}

```

El valor que vamos escribiendo dentro del input lo vamos a obtener de **event.target.value**, y se lo vamos a asignar a la propiedad nombre para ir modificandola al igual que ocurre con la directiva ngModel.

/angular-data-binding-crea-tu-ngmodel-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  nombre: string = 'Trafalgar D. Law'

  changeNombre(event: any): void {
    this.nombre = event.target.value
  }
}

```

Y con esto ya deberíamos de poder ver como se va modificando el valor del párrafo según vamos cambiando el valor del input.

Al final, la directiva ngModel o el Two Way Data Binding se encarga de asignarle el valor a la etiqueta **input** a la que se le asigna, y también va cambiando el valor de la propiedad que se le asigna al **[(ngModel)]** cuando se detecta un evento **input**.

Chapter 10. Variables de plantilla (Referencias)

Con las **variables de plantilla** o **referencias** vamos a poder acceder a los elementos del DOM.

Equivaldrían a utilizar métodos como `getElementById` o `querySelector` del objeto `window.document`, pero como aquí es Angular el que se encarga de trabajar con el DOM, tenemos que evitar esos métodos a toda costa ya que pueden llegar a darnos algún problema. En su lugar usaremos las variables de plantilla.

Estas variables de plantilla se utilizan con alguna directiva de Angular como el `ngIf` cuando le añadimos la parte del `else`.

Pero nosotros sobre todo las podremos utilizar cuando necesitemos trabajar con etiquetas de HTML de media, como la etiqueta **audio** o **video**.

Para poner una variable de plantilla solo hay que poner dentro de la etiqueta a la que queremos acceder a través de la referencia # seguida del nombre que le vamos a dar a la referencia.

A la hora de acceder a esta referencia, usaremos el nombre de la referencia (sin la #).

`/src/app/app.component.html`

```
<input type="text" #miInput>
<button type="button" (click)="mostrarValorInput(miInput)">Mostrar valor del input</button>
```

`/src/app/app.component.ts`

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  mostrarValorInput(elementoInput: any) {
    console.log(elementoInput.value); // Muestra el valor escrito en el input
    console.log(elementoInput.type); // Muestra el valor del atributo type del input
  }
}
```

10.1. Lab: Variables de plantilla (Referencias)

En este laboratorio vamos a crear los controles de reproducción de la etiqueta **video** utilizando las variables de plantilla. Podremos realizar las siguientes acciones:

- Reproducir el sonido
- Pausar el sonido
- Cambiar el volumen

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-variables-de-plantilla-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-variables-de-plantilla-lab  
$ ng s
```

Vamos a empezar por descargarnos un video y lo meteremos dentro de la carpeta **src/assets**.



Podemos descargarnos el video de: <https://www.pexels.com/es-es/videos/>

Ahora vamos a añadir la etiqueta **video** dentro de nuestro componente App, además, vamos a añadir dos botones (para reproducir y pausar el video) y un input de tipo range (para subir y bajar el volumen).

/angular-variables-de-plantilla-lab/src/app/app.component.html

```
<video src="" width="200"></video>  
  
<button type="button">Play</button>  
<button type="button">Pause</button>  
<input type="range" min="0" max="100">
```

Por ahora ya deberíamos de ver nuestros controles, el siguiente paso es mostrar el video, por lo que vamos a añadir la dirección donde lo hemos descargado como valor del atributo **src** de la etiqueta video que tenemos en el HTML.

/angular-variables-de-plantilla-lab/src/app/app.component.html

```
<video src="assets/video.mp4" width="200"></video>  
  
<button type="button">Play</button>
```

```
<button type="button">Pause</button>
<input type="range" min="0" max="100">
```

Ahora ya tenemos toda la estructura, nos falta ir añadiendo la funcionalidad. Vamos a empezar por añadir la funcionalidad para poder reproducir el video.

Esta etiqueta tiene internamente un método **play()** el cual se encarga de empezar a reproducir el video, pero para poder ejecutar esta función, primero tenemos que tener acceso a la etiqueta video.

Por tanto, empezamos añadiendo una variable de plantilla **#videoRef** sobre la etiqueta video.

```
/angular-variables-de-plantilla-lab/src/app/app.component.html
```

```
<video src="assets/video.mp4" width="200" #videoRef></video>

<button type="button">Play</button>
<button type="button">Pause</button>
<input type="range" min="0" max="100">
```

El siguiente paso es detectar el evento **click** sobre el botón de Play, para llamar a una función **reproducirVideo()** que después implementaremos en el archivo de TypeScript.

A esta función le vamos a pasar como parámetro **videoRef**, la referencia del video, para poder llamar a la función **play()** que lleva de forma interna.

```
/angular-variables-de-plantilla-lab/src/app/app.component.html
```

```
<video src="assets/video.mp4" width="200" #videoRef></video>

<button type="button" (click)="reproducirVideo(videoRef)">Play</button>
<button type="button">Pause</button>
<input type="range" min="0" max="100">
```

Ahora en el archivo **app.component.ts** vamos a implementar el método **reproducirVideo**, el cual recibe un video como parámetro.

```
/angular-variables-de-plantilla-lab/src/app/app.component.ts
```

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

```
reproducirVideo(video: any): void {  
}  
}  
}
```

Ahora toca llamar a la función **play** del video para que se ponga a reproducir el video.

/angular-variables-de-plantilla-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';  
import { CommonModule } from '@angular/common';  
import { RouterOutlet } from '@angular/router';  
  
@Component({  
  selector: 'app-root',  
  standalone: true,  
  imports: [CommonModule, RouterOutlet],  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css'  
})  
export class AppComponent {  
  
  reproducirVideo(video: any): void {  
    video.play()  
  }  
}
```

Si probamos a pulsar el botón, ya deberíamos de poder ver el video en nuestro navegador.

Ahora vamos a hacer lo mismo, pero esta vez para pausarlo.

En el archivo de HTML, vamos a detectar el evento **click** sobre el botón de **Pause**, y llamaremos a una función **pausarVideo()** pasandole como parámetro la referencia al video como hemos hecho con el anterior botón.

/angular-variables-de-plantilla-lab/src/app/app.component.html

```
<video src="assets/video.mp4" width="200" #videoRef></video>  
  
<button type="button" (click)="reproducirVideo(videoRef)">Play</button>  
<button type="button" (click)="pausarVideo(videoRef)">Pause</button>  
<input type="range" min="0" max="100">
```

Ahora en el TypeScript nos toca implementar esta función. Esta vez, en lugar de llamar al método **play**, vamos a llamar al método **pause** del video.

/angular-variables-de-plantilla-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  reproducirVideo(video: any): void {
    video.play()
  }

  pausarVideo(video: any): void {
    video.pause()
  }
}
```

Con esto ya podemos tanto reproducir el video, como pausarlo.

Solo nos falta controlar el volumen con el input de tipo range. Para ello no hay método del propio video que permita cambiar el volumen, sino que tenemos que modificar el valor de una propiedad **volume** interna de la etiqueta video.

Esta vez vamos a detectar el evento **input** para que se vaya ejecutando la función **cambiarVolumen()** que le vamos a asignar según desplacemos la barra por el input.

Como necesitamos saber el valor del input (que va de 0 a 100), necesitaremos pasarle el objeto del evento, es decir \$event, como parámetro de la función. Además le pasaremos la referencia del video.

/angular-variables-de-plantilla-lab/src/app/app.component.html

```
<video src="assets/video.mp4" width="200" #videoRef></video>

<button type="button" (click)="reproducirVideo(videoRef)">Play</button>
<button type="button" (click)="pausarVideo(videoRef)">Pause</button>
<input type="range" min="0" max="100" (input)="cambiarVolumen($event, videoRef)">
```

Ahora en el archivo de TypeScript, vamos a implementar esta función.

/angular-variables-de-plantilla-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  reproducirVideo(video: any): void {
    video.play()
  }

  pausarVideo(video: any): void {
    video.pause()
  }

  cambiarVolumen(event: any, video: any): void {
    }

}
```

El primer paso es obtener el valor del input, para lo que vamos a acceder a **event.target.value**.

/angular-variables-de-plantilla-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  reproducirVideo(video: any): void {
    video.play()
  }

}
```

```

    pausarVideo(video: any): void {
      video.pause()
    }

    cambiarVolumen(event: any, video: any): void {
      const volumen = event.target.value
    }

}

```

Una vez tenemos el volumen, vamos a asignarselo a la propiedad **volume** del video. Pero esta propiedad solo puede recibir valores entre el 0 y el 1, por lo que tendremos que dividir el valor del input entre 100.

/angular-variables-de-plantilla-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  reproducirVideo(video: any): void {
    video.play()
  }

  pausarVideo(video: any): void {
    video.pause()
  }

  cambiarVolumen(event: any, video: any): void {
    const volumen = event.target.value
    video.volume = volumen / 100
  }

}

```

Y con esto ya podemos modificar el volumen del video.

Chapter 11. Decorador @Input()

Como hemos visto, nuestras aplicaciones estarán formadas por diferentes componentes, y la idea de hacerlos así es porque una vez creado un componente, este se puede reutilizar para pintar lo mismo, con la misma estructura, funcionalidad y estilos, pero con diferentes datos.

Por ejemplo, imaginemos que tenemos un componente para mostrar notificaciones en nuestra aplicación. Una notificación tendrá como propiedades el texto a mostrar y el tipo de notificación (info, success, warning y danger).

Dentro del componente podremos inicializar estas propiedades para que se pinten con unos valores por defecto.

Componente notificación

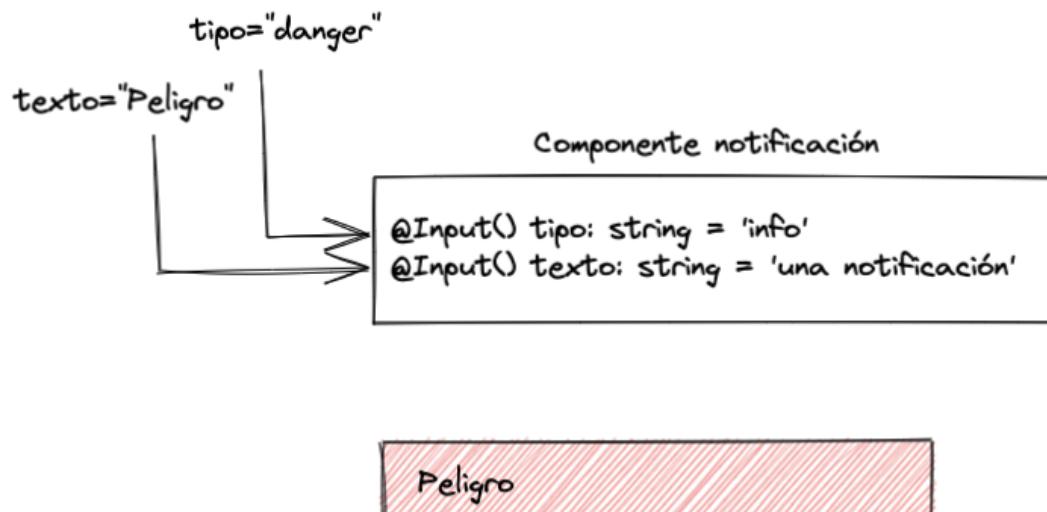
```
tipo: string = 'info'  
texto: string = 'una notificación'
```

una notificación

Pero tal cual hemos declarado este componente, cuando vayamos a pintarlo en nuestra aplicación, siempre se va a pintar con los mismos datos, con el texto "una notificación" y del tipo "info".

La idea es que nosotros podamos reutilizar este componente, como hemos dicho al principio, y para poder reutilizarlo nosotros tenemos que ser capaces de pasarle unos valores distintos a las propiedades internas del componente (tipo y texto).

Aquí es donde entra el decorador **@Input()**. Este decorador se añade delante de las propiedades que queremos que puedan recibir valores desde fuera del propio componente, como es nuestro caso. Con él podremos indicarle que queremos pintar una notificación del tipo "danger" y con el texto "Peligro".



Ahora ya podríamos crear todas las通知 a partir de un único componente, y solo tenemos que pasarle los distintos valores a las propiedades internas del componente.

<pre> <app-notificacion tipo="success" texto="Un mensaje de success" ></app-notificacion> </pre>	Un mensaje de success
<pre> <app-notificacion tipo="warning" texto="Un mensaje de warning" ></app-notificacion> </pre>	Un mensaje de warning
<pre> <app-notificacion tipo="danger" texto="Un mensaje de danger" ></app-notificacion> </pre>	Un mensaje de danger
<pre> <app-notificacion tipo="info" texto="Un mensaje de info" ></app-notificacion> </pre>	Un mensaje de info

Esto en el código quedaría de la siguiente forma:

`/src/app/notificacion/notificacion.component.html`

```

<div [ngStyle]="{{backgroundColor: color}}>
  <p>{{texto}}</p>
</div>

```

/src/app/notificacion/notificacion.component.ts

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

@Component({
  selector: 'app-notificacion',
  standalone: true,
  imports: [NgStyle],
  templateUrl: './notificacion.component.html',
  styleUrls: ['./notificacion.component.css']
})
export class NotificacionComponent implements OnInit {

  @Input() tipo: string = 'success'
  @Input() texto: string = 'Una notificación'

  constructor() { }

  ngOnInit(): void {
  }

  get color(): string {
    switch (this.tipo) {
      case 'success':
        return 'green'
      case 'danger':
        return 'red'
      case 'info':
        return 'blue'
      case 'warning':
        return 'yellow'
    }
    return 'white'
  }
}
```

Y en el componente donde queramos pintar notificaciones, pondremos la etiqueta del componente y le pasaremos las propiedades con los valores que queremos asignarles.

/src/app/app.component.html

```
<app-notificacion texto="Cuidado!" tipo="warning"></app-notificacion>
<app-notificacion texto="Esta es una notificación informativa" tipo="info"></app-notificacion>
```

Para que se reconozca dicha etiqueta, tenemos que importar el componente de la notificación en el componente App.

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { NotificationComponent } from './notification/notification.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, NotificationComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

11.1. Lab: Decorador @Input()

En este laboratorio vamos a crear un componente Sugus con el que pretendemos pintar los distintos tipos de sugus que existen.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-decorador-input-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y dentro de ella vamos a empezar creando el componente del sugus y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-decorador-input-lab  
$ ng g c sugus  
$ ng s
```

Dentro del archivo de typescript del componente, vamos a declarar las dos propiedades que vamos a necesitar para pintar un sugus, el sabor y el color, ambas del tipo string.

/angular-decorador-input-lab/src/app/sugus/sugus.component.ts

```
import { Component, OnInit } from '@angular/core';  
  
@Component({  
  selector: 'app-sugus',  
  standalone: true,  
  imports: [],  
  templateUrl: './sugus.component.html',  
  styleUrls: ['./sugus.component.css']  
})  
export class SugusComponent implements OnInit {  
  
  sabor: string = 'limón'  
  color: string = '#FDE23A'  
  
  constructor() {}  
  
  ngOnInit(): void {}  
}
```

Ahora vamos a crear la estructura del sugus en la plantilla, donde pondremos un div que hará de envoltorio y un párrafo para poner el sabor de los sugus.

Usaremos el String Interpolation para pintar el sabor dentro del párrafo, y al div le vamos a añadir una clase **sugus** que utilizaremos para aplicarle unos estilos desde el CSS, además de utilizar la directiva **ngStyle** para añadirle el color de forma dinámica.

/angular-decorador-input-lab/src/app/sugus/sugus.component.html

```
<div class="sugus" [ngStyle]="{backgroundColor: color}">
  <p>{{sabor}}</p>
</div>
```

Esto nos da un error porque no reconoce la directiva **ngStyle**, por lo que vamos a importarla desde **@angular/common** en el componente Sugus.

/angular-decorador-input-lab/src/app/sugus/sugus.component.ts

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

@Component({
  selector: 'app-sugus',
  standalone: true,
  imports: [NgStyle],
  templateUrl: './sugus.component.html',
  styleUrls: ['./sugus.component.css']
})
export class SugusComponent implements OnInit {

  sabor: string = 'limón'
  color: string = '#FDE23A'

  constructor() { }

  ngOnInit(): void {
  }

}
```

Ahora vamos a añadir los siguientes estilos dentro del archivo de CSS correspondiente al componente sugus.

/angular-decorador-input-lab/src/app/sugus/sugus.component.css

```
.sugus {
  border: 1px solid black;
  width: 100px;
  height: 100px;
  border-radius: 5px;
  color: white;
  position: relative;
  margin: 10px;
  overflow: hidden;
}
```

```
.sugus > p {
  text-align: center;
  transform-origin: center center;
  transform: rotate(-45deg);
  position: absolute;
  top: 25px;
  left: 30px;
  text-shadow: 60px 0px 0px, -60px 0px 0px, -25px 30px 0px, 25px -30px 0px, 30px 30px 0px, -30px -30px 0px, 0px 60px 0px,
  0px -60px 0px;
}
```

El siguiente paso es utilizar este componente dentro del componente App para ver si se muestra correctamente o no.

Primero lo importamos en el archivo de typescript del componente App.

/angular-decorador-input-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { SugusComponent } from './sugus/sugus.component';

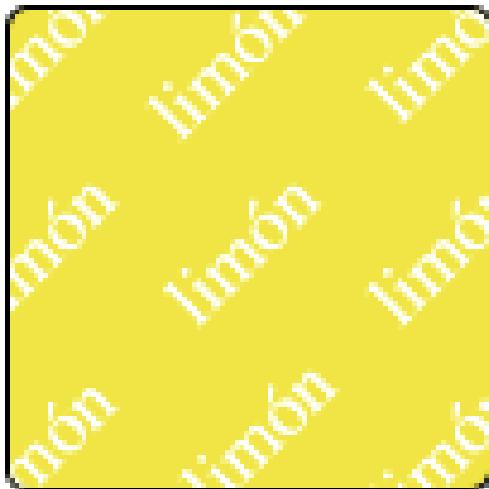
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, SugusComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

Y ahora ponemos el componente para mostrarlo.

/angular-decorador-input-lab/src/app/app.component.html

```
<app-sugus></app-sugus>
```

Deberíamos de ver algo como la siguiente imagen:



Entonces, como hemos comentado, la idea de utilizar componentes, es que podamos reutilizarlos de alguna forma.

En nuestro caso, el sugus siempre va a tener la misma estructura y los mismos estilos y si volvemos a poner el componente del sugus en la plantilla, veremos que siempre se pinta el mismo.

`/angular-decorador-input-lab/src/app/app.component.html`

```
<app-sugus></app-sugus>
<app-sugus></app-sugus>
<app-sugus></app-sugus>
<app-sugus></app-sugus>
<app-sugus></app-sugus>
```

Si ahora queremos que cada uno de estos sugus sea distinto, es decir, queremos pintar los sugus de naranja, limón, fresa, piña y cereza, no podemos, hacerlo porque hemos dejado unos valores iniciales dentro del componente sugus para pintar siempre el de limón.

Lo que podemos hacer es sobreescibir esos valores internos con los valores que le podemos pasar al componente desde el exterior, mediante atributos de la etiqueta como vamos a ver a continuación.

A los componentes podemos pasárselos atributos o propiedades al igual que lo hacemos con otras etiquetas de HTML (por ejemplo, un input lleva un atributo type, y dicho atributo puede coger como valores "text", "number", "color", "email"...).

Al componente `sugus` le vamos a pasar como atributos el sabor y el color que son las propiedades internas que hemos definido y aquellas que indican como se tiene que pintar.

`/angular-decorador-input-lab/src/app/app.component.html`

```
<app-sugus sabor="limón" color="#FDE23A"></app-sugus>
<app-sugus sabor="naranja" color="#F28E40"></app-sugus>
<app-sugus sabor="piña" color="#227BBE"></app-sugus>
<app-sugus sabor="cereza" color="#AD3B52"></app-sugus>
<app-sugus sabor="fresa" color="#EA464C"></app-sugus>
```

Si volvemos al navegador, veremos que ahora tenemos 5 `sugus` de limón, es decir, estos valores que le estamos pasando no le llegan al `sugus`, y por tanto este sigue utilizando los valores iniciales que le habíamos puesto.



Para solucionarlo, solo tenemos que añadir el decorador `@Input()` delante de la declaración de las

propiedades del componente sugus. Además hay que **importar el decorador Input desde @angular/core**.

Ahora estamos permitiendo que estas propiedades (las que llevan el decorador) puedan recibir el valor desde el exterior, permitiéndonos reutilizar un componente.

/angular-decorador-input-lab/src/app/sugus/sugus.component.ts

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

@Component({
  selector: 'app-sugus',
  standalone: true,
  imports: [NgStyle],
  templateUrl: './sugus.component.html',
  styleUrls: ['./sugus.component.css']
})
export class SugusComponent implements OnInit {

  @Input() sabor: string = 'limón'
  @Input() color: string = '#FDE23A'

  constructor() { }

  ngOnInit(): void {
  }
}
```

Después de realizar este cambio, deberíamos ver correctamente los 5 sugus, cada uno de ellos con su color y su sabor.



Chapter 12. Decorador @Output()

Al igual que hay un decorador `@Input`, también hay un decorador `@Output`.

Este se utiliza justo para lo contrario, es decir, para poder enviar datos hacia fuera del componente, en este caso emitiendo eventos.

Para ello se aplica a las instancias de `EventEmitter`, y luego desde ellas usamos el método `emit()` para emitir los datos, los cuales se le pasan como parámetro.

A la hora de recibirlos fuera del componente, tenemos que hacerlo al igual que hacemos con el resto de eventos, usando el `Event Binding`, y como nombre del evento tenemos que usar el nombre de la instancia del `EventEmitter`.

12.1. Lab: Decorador @Output()

En este laboratorio vamos a ver como utilizar el decorador **@Output()** para poder emitir eventos con **EventEmitter** desde dentro de los componentes.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-decorador-output-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, creamos un componente y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-decorador-output-lab  
$ ng g c mi-botón  
$ ng s
```

Hemos creado un componente **MiBotonComponent** desde el cual queremos emitir un evento **customClick** cada vez que se haga click en el botón.

Para emitir eventos en Angular, tenemos la clase **EventEmitter** de **@angular/core**. Esta clase nos permite emitir eventos con el método **emit()** al que le pasaremos como parámetro el evento o datos a emitir a quien esté escuchando dicho evento.

Por tanto, dentro de la plantilla del componente que hemos creado, vamos a crear un **button** al que le vamos a poner el evento **click** para llamar a una función **emitterEvento** que crearemos después en el TypeScript.

/angular-decorador-output-lab/src/app/mi-botón/mi-botón.component.html

```
<button type="button" (click)="emitterEvento()">Pulsa aquí para emitir tu propio evento</button>
```

Dentro del TypeScript vamos a añadir esta función a la que se llama desde la plantilla.

/angular-decorador-output-lab/src/app/mi-botón/mi-botón.component.ts

```
import { Component, OnInit } from '@angular/core';  
  
@Component({  
  selector: 'app-mi-botón',  
  standalone: true,  
  imports: [],  
  templateUrl: './mi-botón.component.html',  
  styleUrls: ['./mi-botón.component.css']  
})  
export class MiBotónComponent implements OnInit {
```

```

constructor() { }

ngOnInit(): void {
}

emitirEvento() {

}

}

```

Ahora necesitamos una instancia del **EventEmitter** para poder emitir eventos, por lo que vamos a crearla, y le indicaremos que va a emitir datos del tipo **string**. El nombre que le demos a la instancia será el nombre del evento que tendremos que detectar.

/angular-decorador-output-lab/src/app/mi-boton/mi-boton.component.ts

```

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

@Component({
  selector: 'app-mi-boton',
  standalone: true,
  imports: [],
  templateUrl: './mi-boton.component.html',
  styleUrls: ['./mi-boton.component.css']
})
export class MiBotonComponent implements OnInit {
  customClick = new EventEmitter<string>();

  constructor() { }

  ngOnInit(): void {
  }

  emitirEvento() {

  }

}

```

Como el evento lo vamos a querer detectar fuera de este componente, tendremos que decorarlo con **@Output()**.

/angular-decorador-output-lab/src/app/mi-boton/mi-boton.component.ts

```

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

@Component({
  selector: 'app-mi-boton',

```

```

standalone: true,
imports: [],
templateUrl: './mi-boton.component.html',
styleUrl: './mi-boton.component.css'
})
export class MiBotonComponent implements OnInit {
@Output() customClick = new EventEmitter<string>();

constructor() { }

ngOnInit(): void {
}

emitirEvento() {

}
}

```

Dentro de este componente, solo nos queda emitir un evento (o string en este caso). Por lo que dentro de la función **emitirEvento**, vamos a llamar a la función **emit** del EventEmitter y enviaremos el texto que queramos.

/angular-decorador-output-lab/src/app/mi-boton/mi-boton.component.ts

```

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

@Component({
selector: 'app-mi-boton',
standalone: true,
imports: [],
templateUrl: './mi-boton.component.html',
styleUrl: './mi-boton.component.css'
})
export class MiBotonComponent implements OnInit {
@Output() customClick = new EventEmitter<string>();

constructor() { }

ngOnInit(): void {
}

emitirEvento() {
const mensajes: Array<string> = ['Cowabunga', 'Jawsome', 'Chispas']
const posRandom: number = Math.floor(Math.random() * mensajes.length)
const textoRandom: string = mensajes[posRandom]
this.customClick.emit(textoRandom);
}
}

```

Una vez ya podemos enviar el evento, tenemos que detectarlo en la etiqueta del mismo componente donde se ha declarado. Por tanto, dentro de la plantilla del componente App, tenemos que poner el componente **mi-botón** y detectar el evento **customClick**.

/angular-decorador-output-lab/src/app/app.component.html

```
<app-mi-botón (customClick)="mostrarMsg($event)"></app-mi-botón>
```

Y que no se nos olvide importar el componente **MiBotonComponent** en el componente **AppComponent**.

/angular-decorador-output-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { MiBotonComponent } from './mi-botón/mi-botón.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, MiBotonComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

Para terminar, dentro de la función **mostrarMsg** que vamos a crear en el componente **app.component.ts**, vamos a recibir el mensaje que se emite al pulsar el botón y lo mostraremos en un popup del navegador.

/angular-decorador-output-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { MiBotonComponent } from './mi-botón/mi-botón.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, MiBotonComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  mostrarMsg(event: string): void {
```

```
    alert(event)
}
}
```

Con esto ya podemos emitir eventos propios desde dentro de los componentes. En este caso, al pulsar sobre el botón, debe de mostrarse un mensaje en un popup.

Chapter 13. Directivas

Las **directivas** en angular son componentes que no tienen una plantilla asociada y que se añaden sobre otras etiquetas o componentes para añadirles algo de funcionalidad o para modificar la estructura del DOM.

Dentro de Angular podemos diferenciar 2 tipos de directivas:

- Directivas de atributo
- Directivas estructurales

13.1. Directivas de atributo

Las **directivas de atributo** interaccionan con el elemento al que se le aplica la directiva para alterar su apariencia o su comportamiento, por ejemplo para añadirle una clase o cambiarle un estilo.



Tenemos que añadir en las importaciones de los componentes standalone las directivas que vamos a utilizar o en su defecto el módulo antiguo donde están definidas.

- **ngModel**: implementa el mecanismo del Two Way Data Binding.

/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    FormsModule,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  persona: any = {
    nombre: 'Charles',
    apellido: 'Falco'
  }
}
```

/src/app/app.component.html

```
<input [(ngModel)]="persona.nombre">
```

- **ngClass**: esta directiva permite añadir o quitar varias clases de forma simultánea y dinámica de un elemento

/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    FormsModule,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  persona: any = {
    nombre: 'Charles',
    apellido: 'Falco'
  }
}
```

```

    NgClass,
],
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})
export class AppComponent {
persona: any = {
  nombre: 'Charles',
  apellido: 'Falco'
}

}

```

/src/app/app.component.html

```
<p [ngClass]="{bordeRojo: true, letraAzul: false}">Este párrafo tiene el borde rojo y la letra no es azul</p>
```

- **ngStyle**: esta directiva permite añadir varios estilos en línea a un elemento.

/src/app/app.component.ts

```

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    FormsModule,
    NgClass,
    NgStyle,
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
persona: any = {
  nombre: 'Charles',
  apellido: 'Falco'
}

}

```

/src/app/app.component.html

```
<p [ngStyle]="{'font-size': '15px', textDecoration: 'line-through'}">Este párrafo tiene una letra de 15px y aparece tachado</p>
```

Como se puede observar, tanto el ngClass como el ngStyle esperan que su valor sea un objeto de JavaScript (`{}`). En los objetos de JavaScript, si una de las claves tiene un `-`, esa clave va entre " o la

tenemos que escribir en camelCase (**textDecoration**, **fontSize**, ...).

13.2. Crear una directiva de atributo

Angular nos permite crear nuestras propias directivas, para ello hay que lanzar alguno de los siguientes comandos:

```
$ ng generate directive nombre-directiva  
$ ng g d nombre-directiva
```

Estos comandos generan un archivo acabado en **directive.ts**, además de que añade dicha directiva dentro del array de **declarations** del módulo de la aplicación para que Angular reconozca la directiva cuando la utilicemos sobre alguna etiqueta.

Dentro del archivo de la directiva, nos encontramos una clase con un decorador **@Directive({})**. Como ya hemos comentado en algún momento, las directivas son componentes que no tienen plantilla, por lo que podemos ver que como opción del decorador tenemos el **selector** que hace referencia al nombre que hay que usar para aplicar la directiva, y esta vez no tendremos las opciones de **templateUrl** ni de **stylesUrls**.

/src/app/mi-directiva.directive.ts

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

@Directive({
  selector: '[appMiDirectiva]',
  standalone: true
})
export class MiDirectivaDirective {

  constructor() {

  }

}
```

Dentro de esta directiva ya podemos ir añadiendo el código para que haga los cambios que queremos realizar sobre las etiquetas.

Podemos ver como crear una en el siguiente laboratorio.

13.3. Lab: Crear una directiva de atributo

En este laboratorio vamos a crear una directiva para marcar los párrafos de un color dado cuando pasemos el ratón por encima de ellos.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-directivas-de-atributo-crear-una-directiva-lab  
? Would you like to add Angular routing? N  
? Which stylesheet format would you like to use? CSS
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, crearemos una directiva **marcar** y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-directivas-de-atributo-crear-una-directiva-lab  
$ ng g d marcar  
$ ng s
```

Como vamos a utilizar la directiva en la plantilla del componente App, tenemos que añadirla en el array de imports del componente standalone.

/angular-directivas-de-atributo-crear-una-directiva-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { MarcarDirective } from './directives/marcar.directive';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, MarcarDirective],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

Dentro de la plantilla del componente App vamos a poner un párrafo al que le vamos a aplicar la directiva que hemos creado, y que todavía no hace nada. Para aplicar la directiva, solo tenemos que poner el valor del **selector** que encontramos dentro del archivo de la directiva que hemos creado **marcar.directive.ts**.

/angular-directivas-de-atributo-crear-una-directiva-lab/src/app/app.component.html

```
<p appMarcar>Si pasas el ratón por encima, el párrafo se marcará en amarillo</p>
```

Ahora vamos a ir al archivo de la directiva para empezar a añadirle la lógica necesaria para poder ir cambiando el color de los elementos a los que les añadamos esta directiva.

Empezaremos añadiendo una propiedad **color** para almacenar el color con el que vamos a pintar el fondo del párrafo.

/angular-directivas-de-atributo-crear-una-directiva-lab/src/app/marcar.directive.ts

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

@Directive({
  selector: '[appMarcar]',
  standalone: true
})
export class MarcarDirective {

  color: string = 'yellow'

  constructor() { }

}
```

Ahora vamos a hacer que se pinte el párrafo con dicho color. Para ello, vamos a necesitar importar desde **@angular/core** el decorador **HostBinding**.

Este decorador se lo vamos a poner a otra propiedad a la que llamaremos **colorFondo**, y le pasaremos como parámetro de aquella parte que queremos modificar de la etiqueta a la que le hemos asignado la directiva. En este caso como queremos modificar el color de fondo del párrafo, le indicaremos que lo que vamos a modificar es **style.backgroundColor**.

Y dentro del constructor inicializaremos la propiedad **colorFondo** con el valor de la propiedad **color**.

 El decorador HostBinding se encarga de recoger el valor que se le asigna a la propiedad de la directiva y asignarselo a la propiedad de la etiqueta (donde se ha puesto la directiva) que se le pasa como parámetro.

Cada vez que cambia la propiedad de la directiva, se cambia el valor de la propiedad de la etiqueta.

/angular-directivas-de-atributo-crear-una-directiva-lab/src/app/marcar.directive.ts

```
import { Directive, HostBinding } from '@angular/core';

@Directive({
  selector: '[appMarcar]',
  standalone: true
})
export class MarcarDirective {
```

```

@HostBinding('style.backgroundColor') colorFondo: string;
color: string = 'yellow'

constructor() {
  this.colorFondo = this.color;
}

}

```

Si entramos al navegador (<http://localhost:4200>) ya deberíamos de ver el párrafo con el fondo de amarillo.

Pues el siguiente paso es conseguir que solo se pinte de amarillo cuando tengamos el ratón por encima del párrafo.

Para esto vamos a utilizar otro decorador, el **HostListener** que se importa desde el mismo lado que el anterior.

 El decorador HostListener se le asigna a una función, y esta función se va a ejecutar cada vez que se detecte un evento sobre la etiqueta a la que le hemos puesto la directiva.

El evento se lo indicamos nosotros pasandole el nombre como parámetro del decorador.

Vamos a añadir una función **onMouseOver** con el HostListener escuchando cuando ocurre el evento **mouseover**. Dentro de la función añadiremos la inicialización que habíamos puesto en el constructor.

Del constructor vamos a quitar dicha inicialización, y le vamos a asignar **white** como valor inicial a la propiedad **colorFondo**.

/angular-directivas-de-atributo-crear-una-directiva-lab/src/app/marcar.directive.ts

```

import { Directive, HostBinding, HostListener } from '@angular/core';

@Directive({
  selector: '[appMarcar]',
  standalone: true
})
export class MarcarDirective {

  @HostBinding('style.backgroundColor') colorFondo: string = 'white';
  color: string = 'yellow'

  constructor() {

  }

  @HostListener('mouseover') onMouseOver() {

```

```

    this.colorFondo = this.color;
}

}

```

Ya podemos ver que inicialmente el párrafo aparece en blanco, y si pasamos el ratón por encima se cambia el color al amarillo.

Pero nos encontramos con un problema, y es que si quitamos el ratón de encima del párrafo, este se queda de amarillo, cuando deberíamos de volverlo a dejar de color blanco.

Para solucionar esto, añadiremos otro **HostListener** para el evento **mouseleave** con el que cambiaremos el valor del **colorFondo** a **white**.

/angular-directivas-de-atributo-crear-una-directiva-lab/src/app/marcar.directive.ts

```

import { Directive, HostBinding, HostListener } from '@angular/core';

@Directive({
  selector: '[appMarcar]',
  standalone: true
})
export class MarcarDirective {

  @HostBinding('style.backgroundColor') colorFondo: string = 'white';
  color: string = 'yellow'

  constructor() {

  }

  @HostListener('mouseover') onMouseOver() {
    this.colorFondo = this.color;
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.colorFondo = 'white';
  }
}

```

Ahora ya funciona como se espera que lo haga. Solo se pinta de amarillo mientras mantengamos el ratón por encima del párrafo.

El siguiente paso es permitir que estos elementos se puedan pintar de distintos colores, y que seamos nosotros quienes le indiquemos de qué color queremos que se pinten asignándoselo de alguna forma a la directiva, al igual que ocurre con otras directivas que ya hemos visto.

De momento no le vamos a asignar los colores directamente a la directiva, como hemos visto que se hace con otras como el **ngStyle** a la que se le asignan los estilos a aplicar.

En nuestro caso, vamos a empezar por añadirsela a la propiedad **color** que ya tenemos declarada dentro de la directiva, y para permitir que esta propiedad reciba valores desde el exterior, le vamos a añadir el decorador **@Input**.

/angular-directivas-de-atributo-crear-una-directiva-lab/src/app/marcar.directive.ts

```
import { Directive, HostBinding, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[appMarcar]',
  standalone: true
})
export class MarcarDirective {

  @HostBinding('style.backgroundColor') colorFondo: string = 'white';
  @Input() color: string = 'yellow'

  constructor() {

  }

  @HostListener('mouseover') onMouseOver() {
    this.colorFondo = this.color;
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.colorFondo = 'white';
  }
}
```

Ahora en nuestra etiqueta del párrafo podemos añadirle una propiedad **color** e igualarle cualquier color con el que queramos que se marque la etiqueta.

Vamos a añadir un par de párrafos más con la directiva y distintos colores.

/angular-directivas-de-atributo-crear-una-directiva-lab/src/app/app.component.html

```
<p appMarcar>Si pasas el ratón por encima, el párrafo se marcará en amarillo</p>
<p appMarcar color="red">Si pasas el ratón por encima, el párrafo se marcará en rojo</p>
<p appMarcar color="blue">Si pasas el ratón por encima, el párrafo se marcará en azul</p>
```

Si probamos a pasar el ratón por encima de los tres párrafos, veremos que cada uno de ellos debe de pintarse del color que se le ha indicado, y en el caso del primer párrafo al que no se le está pasando el color desde el exterior entonces se va a quedar con el color que le hemos puesto por defecto dentro de la directiva (el amarillo).

El siguiente paso ya sería poder asignarle estos colores directamente a la directiva **appMarcar** que hemos puesto en las etiquetas.

El decorador **@Input** puede recibir un parámetro con el cual le estaríamos dando un alias, es decir, que en lugar de asignarle el color al atributo **color**, ahora se lo podríamos asignar a aquel nombre que le pasemos como parámetro.

Pues vamos a pasarle al **@Input** como parámetro el nombre de la directiva.

/angular-directivas-de-atributo-crear-una-directiva-lab/src/app/marcar.directive.ts

```
import { Directive, HostBinding, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[appMarcar]',
  standalone: true
})
export class MarcarDirective {

  @HostBinding('style.backgroundColor') colorFondo: string = 'white';
  @Input('appMarcar') color: string = 'yellow'

  constructor() {

  }

  @HostListener('mouseover') onMouseOver() {
    this.colorFondo = this.color;
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.colorFondo = 'white';
  }
}
```

Después de este cambio, ninguno de los párrafos debe de marcarse, porque ahora ya no espera recibir los colores sobre el atributo **color**, sino sobre el nombre de la directiva, como se muestra a continuación.

/angular-directivas-de-atributo-crear-una-directiva-lab/src/app/app.component.html

```
<p appMarcar>Si pasas el ratón por encima, el párrafo se marcará en amarillo</p>
<p appMarcar="red">Si pasas el ratón por encima, el párrafo se marcará en rojo</p>
<p appMarcar="blue">Si pasas el ratón por encima, el párrafo se marcará en azul</p>
```

Ahora ya lo hemos solucionado, pero no del todo, porque si nos fijamos, en el primero que es al que no le hemos asignado ningún valor, no se marca de ningún color.

Esto se debe al ciclo de vida de los componentes y directivas, cuando se pinta el componente y se aplica la directiva, lo primero que se hace es crear la instancia de la clase, inicializando las propiedades con los valores asignados directamente al declararlas, o en el constructor.

Después de esto, las propiedades se sobrescriben con aquellos valores que reciben desde el exterior (con el `@Input`), por lo que en nuestro caso el `yellow` que le hemos puesto inicialmente, se está sobrescribiendo por un string vacío porque no le hemos asignado al `appMarcar` ningún valor.

Y después de inicializar estas propiedades, es cuando se va a ejecutar el método del ciclo de vida `ngOnInit`. Por lo que para solucionar este problema, tendremos que añadir este método que viene de implementar la **interface OnInit**.

Dentro de este método vamos a inicializar la propiedad `color` con el valor `yellow` en el caso de que el valor que recibe sea un valor **falsy**.



Los valores falsy son los valores que se evalúan a false, como el propio false, un string "", el número 0, null o undefined.

`/angular-directivas-de-atributo-crear-una-directiva-lab/src/app/marcar.directive.ts`

```
import { Directive, HostBinding, HostListener, Input, OnInit } from '@angular/core';

@Directive({
  selector: '[appMarcar]',
  standalone: true
})
export class MarcarDirective implements OnInit {

  @HostBinding('style.backgroundColor') colorFondo: string = 'white';
  @Input('appMarcar') color: string = 'yellow'

  constructor() {}

  ngOnInit() {
    if (!this.color) {
      this.color = 'yellow'
    }
  }

  @HostListener('mouseover') onMouseOver() {
    this.colorFondo = this.color;
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.colorFondo = 'white';
  }
}
```

Y con esta última modificación ya vuelve a funcionar todo como cuando le asignábamos los colores a la propiedad `color`.

13.4. Directivas estructurales

Las **directivas estructurales** interaccionan con la vista actual **cambiando la estructura del DOM**. Pueden añadir, eliminar y reemplazar elementos en el DOM, y se reconocen fácilmente porque van precedidas de un @.



En versiones anteriores de Angular, se utilizaba una sintaxis distinta e iban precedidos por *.

13.5. @if

Esta directiva crea o elimina un elemento del DOM dependiendo de si se cumple o no la condición.

/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  mostrar: boolean = false;
}
```

/src/app/app.component.html

```
@if (mostrar) {
  <p>Este párrafo no se muestra</p>
}
```

13.6. @if con @else

En las primeras versiones de Angular no había nada parecido al else dentro de las directivas, por lo que si queríamos mostrar algo cuando la condición del if no se cumplía, teníamos que volver a utilizar la directiva if con la condición negada.

/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  mostrar: boolean = false;
}
```

/src/app/app.component.html

```
@if (mostrar) {
  <p>Se muestra si la propiedad mostrar es igual a true</p>
}

@if (!mostrar) {
  <p>Se muestra si la propiedad mostrar es igual a false</p>
}
```

El problema de utilizar dos veces la directiva if es que el código no es eficiente ya que cada vez que cambia el valor de la propiedad usada como condición, se tiene que evaluar dos veces la condición y mantener el estado de dos elementos.

Ahora para utilizar el else, tenemos que poner a continuación del if el **@else**.

/src/app/app.component.html

```
@if (mostrar) {
  <p>Se muestra si la propiedad mostrar es igual a true</p>
} @else {
  <p>Se muestra si la propiedad mostrar es igual a false</p>
}
```

13.7. Lab: ngIf

En este laboratorio vamos a utilizar la directiva ngIf para mostrar los botones con los que cambiar entre el modo oscuro y el modo claro de las aplicaciones.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-directivas-estructurales-ngif-lab  
? Would you like to add Angular routing? N  
? Which stylesheet format would you like to use? CSS
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-directivas-estructurales-ngif-lab  
$ ng s
```

Vamos a empezar por añadir en el TypeScript la propiedad **darkModeActivado** para conocer el estado del tema de color que habría que aplicar en la aplicación. Además vamos a añadir un método **toggleDarkMode** que recibirá un booleano como parámetro para poder cambiar entre el dark mode y el light mode.

/angular-directivas-estructurales-ngif-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'app-root',  
  standalone: true,  
  imports: [],  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
  darkModeActivado: boolean = true;  
  
  toggleDarkMode(activado: boolean) {  
    this.darkModeActivado = activado  
  }  
}
```

Ahora vamos a añadir en la plantilla dos botones, uno para activar el dark mode y el otro para activar el light mode.

/angular-directivas-estructurales-ngif-lab/src/app/app.component.html

```
<button type="button">Activar dark mode </button>
```

```
<button type="button">Activar light mode ☀</button>
```

Pero no queremos que se muestren al mismo tiempo, sino que el primero debería de mostrarse cuando **darkModeActivado** sea false y el segundo cuando sea true. Por lo que vamos a añadir la directiva **ngIf** con la propiedad anterior como condición.

/angular-directivas-estructurales-ngif-lab/src/app/app.component.html

```
@if (darkModeActivado) {  
  <button type="button">Activar light mode ☀</button>  
} @else {  
  <button type="button">Activar dark mode ☁</button>  
}
```

Solo nos queda añadir el evento **click** a los dos botones para ejecutar la función **toggleDarkMode** que añadimos al principio, y de esta forma cambiar el valor de la propiedad **darkModeActivado** para que vayan cambiando los botones a mostrar.

/angular-directivas-estructurales-ngif-lab/src/app/app.component.html

```
@if (darkModeActivado) {  
  <button type="button" (click)="toggleDarkMode(false)">Activar light mode ☀</button>  
} @else {  
  <button type="button" (click)="toggleDarkMode(true)">Activar dark mode ☁</button>  
}
```

13.8. @for

La directiva **@for** se encarga de replicar la etiqueta donde se ha añadido tantas veces como elementos haya en el array que va a iterar.

Esta directiva es como un **for of** de JavaScript, con el que se itera un array que va a ir guardando **los valores** de este en una variable que declaramos dentro del **for**.

```
const items = ['Item 1', 'Item 2', 'Item 3']

for (let item of items) {
  console.log(item)
}

// En la primera iteración muestra -> Item 1
// En la segunda iteración muestra -> Item 2
// En la tercera iteración muestra -> Item 3
```

Pues la sintaxis de la directiva es muy parecida, lo que hemos puesto dentro de los paréntesis del **for of**, es lo que pondremos como valor de la directiva **@for**.

/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  items: Array<string> = ['Item 1', 'Item 2', 'Item 3']
}
```

/src/app/app.component.html

```
<ul>
  @for (let item of items; track $index) {
    <li>{{item}}</li>
  }
</ul>
```

Lo que ocurre es que esta directiva itera sobre el array de **items**, y en la primera iteración va a guardar en la variable **item** el primer elemento del array **Item 1**. Esta variable podemos utilizarla en cualquier lugar del elemento donde hemos puesto la directiva **@or**, ya sea para asignarle valores

a los atributos de la etiqueta (en este caso el **li**), o dentro de esta etiqueta (como hemos hecho en el código de ejemplo de arriba). Y luego sigue iterando el array hasta que ya no queden elementos dentro de este.

Dentro del valor de esta directiva podemos declarar otras variables separándolas con ; para acceder a otros datos como la posición, saber si es el primer elemento o el último... Para ello tendremos que asignarle a las variables las siguientes palabras clave:

- \$index: devuelve el índice del elemento en el array.
- \$first: devuelve un true si es el primer elemento del array.
- \$last: devuelve un true si es el último elemento del array.
- \$even: devuelve un true si el elemento está en una posición par dentro del array.
- \$odd: devuelve un true si el elemento está en una posición impar dentro del array.

/src/app/app.component.html

```
<ul>
  @for (let item of items; track $index; let posicion = $index; let esUltimo = $last;) {
    <li>{{posicion}}: {{item}} {{esUltimo ? '- fin' : ''}}</li>
  }
</ul>
```

13.9. Lab: ngFor

En este laboratorio vamos a ver como utilizar la directiva ngFor para mostrar una tabla con algunos datos del top 10 de criptomonedas.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-directivas-estructurales-ngfor-lab  
? Would you like to add Angular routing? N  
? Which stylesheet format would you like to use? CSS
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-directivas-estructurales-ngfor-lab  
$ ng s
```

Vamos a empezar añadiendo una propiedad **criptomonedas** que tendrá como valor un array de objetos, donde cada objeto va a representar una criptomoneda. Estos objetos van a tener el nombre de la criptomoneda, el precio y el símbolo.

/angular-directivas-estructurales-ngfor-lab/src/app/app.component.ts

```
import { Component, NgClass } from '@angular/core';  
  
@Component({  
  selector: 'app-root',  
  standalone: true,  
  imports: [NgClass],  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
  criptomonedas: Array<any> = [  
    { nombre: 'Bitcoin', precio: 43381.89, simbolo: 'BTC' },  
    { nombre: 'Ethereum', precio: 3771.90, simbolo: 'ETH' },  
    { nombre: 'Binance Coin', precio: 513.26, simbolo: 'BNB' },  
    { nombre: 'Tether', precio: 0.884, simbolo: 'USDT' },  
    { nombre: 'Solana', precio: 164.53, simbolo: 'SOL' },  
    { nombre: 'Cardano', precio: 1.18, simbolo: 'ADA' },  
    { nombre: 'Ripple', precio: 0.795, simbolo: 'XRP' },  
    { nombre: 'USD Coin', precio: 0.88, simbolo: 'USDC' },  
    { nombre: 'Polkadot', precio: 24.90, simbolo: 'DOT' },  
    { nombre: 'Terra', precio: 61.79, simbolo: 'LUNA' },  
  ]
```

```
}
```

Ahora vamos a crear en la plantilla la estructura básica de la tabla que vamos a llenar con los datos de las criptomonedas.

La tabla va a tener una cabecera (etiqueta **thead**) con una fila (etiqueta **tr**) y esta con 3 columnas (etiqueta **th**). También le vamos a añadir el cuerpo de la tabla (etiqueta **tbody**).

```
/angular-directivas-estructurales-ngfor-lab/src/app/app.component.html
```

```
<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Símbolo</th>
      <th>Precio</th>
    </tr>
  </thead>
  <tbody>

  </tbody>
</table>
```

Dentro del cuerpo de la tabla vamos a crear una fila (etiqueta **tr**) con 3 celdas (etiquetas **td**) por cada criptomoneda del array que hemos creado.

Será en la etiqueta **tr** donde vamos a añadir la directiva **ngFor** ya que lo que queremos replicar es la fila con sus 3 celdas por cada dato que hay en el array.

Dentro de cada celda vamos a mostrar los atributos **nombre**, **símbolo** y **precio** de cada criptomoneda.

```
/angular-directivas-estructurales-ngfor-lab/src/app/app.component.html
```

```
<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Símbolo</th>
      <th>Precio</th>
    </tr>
  </thead>
  <tbody>
    @for (let cripto of criptomonedas; track $index) {
      <tr>
        <td>{{cripto.nombre}}</td>
        <td>{{cripto.simbolo}}</td>
        <td>{{cripto.precio}}</td>
      </tr>
    }
  </tbody>
</table>
```

```
</tbody>  
</table>
```

Con esta ya debería de mostrarse la tabla, pero vamos a utilizar algunas de las propiedades del `ngFor` para mejorar la visibilidad de los datos.

Empezaremos por obtener la propiedad **even** (nos devuelve true cuando la cripto está en una posición par del array), y le vamos a cambiar el color de fondo de las filas cuando las filas sean pares.

Para darle los estilos, vamos a aplicar la clase **filaPar** con la directiva **ngClass**.

`/angular-directivas-estructurales-ngfor-lab/src/app/app.component.html`

```
<table>  
  <thead>  
    <tr>  
      <th>Nombre</th>  
      <th>Símbolo</th>  
      <th>Precio</th>  
    </tr>  
  </thead>  
  <tbody>  
    @for (let cripto of criptomonedas; track $index; let esPar = $even) {  
      <tr [ngClass]="{filaPar: esPar}">  
        <td>{{cripto.nombre}}</td>  
        <td>{{cripto.simbolo}}</td>  
        <td>{{cripto.precio}}</td>  
      </tr>  
    }  
  </tbody>  
</table>
```

Ahora toca añadir la clase dentro del archivo de CSS del componente App.

Vamos a aprovechar para añadir 3 clases:

- **filaPar**: ponemos el color de fondo a gris claro cuando la fila está en una posición par.
- **primeraFila**: ponemos el borde superior de color negro y con un ancho de 1px, solo para la primera fila de la tabla.
- **ultimaFila**: ponemos el borde inferior de color negro y con un ancho de 1px, solo para la última fila de la tabla.



Para que las etiquetas **tr** muestren los estilos de sus bordes, tenemos que hacer que los bordes de las celdas (etiquetas **td** y **th**) se colapsen, por lo que hay que añadir la propiedad CSS **border-collapse: collapse;** sobre la tabla.

/angular-directivas-estructurales-ngfor-lab/src/app/app.component.css

```
table {  
  border-collapse: collapse;  
}  
  
.filaPar {  
  background-color: lightgray;  
}  
  
.primeraFila {  
  border-top: 1px solid black;  
}  
  
.ultimaFila {  
  border-bottom: 1px solid black;  
}
```

Ahora vamos a obtener con las palabras clave **first** y **last** del ngFor dos booleanos para saber cuales son las filas que ocupan la primera y última posición.

/angular-directivas-estructurales-ngfor-lab/src/app/app.component.html

```
<table>  
  <thead>  
    <tr>  
      <th>Nombre</th>  
      <th>Símbolo</th>  
      <th>Precio</th>  
    </tr>  
  </thead>  
  <tbody>  
    @for (let cripto of criptomonedas; track $index; let esPar = $even; let esPrimeraFila = $first; let esUltimaFila = $last) {  
      <tr [ngClass]="{filaPar: esPar}">  
        <td>{{cripto.nombre}}</td>  
        <td>{{cripto.simbolo}}</td>  
        <td>{{cripto.precio}}</td>  
      </tr>  
    }  
  </tbody>  
</table>
```

Una vez tenemos estos dos booleanos, vamos a añadirle a la fila un par de clases más, aquellas que hemos puesto en nuestro archivo de estilos, y el valor para estas clases será el de las variables **esPrimeraFila** y **esUltimaFila**.

/angular-directivas-estructurales-ngfor-lab/src/app/app.component.html

```
<table>  
  <thead>  
    <tr>  
      <th>Nombre</th>  
      <th>Símbolo</th>  
      <th>Precio</th>  
    </tr>
```

```

</thead>
<tbody>
  @for (let cripto of criptomonedas; track $index; let esPar = $even; let esPrimeraFila = $first; let esUltimaFila = $last) {
    <tr [ngClass]="{filaPar: esPar, primeraFila: esPrimeraFila, ultimaFila: esUltimaFila}">
      <td>{{cripto.nombre}}</td>
      <td>{{cripto.simbolo}}</td>
      <td>{{cripto.precio}}</td>
    </tr>
  }
</tbody>
</table>

```

Ya deberíamos de ver la tabla con los datos y sus distintos estilos.

Para terminar, vamos a añadir una última columna al principio de la tabla, el número que ocupa en la lista cada una de las criptomonedas.

Para añadir este nuevo valor, tenemos que añadir una etiqueta **th** más con el texto **Nº**, y una celda más dentro de la fila que lleva la directiva ngFor.

/angular-directivas-estructurales-ngfor-lab/src/app/app.component.html

```

<table>
  <thead>
    <tr>
      <th>Nº</th>
      <th>Nombre</th>
      <th>Símbolo</th>
      <th>Precio</th>
    </tr>
  </thead>
  <tbody>
    @for (let cripto of criptomonedas; track $index; let esPar = $even; let esPrimeraFila = $first; let esUltimaFila = $last) {
      <tr [ngClass]="{filaPar: esPar, primeraFila: esPrimeraFila, ultimaFila: esUltimaFila}">
        <td></td>
        <td>{{cripto.nombre}}</td>
        <td>{{cripto.simbolo}}</td>
        <td>{{cripto.precio}}</td>
      </tr>
    }
  </tbody>
</table>

```

Ahora necesitamos obtener la posición de cada una de las criptomonedas, y lo haremos añadiendo otra variable más a la que le vamos a asignar el valor de **index**, otra de las palabras clave que podemos utilizar dentro de la directiva ngFor.

Como la primera posición empieza siendo un **0**, vamos a sumarle 1 a la hora de mostrar este nuevo dato dentro del **td** que hemos añadido justo en el paso anterior.

/angular-directivas-estructurales-ngfor-lab/src/app/app.component.html

```

<table>
  <thead>
    <tr>
      <th>Nº</th>
      <th>Nombre</th>

```

```
<th>Símbolo</th>
<th>Precio</th>
</tr>
</thead>
<tbody>
@for (let cripto of criptomonedas; track $index; let esPar = $even; let esPrimeraFila = $first; let esUltimaFila = $last; let posicion = $index) {
  <tr [ngClass]="{{filaPar: esPar, primeraFila: esPrimeraFila, ultimaFila: esUltimaFila}}">
    <td>{{posicion + 1}}</td>
    <td>{{cripto.nombre}}</td>
    <td>{{cripto.simbolo}}</td>
    <td>{{cripto.precio}}</td>
  </tr>
}
</tbody>
</table>
```

13.10. @switch

La última directiva estructural que podemos encontrar en Angular es el **@switch**, una directiva que funciona como una instrucción **switch** de JavaScript. En este caso se encarga de crear/eliminar las etiquetas en función del **@case** que se vaya a ejecutar.

Esta directiva es un poco distinta a las anteriores ya que junto a ella nos encontramos 3 directivas distintas, **@switch**, **@case** y **@default**.

/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  resultado: number = 100;

}
```

/src/app/app.component.html

```
@switch (resultado) {
  @case (10) {
    <p>10</p>
  }
  @case (100) {
    <p>100</p>
  }
  @default {
    <p>Valor por defecto</p>
  }
}
```

13.11. Lab: ngSwitch

En este laboratorio vamos a utilizar la directiva ngSwitch para mostrar la clase de mascota que tenemos y que podremos elegir desde un desplegable.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-directivas-estructurales-ngswitch-lab  
? Would you like to add Angular routing? N  
? Which stylesheet format would you like to use? CSS
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-directivas-estructurales-ngswitch-lab  
$ ng s
```

Empezaremos por añadir las propiedades que vamos a necesitar en nuestro componente App. Estas propiedades serán **mascotas**, un array de strings con los tipos de mascota que vamos a poder seleccionar del desplegable, y luego una propiedad **mascotaSeleccionada** en la que guardaremos la opción del desplegable que hay seleccionada.

/angular-directivas-estructurales-ngswitch-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'app-root',  
  standalone: true,  
  imports: [],  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
  mascotas: Array<string> = [  
    'perro',  
    'gato',  
    'canario',  
    'tortuga'  
  ]  
  mascotaSeleccionada: string = 'canario'  
}
```

Una vez que tenemos las propiedades, vamos a empezar por añadir el desplegable en la plantilla del componente.

/angular-directivas-estructurales-ngswitch-lab/src/app/app.component.html

```
<select>
  <option value="perro">Perro</option>
  <option value="gato">Gato</option>
  <option value="canario">Canario</option>
  <option value="tortuga">Tortuga</option>
</select>
```

Ahora para poder seleccionar las distintas opciones y que se vaya cambiando el valor de la propiedad **mascotaSeleccionada** vamos a tener que utilizar la directiva **ngModel**, por lo que tendremos que importar el módulo de los formularios en el componente.

/angular-directivas-estructurales-ngswitch-lab/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  mascotas: Array<string> = [
    'perro',
    'gato',
    'canario',
    'tortuga'
  ]

  mascotaSeleccionada: string = 'canario'
}
```

Ya podemos añadir la directiva **ngModel** sobre la etiqueta select asignándole como valor la propiedad de la cual se va a coger el valor inicial del desplegable, y en la que vamos a guardar el valor de la opción que seleccionemos desde este.

/angular-directivas-estructurales-ngswitch-lab/src/app/app.component.html

```
<select [(ngModel)]="mascotaSeleccionada">
  <option value="perro">Perro</option>
  <option value="gato">Gato</option>
  <option value="canario">Canario</option>
  <option value="tortuga">Tortuga</option>
</select>
```

Si nos fijamos, todavía no hemos usado el array de mascotas que habíamos declarado como propiedad, y las distintas etiquetas **option** que hemos puesto son idénticas en estructura, pero cambian los datos, por lo que vamos a sustituir este código por una etiqueta **option** con la directiva **ngFor**.

Para darle el valor a value usaremos el **property binding**, es decir, meteremos el atributo entre corchetes, y luego para mostrar el tipo de mascota como texto, usaremos el **string interpolation** (las dobles llaves).

/angular-directivas-estructurales-ngswitch-lab/src/app/app.component.html

```
<select [(ngModel)]="mascotaSeleccionada">
  @for (let mascota of mascotas; track $index) {
    <option [value]="mascota">{{mascota}}</option>
  }
</select>
```

El siguiente paso es añadir los distintos párrafos que queremos mostrar dentro de un div sobre el que luego pondremos la directiva **ngSwitch**.

/angular-directivas-estructurales-ngswitch-lab/src/app/app.component.html

```
<select [(ngModel)]="mascotaSeleccionada">
  @for (let mascota of mascotas; track $index) {
    <option [value]="mascota">{{mascota}}</option>
  }
</select>

<div>
  <p>Tengo un perro</p>
  <p>Tengo un gato</p>
  <p>Tengo otra mascota</p>
</div>
```

Dentro del div vamos a añadir la directiva **ngSwitch** a la que le vamos a asignar el valor con el cual se irán comparando los distintos **case** que pondremos después. En este caso le asignaremos el valor de la propiedad **mascotaSeleccionada**.

/angular-directivas-estructurales-ngswitch-lab/src/app/app.component.html

```
<select [(ngModel)]="mascotaSeleccionada">
  @for (let mascota of mascotas; track $index) {
    <option [value]="mascota">{{mascota}}</option>
  }
</select>

<div>
  @switch (mascotaSeleccionada) {
    <p>Tengo un perro</p>
    <p>Tengo un gato</p>
```

```
<p>Tengo otra mascota</p>
}
</div>
```

Ahora vamos a añadir las directivas de **ngSwitchCase** sobre los dos primeros párrafos, y les asignaremos **los strings** "perro" y "gato".

/angular-directivas-estructurales-ngswitch-lab/src/app/app.component.html

```
<select [(ngModel)]="mascotaSeleccionada">
  @for (let mascota of mascotas; track $index) {
    <option [value]="mascota">{{mascota}}</option>
  }
</select>

<div>
  @switch (mascotaSeleccionada) {
    @case ('perro') {
      <p>Tengo un perro</p>
    }
    @case ('gato') {
      <p>Tengo un gato</p>
    }
    <p>Tengo otra mascota</p>
  }
</div>
```

Por último, en el tercer párrafo vamos a añadirle la directiva **ngSwitchDefault** para que este párrafo se muestre cuando la mascota seleccionada sea distinta a perro y gato.

/angular-directivas-estructurales-ngswitch-lab/src/app/app.component.html

```
<select [(ngModel)]="mascotaSeleccionada">
  @for (let mascota of mascotas; track $index) {
    <option [value]="mascota">{{mascota}}</option>
  }
</select>

<div>
  @switch (mascotaSeleccionada) {
    @case ('perro') {
      <p>Tengo un perro</p>
    }
    @case ('gato') {
      <p>Tengo un gato</p>
    }
    @default {
      <p>Tengo otra mascota</p>
    }
  }
</div>
```

```
</div>
```

Chapter 14. Pipes

Los **pipes** permiten coger un dato de entrada y transformarlo sin cambiar su valor, solo la forma en que se visualiza. Se llaman pipes porque para usarlos vamos a poner el símbolo | (barra de la tecla del 1).

La sintaxis para utilizar los pipes es como podemos ver a continuación:

```
 {{ datoDeEntrada | nombreDelPipe }}
```

Normalmente los utilizaremos junto al **String Interpolation** como podemos ver justo encima, pero no tiene porque ser siempre así, también podemos utilizarlos dentro de directivas como el **ngFor** como ya veremos más adelante.

Los datos que vamos a transformar los tendremos que declarar previamente en los archivos de TypeScript de nuestros componentes, o podemos poner los valores directamente en el propio String Interpolation.

```
 {{ propiedadDelComponente | nombreDelPipe }}  
 {{ 'un string' | nombreDelPipe }}  
 {{ 123 | nombreDelPipe }}
```

Los pipes pueden recibir parámetros para modificar de alguna forma como se tiene que mostrar los datos después de la transformación que se va a realizar, por ejemplo, para cambiar el formato de las fechas que vamos a mostrar.

Para pasarle los parámetros a los pipes, los vamos separando con : después del nombre del pipe. Estos parámetros también pueden ser propiedades que se hayan declarado dentro de la clase del componente, pero también podemos pasarle los valores directamente.

```
 {{ propiedadDelComponente | nombreDelPipe:param1:param2 }}  
 {{ 'un string' | nombreDelPipe:[1, 2]:'un parámetro del tipo string' }}
```

Además de lo indicado antes, podemos concatenar diferentes pipes, donde la salida de un pipe será el valor de entrada del pipe que se aplique a continuación.

```
 {{ propiedadDelComponente | nombreDelPipe1:param1 | nombreDelPipe2:2:param2 }}
```



Al concatenar los pipes, hay que tener cuidado con el orden en que los ponemos, ya que si el primer pipe devuelve un valor del tipo string, y el siguiente pipe espera recibir como valor de entrada uno del tipo numerico, esto no funcionará.

Angular ya trae internamente una serie de pipes, donde algunos de ellos son:

- lowercase
- uppercase
- titlecase
- slice
- currency
- date
- json
- async
- ...

Podemos encontrar todos los pipes en <https://angular.io/api?type=pipe>.

Los pipes se han vuelto también **standalone**, por lo que tendremos que importar estos pipes dentro de los **componentes standalone** donde se vayan a utilizar:

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, TitleCasePipe, CurrencyPipe, DatePipe, SlicePipe],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
...
```

14.1. Pipe uppercase

El pipe **uppercase** se encarga de transformar un string a mayúsculas.

/ proyecto-angular/src/app/app.component.ts

```
valor: string = 'esto es un string';
```

/ proyecto-angular/src/app/app.component.html

```
<p>{{ valor }}</p> <!-- esto es un string -->
<p>{{ valor | uppercase }}</p> <!-- ESTO ES UN STRING -->
```

14.2. Pipe lowercase

El pipe **lowercase** se encarga de transformar un string a minúsculas.

/ proyecto-angular/src/app/app.component.ts

```
valor: string = 'Esto es un STRING';
```

/ proyecto-angular/src/app/app.component.html

```
<p>{{ valor }}</p> <!-- Esto es un STRING -->
<p>{{ valor | lowercase }}</p> <!-- esto es un string -->
```

14.3. Pipe titlecase

El pipe **titlecase** se encarga de poner la primera letra de cada palabra de un string en mayúsculas.

/ proyecto-angular/src/app/app.component.ts

```
valor: string = 'esto es un string';
```

/ proyecto-angular/src/app/app.component.html

```
<p>{{ valor }}</p> <!-- esto es un string -->
<p>{{ valor | titlecase }}</p> <!-- Esto Es Un String -->
```

14.4. Pipe currency

El pipe **currency** se encarga de formatear números añadiendo dos decimales y el símbolo de una moneda. Por defecto añade el símbolo del \$, aunque esto podemos cambiarlo añadiéndole un parámetro con el código de la moneda que queremos que se muestre.

/ proyecto-angular/src/app/app.component.ts

```
precio: number = 10
```

/ proyecto-angular/src/app/app.component.html

```
<p>{{ precio }}</p> <!-- 10 -->
<p>{{ precio | currency }}</p> <!-- $10.00 -->
<p>{{ precio | currency:'EUR' }}</p> <!-- €10.00 -->
```

Podemos ver todos los códigos de las monedas en https://en.wikipedia.org/wiki/ISO_4217.

14.5. Pipe date

El pipe **date** se encarga de formatear una fecha dada. A este pipe le podemos pasar como parámetro el formato (un string) en el que queremos que se muestre la fecha.

Para crear el formato utilizaremos una serie de letras para definir el tipo de dato a mostrar donde:

- **d**: representa el número del día
- **M**: representa el mes
- **y**: representa el año
- **h**: representa la hora
- **m**: representa los minutos
- ...

Podemos encontrar más información sobre como crear los formatos utilizando estas letras en <https://angular.io/api/common/DatePipe#custom-format-options>.

Para hacernos una idea de algunos de los más usados:

- **dd**: el número del día con dos dígitos
- **MM**: el número del mes con dos dígitos
- **MMM**: las tres primeras iniciales del mes
- **MMMM**: el nombre completo del mes
- **yyyy**: el número del año con cuatro dígitos

/ proyecto-angular/src/app/app.component.ts

```
fecha = new Date(2021, 11, 2);
```

/ proyecto-angular/src/app/app.component.html

```
<p>{{ fecha }}</p> <!-- Thu Dec 02 2021 00:00:00 GMT+0100 (hora estándar de Europa central) -->
<p>{{ fecha | date }}</p> <!-- Dec 2, 2021 -->
<p>{{ fecha | date:'dd/MM/yyyy' }}</p> <!-- 02/12/2021 -->
<p>{{ fecha | date:'dd MMMM yyyy' }}</p> <!-- 02 December 2021 -->
```

14.6. Pipe slice

El pipe **slice** devuelve un substring del valor al que se aplica. Este pipe puede recibir un primer parámetro con el que indicamos a partir de que letra va a comenzar el substring, y un segundo parámetro indicando cuantas letras queremos obtener (empezando a contar desde el valor del primer parámetro).

/ proyecto-angular/src/app/app.component.ts

```
valor: string = 'Esto es un string'
```

/ proyecto-angular/src/app/app.component.html

```
<p>{{ valor }}</p><!-- Esto es un string -->
<p>{{ valor | slice:5 }}</p><!-- es un string-->
<p>{{ valor | slice:3:9 }}</p><!-- o es u -->
<p>{{ valor | slice:0:4 }}</p><!-- Esto -->
```

14.7. Pipe json

El pipe **json** se encarga de mostrar un objeto/array de TypeScript en formato JSON.



Internamente, el pipe json utilizará la función `JSON.stringify(miObjeto)`.

`/ proyecto-angular/src/app/app.component.ts`

```
serie = {  
    titulo: 'Manhunt: Unabomber',  
    temporadas: 2,  
    finalizada: true  
}
```

`/ proyecto-angular/src/app/app.component.html`

```
<pre>{{ serie }}</pre>  
<pre>{{ serie | json }}</pre>
```

[object Object]

```
{  
    "titulo": "Manhunt: Unabomber",  
    "temporadas": 2,  
    "finalizada": true  
}
```

14.8. Lab: Pipes

En este laboratorio vamos a utilizar los pipes vistos antes para mostrar los datos de un producto con un formato más legible.

Utilizaremos como datos del producto el siguiente objeto:

```
producto = {  
    nombre: 'one plus 8t',  
    descripcion: 'OnePlus 8T 5G - Smartphone FHD de 6.55 "120 Hz + pantalla fluida, 8 GB de RAM + 128 GB de espacio de  
almacenamiento, cámara cuádruple, carga Warp de 65 W, SIM dual, 5G, Plata (Lunar Silver)',  
    precio: 517.9,  
    fechaEntrega: new Date(2021, 8, 12)  
}
```

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-pipes-lab  
  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-pipes-lab  
$ ng s
```

Lo primero que vamos a hacer es declarar una propiedad producto como la que tenemos en el enunciado del laboratorio, dentro del archivo de **app.component.ts**.

/angular-pipes-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';  
import { CurrencyPipe, DatePipe, SlicePipe, TitleCasePipe } from '@angular/common';  
import { RouterOutlet } from '@angular/router';  
  
@Component({  
    selector: 'app-root',  
    standalone: true,  
    imports: [RouterOutlet, TitleCasePipe, CurrencyPipe, DatePipe, SlicePipe],  
    templateUrl: './app.component.html',  
    styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
    producto = {  
        nombre: 'one plus 8t',  
        descripcion: 'OnePlus 8T 5G - Smartphone FHD de 6.55 "120 Hz + pantalla fluida, 8 GB de RAM + 128 GB de espacio de  
almacenamiento, cámara cuádruple, carga Warp de 65 W, SIM dual, 5G, Plata (Lunar Silver)',  
        precio: 517.9,  
        fechaEntrega: new Date(2021, 8, 12)  
    }  
}
```

Ahora vamos a crear la estructura para pintar este producto dentro de la plantilla del componente App.

/angular-pipes-lab/src/app/app.component.html

```
<div>
  <p>{{producto.nombre}}</p>
  <div>
    <p>{{producto.precio}}</p>
    <p>{{producto.fechaEntrega}}</p>
  </div>
  <p>{{producto.descripcion}}</p>
</div>
```

Para empezar, queremos que el nombre del producto aparezca con la primera letra de cada palabra en mayúscula, por lo que sobre este primer dato vamos a utilizar el pipe **titlecase**.

/angular-pipes-lab/src/app/app.component.html

```
<div>
  <p>{{producto.nombre | titlecase}}</p>
  <div>
    <p>{{producto.precio}}</p>
    <p>{{producto.fechaEntrega}}</p>
  </div>
  <p>{{producto.descripcion}}</p>
</div>
```

La siguiente mejora que vamos a hacer es poner el símbolo del euro junto al precio, además de añadirle el segundo decimal. Aquí utilizaremos el pipe **currency** pasandole como parámetro el código **EUR** para que utilice el símbolo de € en lugar del \$.

/angular-pipes-lab/src/app/app.component.html

```
<div>
  <p>{{producto.nombre | titlecase}}</p>
  <div>
    <p>{{producto.precio | currency:'EUR'}}</p>
    <p>{{producto.fechaEntrega}}</p>
  </div>
  <p>{{producto.descripcion}}</p>
</div>
```

Como podemos ver, la fecha muestra también la hora y la zona horaria, datos que no necesitamos mostrar, por lo que vamos a proceder a quitarlos con el pipe **date** el cual se va a quedar solo con la parte de la fecha para mostrarla.

También queremos cambiar el formato en el que vamos a mostrar la fecha, por lo que vamos a pasarselo como parámetro del pipe. En este caso queremos que la fecha se muestre como 12

September 2021, por lo que utilizaremos como formato 'dd MMMM yyyy'.

/angular-pipes-lab/src/app/app.component.html

```
<div>
  <p>{{producto.nombre | titlecase}}</p>
  <div>
    <p>{{producto.precio | currency:'EUR'}}</p>
    <p>{{producto.fechaEntrega | date:'dd MMMM yyyy'}}</p>
  </div>
  <p>{{producto.descripcion}}</p>
</div>
```

Y por último, de la descripción no queremos mostrar tantas letras, nos vamos a quedar solo con las 150 primeras con el pipe del **slice**, al que le vamos a pasar dos parámetros.

/angular-pipes-lab/src/app/app.component.html

```
<div>
  <p>{{producto.nombre | titlecase}}</p>
  <div>
    <p>{{producto.precio | currency:'EUR'}}</p>
    <p>{{producto.fechaEntrega | date:'dd MMMM yyyy'}}</p>
  </div>
  <p>{{producto.descripcion | slice:0:150}}...</p>
</div>
```

Ahora la información que estamos mostrando queda más legible para el usuario que la va a ver, aunque se puede mejorar todavía un poco más añadiéndole un poco de CSS.

/angular-pipes-lab/src/app/app.component.css

```
.producto {
  border: 1px solid black;
  border-radius: 5px;
  box-shadow: 3px 3px 6px black;
  width: 300px;
  padding: 10px 20px;
  background: linear-gradient(90deg, rgba(255,255,255,1) 0%, rgba(224,200,200,1) 68%);
}

.producto > p.titulo {
  text-align: center;
  font-size: 20px;
  font-weight: bolder;
}

.producto > .otros-datos {
  display: flex;
  justify-content: space-evenly;
```

```
}

.producto > .otros-datos > p {
  border: 1px solid black;
  border-radius: 20px;
  padding: 5px;
}

.producto > p.descripcion {
  text-align: justify;
}
```

Y ahora para aplicar estos estilos vamos a añadir las clases en las distintas etiquetas que teníamos en el HTML.

/angular-pipes-lab/src/app/app.component.html

```
<div class="producto">
  <p class="titulo">{{producto.nombre | titlecase}}</p>
  <div class="otros-datos">
    <p>{{producto.precio | currency:'EUR'}}</p>
    <p>{{producto.fechaEntrega | date:'dd MMMM yyyy'}}</p>
  </div>
  <p class="descripcion">{{producto.descripcion | slice:0:150}}...</p>
</div>
```

14.9. Crear un pipe

Es posible crear pipes propios con cualquiera de los siguientes comandos:

```
$ ng generate pipe mi-nuevo-pipe  
$ ng g p mi-nuevo-pipe
```

Con este comando, se generan dos archivos y se actualiza otro:

- Se crea **mi-nuevo-pipe.pipe.ts** donde vamos a añadir la funcionalidad del pipe.
- Se crea **mi-nuevo-pipe.pipe.spec.ts** para los tests.
- Se actualiza **app.module.ts** para añadir este pipe en las declaraciones del módulo para que al utilizar este pipe angular lo reconozca.

En el archivo donde se define el pipe se encuentra el decorator **@Pipe** en el cual nos encontramos la propiedad **name** que nos indica el nombre con el que vamos a llamar al pipe cuando lo queramos aplicar sobre algún dato.

/src/app/mi-nuevo-pipe.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';  
  
@Pipe({  
  name: 'miNuevoPipe',  
  standalone: true  
})  
export class MiNuevoPipe implements PipeTransform {  
  transform(value: unknown, ...args: unknown[]): unknown {  
    return null;  
  }  
}
```

La función de **transform** viene de implementar la interfaz **PipeTransform**, y esta función se ejecuta cada vez que aplicamos el pipe sobre algún dato.

El primer parámetro que se recibe es el valor que va antes del símbolo del | cuando aplicamos estos.

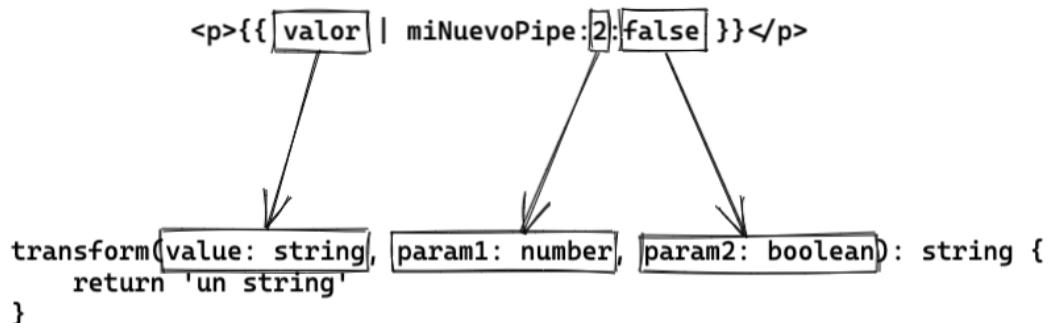
Luego puede recibir más parámetros que se guardarán en la lista de **args**, aunque esta definición la podemos cambiar, y poner el número exacto de parámetros y sus tipos de datos que vayamos a necesitar.

/src/app/mi-nuevo-pipe.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';  
  
@Pipe({  
  name: 'miNuevoPipe',  
  standalone: true  
})  
export class MiNuevoPipe implements PipeTransform {  
  transform(value: unknown, ...args: unknown[]): unknown {  
    return value + args[0];  
  }  
}
```

```
    standalone: true
})
export class MiNuevoPipe implements PipeTransform {
  transform(value: string, param1: number, param2: boolean): string {
    return null;
  }
}
```

En el dibujo que se muestra a continuación podemos ver que valores de los usados al aplicar el pipe llegarían a que parámetros de la función transform del pipe.



14.10. Lab: Crear custom pipes

En este laboratorio vamos a crear los siguientes pipes:

- Crear un pipe **reverse** que le de la vuelta a los strings:
 - `{{ 'hola mundo' | reverse }} ⇒ 'odnum aloh'`
- Crear un pipe **ocultar** que cambie las palabras que se le pasan por parámetro (en un array) por asteriscos:
 - `{{ 'La noche se avecina, ahora empieza mi guardia. No terminará hasta el día de mi muerte. No tomaré esposa, no poseeré tierras, no engendraré hijos. No llevaré corona, no alcanzaré la gloria. Viviré y moriré en mi puesto. Soy la espada en la oscuridad. Soy el vigilante del Muro. Soy el fuego que arde contra...' | ocultar:['no', 'soy'] }} ⇒ 'La $che se avecina, ahora empieza mi guardia. $ terminará hasta el día de mi muerte. $ tomaré esposa, $ poseeré tierras, $ engendraré hijos. $ llevaré corona, $ alcanzaré la gloria. Viviré y moriré en mi puesto. $ la espada en la oscuridad. $ el vigilante del Muro. $ el fuego que arde contra...'`

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-pipes-crear-un-pipe-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y dentro de ella vamos a empezar creando los dos pipes con los siguientes comandos:

```
$ cd angular-pipes-crear-un-pipe-lab  
$ ng g p reverse  
$ ng g p ocultar  
$ ng s
```

Vamos a empezar a crear el primero, el pipe **reverse**, para ello vamos a abrir el archivo de **reverse.pipe.ts** donde nos encontramos la función de **transform** que tenemos que rellenar.

Vamos a empezar cambiando la definición de esta función, indicando que el valor que vamos a recibir es un string, que no va a recibir ningún parámetro extra y por último que este pipe va a retornar un string.

/angular-pipes-crear-un-pipe-lab/src/app/reverse.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';  
  
@Pipe({  
  name: 'reverse',  
  standalone: true  
})
```

```

export class ReversePipe implements PipeTransform {

  transform(value: string): string {
    return null;
  }

}

```

Vamos a ir añadiendo la funcionalidad del pipe, paso a paso. Primero, vamos a coger el **value** que es el string al que queremos darle la vuelta, y lo vamos a separar en un array de letras. Utilizaremos la función de **split(separador)** de los strings, y le pasaremos como separados un string vacío indicando que hay que separar dicho string letra a letra.

/angular-pipes-crear-un-pipe-lab/src/app/reverse.pipe.ts

```

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

@Pipe({
  name: 'reverse',
  standalone: true
})
export class ReversePipe implements PipeTransform {

  transform(value: string): string {
    const arrayLetras = value.split('')
    return null;
  }

}

```

Los arrays tienen un método **reverse** que se encarga de darle la vuelta a todos los elementos, por tanto, vamos a utilizarlo sobre el array de letras que hemos obtenido en el paso anterior.

/angular-pipes-crear-un-pipe-lab/src/app/reverse.pipe.ts

```

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

@Pipe({
  name: 'reverse',
  standalone: true
})
export class ReversePipe implements PipeTransform {

  transform(value: string): string {
    const arrayLetras = value.split('')
    const arrayLetrasAlReves = arrayLetras.reverse()
    return null;
  }

}

```

```
}
```

Ya casi lo tenemos, el último paso es coger dicho array de letras y volverlas a unir para formar un string. Esto lo vamos a conseguir con la función de **join(separador)** que podemos utilizar con los arrays, donde el parámetro separador será un string vacío para que deje todas las letras juntas.

/angular-pipes-crear-un-pipe-lab/src/app/reverse.pipe.ts

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

@Pipe({
  name: 'reverse',
  standalone: true
})
export class ReversePipe implements PipeTransform {

  transform(value: string): string {
    const arrayLetras = value.split('')
    const arrayLetrasAlReves = arrayLetras.reverse()
    return arrayLetrasAlReves.join('');
  }
}
```

Y con esto ya tenemos nuestro pipe, ahora solo tenemos que probar que hace lo que debe. Por tanto, dentro de nuestro componente App, vamos a crear un string al que luego le vamos a dar la vuelta con este pipe que acabamos de crear.

/angular-pipes-crear-un-pipe-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  texto: string = 'Hola mundo'
}
```

Para poder utilizar el pipe, tenemos que importarlo, por lo que vamos a ponerlo en el array de imports del componente.

/angular-pipes-crear-un-pipe-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { ReversePipe } from './reverse.pipe';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, ReversePipe],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  texto: string = 'Hola mundo'
}
```

Ahora en el HTML del componente App, vamos a aplicar el pipe sobre esta propiedad texto que acabamos de añadir.

/angular-pipes-crear-un-pipe-lab/src/app/app.component.html

```
<p>{{texto | reverse}}</p>
```

Y ya deberíamos de ver en nuestro navegador <http://localhost:4200/> el siguiente texto **odnum aloH**.

Ya tenemos el primer pipe, ahora vamos a por el segundo.

Abriremos el archivo **ocultar.pipe.ts**, donde vamos a modificar la definición de la función transform del pipe. Le indicaremos que:

- El valor que recibe es un string.
- El valor que va a retornar es un string.
- Recibe un segundo parámetro con las palabras que queremos ocultar, y el tipo será un array de strings.

/angular-pipes-crear-un-pipe-lab/src/app/ocultar.pipe.ts

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

@Pipe({
  name: 'ocultar',
  standalone: true
})
export class OcultarPipe implements PipeTransform {

  transform(value: string, palabras: Array<string>): string {
    return null;
}
```

```
}
```

```
}
```

La idea es recorrer el array de palabras e ir modificando el value reemplazando las coincidencias de estas palabras con tantos asteriscos como letras tenga cada palabra.

El primer paso va a ser recorrer el array de palabras.

/angular-pipes-crear-un-pipe-lab/src/app/ocultar.pipe.ts

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

@Pipe({
  name: 'ocultar',
  standalone: true
})
export class OcultarPipe implements PipeTransform {

  transform(value: string, palabras: Array<string>): string {
    palabras.forEach(palabra => {

    })
    return null;
  }
}
```

Ahora vamos a crear una expresión regular para buscar la palabra dentro del value. A esta expresión regular le añadiremos los flags:

- g: para que busque todas las coincidencias
- i: para que no tenga en cuenta mayúsculas/minúsculas

/angular-pipes-crear-un-pipe-lab/src/app/ocultar.pipe.ts

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

@Pipe({
  name: 'ocultar',
  standalone: true
})
export class OcultarPipe implements PipeTransform {

  transform(value: string, palabras: Array<string>): string {
    palabras.forEach(palabra => {
      const regExp = new RegExp(palabra, 'gi')
    })
  }
}
```

```
    return null;
}

}
```

Ahora vamos a utilizar la función **replace** sobre el value, en la que le indicaremos que queremos reemplazar las coincidencias que haya con la expresión regular que acabamos de crear, y como segundo parámetro le vamos a pasar el texto con el que queremos reemplazar dichas coincidencias.

/angular-pipes-crear-un-pipe-lab/src/app/ocultar.pipe.ts

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

@Pipe({
  name: 'ocultar',
  standalone: true
})
export class OcultarPipe implements PipeTransform {

  transform(value: string, palabras: Array<string>): string {
    palabras.forEach(palabra => {
      const regExp = new RegExp(palabra, 'gi')
      value = value.replace(regExp, '*')
    })

    return null;
  }
}
```

Ahora mismo cada coincidencia que haya se va a reemplazar por un único asterisco, pero nosotros queremos que sea un asterisco por cada letra que tenga la palabra a reemplazar, por lo que vamos a utilizar la función **repeat** de los strings, y le pasaremos como parámetro la longitud de la palabra.

/angular-pipes-crear-un-pipe-lab/src/app/ocultar.pipe.ts

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

@Pipe({
  name: 'ocultar',
  standalone: true
})
export class OcultarPipe implements PipeTransform {

  transform(value: string, palabras: Array<string>): string {
    palabras.forEach(palabra => {
      const regExp = new RegExp(palabra, 'gi')
      value = value.replace(regExp, '*'.repeat(palabra.length))
    })
  }
}
```

```

        return null;
    }

}

```

El último paso es retornar el **value** en lugar del null.

/angular-pipes-crear-un-pipe-lab/src/app/ocultar.pipe.ts

```

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

@Pipe({
  name: 'ocultar',
  standalone: true
})
export class OcultarPipe implements PipeTransform {

  transform(value: string, palabras: Array<string>): string {
    palabras.forEach(palabra => {
      const regExp = new RegExp(palabra, 'gi')
      value = value.replace(regExp, '*'.repeat(palabra.length))
    })

    return value;
  }
}

```

Pues solo tenemos que ir al componente de App, para añadir el nuevo texto y aplicarle el nuevo pipe con una serie de palabras como parámetro.

/angular-pipes-crear-un-pipe-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { ReversePipe } from './reverse.pipe';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, ReversePipe],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  texto: string = 'Hola mundo'
  juramento: string = 'Escuchad mis palabras, sed testigos de mi juramento ... La noche se avecina, ahora empieza mi guardia. No terminará hasta el día de mi muerte. No tomaré esposa, no poseeré tierras, no engendraré hijos. No llevaré corona, no alcanzaré la gloria. Viviré y moriré en mi puesto. Soy la espada en la oscuridad. Soy el vigilante del Muro. Soy el fuego que arde contra el frío, la luz que trae el amanecer, el cuerno que despierta a los durmientes, el escudo que defiende los reinos de los hombres. Entrego mi vida y mi honor a la Guardia de la Noche, durante esta noche y todas las que estén por venir.'
}

```

También tenemos que importar el pipe **ocultar** en el array de imports del componente.

/angular-pipes-crear-un-pipe-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { ReversePipe } from './reverse.pipe';
import { OcultarPipe } from './ocultar.pipe';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, ReversePipe, OcultarPipe],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  texto: string = 'Hola mundo'
  juramento: string = 'Escuchad mis palabras, sed testigos de mi juramento ... La noche se avecina, ahora empieza mi guardia. No terminará hasta el día de mi muerte. No tomaré esposa, no poseeré tierras, no engendraré hijos. No llevaré corona, no alcanzaré la gloria. Viviré y moriré en mi puesto. Soy la espada en la oscuridad. Soy el vigilante del Muro. Soy el fuego que arde contra el frío, la luz que trae el amanecer, el cuerno que despierta a los durmientes, el escudo que defiende los reinos de los hombres. Entrego mi vida y mi honor a la Guardia de la Noche, durante esta noche y todas las que estén por venir.'
}
```

Ahora en el HTML del componente App, vamos a aplicarle al **juramento** el pipe **ocultar** pasándole como parámetro un array con las palabras:

- no
- soy

/angular-pipes-crear-un-pipe-lab/src/app/app.component.html

```
<p>{{texto | reverse}}</p>
<p>{{juramento | ocultar:['no', 'soy']}}</p>
```

Ya deberíamos de ver el texto con las palabras ocultas, quedando como se muestra a continuación:

odnum aloH

Escuchad mis palabras, sed testigos de mi juramento ... La **che se avecina, ahora empieza mi guardia. ** terminará hasta el día de mi muerte. ** tomaré esposa, ** poseeré tierras, ** engendraré hijos. ** llevaré corona, ** alcanzaré la gloria. Viviré y moriré en mi puesto. *** la espada en la oscuridad. *** el vigilante del Muro. *** el fuego que arde contra el frío, la luz que trae el amanecer, el cuer** que despierta a los durmientes, el escudo que defiende los rei**s de los hombres. Entrego mi vida y mi ho**r a la Guardia de la **che, durante esta **che y todas las que estén por venir.

14.11. Pipes puros e impuros

Todos los pipes vistos anteriormente se consideran **pipes puros**, es decir, pipes que solo se va a ejecutar cuando cambia el valor al que se aplican, ya sea un valor primitivo, o un valor por referencia, o cuando cambia alguno de los parámetros que recibe.

En los valores por referencia, los pipes puros se aplican cuando cambia la referencia, no el contenido de esta.



Por ejemplo, el método **push** de los arrays mutan este, pero no cambia su referencia, por lo que no detectaría dicho cambio y no se volvería a aplicar el pipe.

Para que se aplique el pipe en el caso anterior, tendríamos que crear un nuevo array con el contenido que tenía, agregarle el nuevo elemento al final y asignarselo al dato al cual se le ha aplicado el pipe.

Al igual que tenemos los pipes puros, nos encontramos con los **pipes impuros** los cuales se ejecutan cada vez que se modifica cualquier elemento de la aplicación, tenga o no tenga que ver con los datos que se usan en el pipe. Por lo que este tipo de pipes son muy inefficientes ya que con cada cambio en la aplicación se ejecutará esta función.

Para indicar que un pipe es impuro, solo tenemos que añadir la propiedad **pure: false** dentro del decorador del pipe, como podemos ver a continuación:

/src/app/mi-pipe.ts

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

@Pipe({
  name: 'miPipe',
  standalone: true,
  pure: false
})
export class MiPipePipe implements PipeTransform {

  transform(value: string): string {
    return 'un pipe';
  }
}
```

14.12. Lab: Pipes puros e impuros

En este laboratorio vamos a ver como utilizar un pipe impuro, y después veremos como podemos pasar ese pipe a un pipe puro para que sea más eficiente nuestro código.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-pipes-puros-e-impuros-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado y dentro de ella vamos a empezar creando el pipe filtro y después levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-pipes-puros-e-impuros-lab  
$ ng g p filtro  
$ ng s
```

Vamos a empezar por crear una lista de personajes en nuestro componente App e importando el pipe que hemos creado.

/angular-pipes-puros-e-impuros-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FiltroPipe } from './filtro.pipe';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FiltroPipe],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  personajes: Array<string> = [
    'Octavia Blake',
    'Tony Stark',
    'Arya Stark',
    'Charles Falco',
  ]
}
```

Una vez tenemos el array, vamos a mostrar estos datos en una lista, y para ello usaremos la directiva ***ngFor** con la que vamos a iterar **personajes** guardando cada uno de ellos en la variable

personaje por cada iteración.

/angular-pipes-puros-e-impuros-lab/src/app/app.component.html

```
<ul>
  @for (let personaje of personajes; track $index) {
    <li>{{personaje}}</li>
  }
</ul>
```

Ahora vamos a añadir un campo de texto desde el que vamos a añadir nuevos personajes en el array de personajes. Sobre este campo, vamos a detectar el evento **change** y le pasaremos el objeto **\$event** a la función a la que se llamará cuando hayamos escrito el valor y pulsemos la tecla de Enter, o pulsemos fuera del input.

/angular-pipes-puros-e-impuros-lab/src/app/app.component.html

```
<div>
  <label for="nuevo-personaje">Nuevo personaje:</label>
  <input type="text" (change)="addPersonaje($event)">
</div>
<ul>
  @for (let personaje of personajes; track $index) {
    <li>{{personaje}}</li>
  }
</ul>
```

Ahora en el typescript del componente vamos a añadir la función de **addPersonaje** la que va a recibir como parámetro un objeto **event** con la información del evento que se ha detectado en el input.

Dentro de la función, vamos a extraer del evento el valor que hemos escrito en el input, y luego lo añadiremos al final del array de personajes que ya tenemos.

/angular-pipes-puros-e-impuros-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FiltroPipe } from './filtro.pipe';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FiltroPipe],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  personajes: Array<string> = [
```

```

    'Octavia Blake',
    'Tony Stark',
    'Arya Stark',
    'Charles Falco',
]

addPersonaje(event: any): void {
  const nuevoPersonaje = event.target.value
  this.personajes.push(nuevoPersonaje)
}
}

```

Ahora ya podemos probar a añadir nuevos personajes al array, y veremos que al hacerlo la lista se va a actualizando automáticamente.

El siguiente paso es duplicar la lista en la plantilla del componente y añadir otro campo de texto, pero esta vez lo vamos a utilizar para poner el texto por el cual queremos filtrar la esta segunda lista.

/angular-pipes-puros-e-impuros-lab/src/app/app.component.html

```

<div>
  <label for="nuevo-personaje">Nuevo personaje:</label>
  <input type="text" (change)="addPersonaje($event)">
</div>
<ul>
  @for (let personaje of personajes; track $index) {
    <li>{personaje}</li>
  }
</ul>

<hr>

<div>
  <label for="nuevo-personaje">Filtrar por:</label>
  <input type="text" [(ngModel)]="filtroTxt">
</div>
<ul>
  @for (let personaje of personajes; track $index) {
    <li>{personaje}</li>
  }
</ul>

```

Como estamos usando el **two-way data binding** (directiva **ngModel**) necesitaremos importar el módulo de los formularios, el **FormsModule**, en el **app.component.ts**.

/angular-pipes-puros-e-impuros-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';

```

```

import { RouterOutlet } from '@angular/router';
import { FiltroPipe } from './filtro.pipe';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FiltroPipe, FormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  personajes: Array<string> = [
    'Octavia Blake',
    'Tony Stark',
    'Arya Stark',
    'Charles Falco',
  ]

  addPersonaje(event: any): void {
    const nuevoPersonaje = event.target.value
    this.personajes.push(nuevoPersonaje)
  }
}

```

También hemos añadido en el input una propiedad **filtroTxt** como valor del **ngModel**, por tanto tenemos que declarar e inicializar dicha propiedad en el typescript del componente App.

/angular-pipes-puros-e-impuros-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FiltroPipe } from './filtro.pipe';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FiltroPipe, FormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  personajes: Array<string> = [
    'Octavia Blake',
    'Tony Stark',
    'Arya Stark',
    'Charles Falco',
  ]
}

```

```

filtroTxt: string = ''

addPersonaje(event: any): void {
  const nuevoPersonaje = event.target.value
  this.personajes.push(nuevoPersonaje)
}

}

```

Una vez que tenemos la estructura y la funcionalidad para añadir nuevos personajes en el array, vamos a crear implementar el pipe **filtro** que habíamos creado al principio.

Abrimos el archivo **filtro.pipe.ts**, y vamos a cambiar los tipos de datos.

Este pipe se va a aplicar a un array de strings (el array de personajes), y va a recibir un segundo parámetro de tipo string que va a ser el filtro. Finalmente, como valor de retorno tendremos un array de strings, con aquellos personajes que cumplan con el filtro.

/angular-pipes-puros-e-impuros-lab/src/app/filtro.pipe.ts

```

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

@Pipe({
  name: 'filtro',
  standalone: true
})
export class FiltroPipe implements PipeTransform {

  transform(value: Array<string>, filtro: string): Array<string> {
    return null;
  }
}

```

Para filtrar el array, vamos a utilizar la función de **filter** sobre este, en la que vamos a comprobar que el personaje incluye el valor del filtro. Esto lo haremos con la función de **includes**, la cual comprueba si un substring se encuentra dentro de un string.

/angular-pipes-puros-e-impuros-lab/src/app/filtro.pipe.ts

```

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

@Pipe({
  name: 'filtro',
  standalone: true
})
export class FiltroPipe implements PipeTransform {

  transform(value: Array<string>, filtro: string): Array<string> {

```

```

    const arrFiltrado = value.filter(str => {
      return str.includes(filtro)
    })

    return null;
}

}

```

Y por último, devolvemos el array que retorna la función de filter (con aquellos personajes que cumplen la condición).

/angular-pipes-puros-e-impuros-lab/src/app/filtro.pipe.ts

```

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

@Pipe({
  name: 'filtro',
  standalone: true
})
export class FiltroPipe implements PipeTransform {

  transform(value: Array<string>, filtro: string): Array<string> {

    const arrFiltrado = value.filter(str => {
      return str.includes(filtro)
    })

    return arrFiltrado;
  }
}

```

Una vez tenemos el pipe de filtro implementado, vamos a aplicarlo en la segunda lista de personajes, y lo aplicaremos sobre el array que se itera dentro del ***ngFor**. Al pipe le vamos a pasar como parámetro el **filtroTxt** para indicarle que personajes estamos buscando mostrar con la lista.

/angular-pipes-puros-e-impuros-lab/src/app/app.component.html

```

<div>
  <label for="nuevo-personaje">Nuevo personaje:</label>
  <input type="text" (change)="addPersonaje($event)">
</div>
<ul>
  @for (let personaje of personajes; track $index) {
    <li>{{personaje}}</li>
  }
</ul>

<hr>

```

```

<div>
  <label for="nuevo-personaje">Filtrar por:</label>
  <input type="text" [(ngModel)]="filtroTxt">
</div>
<ul>
  @for (let personaje of personajes | filtro:filtroTxt; track $index) {
    <li>{{personaje}}</li>
  }
</ul>

```

Si probamos a escribir en el input del filtro **Stark**, veremos que en la lista se muestran tanto **Tony** como **Arya**. Parece que funciona.

Pero vamos a hacer lo siguiente, teniendo en el campo de filtro **Stark**, vamos a añadir ahora un nuevo personaje **Robb Stark** en el input del nuevo personaje.

Si nos fijamos, el personaje se añade a la lista de arriba, donde mostramos todos los personajes, pero no aparece en la lista filtrada, a no ser que volvamos a escribir el filtro de nuevo.

Nuevo personaje: Robb Stark

- **Octavia Blake**
- **Tony Stark**
- **Arya Stark**
- **Charles Falco**
- **Robb Stark**

Filtrar por: Stark

- **Tony Stark**
- **Arya Stark**

Esto se debe a que cuando añadimos el nuevo personaje, estamos modificando el contenido del array, por lo que el pipe de **filtro** no detecta que el array haya cambiado.

Una forma de solucionar este problema, es que convirtamos el pipe en un pipe impuro añadiéndole

la propiedad **pure: false** en el decorador.

/angular-pipes-puros-e-impuros-lab/src/app/filtro.pipe.ts

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

@Pipe({
  name: 'filtro',
  standalone: true,
  pure: false
})
export class FiltroPipe implements PipeTransform {

  transform(value: Array<string>, filtro: string): Array<string> {

    const arrFiltrado = value.filter(str => {
      return str.includes(filtro)
    })

    return arrFiltrado;
  }
}
```

Una vez convertido en pipe impuro, si volvemos a seguir los pasos descritos antes, ahora si deberíamos de ver a **Robb Stark** en las dos listas, en la primera porque se muestran todos los personajes, y en la segunda porque este personaje cumple con el filtro, tiene incluido en el string el substring **Stark**.

Nuevo personaje: Robb Stark

- Octavia Blake
- Tony Stark
- Arya Stark
- Charles Falco
- Robb Stark

Filtrar por: Stark

- Tony Stark
- Arya Stark
- Robb Stark

Pero esta solución no es la mejor de todas, ya que con cualquier cambio de datos en la aplicación, incluso datos que no afecten en nada a esa lista de personajes o al filtroTxt, se estará ejecutando el pipe.

Por tanto, tenemos una segunda solución que es más eficiente.

Vamos a empezar quitando el **pure: false** que hemos puesto.

/angular-pipes-puros-e-impuros-lab/src/app/filtro.pipe.ts

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

@Pipe({
  name: 'filtro',
  standalone: true,
})
export class FiltroPipe implements PipeTransform {

  transform(value: Array<string>, filtro: string): Array<string> {

    const arrFiltrado = value.filter(str => {
      return str.includes(filtro)
    })
  }
}
```

```
    return arrFiltrado;
}

}
```

Y ahora, en la función de **addPersonaje**, en lugar de añadir el nuevo personaje con el método **push** que muta el array, vamos a crear un nuevo array con el contenido que tenía y vamos a añadirle el nuevo personaje al final de este.

/angular-pipes-puros-e-impuros-lab/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  personajes: Array<string> = [
    'Octavia Blake',
    'Tony Stark',
    'Arya Stark',
    'Charles Falco',
  ]

  filtroTxt: string = ''

  addPersonaje(event: any): void {
    const nuevoPersonaje = event.target.value
    this.personajes = [...this.personajes, nuevoPersonaje]
  }
}
```

Y si volvemos a repetir los pasos de antes, el resultado tiene que ser el mismo que cuando usabamos el pipe como impuro.

14.13. Pipe async

El pipe **async** es un pipe impuro que puede recibir una promesa o un observable como entrada, a la que se subscribe automáticamente. Al estar suscrito, mostrará los datos según los va enviando el observable o en el caso de la promesa, cuando esta se resuelve.

/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { AsyncPipe } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, AsyncPipe],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  mensaje = new Promise<string>(resolve => {
    setTimeout(() => {
      resolve('El canario está en la jaula!');
    }, 2500)
  });

}
```

Como se muestra a continuación, si no ponemos el pipe **async** en la propiedad `mensaje`, se nos muestra que es un objeto de tipo `Promise`, ya que en realidad, el `mensaje` no es un `string`, sino una promesa que al resolverse devolverá un `string`.

Al aplicar el pipe **async**, veremos que no se muestra nada hasta que la promesa se resuelve, en este caso a los 2500ms. Y tras pasar este tiempo se mostrará el valor devuelto por la promesa.

/src/app/app.component.html

```
<p>Mensaje: {{ mensaje }}</p><!-- [object Promise] -->
<p>Mensaje: {{ mensaje | async }}</p><!-- El canario está en la jaula! -->
```

Chapter 15. Formularios

Los formularios son una de las partes principales de una aplicación. Los formularios se usan para acciones como el log in, la reserva de un vuelo, establecer una reunión...

Angular 2 permite manejar el uso de formularios de dos formas:

- **Formularios basados en plantilla**
- **Formularios reactivos**

Al ser un framework, también da soporte a las validaciones y al control de errores.

15.1. Formularios de plantilla (FormsModule)

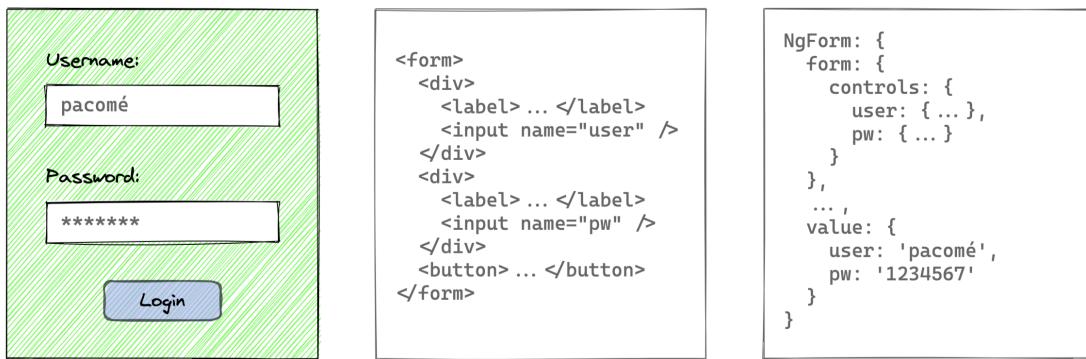
Los formularios de plantilla son un tipo de formularios en los que vamos a poner prácticamente toda su funcionalidad (valores iniciales, validaciones...) en la plantilla del componente, es decir, en el HTML.

Estos formularios van a hacer uso de la directiva **ngModel** (Two Way Data Binding) para recoger los valores iniciales de cada uno de los campos de texto y asignarlos como valores de los **inputs**.



Es necesario importar en el módulo de App el **FormsModule** desde **@angular/forms**, y por supuesto hay que añadirlo dentro del array de **imports**.

Cuando usamos un **ngModel** dentro de una etiqueta **form**, Angular nos pide que añadamos la propiedad **name** sobre los distintos campos para que pueda guardar los valores escritos en los inputs dentro del valor del formulario. En el **value** del formulario se usa el valor de los atributos **name** como clave donde almacenar los valores de los campos.



Dentro de estos formularios usamos variables de plantilla tanto para la etiqueta **form** como para los campos, de tal forma que podamos acceder a todas las propiedades que Angular gestiona para cada uno de estos elementos. A estas variables de plantilla se les asigna **ngForm** y **ngModel** para convertir las referencias a las etiquetas en objetos con las propiedades de estos.

Algunas de las propiedades que tenemos que conocer son:

Table 3. Clases asignadas por Angular a los campos de un formulario

Propiedad	Clase CSS	Significado
valid	ng-valid	El campo es válido
invalid	ng-invalid	El campo es inválido
touched	ng-touched	El campo ha perdido el foco
untouched	ng-untouched	El campo no ha perdido el foco
pristine	ng-pristine	Tiene el valor inicial
dirty	ng-dirty	Se ha modificado el valor inicial

También tenemos acceso a una propiedad **errors**, tanto en el formulario como en cada campo, en la que se van almacenando los errores de aquellas validaciones que no se cumplen.

Por último, también tenemos que tener en cuenta que para utilizar de forma correcta estos formularios deberíamos de utilizar el evento **ngSubmit** de Angular sobre el formulario (etiqueta **form**) para poder pasarlo como parámetro a la función la variable de plantilla del formulario y así sacar de este los valores de los campos.

Veremos el funcionamiento de este tipo de formularios en el siguiente laboratorio.

15.2. Lab: Formularios (FormsModule)

En este laboratorio vamos a crear un formulario de registro usando los formularios basados en plantillas de Angular.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-formularios-de-plantilla-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? no
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, creamos un componente y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-formularios-de-plantilla-lab  
$ ng g c errores-form  
$ ng s
```

Vamos a empezar por crear la estructura del formulario en el archivo de HTML del componente App. Añadiremos de momento cuatro campos:

- Username
- Email
- Contraseña
- Confirmación de contraseña

/angular-formularios-de-plantilla-lab/src/app/app.component.html

```
<form>  
  <div>  
    <label for="username">Usuario:</label>  
    <input type="text" id="username">  
  </div>  
  <div>  
    <label for="email">Email:</label>  
    <input type="email" id="email">  
  </div>  
  <div>  
    <label for="password">Contraseña:</label>  
    <input type="password" id="password">  
  </div>  
  <div>  
    <label for="confirmarPassword">Confirmar contraseña:</label>  
    <input type="password" id="confirmarPassword">  
  </div>
```

```
</form>
```

Lo primero que haremos con el formulario será asignarle unos valores iniciales a estos campos, todos ellos los dejaremos como strings vacíos a excepción de uno para comprobar que se muestra en el navegador y por tanto los está asignando correctamente.

Vamos a generar un objeto dentro del componente con estos valores iniciales.

```
/angular-formularios-de-plantilla-lab/src/app/app.component.ts
```

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  datosInicialesForm = {
    username: 'doflamingo',
    email: '',
    password: '',
    confirmarPassword: ''
  }
}
```

Ahora vamos a necesitar utilizar la directiva **ngModel** para asignar estos valores a cada uno de los campos, por lo que primero tendremos que importar en el componente de App el **FormsModule**.

```
/angular-formularios-de-plantilla-lab/src/app/app.component.ts
```

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, FormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  datosInicialesForm = {
```

```

        username: 'doflamingo',
        email: '',
        password: '',
        confirmPassword: ''
    }

}

```

En nuestro formulario vamos a añadir a cada uno de los campos el valor inicial que se encuentra en el objeto de **datosInicialesForm** utilizando la directiva **ngModel**.

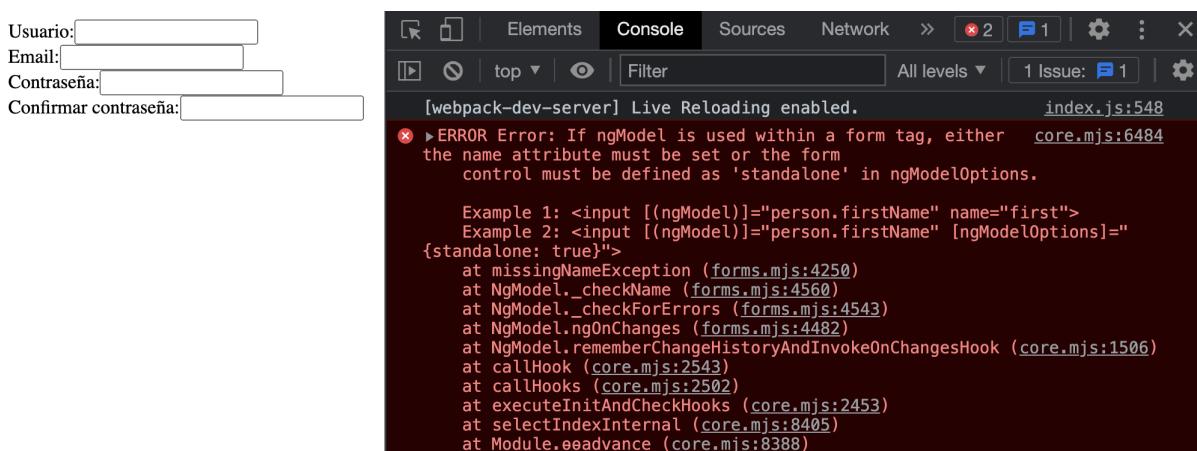
/angular-formularios-de-plantilla-lab/src/app/app.component.html

```

<form>
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" [(ngModel)]="datosInicialesForm.username">
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" [(ngModel)]="datosInicialesForm.email">
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" [(ngModel)]="datosInicialesForm.password">
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" [(ngModel)]="datosInicialesForm.confirmarPassword">
  </div>
</form>

```

Si nos vamos al navegador, <http://localhost:4200/>, podemos comprobar que en el campo de username no se muestra el valor inicial, y si abrimos la consola del navegador veremos un error como el siguiente:



Como se nos indica en el error, cuando usamos la directiva **ngModel** dentro de las etiquetas de HTML **form**, tenemos que añadir un atributo **name** a los campos input. El valor dado a este atributo será la clave en la que se va a guardar el valor del input dentro de un objeto formulario

que veremos ahora después.

Por tanto, para corregir este error, vamos a añadir en cada uno de los campos un atributo **name**.

/angular-formularios-de-plantilla-lab/src/app/app.component.html

```
<form>
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" name="username" [(ngModel)]="datosInicialesForm.username">
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" [(ngModel)]="datosInicialesForm.email">
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" name="password" [(ngModel)]="datosInicialesForm.password">
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" name="confirmarPassword" [(ngModel)]=
"datosInicialesForm.confirmarPassword">
  </div>

</form>
```

Con este cambio debería de haber desaparecido el error anterior, y ya debería de verse el valor inicial del primer campo.

Ahora que ya tenemos los valores iniciales asignados, vamos a ver como podemos obtener el valor del formulario, es decir, un objeto con el valor final de todos los campos. Este objeto tendrá dichos valores asociados a unas claves, donde estas claves son los valores que hemos puesto en los atributos **name** de cada campo.

Para poder sacar este valor, vamos a añadir primero un botón de tipo **submit** en el formulario.

/angular-formularios-de-plantilla-lab/src/app/app.component.html

```
<form>
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" name="username" [(ngModel)]="datosInicialesForm.username">
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" [(ngModel)]="datosInicialesForm.email">
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" name="password" [(ngModel)]="datosInicialesForm.password">
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" name="confirmarPassword" [(ngModel)]=
"datosInicialesForm.confirmarPassword">
  </div>
  <button type="submit">Guardar</button>
</form>
```

Cuando se pulsa un botón de este tipo dentro de un formulario, el formulario emite un evento **submit** que es el que vamos a detectar, pero esta vez no vamos a utilizar el nombre **submit**, sino que vamos a utilizar **ngSubmit**. Es el mismo evento pero este es propio de Angular.

/angular-formularios-de-plantilla-lab/src/app/app.component.html

```
<form (ngSubmit)="guardar()">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" name="username" [(ngModel)]="datosInicialesForm.username">
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" [(ngModel)]="datosInicialesForm.email">
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" name="password" [(ngModel)]="datosInicialesForm.password">
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" name="confirmarPassword" [(ngModel)]=
"datosInicialesForm.confirmarPassword">
  </div>
  <button type="submit">Guardar</button>
</form>
```

Ahora tenemos que crear la función **guardar** dentro del TS. Esta función vamos a hacer que reciba un parámetro que va a contener todos los datos del formulario, y por el momento vamos a mostrarlo por consola.

/angular-formularios-de-plantilla-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, FormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  datosInicialesForm = {
    username: 'doflamingo',
    email: '',
    password: '',
    confirmarPassword: ''
  }

  guardar(form: any) {
    console.log(form)
  }
}
```

```
}
```

```
}
```

Este parámetro **form** que le estamos pasando a la función es una referencia al formulario que tenemos en el HTML, por lo que tenemos que crear un variable de plantilla y pasar la referencia como parámetro de la función.

/angular-formularios-de-plantilla-lab/src/app/app.component.html

```
<form (ngSubmit)="guardar(form)" #form>
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" name="username" [(ngModel)]="datosInicialesForm.username">
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" [(ngModel)]="datosInicialesForm.email">
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" name="password" [(ngModel)]="datosInicialesForm.password">
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" name="confirmarPassword" [(ngModel)]="datosInicialesForm.confirmarPassword">
  </div>
  <button type="submit">Guardar</button>
</form>
```

Con esto ya podemos pulsar sobre el botón del formulario y comprobar que es lo que se muestra por consola.

```
app.component.ts:18
▼<form _ngcontent-wfo-c43 novalidate class="ng-valid ng-dirty ng-touched n
g-submitted">
  ▶<div _ngcontent-wfo-c43>...</div>
  ▶<div _ngcontent-wfo-c43>...</div>
  ▶<div _ngcontent-wfo-c43>...</div>
  ▶<div _ngcontent-wfo-c43>...</div>
    <button _ngcontent-wfo-c43 type="submit">Guardar</button>
</form>
```

Como podemos ver en la imagen anterior, la referencia nos da como valor la etiqueta form y su contenido, pero nosotros no queremos obtener este tipo de datos, sino que queremos que nos llegue un objeto con los datos internos del formulario. Esto lo vamos a conseguir asignándole a la variable de plantilla la directiva **ngForm**, la cual se encarga de generar este objeto a partir de la plantilla del formulario y asignar este valor a dicha variable de plantilla.

/angular-formularios-de-plantilla-lab/src/app/app.component.html

```
<form (ngSubmit)="guardar(form)" #form="ngForm">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" name="username" [(ngModel)]="datosInicialesForm.username">
```

```

</div>
<div>
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" [(ngModel)]="datosInicialesForm.email">
</div>
<div>
  <label for="password">Contraseña:</label>
  <input type="password" id="password" name="password" [(ngModel)]="datosInicialesForm.password">
</div>
<div>
  <label for="confirmarPassword">Confirmar contraseña:</label>
  <input type="password" id="confirmarPassword" name="confirmarPassword" [(ngModel)]=
"datosInicialesForm.confirmarPassword">
</div>
<button type="submit">Guardar</button>
</form>

```

Si volvemos a pulsar sobre el botón, ya podemos ver un objeto con todos los datos que gestiona Angular para los formularios.

```

app.component.ts:18
▼ NgForm {_rawValidators: Array(0), _rawAsyncValidators: Array(0), _onDestroyCallbacks: Arra
y(0), submitted: true, _directives: Array(4), ...} ⓘ
  ► form: FormGroup {_pendingDirty: false, _hasOwnPendingAsyncValidator: false, _pendingTou...
  ► ngSubmit: EventEmitter_ {closed: false, observers: Array(1), isStopped: false, hasError...
    submitted: true
  ► __ngContext__: LComponentView(185) [app-root, TView, 147, LRootView(31), null, null, TN...
  ► _directives: (4) [NgModel, NgModel, NgModel, NgModel]
  ► _onDestroyCallbacks: []
  ► _rawAsyncValidators: []
  ► _rawValidators: []
    asyncValidator: (...)

    control: (...)

    controls: (...)

    dirty: (...)

    disabled: (...)

    enabled: (...)

    errors: (...)

    formDirective: (...)

    invalid: (...)

    path: (...)

    pending: (...)

    pristine: (...)

    status: (...)

    statusChanges: (...)

    touched: (...)

    untouched: (...)

    valid: (...)

    validator: (...)

  ► value: Object
    valueChanges: (...)

  ► [[Prototype]]: ControlContainer

```

De aquí ya podemos sacar el valor del formulario si accedemos a la propiedad **value**.

El siguiente paso va a ser añadir una serie de validaciones en los campos. Les añadiremos a todos ellos el atributo **required** para hacerlos campos obligatorios, y a los dos de las contraseñas les vamos a añadir el atributo **minlength** con un valor a 8 para indicar que estos dos campos tienen que tener como mínimo 8 caracteres.

/angular-formularios-de-plantilla-lab/src/app/app.component.html

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

```

<div>
  <label for="username">Usuario:</label>
  <input type="text" id="username" name="username" [(ngModel)]="datosInicialesForm.username" required>
</div>
<div>
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" [(ngModel)]="datosInicialesForm.email" required>
</div>
<div>
  <label for="password">Contraseña:</label>
  <input type="password" id="password" name="password" [(ngModel)]="datosInicialesForm.password" required minlength="8">
</div>
<div>
  <label for="confirmarPassword">Confirmar contraseña:</label>
  <input type="password" id="confirmarPassword" name="confirmarPassword" [(ngModel)]="datosInicialesForm.confirmarPassword" required minlength="8">
</div>
<button type="submit">Guardar</button>
</form>

```

Estas validaciones que acabamos de poner nos van a permitir comprobar si los campos tienen errores y si el formulario es valido o no.

Angular añade una serie de propiedades y clases sobre las etiquetas del formulario y de cada uno de los campos, y las vamos a utilizar para añadirle unos estilos a los campos, deshabilitar el botón del formulario y mostrar los errores.

Empezaremos por añadirle los estilos, y lo que vamos a hacer es añadirles un borde rojo cuando los campos hayan perdido los valores iniciales y ademas sean inválidos, y será de color verde cuando sean válidos.

Dentro del archivo de CSS, vamos a utilizar las clases de **ng-valid**, **ng-invalid** y **ng-dirty** para añadir los colores a los bordes.

/angular-formularios-de-plantilla-lab/src/app/app.component.css

```

input.ng-invalid.ng-dirty {
  border: 1px solid red;
}

input.ng-valid.ng-dirty {
  border: 1px solid green;
}

```

Ahora al ir escribiendo en los distintos campos, veremos que cambian sus bordes de color según cumplan o no las validaciones que les hemos asignado.

Al igual que añade las clases anteriores, sabemos que también añade unas propiedades **valid**, **invalid**, **dirty**... sobre la variable de plantilla que hemos puesto en el formulario. Estas propiedades las vamos a utilizar para mostrar los errores y deshabilitar el botón.

Vamos a empezar por deshabilitar el botón del formulario cuando este sea inválido, por tanto vamos a asignar el valor del atributo **invalid** de la variable de plantilla que habíamos creado sobre

el formulario a la propiedad **disabled** del botón.

/angular-formularios-de-plantilla-lab/src/app/app.component.html

```
<form (ngSubmit)="guardar(form)" #form="ngForm">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" name="username" [(ngModel)]="datosInicialesForm.username" required>
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" [(ngModel)]="datosInicialesForm.email" required>
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" name="password" [(ngModel)]="datosInicialesForm.password" required minlength="8">
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" name="confirmarPassword" [(ngModel)]="datosInicialesForm.confirmarPassword" required minlength="8">
  </div>
  <button type="submit" [disabled]="form.invalid">Guardar</button>
</form>
```

Con esto debería de aparecer deshabilitado hasta que todos los campos cumplan con las validaciones que les hemos añadido anteriormente.

Y por último, ahora tenemos que mostrar los errores de los campos. Vamos a crear una variable de plantilla por cada uno de ellos, y esta vez les vamos a asignar la directiva **ngModel** para que en lugar de darnos una etiqueta HTML nos de un objeto que representa a un campo de formulario con todas sus propiedades.

/angular-formularios-de-plantilla-lab/src/app/app.component.html

```
<form (ngSubmit)="guardar(form)" #form="ngForm">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" name="username" [(ngModel)]="datosInicialesForm.username" required #campoUsername="ngModel">
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" [(ngModel)]="datosInicialesForm.email" required #campoEmail="ngModel">
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" name="password" [(ngModel)]="datosInicialesForm.password" required minlength="8" #campoPassword="ngModel">
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" name="confirmarPassword" [(ngModel)]="datosInicialesForm.confirmarPassword" required minlength="8" #campoConfirmarPassword="ngModel">
  </div>
  <button type="submit" [disabled]="form.invalid">Guardar</button>
</form>
```

Vamos a utilizar estas variables de plantilla para pasárselas al componente de **errores-form** que

creamos al principio. Vamos a poner este componente una vez por cada campo, y le vamos a pasar a una propiedad **errores** la propiedad **errors** de cada campo.

Recordad que antes de usar ese componente, hay que importarlo en el componente donde se va a utilizar, en este caso en el componente App.

/angular-formularios-de-plantilla-lab/src/app/app.component.html

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { RouterOutlet } from '@angular/router';
import { ErroresFormComponent } from './errores-form/errores-form.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, FormsModule, ErroresFormComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  datosInicialesForm = {
    username: 'doflamingo',
    email: '',
    password: '',
    confirmPassword: ''
  }

  guardar(form: any) {
    console.log(form)
  }

}
```

/angular-formularios-de-plantilla-lab/src/app/app.component.html

```
<form (ngSubmit)="guardar(form)" #form="ngForm">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" name="username" [(ngModel)]="datosInicialesForm.username" required #campoUsername="ngModel">
      <app-errores-form [errores]="campoUsername.errors"></app-errores-form>
    </div>
    <div>
      <label for="email">Email:</label>
      <input type="email" id="email" name="email" [(ngModel)]="datosInicialesForm.email" required #campoEmail="ngModel">
        <app-errores-form [errores]="campoEmail.errors"></app-errores-form>
      </div>
    <div>
      <label for="password">Contraseña:</label>
      <input type="password" id="password" name="password" [(ngModel)]="datosInicialesForm.password" required minlength="8" #campoPassword="ngModel">
        <app-errores-form [errores]="campoPassword.errors"></app-errores-form>
      </div>
```

```

<div>
  <label for="confirmarPassword">Confirmar contraseña:</label>
  <input type="password" id="confirmarPassword" name="confirmarPassword" [(ngModel)]=
    "datosInicialesForm.confirmarPassword" required minlength="8" #campoConfirmarPassword="ngModel">
    <app-errores-form [errores]="campoConfirmarPassword.errors"></app-errores-form>
  </div>
  <button type="submit" [disabled]="form.invalid">Guardar</button>
</form>

```

Ahora vamos al componente de **errores-form**, a la parte del TypeScript donde vamos a recibir el valor de los errores con el decorador **@Input**. También tenemos que añadir una propiedad para guardar la lista de mensajes de error a mostrar por cada campo.

/angular-formularios-de-plantilla-lab/src/app/errores-form/errores-form.component.ts

```

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

@Component({
  selector: 'app-errores-form',
  standalone: true,
  imports: [],
  templateUrl: './errores-form.component.html',
  styleUrls: ['./errores-form.component.css']
})
export class ErroresFormComponent {
  @Input() errores: {[key: string]: any} | null = null
  mensajesErrores: Array<string> = []

  constructor() { }

}

```

Vamos a hacer que esta clase implemente la interface **OnChanges** para añadir el método **ngOnChanges** del ciclo de vida de los componentes, el cual se ejecuta cada vez que hay cambios en las propiedades del componente. En nuestro caso queremos utilizarlo para ir actualizando la lista de errores cada vez que cambiemos el valor del campo y este autocalcule la propiedad **errors**.

/angular-formularios-de-plantilla-lab/src/app/errores-form/errores-form.component.ts

```

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

@Component({
  selector: 'app-errores-form',
  standalone: true,
  imports: [],
  templateUrl: './errores-form.component.html',
  styleUrls: ['./errores-form.component.css']
})
export class ErroresFormComponent implements OnChanges {

  @Input() errores: {[key: string]: any} | null = null
  mensajesErrores: Array<string> = []

```

```
constructor() { }

ngOnChanges(): void {
}

}
```

Dentro del método **ngOnChanges** vamos a inicializar la lista de mensajes de error a un array vacío, ya que si no lo hacemos, los mensajes se irán añadiendo sobre una lista que ya tiene esos mensajes de las anteriores veces que hemos ido escribiendo sobre los campos.

/angular-formularios-de-plantilla-lab/src/app/errores-form/errores-form.component.ts

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

@Component({
  selector: 'app-errores-form',
  standalone: true,
  imports: [],
  templateUrl: './errores-form.component.html',
  styleUrls: ['./errores-form.component.css']
})
export class ErroresFormComponent implements OnChanges {

  @Input() errores: {[key: string]: any} | null = null
  mensajesErrores: Array<string> = []

  constructor() { }

  ngOnChanges(): void {
    this.mensajesErrores = []
  }

}
```

Ahora tenemos que recorrer el objeto de errores e iremos comprobando con un **switch** que mensajes hay que ir añadiendo sobre el array de **mensajesErrores**.

/angular-formularios-de-plantilla-lab/src/app/errores-form/errores-form.component.ts

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

@Component({
  selector: 'app-errores-form',
  standalone: true,
  imports: [],
  templateUrl: './errores-form.component.html',
  styleUrls: ['./errores-form.component.css']
})
```

```

})
export class ErroresFormComponent implements OnChanges {

  @Input() errores: {[key: string]: any} | null = null
  mensajesErrores: Array<string> = []

  constructor() { }

  ngOnChanges(): void {
    this.mensajesErrores = []
    for (let keyErr in this.errores) {
      switch (keyErr) {
        case 'required':
          break;
        case 'minlength':
          break;
      }
    }
  }
}

```

Dentro de cada **case** añadiremos un mensaje a mostrar. Para el mensaje de **minlength** vamos a calcular el número de caracteres que nos faltan para que el campo sea válido. Estos datos podemos sacarlos del valor que lleva asociada la clave **minlength** en el que nos encontramos con **actualLength** (la longitud actual del valor del campo) y **requiredLength** (la longitud que tiene que tener el valor para que el campo sea válido).

/angular-formularios-de-plantilla-lab/src/app/errores-form/errores-form.component.ts

```

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

@Component({
  selector: 'app-errores-form',
  standalone: true,
  imports: [],
  templateUrl: './errores-form.component.html',
  styleUrls: ['./errores-form.component.css']
})
export class ErroresFormComponent implements OnChanges {

  @Input() errores: {[key: string]: any} | null = null
  mensajesErrores: Array<string> = []

  constructor() { }

  ngOnChanges(): void {
    this.mensajesErrores = []
    for (let keyErr in this.errores) {
      switch (keyErr) {
        case 'required':
          this.mensajesErrores.push('El campo es obligatorio')
          break;
        case 'minlength':

```

```

        const msg = `El campo necesita ${this.errores[keyErr].requiredLength} caracteres. Te faltan
        ${this.errores[keyErr].requiredLength - this.errores[keyErr].actualLength}.`
        this.mensajesErrores.push(msg)
        break;
    }
}
}

```

Solo nos queda ir a la plantilla y poner la lista de mensajes.

/angular-formularios-de-plantilla-lab/src/app/errores-form/errores-form.component.html

```

<ul>
  @for (msg of mensajesErrores; track $index) {
    <li>{{ msg }}</li>
  }
</ul>

```

Con esto ya deberían de pintarse los mensajes de error, pero ahora tenemos un problemilla y es que se pintan nada más cargar el formulario. Deberíamos de evitar que se pinten los errores nada más cargarlo, y solo mostrarlos cuando ya estemos rellenándolos o cuando el campo pierda el foco.

Para hacer esto, vamos a utilizar la directiva **ngIf** sobre cada componente de errores para hacer que se pinten o no añadiendo como condición que el campo tiene que ser **invalid** y **dirty**, es decir, que hayamos modificado el valor inicial y no cumpla con las validaciones que le hemos añadido.



Si en lugar de querer mostrar los errores según vamos escribiendo en los campos, queremos mostrarlos cuando estos pierden el foco, entonces tenemos que cambiar **dirty** por **touched**.

/angular-formularios-de-plantilla-lab/src/app/app.component.html

```

<form (ngSubmit)="guardar(form)" #form="ngForm">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" name="username" [(ngModel)]="datosInicialesForm.username" required #campoUsername="ngModel">
    @if (campoUsername.invalid && campoUsername.dirty) {
      <app-errores-form [errores]="campoUsername.errors"></app-errores-form>
    }
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" [(ngModel)]="datosInicialesForm.email" required #campoEmail="ngModel">
    @if (campoEmail.invalid && campoEmail.dirty) {
      <app-errores-form [errores]="campoEmail.errors"></app-errores-form>
    }
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" name="password" [(ngModel)]="datosInicialesForm.password" required minlength="8"
    #campoPassword="ngModel">
    @if (campoPassword.invalid && campoPassword.dirty) {
      <app-errores-form [errores]="campoPassword.errors"></app-errores-form>
    }
  </div>

```

```
</div>
<div>
  <label for="confirmarPassword">Confirmar contraseña:</label>
  <input type="password" id="confirmarPassword" name="confirmarPassword" [(ngModel)]=
"datosInicialesForm.confirmarPassword" required minlength="8" #campoConfirmarPassword="ngModel">
  @if (campoConfirmarPassword.invalid && campoConfirmarPassword.dirty) {
    <app-errores-form [errores]="campoConfirmarPassword.errors"></app-errores-form>
  }
</div>
<button type="submit" [disabled]="form.invalid">Guardar</button>
</form>
```

Y con esto ya tenemos nuestro formulario completo.

15.3. Formularios reactivos (ReactiveFormsModule)

Los formularios reactivos facilitan la gestión de los datos puesto que se controlan directamente desde la clase del componente. De esta forma el componente tiene acceso a los datos y a la estructura del formulario, por lo tanto el componente puede reaccionar a los cambios que observa.

Con este tipo de formularios empezamos creando lo que sería el formulario, creando una instancia de **FormGroup**, y dentro de este se van definiendo los distintos campos, que serían instancias de **FormControl**.

El constructor de **FormGroup** recibe como parámetro un objeto en el que se definen los distintos campos del formulario.

El constructor de **FormControl** recibe como parámetros el valor inicial que queremos darle al campo, y luego las validaciones, donde podemos pasarle una sola, o un array de ellas.

Las validaciones para este tipo de formularios las sacamos de **Validators**, donde nos encontramos con aquellas que tenemos en HTML5, aunque luego también podemos crear nuestras propias validaciones.

Función	Descripción
required	El campo es obligatorio.
min(num)	El valor tiene que ser como mínimo <i>num</i> .
max(num)	El valor tiene que ser como máximo <i>num</i> .
minLength(num)	La longitud del valor tiene que ser como mínimo <i>num</i> .
maxLength(num)	La longitud del valor tiene que ser como máximo <i>num</i> .
pattern(exp)	El valor tiene que cumplir con la expresión regular <i>exp</i> .



Todas estas clases (**FormControl**, **FormGroup**, **Validators**...) se importan desde **@angular/forms**.

Todo lo anterior se usa dentro del componente, en el archivo de TypeScript. Pero también tenemos que tocar la plantilla.

Dentro de la plantilla de HTML tenemos que enlazar los distintos campos con los datos que hemos inicializado en el TypeScript. Para ello se utilizan directivas que vienen del módulo **ReactiveFormsModule** que hay que importar en el módulo de App y que también se importa desde **@angular/forms**.

Las directivas que vamos a utilizar son:

- **formGroup**: le damos como valor la propiedad a la que le hemos asignado la instancia del **FormGroup**.

- **formControlName**: le damos como valor la clave que hace referencia a la instancia FormControl de la que hay que sacar las validaciones y el valor inicial.

Podemos acceder a las propiedades del formulario desde la propia instancia del FormGroup, donde tendremos las propiedades de **valid**, **invalid**, **dirty**, **pristine**, **touched**, **untouched**, **value** y **errors** entre otras.

Y para acceder a estas mismas propiedades pero esta vez las de cada uno de los campos tendremos que acceder desde la instancia del FormGroup a **controls['campo']** y luego a la propiedad que queramos.



Dentro de este tipo de formularios no podemos asignarles a las variables de plantilla el **ngModel** para acceder a las propiedades (valid, touched, dirty, errors...) de cada uno de los campos.

Todo esto lo vamos a ver en práctica en el siguiente laboratorio.

15.4. Lab: Formularios reactivos (ReactiveFormsModule)

En este laboratorio vamos a crear un formulario de registro usando los formularios reactivos de Angular.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-formularios-reactivos-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? no
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, creamos un componente y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-formularios-reactivos-lab  
$ ng g c errores-form  
$ ng s
```

Vamos a hacer el mismo formulario que hicimos en el **Lab: Formularios (FormsModule)**, por lo que vamos a poner el mismo código inicial de HTML en la plantilla del componente App, incluyendo ya el botón de tipo **submit**.

/angular-formularios-reactivos-lab/src/app/app.component.html

```
<form>  
  <div>  
    <label for="username">Usuario:</label>  
    <input type="text" id="username">  
  </div>  
  <div>  
    <label for="email">Email:</label>  
    <input type="email" id="email">  
  </div>  
  <div>  
    <label for="password">Contraseña:</label>  
    <input type="password" id="password">  
  </div>  
  <div>  
    <label for="confirmarPassword">Confirmar contraseña:</label>  
    <input type="password" id="confirmarPassword">  
  </div>  
  <button type="submit">Guardar</button>  
</form>
```

El archivo de TypeScript es donde vamos a notar grandes cambios si lo comparamos con el otro tipo

de formularios. Dentro del componente, en el constructor vamos a empezar por crear una instancia de **FormGroup**, es decir, de un formulario o grupo de campos.

/angular-formularios-reactivos-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { FormGroup } from '@angular/forms';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  formulario: FormGroup;

  constructor() {
    this.formulario = new FormGroup({
      })
  }
}
```

Dentro del objeto que recibe el **FormGroup** como parámetro, tenemos que pasarle como claves un nombre identificativo para cada campo y como valores una instancia de **FormControl** por cada uno de los campos.

Al tener 4 campos, tendremos 4 claves con sus respectivas instancias de **FormControl**.

/angular-formularios-reactivos-lab/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  formulario: FormGroup;

  constructor() {
```

```

    this.formulario = new FormGroup({
      username: new FormControl(),
      email: new FormControl(),
      password: new FormControl(),
      confirmPassword: new FormControl(),
    })
  }

}

```

Cada **FormControl** recibe como parámetros, un valor inicial, y luego las validaciones que queremos aplicar sobre el campo. De momento vamos a rellenar con valores iniciales cada uno de los campos, y en el primero vamos a darle un valor para ver si más adelante se muestra en la vista.

/angular-formularios-reactivos-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  formulario: FormGroup;

  constructor() {
    this.formulario = new FormGroup({
      username: new FormControl('bartolomeo'),
      email: new FormControl(''),
      password: new FormControl(''),
      confirmPassword: new FormControl('')
    })
  }
}

```

Ya tenemos una primera configuración del **FormGroup**, nos toca enlazar todos estos datos con la plantilla. Para ello necesitaremos las directivas **formGroup** y **formControlName**, directivas que no vienen en el módulo raíz de Angular, por lo que tendremos que importar en este el módulo que las contiene.

Dentro del componente de la aplicación vamos a importar el **ReactiveFormsModule** que viene de **@angular/forms**.

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, ReactiveFormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  formulario: FormGroup;

  constructor() {
    this.formulario = new FormGroup({
      username: new FormControl('bartolomeo'),
      email: new FormControl(''),
      password: new FormControl(''),
      confirmPassword: new FormControl('')
    })
  }
}
```

Ahora ya podemos añadir al formulario una directiva **formGroup** a la que le vamos a asignar la instancia del FormGroup que hemos creado en el componente, de tal forma que se puedan enlazar después los valores de cada campo con los distintos **input**.

```
<form [formGroup]="formulario">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username">
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email">
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password">
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword">
  </div>
```

```
<button type="submit">Guardar</button>
</form>
```

El siguiente paso es enlazar los campos con las claves de la instancia del formulario. Para hacer esto tenemos que añadir la directiva **FormControlName** sobre cada input y le asignaremos como valor la clave que hemos puesto anteriormente en la instancia del FormGroup.

/angular-formularios-reactivos-lab/src/app/app.component.html

```
<form [FormGroup]="formulario">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" FormControlName="username">
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" FormControlName="email">
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" FormControlName="password">
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" FormControlName="confirmarPassword">
  </div>
  <button type="submit">Guardar</button>
</form>
```

Una vez hecho esto ya deberíamos de ver en el formulario los valores iniciales.

Ahora toca añadir las validaciones. Estas se añaden dentro de las instancias de los **FormControl** como segundo parámetro, y vienen de **Validators**, una clase donde se encuentran todas aquellas validaciones que tenemos en HTML5.

Como segundo parámetro le podemos pasar una sola validación, o un array de validaciones en el caso de que queramos aplicar más de una.

/angular-formularios-reactivos-lab/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, ReactiveFormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

```

export class AppComponent {
  formulario: FormGroup;

  constructor() {
    this.formulario = new FormGroup({
      username: new FormControl('bartolomeo', Validators.required),
      email: new FormControl('', Validators.required),
      password: new FormControl('', [Validators.required, Validators.minLength(8)]),
      confirmPassword: new FormControl('', [Validators.required, Validators.minLength(8)])
    })
  }
}

```

Ahora que tenemos las validaciones podemos añadirle los estilos para poner los bordes de los campos en rojo o verde según sean válidos o inválidos. Las clases que aplica Angular en este caso siguen siendo las mismas que con los otros formularios de Angular que ya vimos en el laboratorio anterior.

/angular-formularios-reactivos-lab/src/app/app.component.css

```

input.ng-invalid.ng-dirty {
  border: 1px solid red;
}

input.ng-valid.ng-dirty {
  border: 1px solid green;
}

```

Antes de añadir la parte de los errores, vamos a poner el evento del **ngSubmit** para llamar a la función de **guardar** que crearemos después dentro del componente.

Esta vez no hay que pasarle ninguna variable de plantilla como parámetro, ya que el formulario lo hemos declarado dentro del TypeScript, y en dicha propiedad tenemos toda la información de este.

/angular-formularios-reactivos-lab/src/app/app.component.html

```

<form [formGroup]="formulario" (ngSubmit)="guardar()">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" formControlName="username">
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" formControlName="email">
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" formControlName="password">
  </div>
</div>

```

```

<label for="confirmarPassword">Confirmar contraseña:</label>
<input type="password" id="confirmarPassword" formControlName="confirmarPassword">
</div>
<button type="submit">Guardar</button>
</form>

```

En el TypeScript vamos a añadir la función de guardar, y dentro de ella ya podemos acceder al **value** del formulario para sacar de ahí los valores.

/angular-formularios-reactivos-lab/src/app/app.component.ts

```

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, ReactiveFormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  formulario: FormGroup;

  constructor() {
    this.formulario = new FormGroup({
      username: new FormControl('bartolomeo', Validators.required),
      email: new FormControl('', Validators.required),
      password: new FormControl('', [Validators.required, Validators.minLength(8)]),
      confirmarPassword: new FormControl('', [Validators.required, Validators.minLength(8)])
    })
  }

  guardar() {
    console.log(this.formulario.value);
  }
}

```

Pues solo nos queda añadir la parte de los errores.

Al principio ya creamos el componente para mostrar los errores, y este componente va a ser igual que el que hicimos en el laboratorio anterior (**Lab: Formularios (FormsModule)**), por lo que vamos a copiar y pegar el código del TypeScript y del HTML en el de este proyecto.

/angular-formularios-reactivos-lab/src/app/errores-form/errores-form.component.ts

```

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

@Component({
  selector: 'app-errores-form',
  standalone: true,

```

```

imports: [],
templateUrl: './errores-form.component.html',
styleUrl: './errores-form.component.css'
})
export class ErroresFormComponent implements OnChanges {

@Input() errores: {[key: string]: any} | null = null
mensajesErrores: Array<string> = []

constructor() { }

ngOnChanges(): void {
  this.mensajesErrores = []
  for (let keyErr in this.errores) {
    switch (keyErr) {
      case 'required':
        this.mensajesErrores.push('El campo es obligatorio')
        break;
      case 'minlength':
        const msg = `El campo necesita ${this.errores[keyErr].requiredLength} caracteres. Te faltan
${this.errores[keyErr].requiredLength - this.errores[keyErr].actualLength}.`
        this.mensajesErrores.push(msg)
        break;
    }
  }
}
}

```

/angular-formularios-reactivos-lab/src/app/errores-form/errores-form.component.html

```

<ul>
  @for (msg of mensajesErrores; track $index) {
    <li>{{ msg }}</li>
  }
</ul>

```

Tenemos que importar este componente en el de la aplicación para poder utilizarlo.

/angular-formularios-reactivos-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule, Validators } from '@angular/forms';
import { RouterOutlet } from '@angular/router';
import { ErroresFormComponent } from './errores-form/errores-form.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, ReactiveFormsModule, ErroresFormComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  formulario: FormGroup;

  constructor() {
    this.formulario = new FormGroup({

```

```

        username: new FormControl('bartolomeo', Validators.required),
        email: new FormControl('', Validators.required),
        password: new FormControl('', [Validators.required, Validators.minLength(8)]),
        confirmPassword: new FormControl('', [Validators.required, Validators.minLength(8)]),
    })
}

guardar() {
    console.log(this.formulario.value);
}

}

```

Lo que cambia con respecto al anterior laboratorio es la forma de pasarle los errores y acceder a las propiedades del formulario (dirty, touched, valid, invalid, ...), ya que dentro de los formularios reactivos no podemos utilizar variables de plantilla y asignarles la directiva **ngModel**.

En este tipo de formularios podemos acceder a estas propiedades a través de la propiedad **formulario** que hemos creado. Dentro de este accederíamos a **controls** (el objeto con las claves asociadas a cada campo), después le pasaríamos la clave del campo y por último la propiedad.

Entonces, para pasar primero los errores en cada uno de los campos, tendríamos que hacer lo siguiente:

/angular-formularios-reactivos-lab/src/app/app.component.html

```

<form [formGroup]="formulario" (ngSubmit)="guardar()">
    <div>
        <label for="username">Usuario:</label>
        <input type="text" id="username" formControlName="username">
        <app-errores-form [errores]="formulario.controls['username'].errors"></app-errores-form>
    </div>
    <div>
        <label for="email">Email:</label>
        <input type="email" id="email" formControlName="email">
        <app-errores-form [errores]="formulario.controls['email'].errors"></app-errores-form>
    </div>
    <div>
        <label for="password">Contraseña:</label>
        <input type="password" id="password" formControlName="password">
        <app-errores-form [errores]="formulario.controls['password'].errors"></app-errores-form>
    </div>
    <div>
        <label for="confirmarPassword">Confirmar contraseña:</label>
        <input type="password" id="confirmarPassword" formControlName="confirmarPassword">
        <app-errores-form [errores]="formulario.controls['confirmarPassword'].errors"></app-errores-form>
    </div>
    <button type="submit">Guardar</button>
</form>

```

Y si ahora queremos hacer que solo se muestren los errores de los campos cuando ya se haya modificado el valor inicial y además el campo sea inválido, tenemos que añadir la directiva **ngIf** accediendo a las propiedades de **invalid** y **dirty**.

/angular-formularios-reactivos-lab/src/app/app.component.html

```
<form [FormGroup]="formulario" (ngSubmit)="guardar()">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" formControlName="username">
    @if (formulario.controls['username'].invalid && formulario.controls['username'].dirty) {
      <app-errores-form [errores]="formulario.controls['username'].errors"></app-errores-form>
    }
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" formControlName="email">
    @if (formulario.controls['email'].invalid && formulario.controls['email'].dirty) {
      <app-errores-form [errores]="formulario.controls['email'].errors"></app-errores-form>
    }
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" formControlName="password">
    @if (formulario.controls['password'].invalid && formulario.controls['password'].dirty) {
      <app-errores-form [errores]="formulario.controls['password'].errors"></app-errores-form>
    }
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" formControlName="confirmarPassword">
    @if (formulario.controls['confirmarPassword'].invalid && formulario.controls['confirmarPassword'].dirty) {
      <app-errores-form [errores]="formulario.controls['confirmarPassword'].errors"></app-errores-form>
    }
  </div>
  <button type="submit">Guardar</button>
</form>
```

Como podemos ver, nuestro código HTML queda demasiado sucio con código de TypeScript, por lo que vamos a mejorarlo un poco.

Para la condición del **if** vamos a crear una función **pintarErrores** a la que le vamos a pasar el nombre del campo y vamos a hacer que devuelva un booleano para indicar si los tiene que pintar o no.

Dentro de la función, vamos a pedir el campo o control con la función **get** de la propiedad **formulario**, y desde este campo ya podemos acceder a las propiedades **invalid** y **dirty**.

/angular-formularios-reactivos-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule, Validators } from '@angular/forms';
import { RouterOutlet } from '@angular/router';
import { ErroresFormComponent } from './errores-form/errores-form.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, ReactiveFormsModule, ErroresFormComponent],
  templateUrl: './app.component.html',
```

```

    styleUrls: './app.component.css'
})
export class AppComponent {
  formulario: FormGroup;

  constructor() {
    this.formulario = new FormGroup({
      username: new FormControl('bartolomeo', Validators.required),
      email: new FormControl('', Validators.required),
      password: new FormControl('', [Validators.required, Validators.minLength(8)]),
      confirmPassword: new FormControl('', [Validators.required, Validators.minLength(8)])
    })
  }

  guardar() {
    console.log(this.formulario.value);
  }

  pintarErrores(controlName: string) {
    const control = this.formulario.get(controlName)!;
    return control.invalid && control.dirty
  }
}

```

Ahora en el HTML vamos a llamar desde cada **ngIf** a esta función pasándole el nombre del campo.

/angular-formularios-reactivos-lab/src/app/app.component.html

```

<form [formGroup]="formulario" (ngSubmit)="guardar()">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" formControlName="username">
    @if (pintarErrores('username')) {
      <app-errores-form [errores]="formulario.controls['username'].errors"></app-errores-form>
    }
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" formControlName="email">
    @if (pintarErrores('email')) {
      <app-errores-form [errores]="formulario.controls['email'].errors"></app-errores-form>
    }
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" formControlName="password">
    @if (pintarErrores('password')) {
      <app-errores-form [errores]="formulario.controls['password'].errors"></app-errores-form>
    }
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" formControlName="confirmarPassword">
    @if (pintarErrores('confirmarPassword')) {

```

```

        <app-errores-form [errores]="formulario.controls['confirmarPassword'].errors"></app-errores-form>
    }
</div>
<button type="submit">Guardar</button>
</form>

```

Ahora nos toca quitar las instrucciones para acceder a los errores, y en este caso, vamos a crear una función igual que la anterior, pero esta vez para obtener los errores.

/angular-formularios-reactivos-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule, Validators } from '@angular/forms';
import { RouterOutlet } from '@angular/router';
import { ErroresFormComponent } from './errores-form/errores-form.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, ReactiveFormsModule, ErroresFormComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  formulario: FormGroup;

  constructor() {
    this.formulario = new FormGroup({
      username: new FormControl('bartolomeo', Validators.required),
      email: new FormControl('', Validators.required),
      password: new FormControl('', [Validators.required, Validators.minLength(8)]),
      confirmarPassword: new FormControl('', [Validators.required, Validators.minLength(8)])
    })
  }

  guardar() {
    console.log(this.formulario.value);
  }

  pintarErrores(controlName: string) {
    const control = this.formulario.get(controlName)!
    return control.invalid && control.dirty
  }

  getErrores(controlName: string) {
    const control = this.formulario.get(controlName)!
    return control.errors
  }
}

```

Y desde el HTML llamamos por cada uno de los campos a esta nueva función, pasándole el nombre

del campo.

/angular-formularios-reactivos-lab/src/app/app.component.html

```
<form [formGroup]="formulario" (ngSubmit)="guardar()">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" formControlName="username">
    @if (pintarErrores('username')) {
      <app-errores-form [errores]="getErrores('username')"></app-errores-form>
    }
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" formControlName="email">
    @if (pintarErrores('email')) {
      <app-errores-form [errores]="getErrores('email')"></app-errores-form>
    }
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" formControlName="password">
    @if (pintarErrores('password')) {
      <app-errores-form [errores]="getErrores('password')"></app-errores-form>
    }
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" formControlName="confirmarPassword">
    @if (pintarErrores('confirmarPassword')) {
      <app-errores-form [errores]="getErrores('confirmarPassword')"></app-errores-form>
    }
  </div>
  <button type="submit">Guardar</button>
</form>
```

Y con esto ya podemos mostrar los errores de cada campo.

Por último, vamos a deshabilitar el botón cuando el formulario es inválido.

/angular-formularios-reactivos-lab/src/app/app.component.html

```
<form [formGroup]="formulario" (ngSubmit)="guardar()">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" formControlName="username">
    @if (pintarErrores('username')) {
      <app-errores-form [errores]="getErrores('username')"></app-errores-form>
    }
  </div>
  <div>
```

```

<label for="email">Email:</label>
<input type="email" id="email" formControlName="email">
@if (pintarErrores('email')) {
    <app-errores-form [errores]="getErrores('email')"></app-errores-form>
}
</div>
<div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" formControlName="password">
    @if (pintarErrores('password')) {
        <app-errores-form [errores]="getErrores('password')"></app-errores-form>
    }
</div>
<div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" formControlName="confirmarPassword">
    @if (pintarErrores('confirmarPassword')) {
        <app-errores-form [errores]="getErrores('confirmarPassword')"></app-errores-form>
    }
</div>
<button type="submit" [disabled]="formulario.invalid">Guardar</button>
</form>

```

Ahora ya tenemos el formulario completo.



En el siguiente laboratorio (**Lab: Crear validaciones**) usaremos este mismo proyecto, en el que vamos a crear nuestras propias validaciones para usarlas en los formularios reactivos.

15.5. Crear una validación personalizada

A veces necesitamos nuestros **propios validadores** para validar un campo de un formulario, porque necesitamos nuestra propia lógica, los que hay no nos sirven...

Estos validadores son muy fáciles de crear, solo son funciones que reciben como parámetro el campo (del tipo **AbstractControl**) que se está validando y tienen que retornar un **null** en caso de que la validación sea correcta, o un **objeto** (del tipo **ValidationErrors**) en el caso de que el campo no cumpla con la validación.

A la hora de usar la validación sobre alguno de los campos, solo tenemos que pasarle la referencia de la función, es decir, sin llegar a ejecutarla.

/src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  formulario: FormGroup;

  constructor() {
    this.formulario = new FormGroup({
      nombre: new FormControl('', [Validators.required, this.empiezaPorMayuscula]),
    })
  }

  empiezaPorMayuscula(control: AbstractControl): ValidationErrors | null {
    const nombre = control.value

    if (nombre[0] === nombre[0].toUpperCase()) {
      return null
    }

    return { empiezaPorMayuscula: true }
  }
}
```

En el caso de que la validación necesite recibir algún dato como parámetro al igual que ocurre por ejemplo con **minLength**, entonces nuestra función tendrá que retornar la función validadora (tipo **ValidatorFn**), para que al ejecutar esta función Angular reciba la función validadora en el array de validaciones.

15.6. Validación de campos cruzados

A veces necesitamos validar un campo cuya validación depende de otro campo del mismo formulario, como por ejemplo:

- Comprobar que los valores de los campos contraseña y confirmación de contraseña son iguales.
- Comprobar que el teléfono introducido lleva el prefijo correspondiente al país seleccionado.
- ...

En estos casos, las validaciones se crean de la misma forma que hemos visto, solo que de alguna forma necesitaremos acceder al valor de estos otros campos de los que dependen las validaciones.

Las instancias de **AbstractControl** que recibimos como parámetro en las funciones validadoras tienen una propiedad **parent** que nos da acceso al elemento superior a los campos, que es el formulario.

Una vez tenemos el formulario ya podemos acceder a los campos del formulario con el método **get()** que recibe como parámetro el nombre del campo. Con el campo ya podemos acceder al **value** para usarlo en nuestra validación.

Veremos como crear una validación de este tipo en el siguiente laboratorio.

15.7. Lab: Crear validaciones

En este laboratorio vamos a ver como crear las siguientes validaciones para los campos de un formulario:

- Una validación que compruebe que la contraseña no sea una de las que se encuentran en una lista negra.
- Una validación cruzada que compruebe que los campos de contraseña y confirmar contraseña contienen el mismo valor.

Para este laboratorio vamos a utilizar el mismo formulario del laboratorio anterior (**Lab: Formularios reactivos (ReactiveFormsModule)**), por lo que vamos a copiar el anterior proyecto, y lo vamos a renombrar a **angular-formularios-crear-validacion-lab**.

Una vez hecho lo anterior, entramos en la carpeta del proyecto y levantaremos el servidor de desarrollo:

```
$ cd angular-formularios-crear-validacion-lab  
$ ng s
```

Una vez tenemos el proyecto preparado y arrancado, vamos a crearnos a mano un archivo **customValidators.ts** dentro de la carpeta **src/app** en el cual vamos a añadir nuestras validaciones.

Como ya se ha comentado anteriormente, las validaciones son funciones del tipo ValidatorFn, por lo que vamos a empezar creando una función **passwordSegura** que cumpla con esa firma.

/angular-formularios-crear-validacion-lab/src/app/customValidators.ts

```
import { AbstractControl, ValidationErrors } from "@angular/forms"

const passwordSegura = (control: AbstractControl): ValidationErrors | null => {

}
```

Del parámetro control, que es el campo que se está validando, vamos a obtener el valor de este accediendo a su propiedad value.

/angular-formularios-crear-validacion-lab/src/app/customValidators.ts

```
import { AbstractControl, ValidationErrors } from "@angular/forms"

const passwordSegura = (control: AbstractControl): ValidationErrors | null => {
  const password = control.value

}
```

Ahora vamos a poner un array con las contraseñas más utilizadas y menos seguras que existen.

/angular-formularios-crear-validacion-lab/src/app/customValidators.ts

```
import { AbstractControl, ValidationErrors } from "@angular/forms"

const passwordSegura = (control: AbstractControl): ValidationErrors | null => {
  const password = control.value
  const passwordsInseguras: Array<string> = ['password', '12345678', '12341234', '1234567890', '0987654321', '987654321',
  'password123', 'qwertyuiop', 'qwerty123']

}
```

Una vez tenemos estos datos, solo tenemos que comprobar si la contraseña introducida se encuentra dentro del array de contraseñas inseguras. Si este es el caso entonces vamos a devolver un objeto del tipo **ValidationErrors**, es decir, un objeto que contiene una propiedad a la que le pondremos de nombre el mismo que el de la función, **passwordSegura**, y como valor le vamos a dar un **true**.

/angular-formularios-crear-validacion-lab/src/app/customValidators.ts

```
import { AbstractControl, ValidationErrors } from "@angular/forms"

const passwordSegura = (control: AbstractControl): ValidationErrors | null => {
  const password = control.value
  const passwordsInseguras: Array<string> = ['password', '12345678', '12341234', '1234567890', '0987654321', '987654321',
  'password123', 'qwertyuiop', 'qwerty123']

  if (passwordsInseguras.includes(password.toLowerCase())) {
    return { passwordSegura: true }
  }

}
```

Por último, si no se pasa por el if, es decir, que la contraseña es una contraseña segura, entonces devolveremos un **null** para indicar que la validación es correcta.

/angular-formularios-crear-validacion-lab/src/app/customValidators.ts

```
import { AbstractControl, ValidationErrors } from "@angular/forms"

const passwordSegura = (control: AbstractControl): ValidationErrors | null => {
  const password = control.value
  const passwordsInseguras: Array<string> = ['password', '12345678', '12341234', '1234567890', '0987654321', '987654321',
  'password123', 'qwertyuiop', 'qwerty123']

  if (passwordsInseguras.includes(password.toLowerCase())) {
    return { passwordSegura: true }
  }

  return null
}
```

Para poder usarla en nuestro formulario, tenemos que exportar esta función.

/angular-formularios-crear-validacion-lab/src/app/customValidators.ts

```
import { AbstractControl, ValidationErrors } from "@angular/forms"
```

```

const passwordSegura = (control: AbstractControl): ValidationErrors | null => {
  const password = control.value
  const passwordsInseguras: Array<string> = ['password', '12345678', '12341234', '1234567890', '0987654321', '987654321',
  'password123', 'qwertyuiop', 'qwerty123']

  if (passwordsInseguras.includes(password.toLowerCase())) {
    return { passwordSegura: true }
  }

  return null
}

export const CustomValidators = {
  passwordSegura,
}

```

Vamos a utilizar esta validación dentro del campo de **password** del formulario reactivo, por lo que tendremos que importar la validación en dicho archivo y pasarle la referencia a esta validación dentro del array de validaciones que ya tenemos.

/angular-formularios-crear-validacion-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators, ReactiveFormsModule } from '@angular/forms';
import { RouterOutlet } from '@angular/router';
import { CustomValidators } from './customValidators';
import { ErroresFormComponent } from './errores-form/errores-form.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, ReactiveFormsModule, ErroresFormComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  formulario: FormGroup;

  constructor() {
    this.formulario = new FormGroup({
      username: new FormControl('bartolomeo', Validators.required),
      email: new FormControl('', Validators.required),
      password: new FormControl('', [Validators.required, Validators.minLength(8), CustomValidators.passwordSegura]),
      confirmPassword: new FormControl('', [Validators.required, Validators.minLength(8)])
    })
  }

  guardar() {
    console.log(this.formulario.value);
  }

  pintarErrores(controlName: string) {
    const control = this.formulario.get(controlName)!
    return control.invalid && control.dirty
  }

  getErrores(controlName: string) {
    const control = this.formulario.get(controlName)!
    return control.errors
  }
}

```

Nuestro componente de errores no está controlando este nuevo error de **passwordSegura**, por lo que tenemos que ir al TypeScript y añadir un nuevo **case** del **switch** para añadir un nuevo mensaje de error para cuando la contraseña que introduzca el usuario sea una contraseña insegura.

/angular-formularios-crear-validacion-lab/src/app/errores-form/errores-form.component.ts

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

@Component({
  selector: 'app-errores-form',
  standalone: true,
  imports: [],
  templateUrl: './errores-form.component.html',
  styleUrls: ['./errores-form.component.css']
})
export class ErroresFormComponent implements OnChanges {

  @Input() errores: {[key: string]: any} | null = null
  mensajesErrores: Array<string> = []

  constructor() { }

  ngOnChanges(): void {
    this.mensajesErrores = []
    for (let keyErr in this.errores) {
      switch (keyErr) {
        case 'required':
          this.mensajesErrores.push('El campo es obligatorio')
          break;
        case 'minlength':
          const msg = `El campo necesita ${this.errores[keyErr].requiredLength} caracteres. Te faltan ${this.errores[keyErr].requiredLength - this.errores[keyErr].actualLength}.`
          this.mensajesErrores.push(msg)
          break;
        case 'passwordSegura':
          this.mensajesErrores.push('La contraseña introducida es insegura')
          break;
      }
    }
  }
}
```

Con esto ya deberíamos de ver el nuevo error cuando introducimos una contraseña que se encuentra en el array de contraseñas inseguras que hemos puesto en la validación.

Vamos a ir un paso más allá, y vamos a modificar esta validación para que el array de contraseñas no tenga que estar definido dentro de la validación y se le pueda pasar como parámetro, al igual que ocurre con otras validaciones como la **minLength**.

Para realizar este cambio, lo que tenemos que tener en cuenta, es que ahora nuestra función validadora va a tener que recibir un parámetro que no es el campo a validar.

Pero a su vez, Angular espera recibir una función del tipo **ValidatorFn**, por lo que vamos a hacer que nuestra nueva función devuelva otra función del tipo **ValidatorFn** en la que vamos a meter la lógica de la validación.

/angular-formularios-crear-validacion-lab/src/app/customValidators.ts

```
import { AbstractControl, ValidationErrors, ValidatorFn } from "@angular/forms"

const passwordSegura = (passwordsInseguras: Array<string>): ValidatorFn => {
  return (control: AbstractControl): ValidationErrors | null => {
    // Aquí va la lógica que teníamos antes
  }
}

export const CustomValidators = {
  passwordSegura,
}
```

Al final nuestra función completa queda como podemos ver a continuación:

/angular-formularios-crear-validacion-lab/src/app/customValidators.ts

```
import { AbstractControl, ValidationErrors, ValidatorFn } from "@angular/forms"

const passwordSegura = (passwordsInseguras: Array<string>): ValidatorFn => {
  return (control: AbstractControl): ValidationErrors | null => {
    const password = control.value

    if (passwordsInseguras.includes(password.toLowerCase())) {
      return { passwordSegura: true }
    }

    return null
  }
}

export const CustomValidators = {
  passwordSegura,
}
```

Después de cambiar nuestra validación, también tenemos que cambiar el lugar donde la estábamos usando, ya que ahora le tenemos que pasar como parámetro el array de las contraseñas inseguras.

/angular-formularios-crear-validacion-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators, ReactiveFormsModule } from '@angular/forms';
import { RouterOutlet } from '@angular/router';
import { CustomValidators } from './customValidators';
import { ErroresFormComponent } from './errores-form/errores-form.component';

@Component({
  selector: 'app-root',
  standalone: true,
```

```

imports: [RouterOutlet, ReactiveFormsModule, ErroresFormComponent],
templateUrl: './app.component.html',
styleUrl: './app.component.css'
})
export class AppComponent {
  formulario: FormGroup;

  constructor() {
    this.formulario = new FormGroup({
      username: new FormControl('bartolomeo', Validators.required),
      email: new FormControl('', Validators.required),
      password: new FormControl('', [Validators.required, Validators.minLength(8)],
        CustomValidators.passwordSegura(['password', '12345678', '12341234', '1234567890', '0987654321', '987654321',
          'password123', 'qwertyuiop', 'qwerty123'])),
      confirmPassword: new FormControl('', [Validators.required, Validators.minLength(8)])
    })
  }

  guardar() {
    console.log(this.formulario.value);
  }

  pintarErrores(controlName: string) {
    const control = this.formulario.get(controlName)!
    return control.invalid && control.dirty
  }

  getErrores(controlName: string) {
    const control = this.formulario.get(controlName)!
    return control.errors
  }
}

```

Ya hemos terminado con esta primera validación, vamos a por la segunda.

Para ello vamos a crear otra función de validación que vamos a llamar **repetirPassword**.

Dentro de ella vamos a empezar por obtener el valor de **confirmarPassword** que es el campo sobre el que la vamos a utilizar después.

/angular-formularios-crear-validacion-lab/src/app/customValidators.ts

```

import { AbstractControl, ValidationErrors, ValidatorFn } from "@angular/forms"

const passwordSegura = (passwordsInseguras: Array<string>): ValidatorFn => {
  return (control: AbstractControl): ValidationErrors | null => {
    const password = control.value

    if (passwordsInseguras.includes(password.toLowerCase()))) {
      return { passwordSegura: true }
    }

    return null
  }
}

const repetirPassword = (control: AbstractControl): ValidationErrors | null => {
  const confirmPassword = control.value

```

```

}

export const CustomValidators = {
  passwordSegura,
  repetirPassword,
}

```

Ahora tenemos que obtener el valor del campo **password**. Para ello, podemos utilizar la propiedad **parent** del control para acceder a lo que sería el formulario, y así poder usar el método **get('password')** para pedir el otro **control** del cual queremos sacar el valor.

/angular-formularios-crear-validacion-lab/src/app/customValidators.ts

```

import { AbstractControl, ValidationErrors, ValidatorFn } from "@angular/forms"

const passwordSegura = (passwordsInseguras: Array<string>): ValidatorFn => {
  return (control: AbstractControl): ValidationErrors | null => {
    const password = control.value

    if (passwordsInseguras.includes(password.toLowerCase())) {
      return { passwordSegura: true }
    }

    return null
  }
}

const repetirPassword = (control: AbstractControl): ValidationErrors | null => {
  const confirmPassword = control.value
  const password = control.parent?.get('password')?.value

}

export const CustomValidators = {
  passwordSegura,
  repetirPassword,
}

```

Solo nos queda comparar estas dos contraseñas para devolver un **null** si son iguales, o un objeto con el error si son distintas.

/angular-formularios-crear-validacion-lab/src/app/customValidators.ts

```

import { AbstractControl, ValidationErrors, ValidatorFn } from "@angular/forms"

const passwordSegura = (passwordsInseguras: Array<string>): ValidatorFn => {
  return (control: AbstractControl): ValidationErrors | null => {
    const password = control.value

```

```

        if (passwordsInseguras.includes(password.toLowerCase())) {
            return { passwordSegura: true }
        }

        return null
    }
}

const repetirPassword = (control: AbstractControl): ValidationErrors | null => {
    const confirmPassword = control.value
    const password = control.parent?.get('password')?.value

    if (password === confirmPassword) {
        return null
    }
    return { repetirPassword: true }
}

export const CustomValidators = {
    passwordSegura,
    repetirPassword,
}

```

Una vez tenemos la validación, vamos a utilizarla en el campo **confirmarPassword** del formulario.

/angular-formularios-crear-validacion-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators, ReactiveFormsModule } from '@angular/forms';
import { RouterOutlet } from '@angular/router';
import { CustomValidators } from './custom-validators';
import { ErroresFormComponent } from './errores-form/errores-form.component';

@Component({
    selector: 'app-root',
    standalone: true,
    imports: [RouterOutlet, ReactiveFormsModule, ErroresFormComponent],
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    formulario: FormGroup;

    constructor() {
        this.formulario = new FormGroup({
            username: new FormControl('bartolomeo', Validators.required),
            email: new FormControl('', Validators.required),
            password: new FormControl('', [Validators.required, Validators.minLength(8)],
                CustomValidators.passwordSegura(['password', '12345678', '12341234', '1234567890', '0987654321', '987654321',
                    'password123', 'qwertyuiop', 'qwerty123'])),
            confirmPassword: new FormControl('', [Validators.required, Validators.minLength(8)],
                CustomValidators.repetirPassword),
        })
    }

    guardar() {
        console.log(this.formulario.value);
    }
}

```

```

pintarErrores(controlName: string) {
  const control = this.formulario.get(controlName)!
  return control.invalid && control.dirty
}

getErrores(controlName: string) {
  const control = this.formulario.get(controlName)!
  return control.errors
}
}

```

Y tendremos que añadir un nuevo mensaje de error para esta nueva validación en nuestro componente de errores.

/angular-formularios-crear-validacion-lab/src/app/errores-form/errores-form.component.ts

```

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

@Component({
  selector: 'app-errores-form',
  standalone: true,
  imports: [],
  templateUrl: './errores-form.component.html',
  styleUrls: ['./errores-form.component.css']
})
export class ErroresFormComponent implements OnChanges {

  @Input() errores: {[key: string]: any} | null = null
  mensajesErrores: Array<string> = []

  constructor() { }

  ngOnChanges(): void {
    this.mensajesErrores = []
    for (let keyErr in this.errores) {
      switch (keyErr) {
        case 'required':
          this.mensajesErrores.push('El campo es obligatorio')
          break;
        case 'minlength':
          const msg = `El campo necesita ${this.errores[keyErr].requiredLength} caracteres. Te faltan
${this.errores[keyErr].requiredLength - this.errores[keyErr].actualLength}.`
          this.mensajesErrores.push(msg)
          break;
        case 'passwordSegura':
          this.mensajesErrores.push('La contraseña introducida es insegura')
          break;
        case 'repetirPassword':
          this.mensajesErrores.push('Esta contraseña no coincide con la anterior')
          break;
      }
    }
  }
}

```

Y con esto ya deberíamos de poder comprobar que los dos campos de las contraseñas coinciden.

Chapter 16. Servicios e inyección de dependencias

Los **servicios** son una pieza fundamental de Angular, toda aquella funcionalidad que pueda estar repetida en distintos componentes se debería de añadir dentro de estos elementos. Algunas de las tareas más comunes de estos servicios son:

- Realizar peticiones HTTP a las APIs
- Gestionar el token de authenticación (interactuar con los storage del navegador)
- Almacenar observables para la comunicación entre componentes
- ...

Imaginemos que en 3 componentes de nuestra aplicación tenemos que realizar una tarea X, y tenemos el código que realiza dicha tarea repetido en estos 3 componentes... En el caso de que un día nuestros superiores nos dijeran de modificar dicha funcionalidad, tendríamos que modificar el código en 3 sitios diferentes.

Sin embargo, si este mismo código lo sacamos de los componentes y lo metemos en un servicio, esta modificación solo tendríamos que realizarla en un único sitio, en el servicio, y los 3 componentes solo tendrían que llamar al método del servicio para realizar la tarea X.



DRY (Don't Repeat Yourself). Si tienes funcionalidad repetida en distintos componentes se debería de centralizar en un servicio. Es más fácil de mantener el código.

Para crear los servicios usamos alguno de los siguientes 2 comandos:

```
$ ng generate service mi-servicio  
$ ng g s mi-servicio
```

Los servicios son clases de TypeScript que llevan un decorador **@Injectable()** dentro del cual se indica que esta clase es un servicio del cual vamos a tener una única instancia para toda la aplicación.

Dentro de la clase es donde hay que poner la lógica de la que se vaya a encargar el servicio.

/src/app/logger.service.ts

```
import { Injectable } from '@angular/core';  
  
@Injectable({  
  providedIn: 'root'  
})  
export class LoggerService {  
  
  constructor() { }
```

```

error(msgError: string): void {
  console.error(`[ERROR] ${msgError}`)
}

warning(msgWarn: string): void {
  console.warn(`[WARNING] ${msgWarn}`)
}
}

```

Para utilizar los servicios en los componentes, u otros elementos como directivas, otros servicios, interceptores... hay que inyectar la dependencia, es decir, la instancia del servicio.

Inyectar estas dependencias es sencillo, solo tenemos que añadirlos como parámetros de los constructores de las clases de estos elementos (sin olvidar ponerles la visibilidad).

/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { LoggerService } from './auth.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  constructor(private logger: LoggerService) {}

  mostrarError(msg: string): void {
    this.logger.error(msg)
  }

  mostrarWarning(msg: string): void {
    this.logger.warning(msg)
  }
}

```

/src/app/app.component.html

```

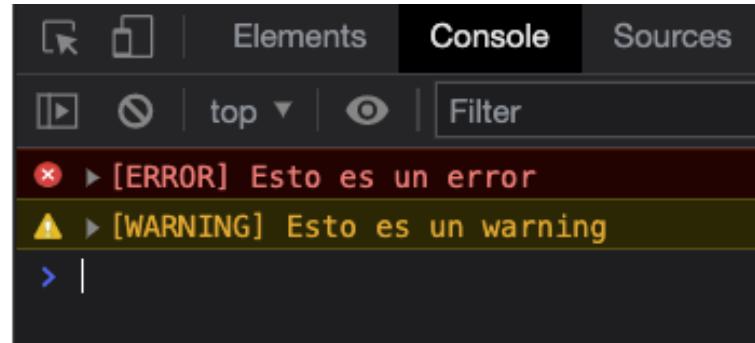
<input type="text" #inputMsg>

<button type="button" (click)="mostrarError(inputMsg.value)">Mostrar mensaje de error</button>
<button type="button" (click)="mostrarWarning(inputMsg.value)">Mostrar mensaje de warning</button>

```

Al escribir un mensaje en el campo de texto y pulsar sobre los botones deberíamos de verlo en la consola del navegador, como se muestra a continuación.

```
Esto es un warning  
Mostrar mensaje de error  
Mostrar mensaje de warning
```



16.1. Lab: Servicios e inyección de dependencias

En este laboratorio vamos a ver como crear un servicio encargado de interactuar con el **LocalStorage** del navegador para guardar en el los tokens de autenticación.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-servicios-lab  
? Would you like to add Angular routing? N  
? Which stylesheet format would you like to use? CSS
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, creamos un servicio y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-servicios-lab  
$ ng g s auth  
$ ng s
```

Vamos a empezar por crear los distintos métodos que vamos a usar para guardar, eliminar, obtener un token y comprobar si existe el token.

/angular-servicios-lab/src/app/auth.service.ts

```
import { Injectable } from '@angular/core';  
  
@Injectable({  
  providedIn: 'root'  
)  
export class AuthService {  
  
  constructor() {}  
  
  setToken(token: string): void {  
  }  
  
  getToken(): string | null {  
  }  
  
  delToken(): void {  
  }  
  
  hasToken(): boolean {  
  }
```

```
}
```

Para interactuar con el **LocalStorage** del navegador, vamos a utilizar los distintos métodos que tiene el objeto **localStorage**.

En el primer método, **setToken**, que es con el que vamos a guardar el token, vamos a utilizar la función **setItem** a la que le vamos a pasar como clave **miToken** y como valor el token que recibirá esta función como parámetro.

/angular-servicios-lab/src/app/auth.service.ts

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

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

  constructor() { }

  setToken(token: string): void {
    localStorage.setItem('miToken', token)
  }

  getToken(): string | null {
  }

  delToken(): void {
  }

  hasToken(): boolean {
  }
}
```

La siguiente función es en la que vamos a pedir el token que se ha guardado con la clave **miToken**, y para ello vamos a usar la función de **localStorage.getItem**.

/angular-servicios-lab/src/app/auth.service.ts

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

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

  constructor() { }
```

```

setToken(token: string): void {
  localStorage.setItem('miToken', token)
}

getToken(): string | null {
  return localStorage.getItem('miToken')
}

delToken(): void {

}

hasToken(): boolean {
}

```

La siguiente función se encarga de eliminar el token, y para hacer esto llamaremos a la función **removeItem** pasandole como parámetro la clave del registro que queremos eliminar, en nuestro caso **miToken**.

/angular-servicios-lab/src/app/auth.service.ts

```

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

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

  constructor() { }

  setToken(token: string): void {
    localStorage.setItem('miToken', token)
  }

  getToken(): string | null {
    return localStorage.getItem('miToken')
  }

  delToken(): void {
    localStorage.removeItem('miToken')
  }

  hasToken(): boolean {
  }
}

```

Y por último, nos queda la función de **hasToken** en la que llamaremos a la de **getToken**. Si esta llamada devuelve un **null** retornaremos un false, ya que el token no existe en el localStorage, pero si devuelve un string, entonces devolveremos un true.

/angular-servicios-lab/src/app/auth.service.ts

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

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

  constructor() { }

  setToken(token: string): void {
    localStorage.setItem('miToken', token)
  }

  getToken(): string | null {
    return localStorage.getItem('miToken')
  }

  delToken(): void {
    localStorage.removeItem('miToken')
  }

  hasToken(): boolean {
    return this.getToken() !== null
  }
}
```

Ya tenemos nuestro servicio con sus funciones hecho. Ahora toca ir al componente de App donde vamos a inyectar la instancia de este servicio en el constructor.

/angular-servicios-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { AuthService } from './auth.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  constructor(private auth: AuthService) {}

}
```

Ahora vamos a poner dos funciones, una para hacer el **login** y otra para hacer el **logout**.

/angular-servicios-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { AuthService } from './auth.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  constructor(private auth: AuthService) {}

  login(): void {
  }

  logout(): void {
  }
}
```

Para hacer el login, tendríamos que hacer una petición POST, pero ya que este laboratorio no va de hacer peticiones HTTP y no tenemos un backend que se encargue de autenticarnos, vamos a simular con **setTimeout** que se hace la petición y que tarda un poquito en recibir una respuesta.

Dentro del **setTimeout** vamos a generar un token (será la parte decimal de un número aleatorio), y vamos a llamar a la función de **setToken** del servicio para guardarla dentro del **localStorage** del navegador.

/angular-servicios-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { AuthService } from './auth.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  constructor(private auth: AuthService) {}

  login(): void {
    console.log('Logueando...')
    setTimeout(() => {

```

```

        console.log('Usuario logueado')
        const token = Math.random().toString().slice(2)
        this.auth.setToken(token)
    }, 1000)
}

logout(): void {
}
}

```

En la función del **logout** solo tendremos que llamar a **delToken** del servicio para eliminarlo del cliente y que este no se pueda seguir enviando al servidor.

/angular-servicios-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { AuthService } from './auth.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  constructor(private auth: AuthService) {}

  login(): void {
    console.log('Logueando...')
    setTimeout(() => {
      console.log('Usuario logueado')
      const token = Math.random().toString().slice(2)
      this.auth.setToken(token)
    }, 1000)
  }

  logout(): void {
    this.auth.delToken()
  }
}

```

El próximo paso es añadir dos botones en el HTML del componente App, uno para el login y otro para el logout.

Utilizaremos la directiva **ngIf** para decidir cual de los dos mostrar. La condición será el valor de una propiedad **isLoggedIn** que añadiremos después.

```
/angular-servicios-lab/src/app/app.component.html
```

```
<button type="button" *ngIf="isLoggedIn; else loginBtn">Logout</button>
<ng-template #loginBtn>
  <button type="button">Login</button>
</ng-template>
```

Además vamos a aprovechar para añadir un evento **click** sobre los dos botones, para que se ejecuten las funciones de **login** y **logout** al pulsar sobre ellos.

```
/angular-servicios-lab/src/app/app.component.html
```

```
<button type="button" *ngIf="isLoggedIn; else loginBtn" (click)="logout()">Logout</button>
<ng-template #loginBtn>
  <button type="button" (click)="login()">Login</button>
</ng-template>
```

Ahora en el TypeScript vamos a añadir una propiedad **isLoggedIn** del tipo booleano que vamos a inicializar a **false**.

```
/angular-servicios-lab/src/app/app.component.ts
```

```
import { Component } from '@angular/core';
import { AuthService } from './auth.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  isLoggedIn: boolean = false

  constructor(private auth: AuthService) {}

  login(): void {
    console.log('Logueando...')
    setTimeout(() => {
      console.log('Usuario logueado')
      const token = Math.random().toString().slice(2)
      this.auth.setToken(token)
    }, 1000)
  }

  logout(): void {
    this.auth.delToken()
  }
}
```

Esta propiedad la vamos a inicializar en el método del ciclo de vida **ngOnInit**, y obtendremos el valor de la función **hasToken** que hay en el servicio.

/angular-servicios-lab/src/app/app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from './auth.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  isLoggedIn: boolean = false

  constructor(private auth: AuthService) {}

  ngOnInit() {
    this.isLoggedIn = this.auth.hasToken()
  }

  login(): void {
    console.log('Logueando...')
    setTimeout(() => {
      console.log('Usuario logueado')
      const token = Math.random().toString().slice(2)
      this.auth.setToken(token)
    }, 1000)
  }

  logout(): void {
    this.auth.delToken()
  }
}
```

Con todo lo que hemos hecho hasta ahora, ya podemos guardar el token y eliminarlo pulsando los botones.

Si pulsamos sobre el de login y entramos en **Herramientas del desarrollador > Application > Local Storage > <http://localhost:4200>**, deberíamos de ver el token que se ha generado.

Key	Value
miToken	9914125254615305

Al pulsar sobre el botón del login, vemos que no se actualiza la vista y se muestra el botón del logout, pero no pasa nada, por ahora esto no lo hemos implementado, lo haremos en el siguiente laboratorio.

Por tanto, para ver si funciona el logout, vamos a refrescar el navegador para que al cargar desde 0 la aplicación, el `isLoggedIn` se inicialice esta vez con un `true` y por tanto aparezca el otro botón.

Al pulsar sobre el botón de **Logout**, tiene que desaparecer el token del Local Storage.

Key	Value
-----	-------

Como hemos visto, todavía hay algo que podemos mejorar, y es que cuando pulsemos sobre cualquiera de los dos botones, el `isLoggedIn` debería de cambiar y con el la vista también pasando a mostrar el otro botón.

Esto es algo que haremos en el siguiente laboratorio.

16.2. Comunicar componentes mediante servicios

Una forma de comunicar un componente con otro es mediante el uso de servicios y el **EventEmitter** que ya vimos anteriormente a la hora de crear eventos.

El **EventEmitter** es una versión especial de los observables (que veremos en un tema más adelante). Básicamente nos va a permitir:

- emitir eventos con una función **emit(dato)**.
- suscribirnos a estos eventos con una función **subscribe((dato) => {})**.

La idea es crear una instancia del EventEmitter en un servicio, y utilizar los métodos comentados antes para emitir unos datos desde un componente, y suscribirnos a estos datos en otro componente.



La comunicación no tiene porque ser solo entre componentes, también podemos enviar datos desde un servicio distinto a un componente.

De esta forma podemos hacer que la aplicación reaccione a estos eventos que se van a ir emitiendo con las acciones que realiza el usuario.

En el siguiente servicio tenemos la instancia del EventEmitter y dentro de la función hemos puesto la llamada a **emit** para emitir un mensaje después de pasar 1 segundo y medio de que se ejecute dicha función.

/src/app/eventos.service.ts

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

@Injectable({
  providedIn: 'root'
})
export class EventosService {
  mensaje$ = new EventEmitter<string>()

  constructor() { }

  pedirMensaje(): void {
    setTimeout(() => {
      this.mensaje$.emit('Este es un mensaje cualquiera')
    }, 1500)
  }
}
```

En el **ngOnInit** se realiza la suscripción al EventEmitter para recibir el mensaje cada vez que se ejecute la función del servicio que lo emite.

/src/app/app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { EventosService } from './eventos.service';

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

  constructor(private eventos: EventosService) {}

  ngOnInit() {
    this.eventos.mensaje$.subscribe((msg: string) => {
      this.mensaje = msg
    })
  }

  pedirMensaje(): void {
    this.eventos.pedirMensaje()
  }
}
```

/src/app/app.component.html

```
<p>Mensaje: {{mensaje}}</p>
<button type="button" (click)="pedirMensaje()">Pedir mensaje</button>
```

16.3. Lab: Comunicar componentes mediante servicios

En este laboratorio vamos a utilizar un EventEmitter en un servicio para poder cambiar los botones de Login y Logout del laboratorio anterior cuando se guarda el token y se elimina.

Para empezar vamos a copiarnos el proyecto de Angular del laboratorio anterior (**Lab: Servicios e inyección de dependencias**) y lo vamos a pegar cambiandole el nombre de la carpeta a **angular-servicios-comunicar-componentes-lab**.

Una vez tenemos el nuevo proyecto, vamos a entrar en la carpeta de este, vamos a crear un nuevo servicio y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-servicios-comunicar-componentes-lab  
$ ng g s eventos  
$ ng s
```

Vamos a empezar por añadir la instancia del EventEmitter dentro del servicio de eventos.



Cuidado con la importación del EventEmitter, ya que cuando se importa de forma automática, muchas veces lo importa desde **stream** o **events**, y este no es el mismo EventEmitter que queremos utilizar.

Al crear la instancia le tenemos que indicar que tipo de datos vamos a emitir con el, a lo que le vamos a indicar que emitiremos booleanos.



El nombre de los observables terminan por convención en \$.

/angular-servicios-comunicar-componentes-lab/src/app/eventos.service.ts

```
import { EventEmitter, Injectable } from '@angular/core';  
  
@Injectable({  
  providedIn: 'root'  
})  
export class EventosService {  
  
  authEvent$ = new EventEmitter<boolean>()  
  
  constructor() { }  
  
}
```

Ahora nos vamos a ir al servicio de **AuthService** que tenemos dentro del proyecto, en el que vamos a inyectar la instancia del servicio anterior.

/angular-servicios-comunicar-componentes-lab/src/app/auth.service.ts

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

```

import { EventosService } from './eventos.service';

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

  constructor(private eventos: EventosService) { }

  setToken(token: string): void {
    localStorage.setItem('miToken', token)
  }

  getToken(): string | null {
    return localStorage.getItem('miToken')
  }

  delToken(): void {
    localStorage.removeItem('miToken')
  }

  hasToken(): boolean {
    return this.getToken() !== null
  }
}

```

Al guardar un token vamos a querer avisar al componente de App, que ahora estamos logueados para que cambie el **isLoggedIn** a **true**, por lo que después de llamar al **setItem**, vamos a acceder al evento y vamos a emitir con el un **true**.

/angular-servicios-comunicar-componentes-lab/src/app/auth.service.ts

```

import { Injectable } from '@angular/core';
import { EventosService } from './eventos.service';

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

  constructor(private eventos: EventosService) { }

  setToken(token: string): void {
    localStorage.setItem('miToken', token)
    this.eventos.authEvent$.emit(true)
  }

  getToken(): string | null {
    return localStorage.getItem('miToken')
  }
}

```

```

    delToken(): void {
      localStorage.removeItem('miToken')
    }

    hasToken(): boolean {
      return this.getToken() !== null
    }
}

```

Haremos lo mismo al eliminar el token, para indicarle al componente que ya no estamos logueados y que tiene que poner el **isLoggedIn** a **false**. Por lo que después de llamar al **removeItem**, vamos a emitir con el **EventEmitter** un valor **false**.

/angular-servicios-comunicar-componentes-lab/src/app/auth.service.ts

```

import { Injectable } from '@angular/core';
import { EventosService } from './eventos.service';

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

  constructor(private eventos: EventosService) { }

  setToken(token: string): void {
    localStorage.setItem('miToken', token)
    this.eventos.authEvent$.emit(true)
  }

  getToken(): string | null {
    return localStorage.getItem('miToken')
  }

  delToken(): void {
    localStorage.removeItem('miToken')
    this.eventos.authEvent$.emit(false)
  }

  hasToken(): boolean {
    return this.getToken() !== null
  }
}

```

Ya tenemos la parte de emisión de eventos. El siguiente paso es recibir estos eventos en el componente App.

Empezaremos por inyectar de nuevo el servicio de **EventosService** dentro del constructor del componente.

/angular-servicios-comunicar-componentes-lab/src/app/app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from './auth.service';
import { EventosService } from './eventos.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  isLoggedIn: boolean = false

  constructor(private auth: AuthService, private eventos: EventosService) {}

  ngOnInit() {
    this.isLoggedIn = this.auth.hasToken()
  }

  login(): void {
    console.log('Logueando...')
    setTimeout(() => {
      console.log('Usuario logueado')
      const token = Math.random().toString().slice(2)
      this.auth.setToken(token)
    }, 1000)
  }

  logout(): void {
    this.auth.delToken()
  }
}
```

Por último, vamos a añadir la suscripción a los eventos que se emitan con **authEvent\$** llamando a la función **subscribe** en el método del **ngOnInit**.

A esta función **subscribe** le vamos a pasar como parámetro una función de callback en la que recibiremos los datos que se emiten con el EventEmitter, es decir, los booleanos que hemos puesto dentro de las funciones **emit** del servicio de **AuthService**.

/angular-servicios-comunicar-componentes-lab/src/app/app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from './auth.service';
import { EventosService } from './eventos.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  isLoggedIn: boolean = false

  constructor(private auth: AuthService, private eventos: EventosService) {}

  ngOnInit() {
    this.isLoggedIn = this.auth.hasToken()

    this.eventos.authEvent$.subscribe(data => {
      this.isLoggedIn = data
    })
  }

  login(): void {
    console.log('Logueando...')
    setTimeout(() => {
      console.log('Usuario logueado')
      const token = Math.random().toString().slice(2)
      this.auth.setToken(token)
    }, 1000)
  }

  logout(): void {
    this.auth.delToken()
  }
}
```

```

    styleUrls: ['./app.component.css']
)
export class AppComponent implements OnInit {
  isLoggedIn: boolean = false

  constructor(private auth: AuthService, private eventos: EventosService) {}

  ngOnInit() {
    this.isLoggedIn = this.auth.hasToken()
    this.eventos.authEvent$.subscribe((hasToken: boolean) => {
      this.isLoggedIn = hasToken
    })
  }

  login(): void {
    console.log('Logueando...')
    setTimeout(() => {
      console.log('Usuario logueado')
      const token = Math.random().toString().slice(2)
      this.auth.setToken(token)
    }, 1000)
  }

  logout(): void {
    this.auth.delToken()
  }
}

```

Una vez hecho esto, ya podemos ver como al pulsar sobre los botones, estos cambian de Login a Logout y viceversa, al mismo tiempo que se añade y elimina el token en el **LocalStorage**.

Chapter 17. Observables (RxJS)

Los **observables** son un tipo de objetos que pueden almacenar valores y emitirlos a sus suscriptores. Estos objetos son los que se utilizan en Angular para trabajar con las operaciones asíncronas, es decir, operaciones que no se resuelven inmediatamente.

Si los comparamos con las promesas, hay una diferencia muy grande, y es que aunque los dos objetos nos permiten trabajar con operaciones asíncronas, las promesas se terminan una vez que la operación asíncrona termina, devolviendo el resultado una sola vez, mientras que los observables pueden seguir enviando más resultados y solo se marcarán como finalizados cuando eliminemos la suscripción a este.

Al crear un observable tenemos dos tipos de objetos:

- El **Observable** es el objeto en el que vamos a definir la lógica de la operación asíncrona.
- El **Subscriber** es el objeto que vamos a usar para comunicarnos con nuestros suscriptores. Con este objeto les vamos a enviar los datos, errores o les vamos a indicar que el observable ya no va a enviar nada más. Para ello usaremos los siguientes métodos:
 - next(datos)
 - error(error)
 - complete()

/src/app/app.component.ts

```
import { Component, OnDestroy } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  constructor() {

    const miObservable = new Observable((subscriber: Subscriber<string>) => {
      setTimeout(() => {
        subscriber.next('Dato 1')
      }, 2000)
      setTimeout(() => {
        subscriber.next('Dato 2')
      }, 4000)
      setTimeout(() => {
        // subscriber.error('Error')
        subscriber.complete()
      }, 6000)
    })
  }
}
```

```
    }, 6000)  
})  
  
}
```

17.1. Suscripciones

Para trabajar con los observables tenemos que conocer otro concepto importante, que son las suscripciones.

Los observables envían datos, pero para poder recibir estos datos, en algún sitio hay que suscribirse a ellos. Esto se consigue con la función **subscribe** de los observables.



Un observable no funciona a no ser que haya al menos una suscripción sobre él.

Esta función recibe como parámetro:

- Una función que se va a ejecutar cuando se llame al **subscriber.next** desde el observable. Esta función recibe como parámetro el dato enviado por el **next**.
- En lugar de una función, si queremos controlar también los errores y cuando se completa el observable, le vamos a pasar como parámetro un objeto con las claves **next**, **error** y **complete**, y cuyos valores serán las funciones a ejecutar cuando desde el interior del observable se llame a esos mismos métodos del objeto **subscriber**.

/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  constructor() {

    const miObservable = new Observable((subscriber: Subscriber<string>) => {
      setTimeout(() => {
        subscriber.next('Dato 1')
      }, 2000)
      setTimeout(() => {
        subscriber.next('Dato 2')
      }, 4000)
      setTimeout(() => {
        // subscriber.error('Error')
        subscriber.complete()
      }, 6000)
    })

    miObservable.subscribe({
      next: (dato: string) => console.log(dato),
    })
  }
}
```

```

        error: (err: string) => console.error(err),
        complete: () => console.log('Se acabó'),
    })
}

}

```

Cuando llamamos a la función **subscribe** de un observable, esta retorna un objeto de tipo **Subscription**. Este objeto lo usaremos para poder eliminar la suscripción en cualquier momento llamando a su método **unsubscribe**.

Normalmente las desuscripciones se realizan en el método del ciclo de vida **ngOnDestroy** para evitar que el observable siga ejecutándose a pesar de que el componente en el que se estaba usando ya no exista.

/src/app/app.component.ts

```

import { Component, OnDestroy } from '@angular/core';
import { Observable, Subscriber, Subscription } from 'rxjs';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnDestroy {
  suscripcion: Subscription;

  constructor() {

    const miObservable = new Observable((subscriber: Subscriber<string>) => {
      setTimeout(() => {
        subscriber.next('Dato 1')
      }, 2000)
      setTimeout(() => {
        subscriber.next('Dato 2')
      }, 4000)
      setTimeout(() => {
        // subscriber.error('Error')
        subscriber.complete()
      }, 6000)
    })

    this.suscripcion = miObservable.subscribe({
      next: (dato: string) => console.log(dato),
      error: (err: string) => console.error(err),
      complete: () => console.log('Se acabó'),
    })
  }
}

```

```
})  
}  
  
ngOnDestroy() {  
  this.suscripcion.unsubscribe();  
}  
  
}
```



Cuando desde dentro de un observable llamamos a los métodos de **error** y **complete**, la suscripción se termina, por lo que no hace falta que nos desuscribamos de ella llamando al método **unsubscribe** de los objetos **Subscription**.

17.2. Operadores

La librería de RxJS nos proporciona una serie de métodos llamados **operadores** que podemos aplicar sobre un observable (antes de suscribirnos) para alterar los datos que nos llegan, por ejemplo transformándolos, filtrándolos, combinándolos con otros datos...

Para utilizar estos operadores, tenemos que llamar a la función **pipe()** sobre el observable y pasarle como parámetros todos los operadores que queramos aplicar.

Algunos de los operadores que podemos usar:

- map: recibe como parámetro un valor emitido por el observable y tiene que retornar otro valor. Normalmente será el mismo valor pero con alguna modificación.
- filter: recibe como parámetro un valor emitido por el observable y tiene que retornar un booleano. Si retorna un **true**, el valor llegará a la suscripción, pero si retorna un **false** el valor no llegará.
- take: emite solo los N primeros valores que manda el observable, siendo N el parámetro que se le pasa a este operador.
- Podemos ver todos los operadores que hay en <https://rxjs.dev/api/operators>.

Los operadores se importan desde **rxjs/operators**.



interval es un observable de la librería RxJS que emite números de 1 en 1 cada vez que pasa el tiempo que se le pasa como parámetro.

/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { interval } from 'rxjs';
import { map, filter } from 'rxjs/operators'

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  constructor() {
    interval(1000)
      .pipe(
        map(n => n * 2),
        filter(n => n < 20),
      )
      .subscribe((num: number) => console.log(num))
  }
}
```

}

17.3. Lab: Observables

En este laboratorio vamos a ver como utilizar los observables para gestionar las suscripciones a distintos servicios de pago.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-observables-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? no
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, vamos a crear un componente, un servicio y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-observables-lab  
$ ng g c suscripcion  
$ ng g s pagos  
$ ng s
```

Dentro del componente de **suscripcion** que hemos creado vamos a empezar declarando las propiedades que vamos a necesitar para gestionar los datos de una suscripción.

Estas propiedades serán:

- **plataforma**: el nombre de la plataforma a la que nos hemos suscrito.
- **precio**: el precio de la suscripción.
- **activa**: un booleano que nos indica si tenemos la suscripción activa o no.
- **fechaProximoPago**: la fecha en la que se renovará de forma automática la suscripción y en la que se intentará realizar un cobro.

E importaremos los pipes de currency y date que vamos a utilizar en la plantilla.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```
import { CurrencyPipe, DatePipe } from '@angular/common';  
import { Component, Input, OnInit } from '@angular/core';  
  
@Component({  
  selector: 'app-suscripcion',  
  standalone: true,  
  imports: [CurrencyPipe, DatePipe],  
  templateUrl: './suscripcion.component.html',  
  styleUrls: ['./suscripcion.component.css'  
})  
export class SuscripcionComponent implements OnInit {  
  @Input() plataforma: string = 'Desconocida'
```

```

@Input() precio: number = 4.99
activa: boolean = false
fechaProximoPago = new Date()

constructor() { }

ngOnInit(): void {

}

}

```

Dentro de la plantilla de este mismo componente vamos a mostrar estas datos. Además de añadir dos botones con los que vamos a activar o cancelar la suscripción.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.html

```

<div>
  <h4>Suscripción a {{plataforma}}</h4>
  <p>Estado: {{activa ? 'activa' : 'no activa'}}</p>
  <p>Próximo pago: {{fechaProximoPago | date:'hh:mm:ss'}}</p>
  <p>Cantidad: {{precio | currency:'EUR'}}</p>
  @if (activa) {
    <button type="button" (click)="cancelarSuscripcion()">Cancelar suscripción</button>
  } @else {
    <button type="button" (click)="activarSuscripcion()">Activar suscripción</button>
  }
</div>

```

Vamos a añadir las funciones de **activarSuscripcion** y **cancelarSuscripcion** dentro del TypeScript, de momento las vamos a dejar vacías.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```

import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()
}

```

```

constructor() { }

ngOnInit(): void {
}

activarSuscripcion() {

}

cancelarSuscripcion() {

}

}

```

Antes de continuar con la lógica de este componente vamos a mostrarlo dentro de la plantilla del componente App, y le pasaremos los datos a los cuales le hemos puesto el decorador `@Input()`.

Tenemos que importarlo primero en el componente para poder utilizarlo.

/angular-observables-lab/src/app/app.component.ts

```

import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { SuscripcionComponent } from './components/suscripcion/suscripcion.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, SuscripcionComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
}

```

/angular-observables-lab/src/app/app.component.html

```
<app-suscripcion plataforma="AngularFlix" [precio]="9.95"></app-suscripcion>
```

Ya deberíamos de ver los datos de la suscripción en el navegador, <http://localhost:4200/>.

Dentro de la función de **activarSuscripcion** vamos a crear un observable que va a recibir una función como parámetro donde vamos a añadir la lógica para realizar la renovación de las suscripciones. Esta función a su vez recibe como parámetro el objeto **subscriber** con el que enviaremos datos a los suscriptores.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```
import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()

  constructor() { }

  ngOnInit(): void {
    }

    activarSuscripcion() {
      new Observable((subscriber: Subscriber<string>) => {
        })
    }

    cancelarSuscripcion() {
    }
}
```

Cuando pulsemos sobre el botón de activar la suscripción, se va a crear el observable, y lo primero que vamos a hacer es cambiar la propiedad **activa** a **true**.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```
import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
```

```

        templateUrl: './suscripcion.component.html',
        styleUrls: ['./suscripcion.component.css']
    })
export class SuscripcionComponent implements OnInit {
    @Input() plataforma: string = 'Desconocida'
    @Input() precio: number = 4.99
    activa: boolean = false
    fechaProximoPago = new Date()

    constructor() { }

    ngOnInit(): void {

    }

    activarSuscripcion() {
        new Observable((subscriber: Subscriber<string>) => {
            this.activa = true
        })
    }

    cancelarSuscripcion() {

    }
}

```

Ahora la idea es que cada 5 segundos se intente realizar un pago (lo simularemos en el servicio que hemos creado). 5 segundos para que podamos probar el ejemplo antes de que se acabe el curso, os imagináis que ponemos la renovación cada mes XD.

Para esto vamos a añadir una función **setInterval** a la que le pasaremos la función con la lógica de renovación de la suscripción y haremos que se ejecute cada 5000ms.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```

import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';

@Component({
    selector: 'app-suscripcion',
    standalone: true,
    imports: [CurrencyPipe, DatePipe],
    templateUrl: './suscripcion.component.html',
    styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
    @Input() plataforma: string = 'Desconocida'
    @Input() precio: number = 4.99

```

```

activa: boolean = false
fechaProximoPago = new Date()

constructor() { }

ngOnInit(): void {

}

activarSuscripcion() {
  new Observable((subscriber: Subscriber<string>) => {
    this.activa = true

    setInterval(() => {
      }, 5000)
  })
}

cancelarSuscripcion() {
}

}

```

Antes de seguir por ahí necesitamos añadir en el servicio un método para simular que el pago se realiza o no se realiza.

Para ello, vamos a crear una función **pagoCorrecto()** que devolverá un booleano. Este booleano será true siempre que el número aleatorio sacado sea par o mayor que 6, de esta forma hacemos que haya menos probabilidades de que el pago no se procese.

/angular-observables-lab/src/app/services/pagos.service.ts

```

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

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

  constructor() { }

  pagoCorrecto(): boolean {
    const randomNum: number = Math.floor(Math.random()*10)
    return randomNum % 2 === 0 || randomNum > 6
  }
}

```

Ahora que tenemos nuestro método, volvemos al componente de la suscripción, donde pondremos

en un **if** la llamada a esta función que acabamos de crear. También tenemos que inyectar el servicio en este componente.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```
import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()

  constructor(
    private pagos: PagosService,
  ) { }

  ngOnInit(): void {
  }

  activarSuscripcion() {
    new Observable((subscriber: Subscriber<string>) => {
      this.activa = true

      setInterval(() => {
        if (this.pagos.pagoCorrecto()) {

          } else {

        }
      }, 5000)
    })
  }

  cancelarSuscripcion() {
  }
}
```

Cuando el pago se ha procesado correctamente, vamos a obtener una nueva fecha de renovación que obtendremos de la fecha actual a la que le vamos a sumar los 5 segundos que hemos dicho antes.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```
import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()

  constructor(
    private pagos: PagosService,
  ) { }

  ngOnInit(): void {

  }

  activarSuscripcion() {
    new Observable((subscriber: Subscriber<string>) => {
      this.activa = true

      setInterval(() => {
        if (this.pagos.pagoCorrecto()) {
          const fecha = new Date()
          fecha.setSeconds(fecha.getSeconds() + 5)
          this.fechaProximoPago = fecha

        } else {
          }
        }, 5000)
      })
    }

    cancelarSuscripcion() {
    }
}
```

```
}
```

```
}
```

Y además de cambiar la fecha del próximo pago, vamos a enviarle la hora en la que se realizará esta renovación al suscriptor. Para enviar este dato desde el observable, vamos a utilizar el **subscriber.next**.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```
import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()

  constructor(
    private pagos: PagosService,
  ) { }

  ngOnInit(): void {
  }

  activarSuscripcion() {
    new Observable((subscriber: Subscriber<string>) => {
      this.activa = true

      setInterval(() => {
        if (this.pagos.pagoCorrecto()) {
          const fecha = new Date()
          fecha.setSeconds(fecha.getSeconds() + 5)
          this.fechaProximoPago = fecha

          subscriber.next(fecha.toLocaleTimeString())
        } else {
        }
      })
    })
  }
}
```

```

    }, 5000)
  })
}

cancelarSuscripcion() {
}

```

Ahora toca llenar la parte del **else**, es decir, que hacer cuando el pago falla. Que el pago falle se puede deber a que la tarjeta de la que se va a cobrar la suscripción esté congelada, dada de baja...

Por tanto, si no se puede cobrar, vamos a cancelar la suscripción, y vamos a enviarle al suscriptor un error con **subscriber.error** indicándole que se ha cancelado su suscripción y que tendrá que volver a suscribirse si quiere continuar utilizando la plataforma.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```

import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})

export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()

  constructor(
    private pagos: PagosService,
  ) { }

  ngOnInit(): void {
  }

  activarSuscripcion() {
    new Observable((subscriber: Subscriber<string>) => {
      this.activa = true

      setInterval(() => {
        if (this.pagos.pagoCorrecto()) {
          const fecha = new Date()
          fecha.setSeconds(fecha.getSeconds() + 5)
          this.fechaProximoPago = fecha

          subscriber.next(fecha.toLocaleTimeString())
        } else {
          this.activa = false
        }
      })
    })
  }
}

```

```

        subscriber.error('No se ha podido realizar el pago. Actualiza el método de pago y vuelve a suscribirte.')
    }
},
},
}

cancelarSuscripcion() {
}

}

```

Ahora vamos a realizar la suscripción a este observable, y para ello vamos a añadir la llamada al método **subscribe** justo donde se cierra el paréntesis de la llamada al **new Observable**.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```

import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()

  constructor(
    private pagos: PagosService,
  ) { }

  ngOnInit(): void {
  }

  activarSuscripcion() {
    new Observable((subscriber: Subscriber<string>) => {
      this.activa = true

      setInterval(() => {
        if (this.pagos.pagoCorrecto()) {
          const fecha = new Date()
          fecha.setSeconds(fecha.getSeconds() + 5)
          this.fechaProximoPago = fecha

          subscriber.next(fecha.toLocaleTimeString())
        } else {
          this.activa = false

          subscriber.error('No se ha podido realizar el pago. Actualiza el método de pago y vuelve a suscribirte.')
        }
      }, 5000)
    }).subscribe()
  }
}

```

```

    }

cancelarSuscripcion() {
}

}

```

A la función **subscribe** le vamos a pasar como parámetro un objeto con las claves **next** y **error** a las que les vamos a asignar dos funciones. Las dos funciones que se van a ejecutar cada vez que se llamen a los métodos **next** y **error** desde el **subscriber** que hay dentro del observable.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```

import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()

  constructor(
    private pagos: PagosService,
  ) { }

  ngOnInit(): void {
  }

  activarSuscripcion() {
    new Observable((subscriber: Subscriber<string>) => {
      this.activa = true

      setInterval(() => {
        if (this.pagos.pagoCorrecto()) {
          const fecha = new Date()
          fecha.setSeconds(fecha.getSeconds() + 5)
          this.fechaProximoPago = fecha

          subscriber.next(fecha.toLocaleTimeString())
        } else {
          this.activa = false

          subscriber.error('No se ha podido realizar el pago. Actualiza el método de pago y vuelve a suscribirte.')
        }
      }, 5000)
    }).subscribe({
      next: (time: string) => {},
      error: (err: string) => {}
    })
  }
}

```

```

    }

cancelarSuscripcion() {
}

}

```

En la función asociada al **next** vamos a mostrar un mensaje de información indicándole al usuario que se ha renovado la suscripción y cuando será el siguiente pago. Y en la función del **error** vamos a mostrar un mensaje de error con el error que enviamos desde el observable.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```

import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()

  constructor(
    private pagos: PagosService,
  ) { }

  ngOnInit(): void {
  }

  activarSuscripcion() {
    new Observable((subscriber: Subscriber<string>) => {
      this.activa = true

      setInterval(() => {
        if (this.pagos.pagoCorrecto()) {
          const fecha = new Date()
          fecha.setSeconds(fecha.getSeconds() + 5)
          this.fechaProximoPago = fecha

          subscriber.next(fecha.toLocaleTimeString())
        } else {
          this.activa = false

          subscriber.error('No se ha podido realizar el pago. Actualiza el método de pago y vuelve a suscribirte.')
        }
      }, 5000)
    }).subscribe({
      next: (time: string) => {
        console.info(`Se ha renovado tu suscripción. El próximo pago se realizará a las ${time}.`)
      },
    })
  }
}

```

```

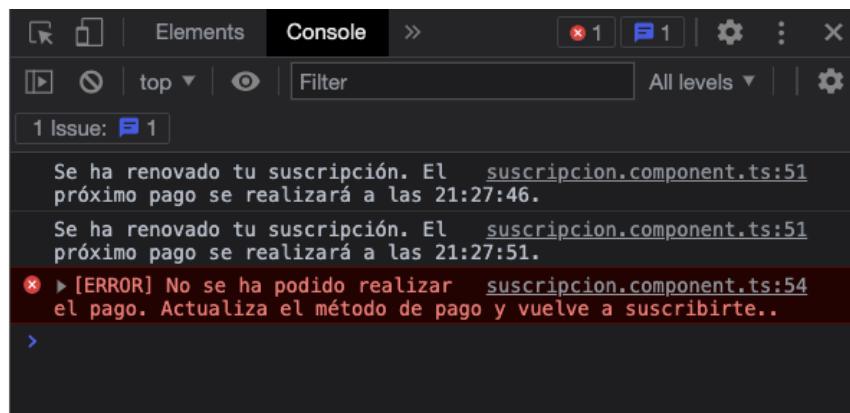
        error: (err: string) => {
          console.error(`[ERROR] ${err}.`)
        }
      })
    }

cancelarSuscripcion() {
}

}

```

Con esto ya deberíamos de poder ver los mensajes de renovación y error cuando pulsamos sobre el botón de **Activar suscripción**. Tiene que salir por la consola del terminal algo como lo que se muestra en la imagen siguiente.



Ahora nos encontramos con un problemilla, y es que el **setInterval** se sigue ejecutando aunque el observable haya dado un error. El observable ya no emite más mensajes, pero si miramos a la hora que aparece al lado del texto **Próximo pago:**, esta se sigue actualizando.

Vamos a eliminar el intervalo llamando a la función de **clearInterval** y pasándole el identificador que retorna la función de **setInterval**, para que ese dato deje de actualizarse.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```

import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()

  constructor(
    private pagos: PagosService,
  ) { }
}

```

```

ngOnInit(): void {
}

activarSuscripcion() {
  new Observable((subscriber: Subscriber<string>) => {
    this.activa = true

    const intervalId = setInterval(() => {
      if (this.pagos.pagoCorrecto()) {
        const fecha = new Date()
        fecha.setSeconds(fecha.getSeconds() + 5)
        this.fechaProximoPago = fecha

        subscriber.next(fecha.toLocaleTimeString())
      } else {
        this.activa = false

        subscriber.error('No se ha podido realizar el pago. Actualiza el método de pago y vuelve a suscribirte.')
      }
    }, 5000)
  }).subscribe({
    next: (time: string) => {
      console.info(`Se ha renovado tu suscripción. El próximo pago se realizará a las ${time}.`)
    },
    error: (err: string) => {
      console.error(`[ERROR] ${err}.`)
    }
  })
}

cancelarSuscripcion() {
}
}

```

Ahora solo nos queda añadir la funcionalidad para poder cancelar la suscripción. Para ello, vamos a crearnos una instancia de **EventEmitter**, que es un tipo de observable, y la vamos a declarar como una nueva propiedad del componente **suscripcionCancelada\$**.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```

import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, EventEmitter, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
}

```

```

activa: boolean = false
fechaProximoPago = new Date()
suscripcionCancelada$ = new Event Emitter<boolean>();

constructor(
  private pagos: PagosService,
) { }

ngOnInit(): void {

}

activarSuscripcion() {
  new Observable((subscriber: Subscriber<string>) => {
    this.activa = true

    const intervalId = setInterval(() => {
      if (this.pagos.pagoCorrecto()) {
        const fecha = new Date()
        fecha.setSeconds(fecha.getSeconds() + 5)
        this.fechaProximoPago = fecha

        subscriber.next(fecha.toLocaleTimeString())
      } else {
        this.activa = false

        subscriber.error('No se ha podido realizar el pago. Actualiza el método de pago y vuelve a suscribirte.')
      }
    }, 5000)
  }).subscribe({
    next: (time: string) => {
      console.info(`Se ha renovado tu suscripción. El próximo pago se realizará a las ${time}.`)
    },
    error: (err: string) => {
      console.error(`[ERROR] ${err}.`)
    }
  })
}

cancelarSuscripcion() {
}
}

```

Ahora desde la función de **cancelarSuscripción** vamos a emitir un **true** para avisar al observable que queremos cancelar la suscripción a la plataforma.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```

import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, Event Emitter, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
}

```

```

        styleUrls: ['./suscripcion.component.css']
    })
export class SuscripcionComponent implements OnInit {
    @Input() plataforma: string = 'Desconocida'
    @Input() precio: number = 4.99
    activa: boolean = false
    fechaProximoPago = new Date()
    suscripcionCancelada$ = new EventEmitter<boolean>();

    constructor(
        private pagos: PagosService,
    ) { }

    ngOnInit(): void {
    }

    activarSuscripcion() {
        new Observable((subscriber: Subscriber<string>) => {
            this.activa = true

            const intervalId = setInterval(() => {
                if (this.pagos.pagoCorrecto()) {
                    const fecha = new Date()
                    fecha.setSeconds(fecha.getSeconds() + 5)
                    this.fechaProximoPago = fecha

                    subscriber.next(fecha.toLocaleTimeString())
                } else {
                    this.activa = false

                    subscriber.error('No se ha podido realizar el pago. Actualiza el método de pago y vuelve a suscribirte.')
                }
            }, 5000)
        }).subscribe({
            next: (time: string) => {
                console.info(`Se ha renovado tu suscripción. El próximo pago se realizará a las ${time}.`)
            },
            error: (err: string) => {
                console.error(`[ERROR] ${err}.`)
            }
        })
    }

    cancelarSuscripcion() {
        this.suscripcionCancelada$.emit(true);
    }
}

```

Dentro del observable tendremos que suscribirnos a este **EventEmitter** para poder reaccionar cuando se emita el booleano para cancelar la suscripción. La suscripción la realizaremos después del **setInterval**.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```

import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, EventEmitter, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';

```

```

import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()
  suscripcionCancelada$ = new Event Emitter<boolean>()

  constructor(
    private pagos: PagosService,
  ) { }

  ngOnInit(): void {
  }

  activarSuscripcion() {
    new Observable((subscriber: Subscriber<string>) => {
      this.activa = true

      const intervalId = setInterval(() => {
        if (this.pagos.pagoCorrecto()) {
          const fecha = new Date()
          fecha.setSeconds(fecha.getSeconds() + 5)
          this.fechaProximoPago = fecha

          subscriber.next(fecha.toLocaleTimeString())
        } else {
          this.activa = false

          subscriber.error('No se ha podido realizar el pago. Actualiza el método de pago y vuelve a suscribirte.')
        }
      }, 5000)

      this.suscripcionCancelada$.subscribe(() => {
      })
    }).subscribe({
      next: (time: string) => {
        console.info(`Se ha renovado tu suscripción. El próximo pago se realizará a las ${time}.`)
      },
      error: (err: string) => {
        console.error(`[ERROR] ${err}.`)
      }
    })
  }

  cancelarSuscripcion() {
    this.suscripcionCancelada$.emit(true);
  }
}

```

Dentro de esta suscripción al EventEmitter vamos a cambiar la propiedad **activa** a **false**, eliminaremos el **setInterval** de la misma forma que hemos hecho antes, y llamaremos al método **complete** del **subscriber** para indicarle al usuario que esta suscripción se ha terminado.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```
import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, EventEmitter, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../../../../services/pagos.service';

@Component({
  selector: 'app-suscripcion',
  standalone: true,
  imports: [CurrencyPipe, DatePipe],
  templateUrl: './suscripcion.component.html',
  styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
  @Input() plataforma: string = 'Desconocida'
  @Input() precio: number = 4.99
  activa: boolean = false
  fechaProximoPago = new Date()
  suscripcionCancelada$ = new EventEmitter<boolean>();

  constructor(
    private pagos: PagosService,
  ) { }

  ngOnInit(): void {
  }

  activarSuscripcion() {
    new Observable((subscriber: Subscriber<string>) => {
      this.activa = true

      const intervalId = setInterval(() => {
        if (this.pagos.pagoCorrecto()) {
          const fecha = new Date()
          fecha.setSeconds(fecha.getSeconds() + 5)
          this.fechaProximoPago = fecha

          subscriber.next(fecha.toLocaleTimeString())
        } else {
          this.activa = false

          subscriber.error('No se ha podido realizar el pago. Actualiza el método de pago y vuelve a suscribirte.')
        }
        clearInterval(intervalId)
      })
    }, 5000)

    this.suscripcionCancelada$.subscribe(() => {
      this.activa = false
      clearInterval(intervalId)
      subscriber.complete()
    })
  })

  }).subscribe({
    next: (time: string) => {
      console.info(`Se ha renovado tu suscripción. El próximo pago se realizará a las ${time}.`)
    }
  })
}
```

```

        },
        error: (err: string) => {
            console.error(`[ERROR] ${err}.`)
        }
    })
}

cancelarSuscripcion() {
    this.suscripcionCancelada$.emit(true);
}

}

```

Ahora en el subscribe del Observable que hemos creado, vamos a añadirle una clave más, **complete** cuyo valor será una función con la que vamos a mostrar otro mensaje indicándole al usuario que se ha cancelado la suscripción.

/angular-observables-lab/src/app/components/suscripcion/suscripcion.component.ts

```

import { CurrencyPipe, DatePipe } from '@angular/common';
import { Component, EventEmitter, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../../services/pagos.service';

@Component({
    selector: 'app-suscripcion',
    standalone: true,
    imports: [CurrencyPipe, DatePipe],
    templateUrl: './suscripcion.component.html',
    styleUrls: ['./suscripcion.component.css']
})
export class SuscripcionComponent implements OnInit {
    @Input() plataforma: string = 'Desconocida'
    @Input() precio: number = 4.99
    activa: boolean = false
    fechaProximoPago = new Date()
    suscripcionCancelada$ = new EventEmitter<boolean>();

    constructor(
        private pagos: PagosService,
    ) { }

    ngOnInit(): void {

    }

    activarSuscripcion() {
        new Observable((subscriber: Subscriber<string>) => {
            this.activa = true

            const intervalId = setInterval(() => {
                if (this.pagos.pagoCorrecto()) {
                    const fecha = new Date()
                    fecha.setSeconds(fecha.getSeconds() + 5)
                    this.fechaProximoPago = fecha

                    subscriber.next(fecha.toLocaleTimeString())
                } else {
                    this.activa = false

                    subscriber.error('No se ha podido realizar el pago. Actualiza el método de pago y vuelve a suscribirte.')
                }
            }, 1000)
        })
    }
}

```

```
    clearInterval(intervalId)
  }
}, 5000)

this.suscripcionCancelada$.subscribe(() => {
  this.activa = false
  clearInterval(intervalId)
  subscriber.complete()
})

}).subscribe({
  next: (time: string) => {
    console.info(`Se ha renovado tu suscripción. El próximo pago se realizará a las ${time}.`)
  },
  error: (err: string) => {
    console.error(`[ERROR] ${err}.`)
  },
  complete: () => {
    console.warn(`Has cancelado tu suscripción a ${this.plataforma}.`)
  }
})
}

cancelarSuscripcion() {
  this.suscripcionCancelada$.emit(true);
}

}
```

Chapter 18. Peticiones Http

En las aplicaciones web es necesario realizar peticiones HTTP para obtener datos, crearlos, actualizarlos... Con las aplicaciones web tradicionales, cada vez que se obtenía una respuesta del servidor se recargaba la página con la información que se había pedido, pero esto no funciona así con las SPAs.

Las SPA no siguen el mismo método que las aplicaciones web tradicionales porque tenemos una sola página y cada vez que se recibe una respuesta se recarga esa página. En su lugar se hace una petición AJAX, es decir, se hace una petición HTTP que normalmente nos devuelve un JSON con los datos, y luego mediante JavaScript modificamos la vista (trabajo del que en este caso se encarga Angular).

Los tipos de peticiones más utilizados son:

- GET: se usa para obtener información sobre el recurso al que identifica la URI a la que se realiza la petición.
- POST: se utiliza para crear nuevos recursos, o mejor dicho un recurso simple del tipo al que se identifica con la URI. Los datos para crear este recurso se envían en el cuerpo de la petición.
- PUT: se usa para actualizar recursos ya existentes y que identificamos con la URI a la que se envía la petición. En realidad si lo usamos bien debería de sobreescribirse el recurso que queremos actualizar con los datos enviados en el cuerpo de la petición.
- PATCH: se usa para actualizar una parte del recurso identificado por la URI a la que se hace la petición.
- DELETE: se usa para eliminar recursos identificados por la URI a la que se envía la petición.

Método	URI	Body	Descripción
GET	/tareas		Obtiene todas las tareas
POST	/tareas	{titulo: 'T1', completada: false}	Crea la tarea con el título T1 y sin completar
PUT	/tareas/9	{titulo: 'T8', completada: true}	Sobreescribe la tarea 9 con los datos enviados. Si solo se envía la propiedad completada, el título se perdería
PATCH	/tareas/26	{completada: true}	Actualiza el valor de completada de la tarea con el id 26
DELETE	/tareas/34		Elimina la tarea con el identificador 34

18.1. HttpClient

El **HttpClient** es un servicio de Angular que nos permite realizar todos los tipos de peticiones HTTP. Para poder usar este servicio, primero hay que añadir el proveedor de Http en la configuración que tenemos en el archivo **app.config.ts**.

Este módulo trae consigo algunas características nuevas:

- La suscripción nos devuelve los datos como objetos de JavaScript.
- Podemos utilizar interceptores.
- Nos permite conocer el progreso de las peticiones (datos enviados y datos descargados).

/src/app/app.config.ts

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient(),
  ]
};
```

Una vez añadido el provider ya podemos utilizar el servicio de HttpClient. Solo necesitamos injectar la instancia en el elemento de Angular (componentes, servicios, ...) donde vayamos a querer realizar las peticiones HTTP.

Este servicio nos da acceso a los siguientes métodos:

Tipo de petición	Definición
GET	get(url, options)
POST	post(url, body, options)
PUT	put(url, body, options)
PATCH	patch(url, body, options)
DELETE	delete(url, options)

/src/app/tareas.service.ts

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

```
export interface ITarea {
  userId: number,
  id: number,
  title: string,
  completed: boolean
}

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

  constructor(private http: HttpClient) { }

  getTareas(): Observable<any> {
    return this.http.get('http://jsonplaceholder.typicode.com/todos')
  }
}
```

18.2. Lab: HttpClient

En este laboratorio vamos a crear una aplicación con la que vamos a poder listar y crear noticias. Para la parte de la persistencia de los datos vamos a utilizar la dependencia de **json-server**.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-http-client-lab  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? no
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, crearemos un servicio, un componente, y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-http-client-lab  
$ ng g s noticias  
$ ng g c noticia  
$ ng s
```

Vamos a empezar con la configuración de **json-server** que es una herramienta que nos permite crear una API Rest sin configuración ninguna en nada de tiempo.

Lo primero será instalarla dentro de la carpeta del proyecto.

```
$ npm i json-server
```

Una vez instalada la dependencia, vamos a crear el archivo que hará de BBDD. Este archivo tiene que ser un archivo JSON, y lo vamos a crear en la raíz del proyecto. El nombre que le vamos a dar es **db.json**.

Dentro de este archivo tenemos que añadir un objeto con una clave por cada tipo de recurso con el que vayamos a interactuar. En nuestro caso solo queremos tener como recurso **noticias** por lo que esta será la clave.

/angular-http-client-lab/db.json

```
{  
  "noticias": [  
    { "id": 1, "titulo": "El niño murciélagos reaparece estas vacaciones en la playa", "contenido": "Lorem ipsum dolor sit amet consectetur adipisicing elit. Adipisci maxime rerum quod accusantium et exercitationem, sed ut necessitatibus quam nostrum atque facilis minus repellat quisquam? Aliquam quidem aut unde nostrum." }  
  ]  
}
```

Cuando realicemos peticiones HTTP sobre el **json-server**, este se encargará de obtener los datos de este archivo JSON, y también de crear nuevos datos, actualizarlos y eliminarlos.

Para terminar, vamos a añadir un script de NPM, **api-json**, para levantar el servidor del **json-server** con la API que genera, dentro del archivo **package.json**.

/angular-http-httpclient-lab/package.json

```
{  
  "name": "angular-http-httpclient-lab",  
  "version": "0.0.0",  
  "scripts": {  
    "ng": "ng",  
    "start": "ng serve",  
    "build": "ng build",  
    "watch": "ng build --watch --configuration development",  
    "test": "ng test",  
    "api-json": "json-server --watch db.json"  
},  
  "private": true,  
  "dependencies": { ... },  
  "devDependencies": { ... }  
}
```

Y ahora vamos a levantar esta api con el siguiente comando:

```
$ npm run api-json  
  
\{^_^\}/ hi!  
  
Loading db.json  

```

Como podemos ver, la API está en el puerto 3000, y si entramos en <http://localhost:3000/noticias> deberíamos de ver los datos que hay en nuestra BBDD JSON.

Ya podemos empezar con el desarrollo de la aplicación.

El primer paso va a ser añadir el provider del HTTP Client en la configuración de la aplicación.

/angular-http-httpclient-lab/src/app/app.config.ts

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';  
import { provideRouter } from '@angular/router';  
  
import { routes } from './app.routes';
```

```
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient()
  ]
};
```

Ahora nos vamos a ir al servicio de noticias en el que vamos a añadir dos métodos, uno para obtener las noticias, y el otro para crear nuevas noticias.

También vamos a dejar creada una interface **INoticia** para describir como son estos objetos, y la vamos a exportar para utilizarla en otros archivos de la aplicación.

/angular-http-httpclient-lab/src/app/noticias.service.ts

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

export interface INoticia {
  id: number,
  titulo: string,
  contenido: string
}

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

  constructor() { }

  getNoticias() {

  }

  createNoticia() {

  }
}
```

Ahora vamos a inyectar el servicio HttpClient dentro del constructor de nuestro servicio de noticias.

/angular-http-httpclient-lab/src/app/noticias.service.ts

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

export interface INoticia {
```

```

    id: number,
    titulo: string,
    contenido: string
}

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

  constructor(private http: HttpClient) { }

  getNoticias() {

  }

  createNoticia() {

}
}

```

Vamos a empezar rellenando el método de **getNoticias**, y en este caso vamos a llamar al método **http.get** pasandole como parámetro la url de noticias que nos ha dado el **json-server**.

/angular-http-httpclient-lab/src/app/noticias.service.ts

```

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

export interface INoticia {
  id: number,
  titulo: string,
  contenido: string
}

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

  constructor(private http: HttpClient) { }

  getNoticias() {
    return this.http.get('http://localhost:3000/noticias')
  }

  createNoticia() {

}
}

```

Para mejorar nuestro código vamos a añadirle el tipo de datos que vamos a obtener con el `get` pasándole `Array<INoticia>` como valor del tipo genérico de este método.

Al mismo tiempo vamos a ponerle el valor de retorno del método `getNoticias`, que será lo que devuelve el `get` envuelto por un **Observable**.

/angular-http-httpclient-lab/src/app/noticias.service.ts

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

export interface INoticia {
  id: number,
  titulo: string,
  contenido: string
}

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

  constructor(private http: HttpClient) { }

  getNoticias(): Observable<Array<INoticia>> {
    return this.http.get<Array<INoticia>>('http://localhost:3000/noticias')
  }

  createNoticia() {

  }
}
```

Como ya tenemos este primer método, vamos a ir a llenar nuestros componentes para ver si se muestran las noticias en la aplicación.

Lo primero va a ser estructurar el componente de **noticia** con el que mostraremos los datos de una noticia.

Dentro del TypeScript vamos a declarar la propiedad **noticia** que recibiremos desde el exterior, por lo que vamos a tener que añadirle el decorador `@Input()`.

/angular-http-httpclient-lab/src/app/noticia/noticia.component.ts

```
import { Component, Input, OnInit } from '@angular/core';
import { INoticia } from '../noticias.service';

@Component({
  selector: 'app-noticia',
  standalone: true,
```

```

imports: [],
templateUrl: './noticia.component.html',
styleUrl: './noticia.component.css'
})
export class NoticiaComponent implements OnInit {
  @Input() noticia: INoticia = { id: 0, titulo: '', contenido: ''};

  constructor() { }

  ngOnInit(): void {
  }
}

```

Y ahora en la plantilla, vamos a pintar el título en una etiqueta **h3** y el contenido dentro de un **p**.

/angular-http-httpclient-lab/src/app/noticia/noticia.component.html

```

<h3>{{noticia.titulo}}</h3>
<p>{{noticia.contenido}}</p>

```

Con esto ya tenemos nuestro componente noticia, ahora vamos a ir al componente **App**, en el que vamos a pedir las noticias a nuestra API para pintarlas con este componente.

Lo primero que vamos a hacer es importar el componente de Noticia y crear una propiedad **listaNoticias** que será inicialmente un array vacío. También vamos a inyectar el servicio de noticias en el constructor, y dejaremos ya puesto el método del ciclo de vida **ngOnInit**.

/angular-http-httpclient-lab/src/app/app.component.ts

```

import { Component, OnInit } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { NoticiaComponent } from './noticia/noticia.component';
import { INoticia, NoticiasService } from './noticias.service';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, NoticiaComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  listaNoticias: Array<INoticia> = []

  constructor(
    private noticias: NoticiasService
  ) {}
}

```

```
ngOnInit() {  
}  
}  
}
```

Dentro del **ngOnInit** vamos a llamar a la función **getNoticias** del servicio, y nos vamos a suscribir al observable, para que cuando obtengamos la respuesta de la petición, podamos coger los datos y guardarlos en la propiedad **listaNoticias**.

/angular-http-httpclient-lab/src/app/app.component.ts

```
import { Component, OnInit } from '@angular/core';  
import { RouterOutlet } from '@angular/router';  
import { NoticiaComponent } from './noticia/noticia.component';  
import { INoticia, NoticiasService } from './noticias.service';  
  
@Component({  
  selector: 'app-root',  
  standalone: true,  
  imports: [RouterOutlet, NoticiaComponent],  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css'  
})  
export class AppComponent implements OnInit {  
  listaNoticias: Array<INoticia> = []  
  
  constructor(  
    private noticias: NoticiasService  
  ) {}  
  
  ngOnInit() {  
    this.noticias.getNoticias()  
      .subscribe((noticias: Array<INoticia>) => {  
        this.listaNoticias = noticias  
      })  
  }  
}
```

Con esto ya tenemos las noticias en dicho array, por lo que ahora toca mostrarlas en la plantilla del componente App, y para ello vamos a iterar con un **@for** este array sobre el componente **noticia**.

/angular-http-httpclient-lab/src/app/app.component.html

```
<h1>Noticias</h1>  
  
@for (noticia of listaNoticias; track $index) {  
  <app-noticia [noticia]="noticia"></app-noticia>  
}
```

Con esto ya deberíamos de poder ver la noticia que habíamos guardado en <http://localhost:4200/>.

Pues ya tenemos la parte de listar los datos, ahora nos toca ponernos con la de guardar nuevas noticias.

Vamos a empezar por añadir un pequeño formulario en el propio componente App. Importaremos el módulo de **ReactiveFormsModule** en las importaciones del componente.

/angular-http-httpclient-lab/src/app/app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { NoticiaComponent } from './noticia/noticia.component';
import { INoticia, NoticiasService } from './noticias.service';
import { ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, NoticiaComponent, ReactiveFormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  listaNoticias: Array<INoticia> = []

  constructor(
    private noticias: NoticiasService
  ) {}

  ngOnInit() {
    this.noticias.getNoticias()
      .subscribe((noticias: Array<INoticia>) => {
        this.listaNoticias = noticias
      })
  }
}
```

Ahora vamos a crear el formulario que va a tener dos campos, el título y el contenido.

/angular-http-httpclient-lab/src/app/app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { NoticiaComponent } from './noticia/noticia.component';
import { INoticia, NoticiasService } from './noticias.service';
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
```

```

imports: [RouterOutlet, NoticiaComponent, ReactiveFormsModule],
templateUrl: './app.component.html',
styleUrl: './app.component.css'
})
export class AppComponent implements OnInit {
  listaNoticias: Array<INoticia> = []
  formularioNoticia: FormGroup

  constructor(
    private noticias: NoticiasService
  ) {
    this.formularioNoticia = new FormGroup({
      titulo: new FormControl(''),
      contenido: new FormControl('')
    })
  }

  ngOnInit() {
    this.noticias.getNoticias()
      .subscribe((noticias: Array<INoticia>) => {
        this.listaNoticias = noticias
      })
  }
}

```

Antes de pasarnos al HTML, vamos a crear la función de **guardarDatos** desde la que vamos a recoger los datos del formulario y los vamos a enviar a nuestra API llamando al método **createNoticia** del servicio de noticias (que implementaremos más adelante).

/angular-http-httpclient-lab/src/app/app.component.ts

```

import { Component, OnInit } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { NoticiaComponent } from './noticia/noticia.component';
import { INoticia, NoticiasService } from './noticias.service';
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, NoticiaComponent, ReactiveFormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  listaNoticias: Array<INoticia> = []
  formularioNoticia: FormGroup

  constructor(
    private noticias: NoticiasService
  ) {
    this.noticias.getNoticias()
      .subscribe((noticias: Array<INoticia>) => {
        this.listaNoticias = noticias
      })
  }

  ngOnInit() {
    this.noticias.getNoticias()
      .subscribe((noticias: Array<INoticia>) => {
        this.listaNoticias = noticias
      })
  }
}

```

```

) {
  this.formularioNoticia = new FormGroup({
    titulo: new FormControl(''),
    contenido: new FormControl(''),
  })
}

ngOnInit() {
  this.noticias.getNoticias()
    .subscribe(noticias: Array<INoticia>) => {
      this.listaNoticias = noticias
    }
}

guardarNoticia() {
  const datosNoticia = this.formularioNoticia.value;
  this.noticias.createNoticia(datosNoticia)
}
}

```

En el HTML del componente App, vamos a crear la estructura del formulario. Pondremos dos campos de texto con sus respectivas etiquetas, y un botón de tipo **submit**.

/angular-http-httpclient-lab/src/app/app.component.html

```

<h1>Noticias</h1>

<form>
  <div>
    <label for="titulo">Título:</label>
    <input type="text" id="titulo">
  </div>
  <div>
    <label for="contenido">Contenido:</label>
    <input type="text" id="contenido">
  </div>
  <button type="submit">Guardar</button>
</form>

@for (noticia of listaNoticias; track $index) {
  <app-noticia [noticia]="noticia"></app-noticia>
}

```

Ahora en la etiqueta **form** vamos a añadirle el evento **ngSubmit** para que cuando se pulse el botón se ejecute la función de **guardarNoticia** que tenemos en el TypeScript.

También vamos a relacionar este formulario con la propiedad **formularioNoticia** que hemos creado en el TypeScript y en la que hemos definido la estructura de los datos. Para esto tenemos que asignarle esta propiedad a la directiva **FormGroup**.

/angular-http-httpclient-lab/src/app/app.component.html

```
<h1>Noticias</h1>

<form (ngSubmit)="guardarNoticia()" [formGroup]="formularioNoticia">
  <div>
    <label for="titulo">Título:</label>
    <input type="text" id="titulo">
  </div>
  <div>
    <label for="contenido">Contenido:</label>
    <input type="text" id="contenido">
  </div>
  <button type="submit">Guardar</button>
</form>

@for (noticia of listaNoticias; track $index) {
  <app-noticia [noticia]="noticia"></app-noticia>
}
```

Ya solo nos queda indicar en los inputs a que claves del **formularioNoticia** van a ir los valores que escribamos en ellos. Para ello vamos a añadirles la directiva **FormControlName** con el valor de las claves.

/angular-http-httpclient-lab/src/app/app.component.html

```
<h1>Noticias</h1>

<form (ngSubmit)="guardarNoticia()" [formGroup]="formularioNoticia">
  <div>
    <label for="titulo">Título:</label>
    <input type="text" id="titulo" formControlName="titulo">
  </div>
  <div>
    <label for="contenido">Contenido:</label>
    <input type="text" id="contenido" formControlName="contenido">
  </div>
  <button type="submit">Guardar</button>
</form>

@for (noticia of listaNoticias; track $index) {
  <app-noticia [noticia]="noticia"></app-noticia>
}
```

Con esto ya tenemos la plantilla, ahora nos vamos a ir al servicio, y lo primero que tenemos que hacer es añadir el parámetro **noticia** en el método de **createNoticia**. Como el valor que enviamos desde el HTML no lleva el identificador, vamos a indicar en la interface **INoticia** que este es opcional poniéndole después del nombre un ?.

/angular-http-httpclient-lab/src/app/noticias.service.ts

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

export interface INoticia {
  id: number,
  titulo: string,
  contenido: string
}

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

  constructor(private http: HttpClient) { }

  getNoticias(): Observable<Array<INoticia>> {
    return this.http.get<Array<INoticia>>('http://localhost:3000/noticias')
  }

  createNoticia(noticia: INoticia) {

  }
}
```

Ahora dentro del método vamos a llamar a la petición **post** pasándole la URL de las noticias y como segundo parámetro la noticia que estamos recibiendo como parámetro.

/angular-http-httpclient-lab/src/app/noticias.service.ts

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

export interface INoticia {
  id: number,
  titulo: string,
  contenido: string
}

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

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

```

getNoticias(): Observable<Array<INoticia>> {
  return this.http.get<Array<INoticia>>('http://localhost:3000/noticias')
}

createNoticia(noticia: INoticia): Observable<INoticia> {
  return this.http.post<INoticia>('http://localhost:3000/noticias', noticia)
}

```

Ya tenemos la petición, si probamos a rellenar desde el navegador los campos del formulario y le damos al botón de **Guardar** no ocurre nada. Como comentamos en el tema de los observables, si no hay ninguna suscripción al observable, este no se ejecuta, por lo que en este caso, no se está realizando la petición POST porque no nos hemos suscrito al observable que retorna el método **createNoticia**.

Nos vamos al componente App, donde hemos llamado a este método y vamos a añadir el **subscribe**.

/angular-http-httpclient-lab/src/app/app.component.ts

```

import { Component, OnInit } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { NoticiaComponent } from './noticia/noticia.component';
import { INoticia, NoticiasService } from './noticias.service';
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, NoticiaComponent, ReactiveFormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  listaNoticias: Array<INoticia> = []
  formularioNoticia: FormGroup

  constructor(
    private noticias: NoticiasService
  ) {
    this.formularioNoticia = new FormGroup({
      titulo: new FormControl(''),
      contenido: new FormControl('')
    })
  }

  ngOnInit() {
    this.noticias.getNoticias()
      .subscribe((noticias: Array<INoticia>) => {
        this.listaNoticias = noticias
      })
  }
}

```

```

    }

    guardarNoticia() {
        const datosNoticia = this.formularioNoticia.value;
        this.noticias.createNoticia(datosNoticia)
            .subscribe(() => {

            })
    }
}

```

En esta suscripción vamos a recibir la noticia que se ha guardado en la BBDD, por lo que podemos aprovechar estos datos para añadirlos en la lista de noticias y que se actualice la vista.

/angular-http-httpclient-lab/src/app/app.component.ts

```

import { Component, OnInit } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { NoticiaComponent } from './noticia/noticia.component';
import { INoticia, NoticiasService } from './noticias.service';
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';

@Component({
    selector: 'app-root',
    standalone: true,
    imports: [RouterOutlet, NoticiaComponent, ReactiveFormsModule],
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
    listaNoticias: Array<INoticia> = []
    formularioNoticia: FormGroup

    constructor(
        private noticias: NoticiasService
    ) {
        this.formularioNoticia = new FormGroup({
            titulo: new FormControl(''),
            contenido: new FormControl('')
        })
    }

    ngOnInit() {
        this.noticias.getNoticias()
            .subscribe((noticias: Array<INoticia>) => {
                this.listaNoticias = noticias
            })
    }

    guardarNoticia() {

```

```
const datosNoticia = this.formularioNoticia.value;
this.noticias.createNoticia(datosNoticia)
    .subscribe((noticiaGuardada: INoticia) => {
        this.listaNoticias.push(noticiaGuardada)
    })
}
```

Ahora si que deberíamos de poder guardar noticias y al hacerlo deberían de aparecer automáticamente en la vista.

Chapter 19. Angular Router

El routing es lo que nos va a permitir cambiar entre las distintas páginas de nuestra aplicación.

Pero recordad que estamos ante aplicaciones de una sola página (SPAs), es decir, aplicaciones que solo tienen una única página de HTML, por lo que no tiene sentido cambiar de página cada vez que cambiamos de URL.

En este caso, el routing funciona un poco diferente a como se hacía con las MPAs (Multi-Page Applications). Ahora lo que se cambia es el contenido a mostrar por la página **index.html**, que serán nuestros componentes.

Angular ya viene con su propio módulo de Routing, el **RouterModule**, que se importa desde **@angular/router**, y dentro de él tenemos una serie de elementos implementados que usaremos para gestionar el routing de la aplicación.

19.1. Definición de rutas

Para definir las rutas de nuestra aplicación hay que crear un array del tipo **Routes**, donde cada una de las rutas serán objetos de tipo **Route**.

Podemos definir las rutas con las siguientes propiedades:

- **path**: el path con el que tiene que coincidir la url.
- **component**: el componente a renderizar cuando el path coincide con la URL a la que hemos navegado.
- **children**: un array con las rutas hijas o rutas anidadas de la ruta a la que se añaden.
- **loadChildren**: una función que permite cargar las rutas de un módulo. Normalmente se utilizar para hacer el lazy loading.
- **canActivate**: un array con las guardias del tipo canActivate que hay que aplicar sobre esta ruta.
- **canDeactivate**: un array con las guardias del tipo canDeactivate que hay que aplicar sobre esta ruta.
- **redirectTo**: string con el path de la ruta a la que queremos hacer una redirección.

Una vez definidas las rutas, hay que configurar el **RouterModule** para que se pueda utilizar en la aplicación. Para ello, solo hay que pasarle el array de rutas al método **forRoot** del **RouterModule** e importarlo en el **app.module.ts**.

19.2. Componente router-outlet

Cuando se configuran las rutas y ya podemos cambiar entre ellas en la aplicación, Angular no sabe en que parte de la página tiene que pintar los componentes asociados a ellas, por lo que tenemos que indicarselo nosotros de alguna forma.

Para decirle el sitio exacto sobre el que pintar estos componentes hay que utilizar el componente **router-outlet** que nos proporcionan.

19.3. Directiva routerLink

Los enlaces de HTML (etiquetas **a**) hacen que se refresque la página cuando queremos cambiar de una ruta a otra, haciendo que la aplicación se vuelva a descargar y ejecutar desde 0, perdiendo el estado que no se hubiese guardado.

Para evitar esto, Angular nos proporciona la directiva **routerLink** que vamos a poner en el enlace para sustituir al atributo **href**. Esta directiva puede recibir dos tipos de valores:

- Un array de segmentos de ruta, por ejemplo: `[routerLink]="/productos", 1, 'opiniones'"`
- Un string con la ruta, por ejemplo: `routerLink="/inicio"`

Lo que hace la directiva es evitar que se cargue la página desde 0, y solo cambia la URL de la barra de direcciones, pinta el componente asociado a la nueva ruta y la añade en el historial de navegación.

19.4. Navegación por código

Ya hemos hablado de como cambiar de rutas con enlaces, pero hay veces en las que queremos cambiar de componente una vez se ha realizado una acción, como por ejemplo cuando guardamos los datos de un formulario. Esta navegación la haríamos a través de código.

Para realizar la navegación por código, hay que inyectar la instancia de **Router** en el constructor del componente en el que la vayamos a realizar.

Este servicio nos permite navegar entre rutas, y para ello utilizaremos cualquiera de los siguientes métodos:

- **navigate**: le pasamos como parámetro un array de segmentos para formar la ruta a la que queremos cambiar.
- **navigateByUrl**: le pasamos como parámetro un string con la ruta a la que queremos cambiar.

19.5. Ruta comodín

El router de Angular nos permite utilizar una ruta que siempre se va a ejecutar. Ojo, siempre que no se haya ejecutado otra antes, por lo que esta ruta tiene que definirse siempre la última en el array de **Routes**.

El valor del path para esta ruta es un doble asterisco *, y lo podemos utilizar con la propiedad ***component** y **redirectTo**.

Un ejemplo donde usar este tipo de ruta es cuando entramos a rutas que no existen. Con ella podemos mostrar un componente de error, o hacer una redirección a otra ruta existente de nuestra aplicación.

19.6. Rutas con parámetros

Cuando tenemos listados de datos, es muy probable que necesitemos ver más en detalle la información de cada elemento en la lista, y aquí es donde entran las rutas con parámetros.

Estas rutas tienen una parte dinámica que va a cambiar dependiendo de los datos que queremos mostrar, donde normalmente se usa el identificador del dato.

Para indicar que una ruta tiene un valor dinámico o parámetro vamos a utilizar : delante del nombre del parámetro en el path.

Por ejemplo, con la ruta **productos/:id** podríamos acceder a rutas como:

- /productos/2
- /productos/9120
- /productos/-M73ljd39udHS92_J0ss2
- ...

Luego dentro de los componentes que se van a pintar necesitaremos acceder a dichos parámetros para poder pedir la información buscada y asociada a estos identificadores.

Para obtener estos valores, hay que injectar la instancia de **ActivatedRoute** en el constructor de nuestro componente. Y desde esta instancia podemos acceder a **params** y **paramMap**, que son **observables** que nos van a dar estos parámetros de la ruta al entrar por primera vez y siempre que alguno de los parámetros cambie.

19.7. Rutas con query params

Otro tipo de parámetros con los que nos podemos encontrar en las rutas de la aplicación son los parámetros de consulta, o **query params**, que son aquellos que van detrás del ? al final de la URL.

Para acceder a estos parámetros de consulta, lo haremos igual que con los parámetros normales, es decir, usando el **ActivatedRoute**, pero esta vez suscribiéndonos a **queryParams** o **queryParamMap**.

19.8. Rutas anidadas

Por ahora hemos hablado de rutas que se encuentran al mismo nivel, pero en algunos casos vamos a necesitar mostrar componentes que comparten parte de la misma url, y en estos casos es cuando podremos utilizar las rutas anidadas.

Por ejemplo, podríamos usarlo con un listado de datos para mostrar al mismo tiempo tanto la lista como la información de alguno de estos datos:

- /usuarios: esta mostrará la lista de usuarios.
- /usuarios/2: esta mostrará la información del usuario con id 2 al mismo tiempo que se sigue mostrando la lista de usuarios.

Otro caso podría ser en el que tenemos un listado de series/películas y podemos filtrarlas por género:

- /categorias: esta mostrará la lista de las categorías.
- /categorias/comedia: esta mostrará las series/películas de la categoría comedia al mismo tiempo que la lista de categorías.
- /categorias/comedia/-M1s1hXmn6_UopYgiIy6: esta mostrará la información de la serie/película que tiene el id -M1s1hXmn6_UopYgiIy6 y pertenece a la categoría comedia, al mismo tiempo que la lista de categorías y la lista de series/películas que son comedias.

Al tener que anidar estas rutas, los componentes asociados a ellas estarán anidados, por lo que tendremos que utilizar el componente **router-outlet** por cada nivel de anidamiento para indicar donde hay que mostrar estos componentes.

19.9. Guards

Las **guards** nos permiten controlar el acceso o salida a una ruta. Es decir, que cuando vayamos a cambiar de una ruta a otra, se comprobará primero si una condición dada se cumple o no se cumple, haciendo que podamos cambiar de ruta o nos quedemos en la que nos encontramos.

Estas **guards** son funciones y las tenemos de varios tipos:

- `CanActivate`: se ejecuta antes de entrar a una ruta.
- `CanDeactivate`: se ejecuta al salir de una ruta.

Estas funciones tendrán que devolver un booleano:

- **true**: se permite entrar en la nueva ruta o salir de la actual.
- **false**: nos quedamos en la ruta actual.

Una vez tenemos las **guards** implementadas, hay que añadirlas sobre las rutas donde queremos realizar las comprobaciones, usando las propiedades de `canActivate` y `canDeactivate*` del objeto que representa la ruta.

Estos elementos van a permitir que podamos comprobar condiciones como:

- Para ver esta página necesitas estar logueado.
- Para ver esta página necesitas tener el rol de Admin.
- Para salir de este formulario tienes que guardar primero los datos.
- ...

19.10. Lab: Angular Router

En este laboratorio vamos a ver como utilizar el router de Angular para crear una aplicación.

Para empezar crearemos un proyecto de Angular con el siguiente comando:

```
$ ng new angular-router-lab  
? Would you like to add Angular routing? N  
? Which stylesheet format would you like to use? CSS
```

Una vez creado el proyecto, vamos a entrar en la carpeta que se ha creado, vamos a crear una serie de elementos que necesitaremos y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-router-lab  
$ ng g c usuarios  
$ ng g c nuevo-usuario  
$ ng g c info-usuario  
$ ng g c editar-usuario  
$ ng g c error404  
$ ng g s usuarios  
$ ng g g info-usuario/info-usuario --implements CanActivate  
$ ng g g editar-usuario/editar-usuario --implements CanDeactivate  
$ ng s
```

Vamos a empezar configurando el módulo del Router de Angular con dos rutas iniciales. Para ello, vamos a crear un archivo **app.routes.ts** en la carpeta **src/app**.

Dentro de este archivo vamos a empezar creando un array con las siguientes dos rutas:

- ruta '' para pintar UsuariosComponent
- ruta 'nuevo-usuario' para pintar NuevoUsuarioComponent

/angular-router-lab/src/app/app.routes.ts

```
import { Routes } from "@angular/router";
import { NuevoUsuarioComponent } from "./nuevo-usuario/nuevo-usuario.component";
import { UsuariosComponent } from "./usuarios/usuarios.component";

const APP_ROUTES: Routes = [
  { path: '', component: UsuariosComponent },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
]
```

Con esto ya tenemos el routing configurado ya que en el archivo de configuración de la aplicación ya se están proveyendo por defecto las rutas.

/angular-router-lab/src/app/app.config.ts

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';

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

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes)
  ]
};
```

Ya que estamos con el archivo de configuración de la aplicación, vamos a aprovechar a proveer el HTTP Client que usaremos más adelante en el servicio que ya hemos creado.

/angular-router-lab/src/app/app.config.ts

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient()
  ]
};
```

Ya tenemos configurado el routing de la aplicación, ahora vamos a añadir un par de enlaces en el componente App para poder cambiar entre las dos rutas que hemos añadido.

Hay que recordar que para que la página no se refresque al cambiar de una ruta a la otra, tenemos que utilizar la directiva **routerLink** y pasarle un array con los segmentos que forman la ruta a la que queremos navegar.

Además, también tenemos que importar esta directiva en las importaciones del componente standalone.

/angular-router-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { RouterLink, RouterOutlet } from '@angular/router';

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

```

standalone: true,
imports: [RouterOutlet, RouterLink],
templateUrl: './app.component.html',
styleUrl: './app.component.css'
})
export class AppComponent {
}

```

/angular-router-lab/src/app/app.component.html

```

<header>
  <ul>
    <li><a [routerLink]="/">Usuarios</a></li>
    <li><a [routerLink]="/nuevo-usuario">Nuevo usuario</a></li>
  </ul>
</header>

```

Pulsando estos dos enlaces ya podemos navegar entre las dos rutas, pero ahora no vemos los componentes que le hemos indicado que se tienen que mostrar cuando navegamos a dichas rutas.

Esto se debe a que Angular no sabe donde tiene que mostrar estos componentes, por lo que se lo tendremos que indicar con el componente **router-outlet**.

/angular-router-lab/src/app/app.component.html

```

<header>
  <ul>
    <li><a [routerLink]="/">Usuarios</a></li>
    <li><a [routerLink]="/nuevo-usuario">Nuevo usuario</a></li>
  </ul>
</header>

<router-outlet></router-outlet>

```

Ahora ya deberíamos de poder navegar entre esas dos rutas y ver los dos componentes que le habíamos indicado.

El siguiente paso es ver como navegar entre estas rutas, pero ahora sin utilizar los enlaces, sino haciéndolo mediante programación.

En el componente de **NuevoUsuario** vamos a añadir un botón de **guardar** como si este botón fuese a guardar unos datos de un formulario (que no vamos a poner).

/angular-router-lab/src/app/nuevo-usuario/nuevo-usuario.component.html

```

<h2>Nuevo usuario</h2>

<button type="button" (click)="guardar()">Guardar</button>

```

Para poder cambiar de ruta mediante código, vamos a necesitar la instancia de **Router** de Angular, por lo que tenemos que inyectar esta dependencia en el constructor del componente.

/angular-router-lab/src/app/nuevo-usuario/nuevo-usuario.component.ts

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

@Component({
  selector: 'app-nuevo-usuario',
  standalone: true,
  imports: [],
  templateUrl: './nuevo-usuario.component.html',
  styleUrls: ['./nuevo-usuario.component.css']
})
export class NuevoUsuarioComponent implements OnInit {

  constructor(private router: Router) { }

  ngOnInit(): void {
  }

  guardar(): void {
  }

}
```

Ahora vamos a simular dentro del método **guardar** que se hace una petición para guardar los datos del nuevo usuario y que tras pasar un segundo y medio estos ya se han guardado y hacemos la navegación al inicio.

Para navegar al inicio vamos a utilizar el método **navigate** de la instancia del **Router**.

/angular-router-lab/src/app/nuevo-usuario/nuevo-usuario.component.ts

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

@Component({
  selector: 'app-nuevo-usuario',
  standalone: true,
  imports: [],
  templateUrl: './nuevo-usuario.component.html',
  styleUrls: ['./nuevo-usuario.component.css']
})
export class NuevoUsuarioComponent implements OnInit {

  constructor(private router: Router) { }

  ngOnInit(): void {
  }

  guardar(): void {
  }

}
```

```

ngOnInit(): void {
}

guardar(): void {
  console.log('guardando el usuario...')
  setTimeout(() => {
    console.log('usuario guardado')
    this.router.navigate([''])
  }, 1500)
}

}

```

El siguiente paso es mostrar una lista de usuarios en el componente de **Usuarios**, por lo que vamos a añadir en el servicio de **Usuarios** un método que nos devuelva un array con los usuarios de la API de <http://jsonplaceholder.typicode.com/>.

/angular-router-lab/src/app/usuarios.service.ts

```

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

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

  constructor(private http: HttpClient) { }

  getUsuarios(): Observable<any> {
    return this.http.get('http://jsonplaceholder.typicode.com/users')
  }
}

```

Dentro del componente de **Usuarios** tenemos que inyectar la instancia de este servicio y pedir estos usuarios en el método **ngOnInit**.

/angular-router-lab/src/app/usuarios/usuarios.component.ts

```

import { Component, OnInit } from '@angular/core';
import { UsuariosService } from '../usuarios.service';

@Component({
  selector: 'app-usuarios',
  standalone: true,
  imports: [],
  templateUrl: './usuarios.component.html',
  styleUrls: ['./usuarios.component.css']
})

```

```

export class UsuariosComponent implements OnInit {

  constructor(private usuariosService: UsuariosService) { }

  ngOnInit(): void {
    this.usuariosService.getUsuarios()
  }
}

```

Como el método **getUsuarios** devuelve un observable, nos vamos a suscribir a él, para obtener los usuarios como parámetro y guardarlos en una propiedad **listaUsuarios** que vamos a inicializar a un array vacío hasta que los obtengamos de la API.

/angular-router-lab/src/app/usuarios/usuarios.component.ts

```

import { Component, OnInit } from '@angular/core';
import { UsuariosService } from '../usuarios.service';

@Component({
  selector: 'app-usuarios',
  standalone: true,
  imports: [],
  templateUrl: './usuarios.component.html',
  styleUrls: ['./usuarios.component.css']
})
export class UsuariosComponent implements OnInit {
  listaUsuarios: Array<any> = []

  constructor(private usuariosService: UsuariosService) { }

  ngOnInit(): void {
    this.usuariosService.getUsuarios()
      .subscribe(usuarios => {
        this.listaUsuarios = usuarios
      })
  }
}

```

Ahora vamos a ir a la plantilla del componente para mostrar una lista con los nombres de estos usuarios.

/angular-router-lab/src/app/usuarios/usuarios.component.html

```

<h2>Usuarios</h2>

<ul>
  @for (usuario of listaUsuarios; track $index) {
    <li>
      {{usuario.name}}
    </li>
  }

```

```
    }
  </ul>
```

Ya tenemos la lista, ahora vamos a añadir junto a los nombres dos enlaces para navegar hasta dos rutas nuevas que añadiremos después. Estas rutas son:

- /usuarios/<id_usuario>/info
- /usuarios/<id_usuario>/editar

También hay que importar **RouterLink** en el componente.

/angular-router-lab/src/app/usuarios/usuarios.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UsuariosService } from '../usuarios.service';
import { RouterLink } from '@angular/router';

@Component({
  selector: 'app-usuarios',
  standalone: true,
  imports: [RouterLink],
  templateUrl: './usuarios.component.html',
  styleUrls: ['./usuarios.component.css']
})
export class UsuariosComponent implements OnInit {
  listaUsuarios: Array<any> = []

  constructor(private usuariosService: UsuariosService) { }

  ngOnInit(): void {
    this.usuariosService.getUsuarios()
      .subscribe(usuarios => {
        this.listaUsuarios = usuarios
      })
  }
}
```

/angular-router-lab/src/app/usuarios/usuarios.component.html

```
<h2>Usuarios</h2>

<ul>
  @for (usuario of listaUsuarios; track $index) {
    <li>
      {{usuario.name}}
      <a [routerLink]="/usuarios", usuario.id, 'info'">Ver +info</a>
      <a [routerLink]="/usuarios", usuario.id, 'editar'">Editar</a>
    </li>
  }
}
```

```
</ul>
```

Como podemos ver, hemos puesto el inicio de la ruta como **/usuarios**, pero nuestra ruta inicial es **/**. Lo que vamos a ver a continuación es como anidar rutas y añadirles parámetros a estas, pero justo antes de hacer esto, añadir una nueva ruta para redireccionar del **/** al **/usuarios** la cual será nuestra ruta inicial a partir de ahora.

Dentro del archivo de **app.routes.ts**, vamos a crear una nueva ruta en la que vamos a utilizar la función **redirectTo** en lugar de **component** para indicar a que otra ruta queremos redireccionar.

Como estamos redireccionando la ruta inicial, Angular nos va a mostrar un error indicando que cuando queremos hacer esto tenemos que añadirle una propiedad **pathMatch** con el valor de **full**.

También vamos a cambiar el path de la ruta inicial y le pondremos como valor **usuarios**.

```
/angular-router-lab/src/app/app.routes.ts
```

```
import { Routes } from '@angular/router';
import { UsuariosComponent } from './usuarios/usuarios.component';
import { NuevoUsuarioComponent } from './nuevo-usuario/nuevo-usuario.component';

export const routes: Routes = [
  { path: 'usuarios', component: UsuariosComponent },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: () => 'usuarios', pathMatch: 'full' }
];
```

Con esto ya tenemos el primer paso, conseguir que la ruta inicial sea **/usuarios**. Ahora ya podemos añadir las rutas anidadas a esta inicial.

Con las rutas anidadas lo que vamos a conseguir es que al mismo tiempo que se muestran las páginas de ver más información y de editar un usuario, se siga mostrando la lista de usuarios.

Vamos a añadir estas dos nuevas rutas para las que hemos puesto los enlaces, pero esta vez lo vamos a hacer en otro array de rutas al que llamaremos **USUARIOS_ROUTES**.

Como son dos rutas anidadas, no hay que ponerle el inicio del path que coincide con **usuarios**, ya que al estar anidadas, este path se va a concatenar sobre el que ya se ha definido en el de las rutas principales.

Además, como hemos comentado, estas rutas tienen un valor que va a cambiar dependiendo del usuario sobre el que pulsemos, el **id**, por lo que tenemos que añadirlo como un parámetro de la ruta. Para esto solo tenemos que darle el nombre que queremos y ponerle justo delante **:**.

```
/angular-router-lab/src/app/app.routes.ts
```

```
import { Routes } from '@angular/router';
import { UsuariosComponent } from './usuarios/usuarios.component';
import { NuevoUsuarioComponent } from './nuevo-usuario/nuevo-usuario.component';
import { InfoUsuarioComponent } from './info-usuario/info-usuario.component';
```

```

import { EditarUsuarioComponent } from './editar-usuario/editar-usuario.component';

const USUARIOS_ROUTES: Routes = [
  { path: ':id/info', component: InfoUsuarioComponent },
  { path: ':id/editar', component: EditarUsuarioComponent },
]

export const routes: Routes = [
  { path: 'usuarios', component: UsuariosComponent },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: () => 'usuarios', pathMatch: 'full' }
];

```

Solo nos queda indicar que este array de rutas son las rutas anidadas de la de **usuarios**. Tenemos que añadir la propiedad **children** y asignarle como valor el array de **USUARIOS_ROUTES** que hemos creado.

/angular-router-lab/src/app/app.routes.ts

```

import { Routes } from '@angular/router';
import { UsuariosComponent } from './usuarios/usuarios.component';
import { NuevoUsuarioComponent } from './nuevo-usuario/nuevo-usuario.component';
import { InfoUsuarioComponent } from './info-usuario/info-usuario.component';
import { EditarUsuarioComponent } from './editar-usuario/editar-usuario.component';

const USUARIOS_ROUTES: Routes = [
  { path: ':id/info', component: InfoUsuarioComponent },
  { path: ':id/editar', component: EditarUsuarioComponent },
]

export const routes: Routes = [
  { path: 'usuarios', component: UsuariosComponent, children: USUARIOS_ROUTES },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: () => 'usuarios', pathMatch: 'full' }
];

```

Ahora ya podemos pulsar sobre los distintos enlaces de **Ver +info** y **Editar** que aparecen al lado de los usuarios, y veremos que las rutas cambian, pero los componentes no se muestran.

Nos está pasando lo mismo que al principio, Angular no sabe en qué parte del componente **Usuarios** tiene que pintar los componentes de las rutas anidadas, por lo que tendremos que decírselo volviendo a poner la etiqueta **router-outlet**.

No olvidemos que hay que importar el componente en la clase también.

/angular-router-lab/src/app/usuarios/usuarios.component.ts

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

```

import { UsuariosService } from '../usuarios.service';
import { RouterLink, RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-usuarios',
  standalone: true,
  imports: [RouterLink, RouterOutlet],
  templateUrl: './usuarios.component.html',
  styleUrls: ['./usuarios.component.css']
})
export class UsuariosComponent implements OnInit {
  listaUsuarios: Array<any> = []

  constructor(private usuariosService: UsuariosService) { }

  ngOnInit(): void {
    this.usuariosService.getUsuarios()
      .subscribe(usuarios => {
        this.listaUsuarios = usuarios
      })
  }
}

```

/angular-router-lab/src/app/usuarios/usuarios.component.html

```

<h2>Usuarios</h2>

<ul>
  @for (usuario of listaUsuarios; track $index) {
    <li>
      {{usuario.name}}
      <a [routerLink]=["/usuarios", usuario.id, 'info']>Ver +info</a>
      <a [routerLink]=["/usuarios", usuario.id, 'editar']>Editar</a>
    </li>
  }
</ul>

<router-outlet></router-outlet>

```

Ya podemos ver los dos componentes cuando cambiamos entre las dos rutas anidadas.

Ahora vamos a ir al componente de **InfoUsuario** en el que queremos mostrar el resto de la información del usuario sobre el que se haya pulsado.

Vamos a añadir un nuevo método en el servicio de **Usuarios** para obtener la información de un usuario dado su identificador.

/angular-router-lab/src/app/usuarios.service.ts

```

import { HttpClient } from '@angular/common/http';

```

```

import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';

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

  constructor(private http: HttpClient) { }

  getUsuarios(): Observable<any> {
    return this.http.get('http://jsonplaceholder.typicode.com/users')
  }

  getUsuarioById(id: string): Observable<any> {
    return this.http.get(`http://jsonplaceholder.typicode.com/users/${id}`)
  }
}

```

Ahora solo necesitamos obtener el identificador del usuario de la ruta, llamar al método **getUsuariosById** que acabamos de crear y mostrar la información en la plantilla.

Para obtener el parámetro **id** de la ruta vamos a necesitar inyectar la instancia de **ActivatedRoute** en el constructor del componente. A través de la instancia vamos a suscribirnos a **paramMap** que es un observable que nos va a dar un Map con los parámetros de la ruta y cada vez que cambie alguno de ellos se volverá a ejecutar la función de la suscripción.

/angular-router-lab/src/app/info-usuario/info-usuario.component.ts

```

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

@Component({
  selector: 'app-info-usuario',
  standalone: true,
  imports: [],
  templateUrl: './info-usuario.component.html',
  styleUrls: ['./info-usuario.component.css']
})
export class InfoUsuarioComponent implements OnInit {
  id: string | null = ''
  constructor(private activatedRoute: ActivatedRoute) { }

  ngOnInit(): void {
    this.activatedRoute.paramMap.subscribe((params) => {
      this.id = params.get('id')
    })
  }
}

```

Una vez tenemos el identificador, ya podemos llamar a la nueva función del servicio para obtener los datos del usuario. Esta vez vamos a guardar el observable en una propiedad del componente, ya que a la hora de mostrar los datos vamos a utilizar el pipe **async**, y así evitamos tener que hacer nosotros la suscripción.

/angular-router-lab/src/app/info-usuario/info-usuario.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs';
import { UsuariosService } from '../usuarios.service';

@Component({
  selector: 'app-info-usuario',
  standalone: true,
  imports: [],
  templateUrl: './info-usuario.component.html',
  styleUrls: ['./info-usuario.component.css']
})
export class InfoUsuarioComponent implements OnInit {
  id: string | null = ''
  datosUsuario$: Observable<any> | null = null

  constructor(
    private activatedRoute: ActivatedRoute,
    private usuariosService: UsuariosService,
  ) { }

  ngOnInit(): void {
    this.activatedRoute.paramMap.subscribe((params) => {
      this.id = params.get('id')
      if (this.id) {
        this.datosUsuario$ = this.usuariosService.getUsuarioById(this.id)
      }
    })
  }
}
```

Ahora vamos a mostrar los datos en la plantilla, para ello, vamos a coger el observable de **datosUsuario\$** y vamos a aplicarle el pipe **async** para que se suscriba y obtenga los datos, y después aplicaremos el pipe **json** para que nos muestre estos datos en formato JSON.

También importaremos estos 2 pipes en el componente:

/angular-router-lab/src/app/info-usuario/info-usuario.component.ts

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

```

import { UsuariosService } from '../usuarios.service';
import { AsyncPipe, JsonPipe } from '@angular/common';

@Component({
  selector: 'app-info-usuario',
  standalone: true,
  imports: [JsonPipe, AsyncPipe],
  templateUrl: './info-usuario.component.html',
  styleUrls: ['./info-usuario.component.css']
})
export class InfoUsuarioComponent implements OnInit {
  id: string | null = ''
  datosUsuario$: Observable<any> | null = null

  constructor(
    private activatedRoute: ActivatedRoute,
    private usuariosService: UsuariosService,
  ) { }

  ngOnInit(): void {
    this.activatedRoute.paramMap.subscribe((params) => {
      this.id = params.get('id')
      if (this.id) {
        this.datosUsuario$ = this.usuariosService.getUsuarioById(this.id)
      }
    })
  }
}

```

/angular-router-lab/src/app/info-usuario/info-usuario.component.html

```

<h3>Usuario con id: {{id}}</h3>

<pre>{{ datosUsuario$ | async | json }}</pre>

```

Ya podemos ir viendo los datos de todos los usuarios según vamos pulsando sobre los distintos enlaces de **Ver +info**.

Ahora vamos a implementar la **guard** que creamos al principio para este componente. Con la **guard** lo que vamos a hacer es controlar si podemos entrar a ver la información de los usuarios o no.

Vamos a abrir el archivo de InfoUsuarioGuard, y en este caso, lo que vamos a hacer es pedir al usuario que confirme si quiere entrar a ver la información o no, utilizando la función **confirm()** de JavaScript.

Como es una guarda, si esta devuelve **true** entonces podemos entrar a ver la información, si devuelve **false** entonces no podemos entrar.



En un caso real podríamos comprobar si el usuario está logueado o si tiene un rol específico, en lugar de usar el **confirm**.

/angular-router-lab/src/app/info-usuario/info-usuario.guard.ts

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

export const infoUsuarioGuard: CanActivateFn = (route, state) => {
  return confirm('¿Quieres ver la información del usuario?');
};
```

Ahora para aplicarla sobre la ruta que muestra la información del usuario, tenemos que ir al archivo de rutas, y añadir sobre esta la propiedad **canActivate** y asignarle un array con las guardas de ese tipo que queremos que se ejecuten cuando se vaya a entrar en dicha ruta.

/angular-router-lab/src/app/app.routes.ts

```
import { Routes } from '@angular/router';
import { UsuariosComponent } from './usuarios/usuarios.component';
import { NuevoUsuarioComponent } from './nuevo-usuario/nuevo-usuario.component';
import { InfoUsuarioComponent } from './info-usuario/info-usuario.component';
import { EditarUsuarioComponent } from './editar-usuario/editar-usuario.component';
import { infoUsuarioGuard } from './info-usuario/info-usuario.guard';

const USUARIOS_ROUTES: Routes = [
  { path: ':id/info', component: InfoUsuarioComponent, canActivate: [infoUsuarioGuard] },
  { path: ':id/editar', component: EditarUsuarioComponent },
]

export const routes: Routes = [
  { path: 'usuarios', component: UsuariosComponent, children: USUARIOS_ROUTES },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: () => 'usuarios', pathMatch: 'full' }
];
```

Ahora cuando pulsemos sobre cualquier enlace de **Ver +info** nos saldrá el popup preguntando si queremos entrar o no a la ruta. Al pulsar sobre **Aceptar** entraremos a ella, pero si pulsamos sobre **Cancelar** entonces nos quedaremos en la ruta en la que estamos.

Ya tenemos una guarda, vamos a implementar otra guarda, la de tipo **canDeactivate**, que nos permite controlar si podemos salir de la ruta actual o no.

En este caso la vamos a utilizar para controlar si podemos salir de la ruta de edición de usuarios o no, dependiendo de si los datos se han guardado o no.

Antes de ir a la guard, vamos a entrar en el componente para añadir una propiedad **datosGuardados** y una función que modifique el valor de esta propiedad.

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

@Component({
  selector: 'app-editar-usuario',
  standalone: true,
  imports: [],
  templateUrl: './editar-usuario.component.html',
  styleUrls: ['./editar-usuario.component.css'
})
export class EditarUsuarioComponent implements OnInit {
  datosGuardados: boolean = false

  constructor() { }

  ngOnInit(): void {
  }

  guardar() {
    this.datosGuardados = true
  }
}
```

En la plantilla vamos a poner un botón para llamar a la función de **guardar** del componente, y mostraremos el valor de la propiedad **datosGuardados**.

```
<h3>Editar usuario</h3>

<p>Actualizaciones guardadas: {{datosGuardados}}</p>

<button type="button" (click)="guardar()">Actualizar</button>
```

Una vez tenemos el componente, vamos a implementar la guard **EditarUsuarioGuard**.

Al igual que la anterior guard, esta tiene que devolver un booleano. Si el valor a devolver es **true** entonces podemos salir de la ruta actual, mientras que si es **false** no.

Dentro de la guard, vamos a crear una interfaz **CanComponentDeactivate** con la que obligaremos a implementar en los componentes un método **canDeactivate** que devuelva un booleano para indicarle a la guarda si podemos salir o no.

Esta interfaz la añadiremos como tipo genérico de la interfaz **CanDeactivate<T>** y como tipo del parámetro **component** del método **canDeactivate**.

/angular-router-lab/src/app/editar-usuario/editar-usuario.guard.ts

```
import { CanDeactivateFn } from '@angular/router';
import { Observable } from 'rxjs';

export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean
}

export const editarUsuarioGuard: CanDeactivateFn<CanComponentDeactivate> = (component, currentRoute, currentState, nextState) => {
  return true;
};
```

En lugar de devolver un **true** como estamos haciendo ahora mismo, vamos a hacer que el método **canDeactivate** devuelva el booleano que devuelva a su vez este mismo método pero dentro del **canDeactivate** de los componentes en los que queremos aplicar la guard.

/angular-router-lab/src/app/editar-usuario/editar-usuario.guard.ts

```
import { CanDeactivateFn } from '@angular/router';
import { Observable } from 'rxjs';

export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean
}

export const editarUsuarioGuard: CanDeactivateFn<CanComponentDeactivate> = (component, currentRoute, currentState, nextState) => {
  return component.canDeactivate()
};
```

Como esta guarda la queremos aplicar sobre el componente **EditarUsuarioComponent** vamos a hacer que la clase de este implemente la interfaz **CanComponentDeactivate** y vamos a crear el método **canDeactivate**.

Dentro de este método vamos a devolver un **true** si los datos ya están guardados, y si no lo están vamos a preguntarle al usuario si quiere salir con un **confirm**.

/angular-router-lab/src/app/editar-usuario/editar-usuario.component.ts

```
import { Component, OnInit } from '@angular/core';
import { CanComponentDeactivate } from './editar-usuario.guard';

@Component({
  selector: 'app-editar-usuario',
  standalone: true,
  imports: [],
  templateUrl: './editar-usuario.component.html',
  styleUrls: ['./editar-usuario.component.css'
})
export class EditarUsuarioComponent implements OnInit, CanComponentDeactivate {
  datosGuardados: boolean = false

  constructor() { }
```

```

ngOnInit(): void {
}

guardar() {
  this.datosGuardados = true
}

canDeactivate() {
  return this.datosGuardados ? true : confirm('¿Seguro que desea salir?')
}

}

```

Por último, solo nos queda indicar sobre que rutas queremos aplicar la guard que hemos creado.

Tenemos que ir al archivo de rutas y añadir sobre la ruta de **editar-usuario** la propiedad **canDeactivate**. El valor que le vamos a dar es un array con todas las guardas de este tipo que queramos aplicar sobre la ruta.

/angular-router-lab/src/app/app.routes.ts

```

import { Routes } from '@angular/router';
import { UsuariosComponent } from './usuarios/usuarios.component';
import { NuevoUsuarioComponent } from './nuevo-usuario/nuevo-usuario.component';
import { InfoUsuarioComponent } from './info-usuario/info-usuario.component';
import { EditarUsuarioComponent } from './editar-usuario/editar-usuario.component';
import { infoUsuarioGuard } from './info-usuario/info-usuario.guard';
import { editarUsuarioGuard } from './editar-usuario/editar-usuario.guard';

const USUARIOS_ROUTES: Routes = [
  { path: ':id/info', component: InfoUsuarioComponent, canActivate: [infoUsuarioGuard] },
  { path: ':id/editar', component: EditarUsuarioComponent, canDeactivate: [editarUsuarioGuard] },
]

export const routes: Routes = [
  { path: 'usuarios', component: UsuariosComponent, children: USUARIOS_ROUTES },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: () => 'usuarios', pathMatch: 'full' }
];

```

Y con esto ya deberíamos de poder salir del componente de editar usuario cuando guardamos los datos, o si no los hemos guardado, pulsando sobre el botón **Aceptar** del popup que se muestra al intentar cambiar a otra ruta.

Para terminar con este laboratorio, vamos a hacer que se muestre una página de error cuando entremos a una ruta que no existe.

/angular-router-lab/src/app/error404/error404.component.ts

```
import { NgStyle } from '@angular/common';
```

```

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

@Component({
  selector: 'app-error404',
  standalone: true,
  imports: [NgStyle],
  templateUrl: './error404.component.html',
  styleUrls: ['./error404.component.css']
})
export class Error404Component {
}

```

/angular-router-lab/src/app/error404/error404.component.html

```
<h2 [ngStyle]="{color: 'red'}">Error 404: Page not found</h2>
```

Para hacer lo que hemos dicho, solo tenemos que añadir la ruta comodín al final de todas las rutas (ya que esta siempre se ejecuta) y asignarle el componente de **Error404** que hemos creado.

/angular-router-lab/src/app/app.routes.ts

```

import { Routes } from '@angular/router';
import { UsuariosComponent } from './usuarios/usuarios.component';
import { NuevoUsuarioComponent } from './nuevo-usuario/nuevo-usuario.component';
import { InfoUsuarioComponent } from './info-usuario/info-usuario.component';
import { EditarUsuarioComponent } from './editar-usuario/editar-usuario.component';
import { infoUsuarioGuard } from './info-usuario/info-usuario.guard';
import { editarUsuarioGuard } from './editar-usuario/editar-usuario.guard';
import { Error404Component } from './error404/error404.component';

const USUARIOS_ROUTES: Routes = [
  { path: ':id/info', component: InfoUsuarioComponent, canActivate: [infoUsuarioGuard] },
  { path: ':id/editar', component: EditarUsuarioComponent, canDeactivate: [editarUsuarioGuard] },
]

export const routes: Routes = [
  { path: 'usuarios', component: UsuariosComponent, children: USUARIOS_ROUTES },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: () => 'usuarios', pathMatch: 'full' },
  { path: '**', component: Error404Component },
];

```

Si probamos a entrar en <http://localhost:4200/no-existe> veremos que se muestra la página de error.