



Angular

David Pestana

Version 13.0.0 2022-02-23

Table of Contents

1. Introducción	1
2. Diferencias entre AngularJS y Angular	2
3. Características	3
4. Single Page Applications (SPAs)	4
5. Lenguaje	7
6. Requisitos básicos	8
7. Arquitectura	9
7.1. Módulos	9
7.2. Componentes	9
7.3. Plantillas	9
7.4. Metadatos	9
7.5. Data Binding	10
7.6. Directivas	10
7.7. Servicios	10
7.8. Inyección de dependencias	10
8. Angular CLI	11
8.1. Instalación	12
8.2. Creación de contenido	13
8.3. Ejecución	14
8.4. Producción	15
8.5. Generar Elementos de Angular	16
8.6. Testing	18
8.7. Añadir librerías externas	19
8.8. Lab: Añadir librerías externas (bootstrap)	20
8.9. Actualizar proyecto y librerías	23
8.10. Hot Module Replacement	24
8.11. Estructura del proyecto	25
9. Componentes	27
9.1. Template	28
9.2. Class	29
9.3. Decorator	30
9.4. Ciclo de vida	31
10. Data Binding	32
10.1. String Interpolation	33
10.2. Lab: String Interpolation	34
10.3. Property Binding	38
10.4. Lab: Property Binding	39
10.5. Event Binding	42

10.6. Lab: Event Binding	44
10.7. Two-Way Data Binding	48
10.8. Lab: Two-Way Data Binding	50
10.9. Lab: ¿Cómo funciona internamente el 2-Way Data Binding?	55
11. Variables de plantilla (Referencias)	58
11.1. Lab: Variables de plantilla (Referencias)	59
12. Decorador @Input()	66
12.1. Lab: Decorador @Input()	69
13. Decorador @Output()	75
13.1. Lab: Decorador @Output0	76
14. Directivas	81
14.1. Directivas de atributo	82
14.2. Crear una directiva de atributo	83
14.3. Lab: Crear una directiva de atributo	84
14.4. Directivas estructurales	92
14.5. ngIf	93
14.6. ngIf con else	94
14.7. ngIf con then/else	96
14.8. Lab: ngIf	97
14.9. ngFor	100
14.10. Lab: ngFor	102
14.11. ngSwitch	108
14.12. Lab: ngSwitch	109
15. Pipes	113
15.1. Pipe uppercase	115
15.2. Pipe lowercase	116
15.3. Pipe titlecase	117
15.4. Pipe currency	118
15.5. Pipe date	119
15.6. Pipe slice	120
15.7. Pipe json	121
15.8. Lab: Pipes	122
15.9. Crear un pipe	126
15.10. Lab: Crear custom pipes	128
15.11. Pipes puros e impuros	136
15.12. Lab: Pipes puros e impuros	137
15.13. Pipe async	149
16. Formularios	150
16.1. Formularios de plantilla (FormsModule)	151
16.2. Lab: Formularios (FormsModule)	153
16.3. Formularios reactivos (ReactiveFormsModule)	168

16.4. Lab: Formularios reactivos (ReactiveFormsModule)	170
16.5. Crear una validación personalizada	186
16.6. Validación de campos cruzados	187
16.7. Lab: Crear validaciones	188
17. Servicios e inyección de dependencias	199
17.1. Lab: Servicios e inyección de dependencias	202
17.2. Comunicar componentes mediante servicios	214
17.3. Lab: Comunicar componentes mediante servicios	216
18. Observables (RxJS)	222
18.1. Suscripciones	224
18.2. Operadores	227
18.3. Lab: Observables	228
19. Peticiones Http	250
19.1. HttpClient	251
19.2. Lab: HttpClient	253
19.3. Interceptores	271
19.4. Lab: Interceptores	273
20. Angular Router	294
20.1. Definición de rutas	295
20.2. Componente router-outlet	296
20.3. Directiva routerLink	297
20.4. Navegación por código	298
20.5. Ruta comodín	299
20.6. Rutas con parámetros	300
20.7. Rutas con query params	301
20.8. Rutas anidadas	302
20.9. Guards	303
20.10. Lab: Angular Router	304
21. Módulos	323
21.1. Añadir elementos a un módulo	325
21.2. Creando un módulo de funcionalidad	328
22. Transclusión	330
22.1. Lab: Transclusión	331
23. Decorador ViewChild	339
23.1. Lab: ViewChild	340
24. Componentes dinámicos	345
24.1. Lab: Componentes dinámicos	346
24.2. Lab: Componentes dinámicos	358
25. Progressive Web Apps (PWAs)	383
25.1. Beneficios de usar una PWA	384
25.2. App Manifest	385

25.3. Service Workers	386
25.4. Lab: Progressive Web Apps (PWAs)	388
25.5. Notificaciones Push	422
25.6. Lab: Notificaciones Push en PWAs	424
26. Angular Universal	443
26.1. Server Side Rendering	444
26.2. Lab: Server Side Rendering	445
26.3. Prerendering	447
26.4. Lab: Prerendering	448
27. Angular Elements	468
27.1. Lab: Angular Elements	469
28. Ngrx	481
28.1. Actions	482
28.2. Reducers	483
28.3. Selectors	484
28.4. Lab: Contador con Ngrx	485
28.5. Effects	495
29. Animaciones	496
29.1. Trigger	497
29.2. State	498
29.3. Style	499
29.4. Transition	500
29.5. Animate	501
29.6. Keyframes	502
29.7. Inicio y fin de la animación	503
29.8. Lab: Animaciones	504
30. Test unitarios	515
30.1. Testing componentes	516
30.2. Lab: Testing componentes	517
30.3. Testing pipes	521
30.4. Lab: Testing pipes	522
30.5. TestBed	526
30.6. Lab: HTML de los componentes	527
30.7. Lab: Lanzar eventos dentro de componentes	536
30.8. Testing servicios	542
30.9. Lab: Testing servicios	543
30.10. Lab: Mocks	553
31. Testing E2E con Cypress	587
31.1. Instalación	588
31.2. Ejecución de tests	589
31.3. Búsqueda de elementos	590

31.4. Lab: Búsqueda de elementos	591
31.5. Navegar por el DOM	596
31.6. Lab: Navegar por el DOM.....	597
31.7. Retry ability	606
31.8. Lab: Retry ability	607
31.9. Interacciones con elementos.....	612
31.10. Screenshots	614
31.11. Lab: Screenshots.....	615
31.12. Alias	618
31.13. Fixtures	619
31.14. Lab: Fixtures	620
31.15. Mocking	633
31.16. Lab: Mocking con spy y stub	634
31.17. Lab: Mocking con intercept	643
31.18. Tick y Clock	651
31.19. Tests visuales	652
31.20. Lab: Tests visuales	653
32. Internacionalización	657
32.1. Lab: internacionalización	658
32.2. Lab: Internacionalización con ngx-translate.....	668
33. Compodoc	684
33.1. Lab: Compodoc	685

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.

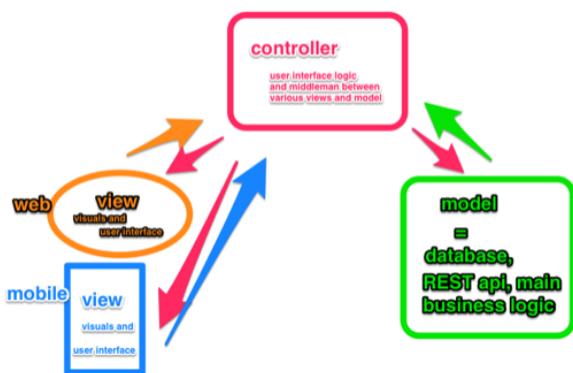


Figure 1. Model View Controller flow

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>

Chapter 2. Diferencias entre AngularJS y Angular

Debido a la reescritura de Angular han aparecido muchas diferencias frente a AngularJS.

- Angular **está orientado a móviles**, mientras que AngularJS necesita de librerías para poderse visualizar en estos dispositivos.
- Los **controladores** y el **\$scope** se han sustituido por componentes.
- La **sintaxis de las directivas estructurales** ha cambiado **ng-repeat** → ***ngFor**.
- Angular usa los **eventos y propiedades estándar del DOM** en lugar de directivas.
- Se ha cambiado el **one-way data binding** por el **property-binding**.
- La **sintaxis del two-way data binding** ha cambiado **ng-model** → **[(ngModel)]**.
- An AngularJS habia varias formas de crear servicios, mientras que en la nueva versión **los servicios se crean usando clases**.

Chapter 3. 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.

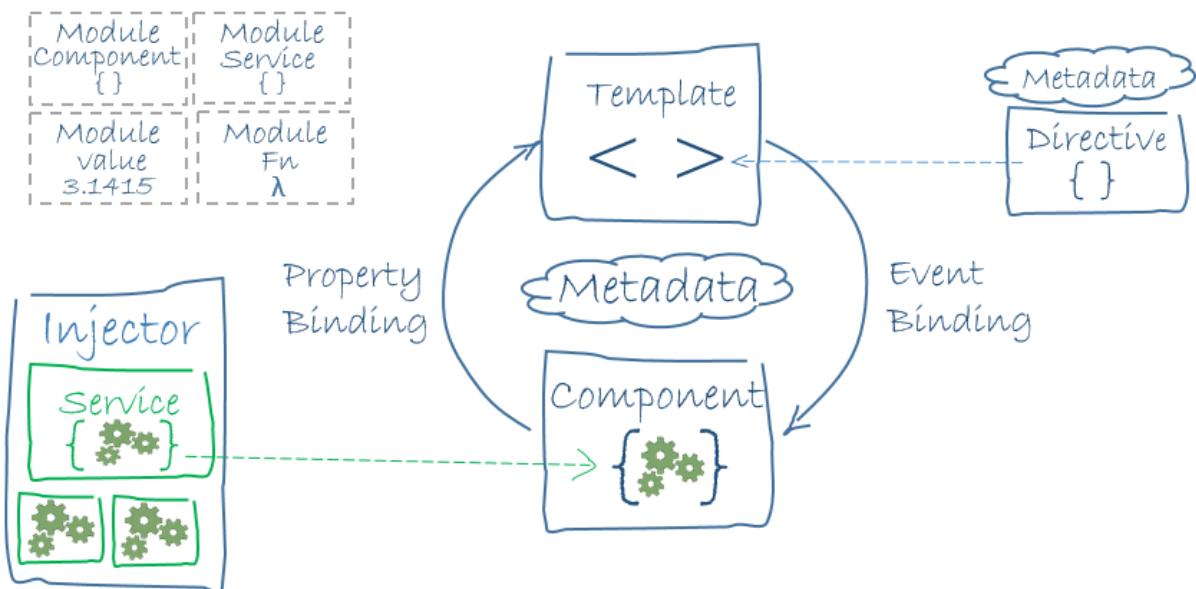


Figure 3. Overview

Chapter 4. Single Page Applications (SPAs)

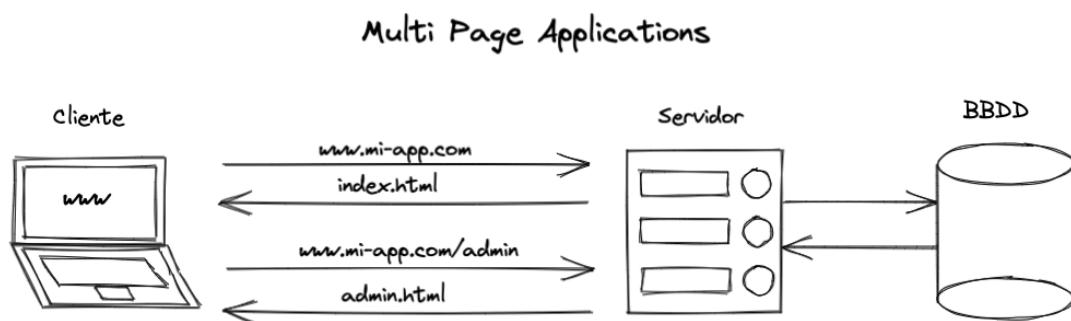
Con Angular estaremos creando aplicaciones que se conocen como Single Page Applications o las siglas SPAs. Estas aplicaciones son aplicaciones que solo tienen una única página de HTML para todo el proyecto.

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.

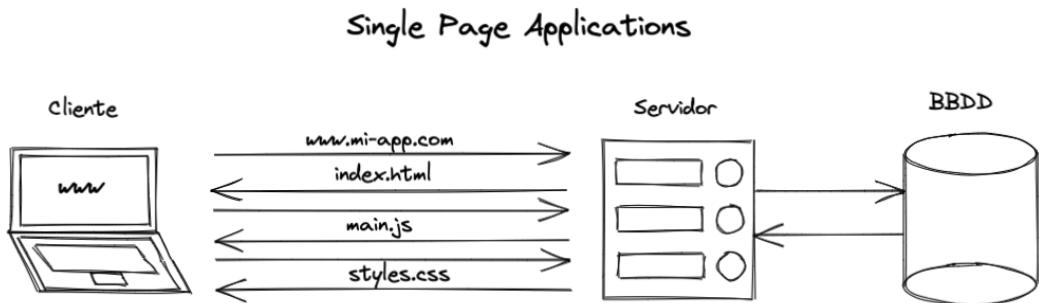
Pero con el paso del tiempo, la tecnología va evolucionando y estas aplicaciones empiezan a tener mucha más funcionalidad, por lo que se empiezan a añadir bastantes archivos de JS para añadirlas.

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

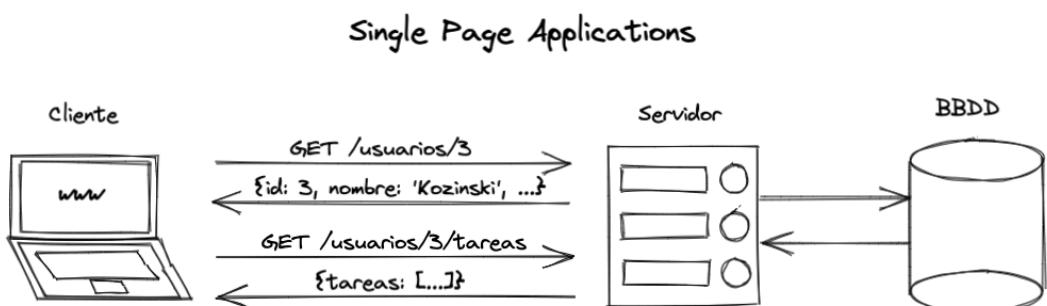
descargar varios de estos últimos. También tienen que gestionar el orden en el que todos estos archivos se van descargando, lo que puede darnos verdaderos quebraderos de cabeza.

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 5. 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 6. 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](#).

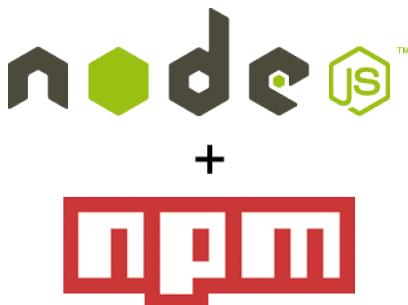


Figure 5. Node y npm

Editores

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

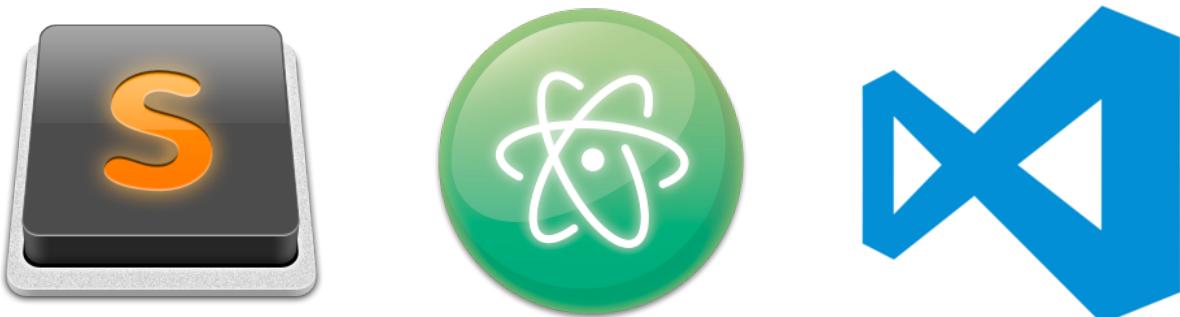


Figure 6. por orden, Sublime, Atom y VSCode

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 7. 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.

7.1. 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.

7.2. 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.

7.3. Plantillas

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

7.4. 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...

7.5. 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**

7.6. 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**

7.7. 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.

7.8. 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 8. 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.

8.1. 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
```

8.2. 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.

8.3. 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
```

8.4. 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
```

8.5. 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**).

8.6. 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
```

8.7. 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
```

8.8. 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.

8.9. 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.

8.10. 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
```

8.11. Estructura del proyecto

Al crear un proyecto de Angular 2, nos crea una carpeta que contiene la siguiente estructura de ficheros:

- **e2e**: aquí se encuentran los archivos de testing End to End, que se ejecutan con la ayuda de Protractor y Jasmine.
- **node_modules**: aquí se encuentran todas las dependencias de nuestro proyecto.
- **src**: es posiblemente la carpeta más importante del proyecto, aquí se encuentra todo el código de la aplicación, y es en la carpeta que pasaremos la mayor parte del tiempo.
 - **app**: en esta carpeta vamos a tener todos los componentes, servicios, plantillas y resto de elementos de la aplicación.
 - **app.module.ts**: este archivo se encarga de entender que componentes y dependencias tenemos en el proyecto.
 - **assets**: aquí guardaremos los assets que no pertenezcan a los componentes de Angular, como imágenes, sonidos...
 - **environments**: en esta carpeta tendremos los archivos encargados de indicar los entornos de trabajo (producción, desarrollo...).
 - **index.html**: este es el archivo donde se inicia la SPA, donde se van a cargar los componentes.
 - **main.ts**: es el primer archivo en ejecutarse, y en él se encuentran los parámetros de configuración de la aplicación y del entorno en el que trabajaremos...
 - **polyfills.ts**: asegura la compatibilidad con los distintos navegadores.
 - **styles.css**: aquí definimos los estilos globales que vamos a darle a la aplicación.
 - **test.ts**: es el punto de entrada a los test unitarios, los haremos con Jasmine.
- **angular-cli.json**: aquí se encuentra la configuración de Angular-CLI.
- **tsconfig.json**: en este archivo se encuentra la configuración de TypeScript, donde se indica como hacer la compilación a JavaScript.
- **karma.conf.js**: aquí se encuentra la configuración de los test unitarios.
- **package.json**: aquí se indican las dependencias del proyecto, las acciones a ejecutar cuando se lanza el proyecto, y alguna información del proyecto.
- **protractor.conf.js**: aquí se encuentra la configuración de los test End to End.

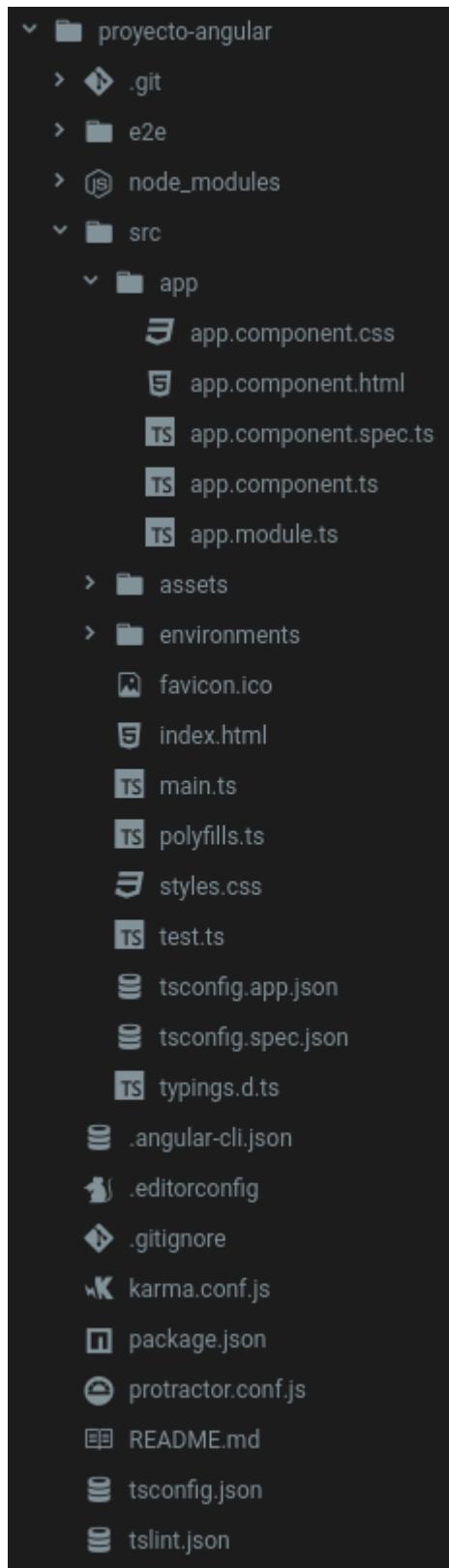


Figure 9. Estructura de proyecto en Angular 2

Chapter 9. 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, hay que asegurarse de que se añade en el array de **declarations** del `app.module.ts` y de que se importa correctamente. Esto hace posible que al usar la etiqueta del componente, Angular 2 sea capaz de encontrarlo y mostrarlo.

`app.module.ts`

```
import { MiComponente } from './mi-componente/mi-componente.component';

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

9.1. 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',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

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

9.2. 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() {  
    // ...  
  }  
}
```

9.3. 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',
  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.
- **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.

9.4. 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[] = [];

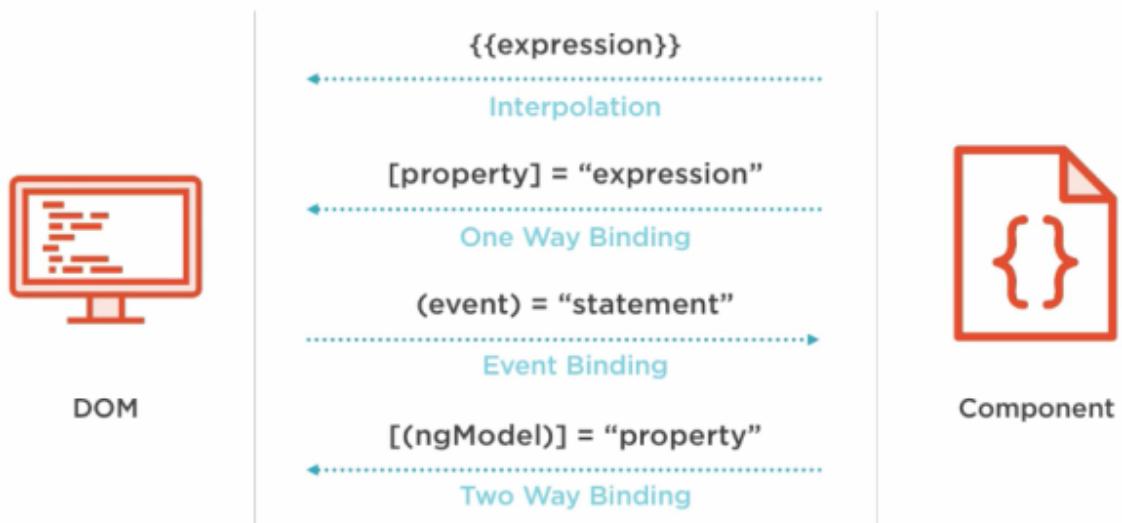
  constructor(private itemService: ItemService) { }

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

Chapter 10. Data Binding

El **data binding** consiste en la sincronización entre los datos del componente y la vista. En Angular 2+ 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



10.1. 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';

@Component({
  selector: 'app-root',
  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. -->
```

10.2. 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  
? 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-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';  
  
@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'  
}
```

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';

@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'

  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.

10.3. 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';

@Component({
  selector: 'app-root',
  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".

10.4. 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  
? 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-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';  
  
@Component({  
  selector: 'app-root',  
  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';

@Component({
  selector: 'app-root',
  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**

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

@Component({
  selector: 'app-root',
  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**.

```
<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**.

```
<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.

10.5. 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';

@Component({
  selector: 'app-root',
  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';

@Component({
  selector: 'app-root',
  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!');
  }
}
```

10.6. 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  
? 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-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';

@Component({
  selector: 'app-root',
  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';

@Component({
  selector: 'app-root',
  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)
  }
}
```

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';

@Component({
  selector: 'app-root',
  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)
  }

}
```

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';

@Component({
  selector: 'app-root',
  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 llamamos a **window.speechSynthesis.speak()** pasandole la configuración anterior y que así nos pueda cantar mal la intro.

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

@Component({
  selector: 'app-root',
  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
    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.

10.7. 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 archivo **app.module.ts** para que Angular reconozca la directiva **ngModel**.

/src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { FiltroPipe } from './filtro.pipe';
import { FormsModule } from '@angular/forms'; <!-- Aquí la importación -->

@NgModule({
  declarations: [
    AppComponent,
    FiltroPipe
  ],
  imports: [
    BrowserModule,
    FormsModule, <-- Añadimos el módulo de Forms en el módulo de la aplicación -->
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

/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 {
  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>
```

10.8. 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  
? 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-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 módulo de la aplicación.

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

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

import { AppComponent } from './app.component';

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

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';

@Component({
  selector: 'app-root',
  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 método **getClaseTema()** 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';

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

  temaSeleccionado: string = 'lightTheme'

  getClaseTema(): 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.

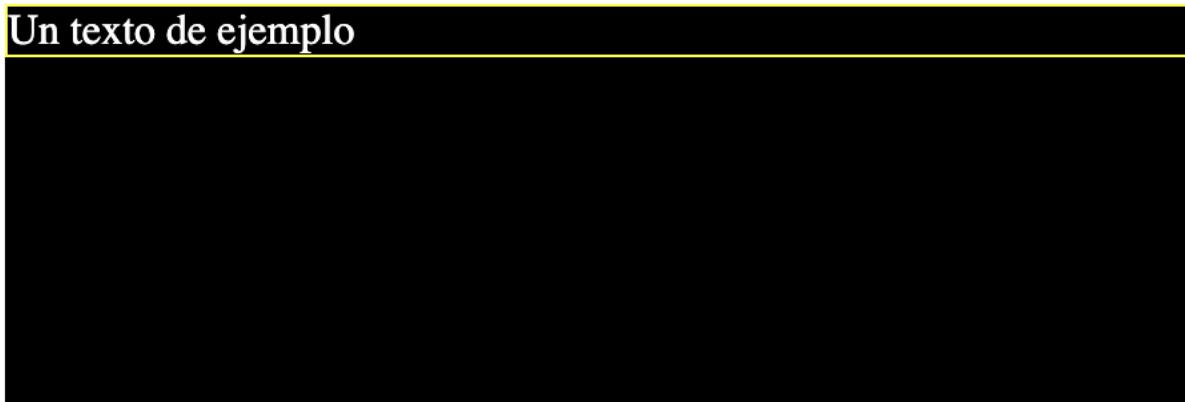
Si el valor es true, la clase de CSS se aplica sobre el elemento, pero si es false, la clase se deja de aplicar.

```
<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]="getClaseTema()">
  <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 ▾



10.9. 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  
? 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-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';  
  
@Component({  
  selector: 'app-root',  
  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';

@Component({
  selector: 'app-root',
  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';

@Component({
  selector: 'app-root',
  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 11. 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';

@Component({
  selector: 'app-root',
  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
  }
}
```

11.1. 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  
? 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-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';

@Component({
  selector: 'app-root',
  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';

@Component({
  selector: 'app-root',
  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';  
  
@Component({  
  selector: 'app-root',  
  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';

@Component({
  selector: 'app-root',
  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**.

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

@Component({
  selector: 'app-root',
  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.

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

@Component({
  selector: 'app-root',
  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 12. 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

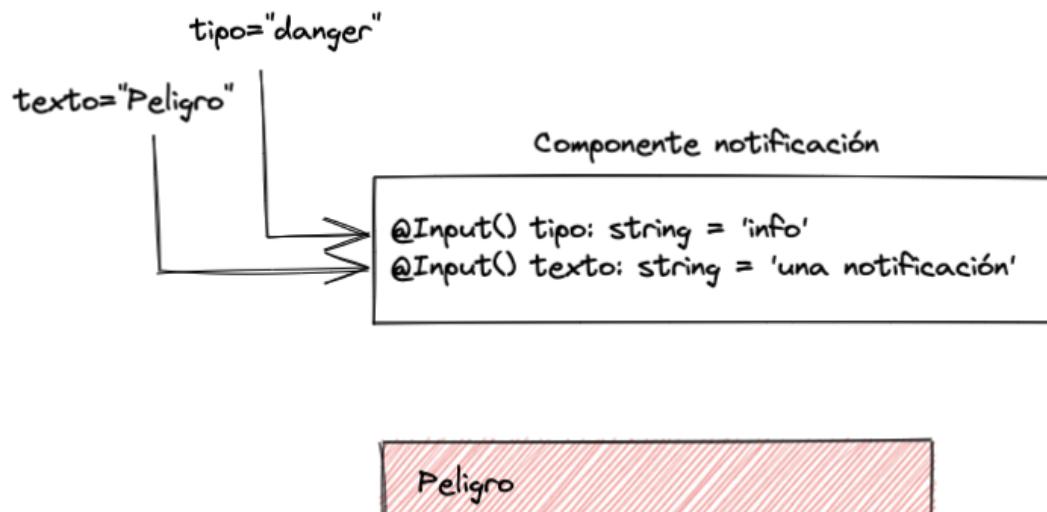
```
tipo: string = 'info'  
texto: string = 'una notificación'
```

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: getColor()}}>
  <p>{{texto}}</p>
</div>

```

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

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

@Component({
  selector: 'app-notificacion',
  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 {
  }

  getColor(): 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>
```

12.1. 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  
? 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 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',  
  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>
```

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;
}

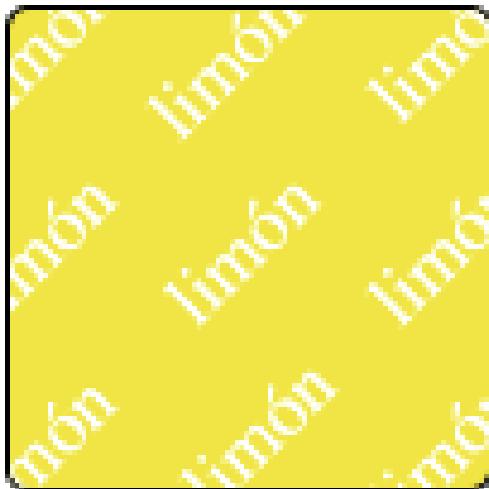
.sugus > p {
  text-align: center;
  transform-origin: center center;
  transform: rotate(-45deg);
  position: absolute;
  top: 25px;
  left: 30px;
  text-shadow: 60px 0px 0px, -60px 0px 0px, -25px 30px 0px, 25px -30px 0px, 30px 30px 0px, -30px -30px 0px, 0px 60px 0px,
  0px -60px 0px;
}
```

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

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

@Component({
  selector: 'app-sugus',
  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 13. 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`.

13.1. 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  
? 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 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-boton/mi-boton.component.ts

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

@Component({
  selector: 'app-mi-boton',
  templateUrl: './mi-boton.component.html',
  styleUrls: ['./mi-boton.component.css']
})
export class MiBotonComponent 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',
  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',
  templateUrl: './mi-boton.component.html',
  styleUrls: ['./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',
  templateUrl: './mi-boton.component.html',
  styleUrls: ['./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-boton** y detectar el evento **customClick**.

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

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

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.

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

@Component({
  selector: 'app-root',
  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 14. 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

14.1. 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.

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

/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 {
  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.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.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**, ...).

14.2. 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]'  
})  
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.

14.3. 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
```

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.

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

@Directive({
  selector: '[appMarcar]'
})
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.

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

@Directive({
  selector: '[appMarcar]'
})
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**.

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

@Directive({
  selector: '[appMarcar]'
})
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**.

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

@Directive({
  selector: '[appMarcar]'
})
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 que color queremos que se pinten asignandoselo 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 añadirselo 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**.

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

@Directive({
  selector: '[appMarcar]'
})
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.

```
<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.

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

@Directive({
  selector: '[appMarcar]'
})
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.

```
<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]'
})
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**.

14.4. 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 *.

14.5. ngIf

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',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  mostrar: boolean = false;
}
```

/src/app/app.component.html

```
<p *ngIf="mostrar">Este parrafo no se muestra</p>
```

14.6. ngIf 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 ngIf no se cumplía, teníamos que volver a utilizar la directiva ngIf con la condición negada.

/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 {
  mostrar: boolean = false;
}
```

/src/app/app.component.html

```
<p *ngIf="mostrar">Se muestra si la propiedad mostrar es igual a true</p>
<p *ngIf="!mostrar">Se muestra si la propiedad mostrar es igual a false</p>
```

El problema de utilizar dos veces la directiva ngIf 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.

En Angular 4 se añadió el **else** para evitar el uso de dos directivas **ngIf**. Para usarlo tenemos que poner después de la condición ; **else variableDePlantillaDelElse**.

En este caso, necesitaremos meter el contenido a mostrar cuando la condición no se cumpla dentro de unas etiquetas **ng-template**, y sobre esta etiqueta tenemos que poner una variable de plantilla.

/src/app/app.component.html

```
<p *ngIf="mostrar; else parrafoElse">Se muestra si la propiedad mostrar es igual a true</p>
<ng-template #parrafoElse>
  <p>Se muestra si la propiedad mostrar es igual a false</p>
</ng-template>
```

La etiqueta **ng-template** es una etiqueta de Angular que equivale a la etiqueta **template** de HTML.



Esta etiqueta se encarga de encapsular otras etiquetas, y el contenido que metemos dentro de ella no se pinta en el navegador directamente, sino que tiene que ser Angular el que va a acceder al contenido, lo va a clonar y lo va a pintar donde toque (en nuestro caso, donde hemos puesto la directiva ngIf).

14.7. ngIf con then/else

Otra forma de utilizar el ngIf con la parte del else, es creando esta vez dos plantillas (etiquetas **ng-template**), una con el contenido a mostrar cuando la condición se cumpla, y la otra con el contenido a mostrar cuando no se cumpla. Ambas plantillas tienen que tener una variable de plantilla asociada.

Después, en una etiqueta **div** pondremos la directiva ngIf con la condición, seguida de ; **then varDePlantillaDellIf else varDePlantillaDelElse**.

/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 {
  mostrar: boolean = false;
}
```

/src/app/app.component.html

```
<div *ngIf="mostrar; then parrafoIf else parrafoElse"></div>

<ng-template #parrafoIf>
  <p>Se muestra si la propiedad mostrar es igual a true</p>
</ng-template>

<ng-template #parrafoElse>
  <p>Se muestra si la propiedad mostrar es igual a false</p>
</ng-template>
```

14.8. 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',  
  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

```
<div *ngIf="darkModeActivado"></div>

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

Y ahora vamos a meter los dos botones dentro de una etiqueta **ng-template** cada uno de ellos con su variable de plantilla correspondiente.

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

```
<div *ngIf="darkModeActivado"></div>

<ng-template #darkModeBtn>
  <button type="button">Activar dark mode </button>
</ng-template>

<ng-template #lightModeBtn>
  <button type="button">Activar light mode </button>
</ng-template>
```

Ahora vamos a añadirle al div la parte del **then** y **else** para indicar que cuando sea true la condición muestre el botón que se encuentra dentro de la plantilla con la referencia **lightModeBtn**, y cuando sea false muestre el de **darkModeBtn**.

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

```
<div *ngIf="darkModeActivado; then lightModeBtn else darkModeBtn"></div>

<ng-template #darkModeBtn>
  <button type="button">Activar dark mode </button>
</ng-template>

<ng-template #lightModeBtn>
  <button type="button">Activar light mode </button>
</ng-template>
```

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.

```
<div *ngIf="darkModeActivado; then lightModeBtn else darkModeBtn"></div>

<ng-template #darkModeBtn>
  <button type="button" (click)="toggleDarkMode(true)">Activar dark mode </button>
</ng-template>

<ng-template #lightModeBtn>
  <button type="button" (click)="toggleDarkMode(false)">Activar light mode </button>
</ng-template>
```

14.9. ngFor

La directiva **ngFor** 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 **ngFor**.

/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 {
  items: Array<string> = ['Item 1', 'Item 2', 'Item 3']
}
```

/src/app/app.component.html

```
<ul>
  <li *ngFor="let item of items">{{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 **ngFor**, 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 separandolas 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>
  <li *ngFor="let item of items; let posicion = index; let esUltimo = last;">{{posicion}}: {{item}} {{esUltimo ? '- fin' : ''}}</li>
</ul>
```

14.10. 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 } from '@angular/core';  
  
@Component({  
  selector: 'app-root',  
  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**, **simbolo** 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>
    <tr *ngFor="let cripto of criptomonedas">
      <td>{{cripto.nombre}}</td>
      <td>{{cripto.simbolo}}</td>
      <td>{{cripto.precio}}</td>
    </tr>
  </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>
    <tr *ngFor="let cripto of criptomonedas; let esPar = even" [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>  
    <tr *ngFor="let cripto of criptomonedas; let esPar = even; let esPrimeraFila = first; let esUltimaFila = last"  
      [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**.

```
<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Símbolo</th>
      <th>Precio</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let cripto of criptomonedas; let esPar = even; let esPrimeraFila = first; let esUltimaFila = last"
        [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.

```
<table>
  <thead>
    <tr>
      <th>Nº</th>
      <th>Nombre</th>
      <th>Símbolo</th>
      <th>Precio</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let cripto of criptomonedas; let esPar = even; let esPrimeraFila = first; let esUltimaFila = last"
        [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.

```
<table>
  <thead>
    <tr>
      <th>Nº</th>
      <th>Nombre</th>
      <th>Símbolo</th>
      <th>Precio</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let cripto of criptomonedas; let esPar = even; let esPrimeraFila = first; let esUltimaFila = last; let posicion = index" [ngClass]="{filaPar: esPar, primeraFila: esPrimeraFila, ultimaFila: esUltimaFila}">
      <td>{{posicion + 1}}</td>
      <td>{{cripto.nombre}}</td>
      <td>{{cripto.símbolo}}</td>
      <td>{{cripto.precio}}</td>
    </tr>
  </tbody>
</table>
```

14.11. ngSwitch

La última directiva estructural que podemos encontrar en Angular es el **ngSwitch**, 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, ngSwitch, ngSwitchCase y ngSwitchDefault. Solo las dos últimas llevan el *.

/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 {
  resultado: number = 100;

}
```

/src/app/app.component.html

```
<div [ngSwitch]="resultado">
  <p *ngSwitchCase="10">10</p>
  <p *ngSwitchCase="100">100</p>
  <p *ngSwitchDefault>Valor por defecto</p>
</div>
```

14.12. 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',  
  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 módulo de la aplicación.

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

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

import { AppComponent } from './app.component';

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

Ya podemos añadir la directiva **ngModel** sobre la etiqueta select asignandole 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 identicas 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">
  <option *ngFor="let mascota of mascotas" [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">
  <option *ngFor="let mascota of mascotas" [value]="mascota">{{mascota}}</option>
</select>

<div>
  <p>Tengo un perro</p>
  <p>Tengo un gato</p>
  <p>Tengo otra mascota</p>
</div>
```

Sobre el 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">
  <option *ngFor="let mascota of mascotas" [value]="mascota">{{mascota}}</option>
</select>

<div [ngSwitch]="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".



Los valores de ngSwitchCase van con "" y ", ya que si quitamos alguna de estas dos comillas, Angular buscará el valor del case en unas propiedades del TypeScript llamadas perro y gato, que no tenemos.

El orden de las comillas no importa, podemos poner tanto "perro" como "perro".

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

```
<select [(ngModel)]="mascotaSeleccionada">
  <option *ngFor="let mascota of mascotas" [value]="mascota">{{mascota}}</option>
</select>

<div [ngSwitch]="mascotaSeleccionada">
  <p *ngSwitchCase="'perro'">Tengo un perro</p>
  <p *ngSwitchCase="'gato'">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">
  <option *ngFor="let mascota of mascotas" [value]="mascota">{{mascota}}</option>
</select>

<div [ngSwitch]="mascotaSeleccionada">
  <p *ngSwitchCase="'perro'">Tengo un perro</p>
  <p *ngSwitchCase="'gato'">Tengo un gato</p>
  <p *ngSwitchDefault>Tengo otra mascota</p>
</div>
```

Chapter 15. 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>.

15.1. 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 -->
```

15.2. 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 -->
```

15.3. 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 -->
```

15.4. 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.

15.5. 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 -->
```

15.6. 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 -->
```

15.7. 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  
}
```

15.8. 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  
? 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-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';  
  
@Component({  
    selector: 'app-root',  
    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>
```

15.9. 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'  
})  
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.

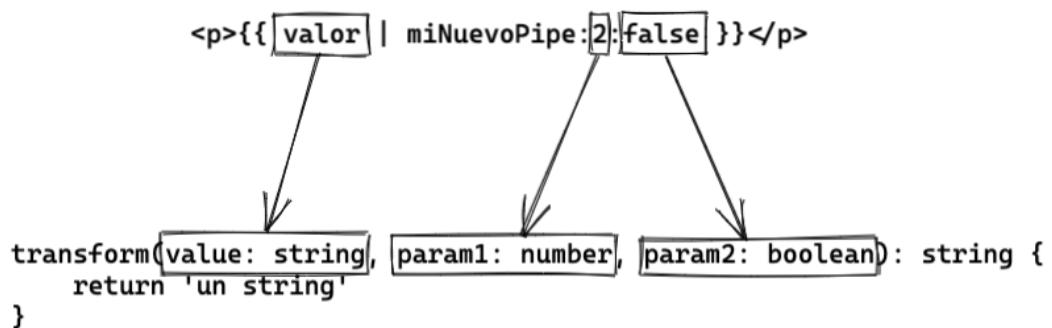
El primer parámetro que se recibe es el valor que va antes del símbolo del | cuando aplicamos estos.

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.

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

@Pipe({
  name: 'miNuevoPipe'
})
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.



15.10. 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  
? 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 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'
})
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'
})
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'
})
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'
})
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';

@Component({
  selector: 'app-root',
  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'
})
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'
})
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'
})
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'
})
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'
})
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'
})
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';

@Component({
  selector: 'app-root',
  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** pasandole 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.

15.11. 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',
  pure: false
})
export class MiPipePipe implements PipeTransform {

  transform(value: string): string {
    return 'un pipe';
  }
}
```

15.12. 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  
? 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 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.

/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',  
  ]  
}
```

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>
  <li *ngFor="let personaje of personajes">{{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>
  <li *ngFor="let personaje of personajes">{{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.

```
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',
  ]

  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.

```
<div>
  <label for="nuevo-personaje">Nuevo personaje:</label>
  <input type="text" (change)="addPersonaje($event)">
</div>
<ul>
  <li *ngFor="let personaje of personajes">{{personaje}}</li>
</ul>

<hr>

<div>
  <label for="nuevo-personaje">Filtrar por:</label>
  <input type="text" [(ngModel)]="filtroTxt">
</div>
<ul>
  <li *ngFor="let personaje of personajes">{{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.module.ts**.

/angular-pipes-puros-e-impuros-lab/src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { FiltroPipe } from './filtro.pipe';
import { FormsModule } from '@angular/forms';

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

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';

@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.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'
})
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'
})
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'
})
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.

```
<div>
  <label for="nuevo-personaje">Nuevo personaje:</label>
  <input type="text" (change)="addPersonaje($event)">
</div>
<ul>
  <li *ngFor="let personaje of personajes">{{personaje}}</li>
</ul>

<hr>

<div>
  <label for="nuevo-personaje">Filtrar por:</label>
  <input type="text" [(ngModel)]="filtroTxt">
</div>
<ul>
  <li *ngFor="let personaje of personajes | filtro:filtroTxt">{{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.

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

@Pipe({
  name: 'filtro',
  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'
})
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.

15.13. 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';

@Component({
  selector: 'app-root',
  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 16. 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.

16.1. Formularios de plantilla (FormsModule)

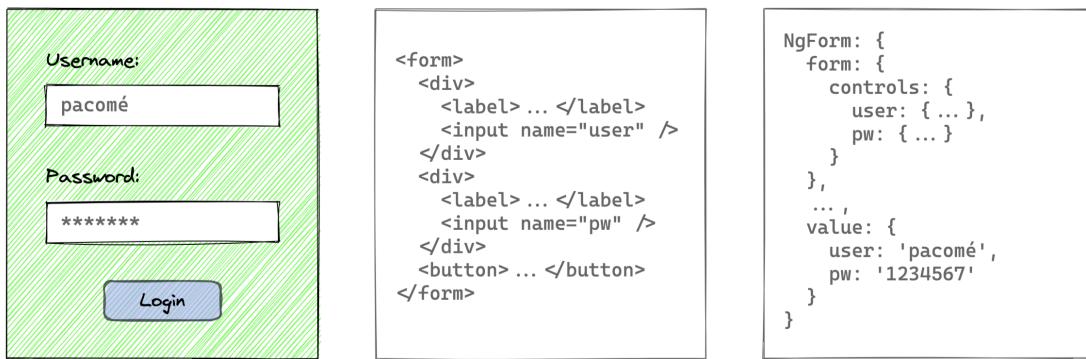
Los formularios de plantilla son un tipo de formularios en los que vamos a poner prácticamente toda su funcionalidad (valores iniciales, validaciones...) en la plantilla del componente, es decir, en el HTML.

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.

16.2. 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  
? 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 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

```
<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.

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

@Component({
  selector: 'app-root',
  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 módulo de App el **FormsModule**.

/angular-formularios-de-plantilla-lab/src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { ErroresFormComponent } from './errores-form/errores-form.component';

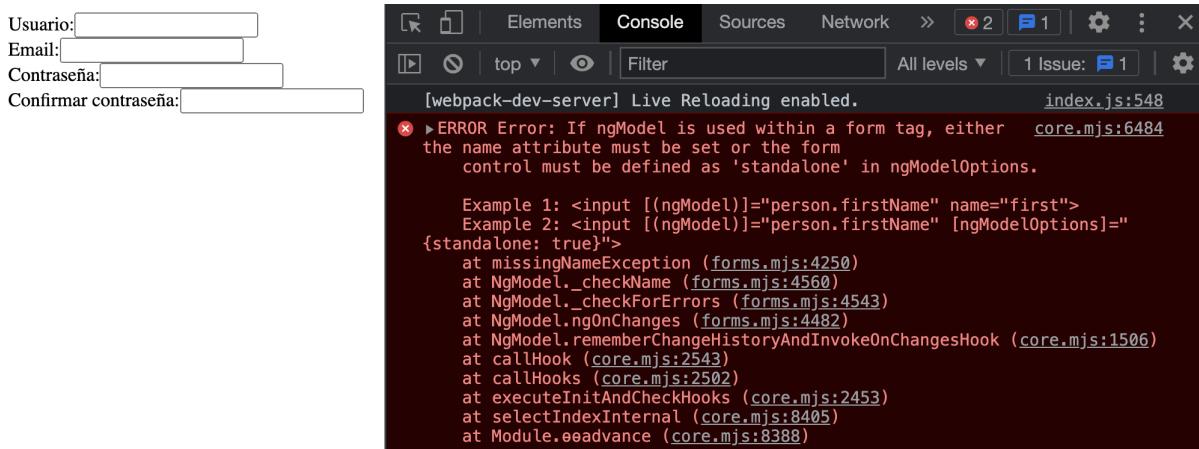
@NgModule({
  declarations: [
    AppComponent,
    ErroresFormComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

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.

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

@Component({
  selector: 'app-root',
  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.

```
<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 _ngcontent-wfo-c43>...
  ►<div _ngcontent-wfo-c43>...
  ►<div _ngcontent-wfo-c43>...
    <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.

```

    ▼ NgForm {_rawValidators: Array(0), _rawAsyncValidators: Array(0), _onDestroyCallbacks: Array(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.

/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>
        </div>
      </div>
    </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',
  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',
  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',
  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',
  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).

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

@Component({
  selector: 'app-errores-form',
  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.

```
<ul>
  <li *ngFor="let msg of mensajesErrores">{{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**.

```
<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" *ngIf="campoUsername.invalid && campoUsername.dirty"></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" *ngIf="campoEmail.invalid && campoEmail.dirty"></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" *ngIf="campoPassword.invalid && campoPassword.dirty"></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" *ngIf="campoConfirmarPassword.invalid && campoConfirmarPassword.dirty"></app-errores-form>
  </div>
  <button type="submit" [disabled]="form.invalid">Guardar</button>
</form>
```

Y con esto ya tenemos nuestro formulario completo.

16.3. 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.

16.4. 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  
? 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 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';

@Component({
  selector: 'app-root',
  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**.

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

@Component({
  selector: 'app-root',
  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(),
      confirmarPassword: new FormControl(),
    })
  }
}
```

Cada **FormControl** recibe como parámetros, un valor inicial, y luego las validaciones que queremos aplicar sobre el campo. De momento vamos a llenar 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.

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

@Component({
  selector: 'app-root',
  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 módulo de la aplicación vamos a importar el **ReactiveFormsModule** que viene de **@angular/forms**.

/angular-formularios-reactivos-lab/src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { ErroresFormComponent } from './errores-form/errores-form.component';

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

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**.

/angular-formularios-reactivos-lab/src/app/app.component.html

```
<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.

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

@Component({
  selector: 'app-root',
  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.

```
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.

```
<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.

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

@Component({
  selector: 'app-root',
  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, OnChanges, Input } from '@angular/core';

@Component({
  selector: 'app-errores-form',
  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;
      }
    }
  }
}
```

/angular-formularios-reactivos-lab/src/app/errores-form/errores-form.component.html

```
<ul>
  <li *ngFor="let msg of mensajesErrores">{{msg}}</li>
</ul>
```

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 pasariamos 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:

```
<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**.

```
<form [FormGroup]="formulario" (ngSubmit)="guardar()">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" formControlName="username">
    <app-errores-form *ngIf="formulario.controls['username'].invalid && formulario.controls['username'].dirty" [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 *ngIf="formulario.controls['email'].invalid && formulario.controls['email'].dirty" [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 *ngIf="formulario.controls['password'].invalid && formulario.controls['password'].dirty" [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 *ngIf="formulario.controls['confirmarPassword'].invalid && formulario.controls['confirmarPassword'].dirty" [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 **ngIf** 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**.

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

@Component({
  selector: 'app-root',
  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.

```
<form [FormGroup]="formulario" (ngSubmit)="guardar()">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" FormControlName="username">
    <app-errores-form *ngIf="pintarErrores('username')" [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 *ngIf="pintarErrores('email')" [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 *ngIf="pintarErrores('password')" [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 *ngIf="pintarErrores('confirmarPassword')" [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.

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

@Component({
  selector: 'app-root',
  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
  }

  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.

```
<form [formGroup]="formulario" (ngSubmit)="guardar()">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" formControlName="username">
    <app-errores-form *ngIf="pintarErrores('username')" [errores]="getErrores('username')"></app-errores-form>
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" formControlName="email">
    <app-errores-form *ngIf="pintarErrores('email')" [errores]="getErrores('email')"></app-errores-form>
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" formControlName="password">
    <app-errores-form *ngIf="pintarErrores('password')" [errores]="getErrores('password')"></app-errores-form>
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" formControlName="confirmarPassword">
    <app-errores-form *ngIf="pintarErrores('confirmarPassword')" [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.

```
<form [formGroup]="formulario" (ngSubmit)="guardar()">
  <div>
    <label for="username">Usuario:</label>
    <input type="text" id="username" formControlName="username">
    <app-errores-form *ngIf="pintarErrores('username')" [errores]="getErrores('username')"></app-errores-form>
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" formControlName="email">
    <app-errores-form *ngIf="pintarErrores('email')" [errores]="getErrores('email')"></app-errores-form>
  </div>
  <div>
    <label for="password">Contraseña:</label>
    <input type="password" id="password" formControlName="password">
    <app-errores-form *ngIf="pintarErrores('password')" [errores]="getErrores('password')"></app-errores-form>
  </div>
  <div>
    <label for="confirmarPassword">Confirmar contraseña:</label>
    <input type="password" id="confirmarPassword" formControlName="confirmarPassword">
    <app-errores-form *ngIf="pintarErrores('confirmarPassword')" [errores]="getErrores('confirmarPassword')"></app-errores-form>
  </div>
  <div>
    <button type="submit" [disabled]="formulario.invalid">Guardar</button>
  </div>
</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.

16.5. 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 } from '@angular/forms';

@Component({
  selector: 'app-root',
  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.

16.6. 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.

16.7. 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', 'qwert123']

}
```

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', 'qwert123']

  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', 'qwert123']

  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 { FormControl, FormGroup, Validators } from '@angular/forms';
import { CustomValidators } from './customValidators';

@Component({
  selector: 'app-root',
  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, OnChanges, Input } from '@angular/core';

@Component({
  selector: 'app-errores-form',
  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.

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

@Component({
  selector: 'app-root',
  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)])
    })
  }

  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.

```
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

  if (confirmPassword !== control.parent!.value) {
    return { repetirPassword: true }
  }

  return null
}

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.

```
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 { repetirPassword: true }
  }

  return null
}

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.

```
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.

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

@Component({
  selector: 'app-root',
  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.

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

@Component({
  selector: 'app-errores-form',
  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 17. 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) {}

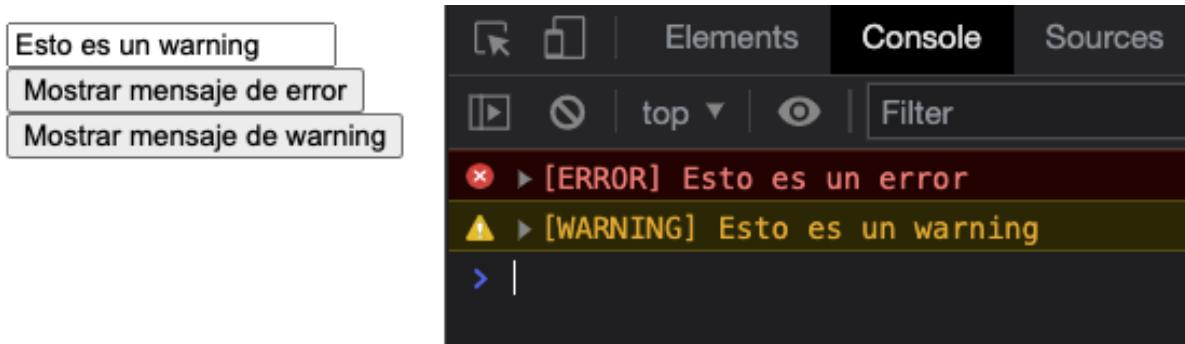
  mostrarError(msg: string): void {
    this.logger.error(msg)
  }

  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.



17.1. 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.

```
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.

```
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**.

```
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 **removeToken** pasandole como parámetro la clave del registro que queremos eliminar, en nuestro caso **miToken**.

```
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**.

```
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.

```
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.

```
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.

```
<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.

```
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.

17.2. 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>
```

17.3. 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.

```
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**.

```
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**.

```
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.

```
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**.

```
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((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 18. 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()`

```
import { Component, OnDestroy } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';

@Component({
  selector: 'app-root',
  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)
    })
  }
}
```

18.1. Suscripciones

Para trabajar con los observables tenemos que conocer otro concepto importante, que son las suscripciones.

Los observables envian 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 el.

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**.

```
import { Component } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';

@Component({
  selector: 'app-root',
  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.

```
import { Component, OnDestroy } from '@angular/core';
import { Observable, Subscriber, Subscription } from 'rxjs';

@Component({
  selector: 'app-root',
  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**.

18.2. 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 transformandolos, filtrandolos, combinandolos 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',
  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))
  }
}
```

18.3. 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  
? 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 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.

/angular-observables-lab/src/app/suscripcion/suscripcion.component.ts

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

@Component({
  selector: 'app-suscripcion',
  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/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>
  <button type="button" *ngIf="activa; else activarBtn" (click)="cancelarSuscripcion()">Cancelar suscripción</button>
  <ng-template #activarBtn>
    <button type="button" (click)="activarSuscripcion()">Activar suscripción</button>
  </ng-template>
</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/suscripcion/suscripcion.component.ts

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

@Component({
  selector: 'app-suscripcion',
  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()**.

/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.

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

@Component({
  selector: 'app-suscripcion',
  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**.

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

@Component({
  selector: 'app-suscripcion',
  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.

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

@Component({
  selector: 'app-suscripcion',
  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.

```
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.

```
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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.

```
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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
        }
      }, 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**.

```
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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** indicandole que se ha cancelado su suscripción y que tendrá que volver a suscribirse si quiere continuar utilizando la plataforma.

/angular-observables-lab/src/app/suscripcion/suscripcion.component.ts

```
import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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)
    })
  }

  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**.

```

import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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.

```

import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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 indicandole 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.

```

import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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

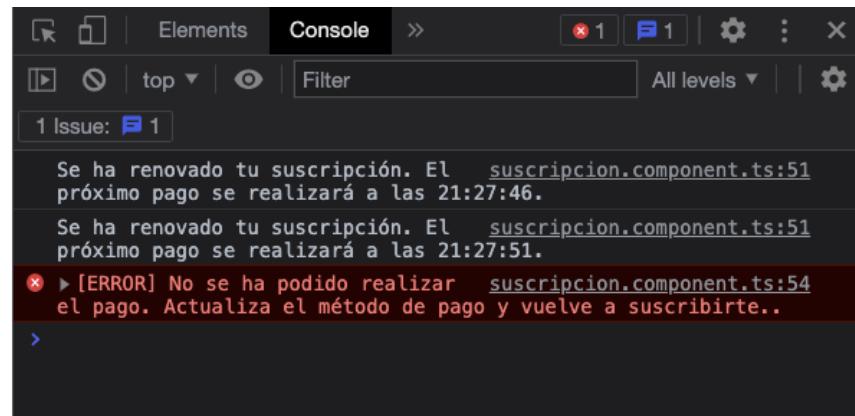
          subscriber.next(fecha.toLocaleTimeString())
        } else {
          this.activa = false

          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 pasandole el identificador que retorna la función de **setInterval**, para que ese dato deje de actualizarse.

```

import { Component, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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.')
        }
        clearInterval(intervalId)
      }, 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\$**.

```
import { Component, EventEmitter, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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)
      }).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.

```

import { Component, EventEmitter, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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**.

```
import { Component, EventEmitter, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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(() => {
      })

    }).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/suscripcion/suscripcion.component.ts

```
import { Component, EventEmitter, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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 indicandole al usuario que se ha cancelado la suscripción.

/angular-observables-lab/src/app/suscripcion/suscripcion.component.ts

```

import { Component, EventEmitter, Input, OnInit } from '@angular/core';
import { Observable, Subscriber } from 'rxjs';
import { PagosService } from '../pagos.service';

@Component({
  selector: 'app-suscripcion',
  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)

      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 19. 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 obtenia 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 envian en el cuerpo de la petición.
- PUT: se usa para actualizar recursos ya existentes y que identificamos con la URI a la que se envia 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 envia 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 envia la propiedad completada, el titulo 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

19.1. 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 importar en el módulo el **HttpClientModule** de Angular.

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.module.ts

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

import { AppComponent } from './app.component';

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

Una vez añadido el módulo ya podemos utilizar el servicio de HttpClient. Solo necesitamos inyectar 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)

```
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')
  }
}
```

19.2. 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  
? 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 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 murcielago 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  
Done  

```

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 importar el módulo de **HttpClientModule** dentro del módulo de la aplicación.

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

import { AppComponent } from './app.component';

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

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.

```
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.

```
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() {

  }

  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**.

```
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** pasandole **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**.

```
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',
  templateUrl: './noticia.component.html',
  styleUrls: ['./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 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 { INoticia, NoticiasService } from './noticias.service';

@Component({
  selector: 'app-root',
  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 { INoticia, NoticiasService } from './noticias.service';

@Component({
  selector: 'app-root',
  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 **ngFor** este array sobre el componente **noticia**.

/angular-http-httpclient-lab/src/app/app.component.html

```
<h1>Noticias</h1>

<app-noticia *ngFor="let noticia of listaNoticias" [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. Usaremos un formulario reactivo para este, por lo que dentro del módulo de App tenemos que importar el módulo de **ReactiveFormsModule**.

/angular-http-httpclient-lab/src/app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { ReactiveFormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { NoticiaComponent } from './noticia/noticia.component';

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

Ahora dentro del componente App, vamos a crear el formulario que va a tener dos campos, el título y el contenido.

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
import { INoticia, NoticiasService } from './noticias.service';

@Component({
  selector: 'app-root',
  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
      })
  }
}
```

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).

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
import { INoticia, NoticiasService } from './noticias.service';

@Component({
  selector: 'app-root',
  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)
  }
}
```

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>

<app-noticia *ngFor="let noticia of listaNoticias" [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>

<app-noticia *ngFor="let noticia of listaNoticias" [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.

```
<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>

<app-noticia *ngFor="let noticia of listaNoticias" [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 ?.

```
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(noticia: INoticia) {

  }
}
```

Ahora dentro del método vamos a llamar a la petición **post** pasandole la URL de las noticias y como segundo parámetro la noticia que estamos recibiendo como parámetro.

```
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(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**.

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
import { INoticia, NoticiasService } from './noticias.service';

@Component({
  selector: 'app-root',
  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.

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
import { INoticia, NoticiasService } from './noticias.service';

@Component({
  selector: 'app-root',
  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.

19.3. Interceptores

Los **interceptores** nos permiten interceptar todas las peticiones HTTP y realizar alguna acción sobre ellas como por ejemplo modificar la petición para añadir alguna cabecera, modificar el cuerpo de la petición...

Los interceptores los creamos con alguno de los siguientes comandos:

```
$ ng generate interceptor nombre-interceptor  
$ ng g interceptor nombre-interceptor
```

Los interceptores son servicios que implementan el método **intercept** de la interfaz **HttpInterceptor**. Este método recibe dos parámetros:

- **req: HttpRequest** que es la petición realizada.
- **next: HttpHandler** que se encarga de llamar al siguiente interceptor que haya en la cadena de interceptores y pasarle la petición a través del método **handle(req)**. El interceptor tiene que devolver lo que devuelva este método que será un observable con la respuesta.

/src/app/log.interceptor.ts

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable()
export class LogInterceptor implements HttpInterceptor {

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {

    console.log(`Method: ${request.method}`);
    console.log(`URL: ${request.url}`);

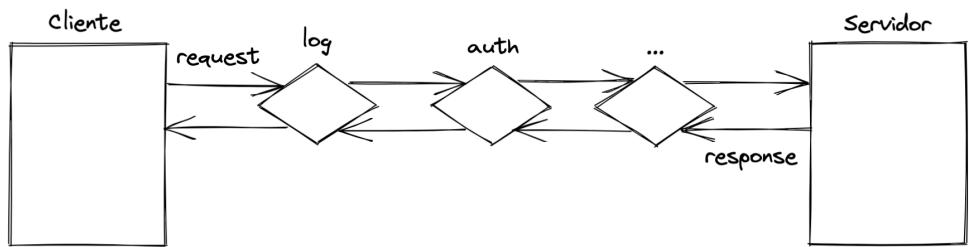
    return next.handle(request);
  }
}
```

Los interceptores son servicios, por lo tanto hay que añadirlos a los providers de la aplicación, y en este caso Angular será el que los ejecute cuando se realice una petición. De esto se encarga Angular y para que funcione correctamente y sepa que servicios son los interceptores, tenemos que registrarlos usando el Token de inyección de dependencias **HTTP_INTERCEPTORS**.

Al registrar los interceptores, necesitamos añadir la propiedad **multi: true** porque es la que va a permitir que se ejecuten más de uno para el mismo token de inyección. La propiedad hay que

añadirla obligatoriamente para que funcionen correctamente.

Interceptores



Los interceptores se van a ir ejecutando en el orden en que los hayamos definido en el array de providers del módulo.

/src/app/app.module.ts

```
import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { LogInterceptor } from './log.interceptor';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: LogInterceptor, multi: true }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

19.4. Lab: Interceptores

En este laboratorio vamos a ver como crear un interceptor para añadir el token de autenticación a las peticiones salientes.

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

```
$ ng new angular-http-interceptores-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 crearemos los siguientes elementos:

- Un servicio auth
- Un servicio de tareas
- Un interceptor auth

También vamos a levantar el servidor de desarrollo con los siguientes comandos.

Usaremos los siguientes comandos para realizar todos los pasos anteriores:

```
$ cd angular-http-interceptores-lab  
$ ng g s auth  
$ ng g s tareas  
$ ng g interceptor auth  
$ ng s
```

Vamos a empezar rellenando el servicio de auth en el que crearemos tres funciones, una para simular un login y guardar el token en el **localStorage** del navegador, otra para simular el logout y eliminar dicho token del localStorage, y la última para obtener el token del localStorage.

Añadimos una función **addToken** en la que crearemos un token de forma aleatoria con la función de **Math.random()**. Este número lo vamos a convertir en un string ya que es el tipo de dato que podemos guardar dentro del localStorage, y le vamos a quitar los dos primeros caracteres con el método **slice**, de esta forma nos quedamos solo con la parte decimal del número aleatorio.



Este token que estamos generando tendría que venir desde el servidor después de haber enviado un usuario y contraseña correctos en una petición POST.

/angular-http-interceptores-lab/src/app/auth.service.ts

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

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

  constructor() { }

  addToken(): void {
    const token = Math.random().toString().slice(2)

  }
}
```

Ahora que tenemos el token vamos a guardarlo en el **localStorage**, por lo que usaremos el método **setItem** del objeto **localStorage** que se encuentra en el navegador, y le vamos a pasar la clave **authToken** como primer parámetro, y como segundo parámetro el valor a guardar que es nuestro número aleatorio.

/angular-http-interceptores-lab/src/app/auth.service.ts

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

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

  constructor() { }

  addToken(): void {
    const token = Math.random().toString().slice(2)
    localStorage.setItem('authToken', token)
  }
}
```

Ahora vamos a añadir la siguiente función con la que queremos eliminar el **authToken** del **localStorage**, de tal forma que cuando se envíen peticiones HTTP al servidor, estas no incluyan ningún token de autenticación para indicarle al servidor que no estamos logueados.

Creamos la segunda función, esta vez vamos a llamar al método **removeItem** del **localStorage** y le vamos a pasar como parámetro el nombre de la clave que queremos eliminar, que en nuestro caso es **authToken**.

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

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

  constructor() { }

  addToken(): void {
    const token = Math.random().toString().slice(2)
    localStorage.setItem('authToken', token)
  }

  delToken(): void {
    localStorage.removeItem('authToken')
  }

}
```

Por último, vamos a añadir la tercera función con la que vamos a obtener el token, en el caso de que se encuentre la clave en el localStorage, o recibiremos un **null** en el caso de que no lo encuentre.

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

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

  constructor() { }

  addToken(): void {
    const token = Math.random().toString().slice(2)
    localStorage.setItem('authToken', token)
  }

  delToken(): void {
    localStorage.removeItem('authToken')
  }

  getToken(): string | null {
    return localStorage.getItem('authToken')
  }
}
```

Con esto ya tenemos nuestro servicio, el siguiente paso es añadir en el servicio de **tareas** una petición GET a <http://jsonplaceholder.typicode.com/todos>.

Pero antes de irnos al servicio, como necesitamos utilizar peticiones HTTP, vamos a tener que importar el **HttpClientModule** en el módulo de la aplicación.

/angular-http-interceptores-lab/src/app/app.module.ts

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

import { AppComponent } from './app.component';

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

Ahora ya podemos ir al servicio de tareas, donde vamos a inyectar la instancia de **HttpClient** en el constructor.

/angular-http-interceptores-lab/src/app/tareas.service.ts

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

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

  constructor(private http: HttpClient) { }

  getTareas(): Observable<any> {
  }
}
```

Ahora dentro del método de **getTareas** vamos a devolver el observable que devuelve la petición GET a la url que hemos indicado antes.

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

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

  constructor(private http: HttpClient) { }

  getTareas(): Observable<any> {
    return this.http.get('http://jsonplaceholder.typicode.com/todos')
  }
}
```

Para no devolver un observable de tipo **any**, vamos a crearnos una interfaz en el mismo servicio para definir la estructura de las tareas y poder indicar que el observable nos va a devolver un array con tareas.

Una tarea tiene los siguientes datos:

- **userId** e **id** que son números
- **title** que es un string
- **completed** que es un booleano

Esta interface la vamos a exportar para luego poder usarla en el resto de la aplicación.

```
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')
  }
}
```

Ahora tanto al tipo de retorno Observable le vamos a indicar que internamente los datos que va a devolvernos es un **array de ITarea**.

```
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<Array<ITarea>> {
    return this.http.get('http://jsonplaceholder.typicode.com/todos')
  }
}
```

Y ahora se queja la petición GET, por lo que le vamos a añadir también su tipo genérico para indicarle que los datos que vamos a obtener con esta petición GET es un **array de ITarea**.

/angular-http-interceptores-lab/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<Array<ITarea>> {
    return this.http.get<Array<ITarea>>('http://jsonplaceholder.typicode.com/todos')
  }
}
```

Una vez tenemos los dos servicios, vamos a ir al componente App en el que vamos a poner 2 botones, uno para loguearnos y otro para desloguearnos.

/angular-http-interceptores-lab/src/app/app.component.html

```
<button type="button" (click)="login()">Login</button>
<button type="button" (click)="logout()">Logout</button>
```

En el TypeScript del componente vamos a crear estas dos funciones, e injectaremos el servicio de auth en el constructor.

/angular-http-interceptores-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() {

  }

  logout() {

  }
}
```

Ahora dentro de la función de login, vamos a llamar a la función **addToken** del servicio de auth con la que vamos a crear el token.

/angular-http-interceptores-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() {
    this.auth.addToken()
  }

  logout() {

  }
}
```

En la función de logout llamaremos a la función de **delToken** del mismo servicio que antes.

```
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 {

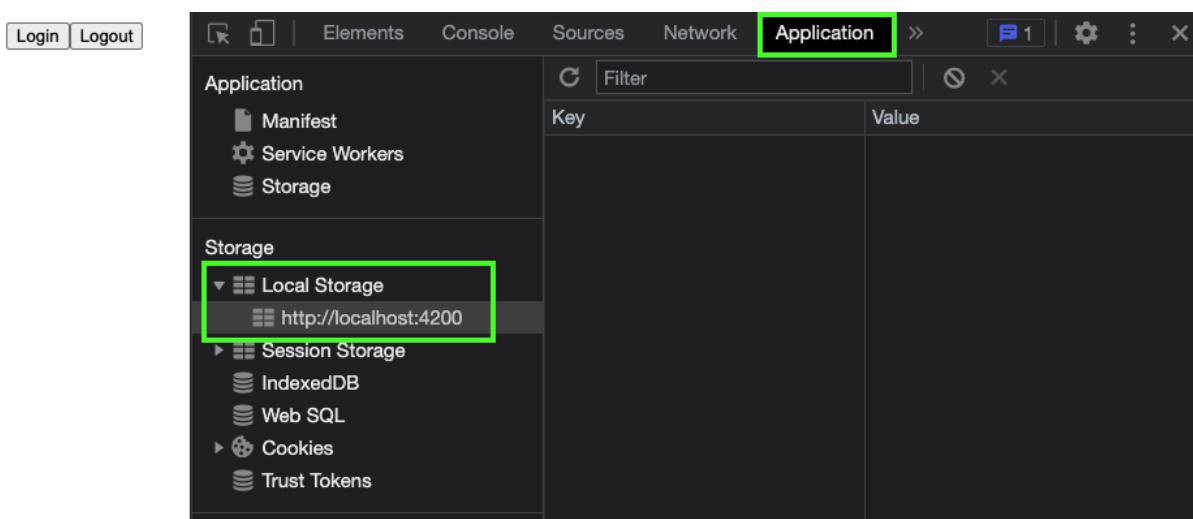
  constructor(private auth: AuthService) {}

  login() {
    this.auth.addToken()
  }

  logout() {
    this.auth.delToken()
  }
}
```

Vamos a comprobar que lo que hemos hecho con el localStorage funciona antes de continuar con el interceptor.

Entramos a la aplicación en <http://localhost:4200> y abrimos las herramientas del desarrollador. Una vez dentro vamos a **Application > Local Storage > http://localhost:4200** y se abre el panel con los datos que se guardan dentro del localStorage, que ahora mismo debería de estar vacío.



Si pulsamos sobre el botón de Login, debería de añadirse una fila con la clave **authToken** y como valor un número aleatorio.

Una vez tenemos este token, si pulsamos sobre el botón de **Logout**, debería de eliminarse del **localStorage**.

Ya que sabemos que esto funciona, así que toca realizar la petición GET dentro del método del ciclo de vida **ngOnInit** del componente App, para que esta petición se realice cuando se inicialice el componente.

/angular-http-interceptores-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 {

  constructor(private auth: AuthService) {}

  ngOnInit() {}

  login() {
    this.auth.addToken()
  }

  logout() {
    this.auth.delToken()
  }
}
```

Para poder llamar a la petición de **getTareas()**, necesitamos injectar primero la instancia del servicio de tareas en el constructor, de la misma forma que hemos hecho con el de auth.

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from './auth.service';
import { TareasService } from './tareas.service';

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

  constructor(private auth: AuthService, private tareas: TareasService) {}

  ngOnInit() {

  }

  login() {
    this.auth.addToken()
  }

  logout() {
    this.auth.delToken()
  }
}
```

El siguiente paso es ejecutar la función **getTareas**, y suscribirnos al observable que devuelve. Dentro de la suscripción recibiremos las tareas y lo único que vamos a hacer con ellas es mostrarlas por la consola del navegador.

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from './auth.service';
import { ITarea, TareasService } from './tareas.service';

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

  constructor(private auth: AuthService, private tareas: TareasService) {}

  ngOnInit() {
    this.tareas.getTareas()
      .subscribe((tareas: Array<ITarea>) => {
        console.log(tareas)
      })
  }

  login() {
    this.auth.addToken()
  }

  logout() {
    this.auth.delToken()
  }
}
```

Con esto ya estamos obteniendo el array con las tareas. Solo nos queda interceptar esta petición con nuestro interceptor de auth para añadirle la cabecera de **Authorization** con el valor del token que tengamos en el localStorage.

Dentro del interceptor recibimos la petición que se va a interceptar como parámetro (**request**), y estas peticiones son objetos inmutables, por lo que no podemos modificar sus datos.

En nuestro caso lo que queremos hacer es modificar sus cabeceras, por lo que clonaremos la petición y al mismo tiempo le indicaremos los cambios que queremos realizar, de esta forma no estaríamos mutando la petición inicial, sino que estamos creando una nueva con los cambios aplicados.

Vamos a empezar por comprobar si tenemos el token o no, y para ello tenemos que inyectar el servicio de **auth** dentro del constructor.

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { Observable } from 'rxjs';
import { AuthService } from './auth.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(private auth: AuthService) {}

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    const token = this.auth.getToken()

    return next.handle(request);
  }
}
```

En el caso de que el token exista vamos a crear el nuevo objeto con las **headers**. Para ello, vamos a utilizar el método **append** sobre las cabeceras de la petición inicial y le vamos a pasar como parámetros el nombre de la cabecera a añadir (que es **Authorization**) y como valor el token que vamos a pedir al servicio de auth.

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { Observable } from 'rxjs';
import { AuthService } from './auth.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(private auth: AuthService) {}

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    const token = this.auth.getToken()

    if (token) {
      const headersConAuthorization = request.headers.append('Authorization', token)
    }

    return next.handle(request);
  }
}
```

El siguiente paso es crear la nueva petición con estas cabeceras. El objeto de **request** tiene un método **clone** al que vamos a llamar y le vamos a pasar como parámetro un objeto con aquellas partes que queremos añadir en la petición clonada. En nuestro caso le vamos a indicar que queremos cambiar las **headers** por el nuevo objeto que acabamos de crear con ellas.

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { Observable } from 'rxjs';
import { AuthService } from './auth.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(private auth: AuthService) {}

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    const token = this.auth.getToken()

    if (token) {
      const headersConAuthorization = request.headers.append('Authorization', token)
      const requestConAuthorization = request.clone({ headers: headersConAuthorization })
    }

    return next.handle(request);
  }
}
```

Solo nos queda enviar esta nueva petición en lugar de enviar la inicial, por lo que dentro del if vamos a llamar a la función de **next.handle** pasandole como parámetro este nuevo objeto **request**.

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { Observable } from 'rxjs';
import { AuthService } from './auth.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(private auth: AuthService) {}

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    const token = this.auth.getToken()

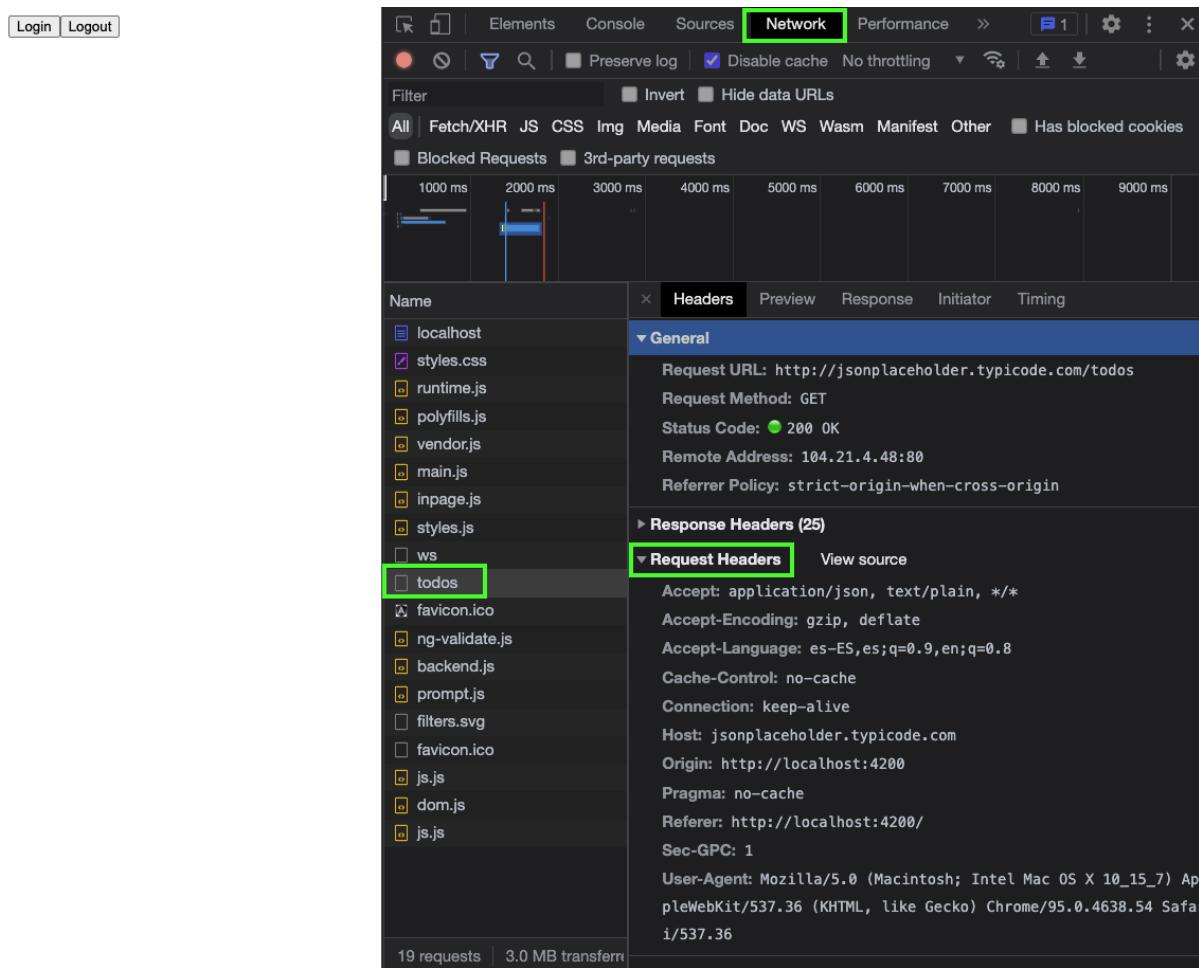
    if (token) {
      const headersConAuthorization = request.headers.append('Authorization', token)
      const requestConAuthorization = request.clone({ headers: headersConAuthorization })
      return next.handle(requestConAuthorization)
    }

    return next.handle(request);
  }
}
```

Y con esto ya tenemos el interceptor, vamos a ver si cuando inspeccionamos las peticiones esta lleva el token o no.

Entramos de nuevo a las herramientas del desarrollador del navegador y buscamos la pestaña de **Network**, refrescamos la página, y saldrán varias peticiones de distintos recursos. Tenemos que buscar una que pone **todos** y pulsar sobre ella.

Si miramos en la sección de **Request Headers** veremos que no aparece por ningún lado la de **Authorization**.



No se está añadiendo la cabecera, o más bien no se están interceptando las peticiones porque nos falta indicar en el módulo de la aplicación que cuando se pida el token de inyección **HTTP_INTERCEPTORS** se provea la instancia de la clase **AuthInterceptor**.

Esta configuración se la vamos a añadir dentro del array de **providers** del módulo App.

Además hay que añadirle la opción de **multi: true** porque el token **HTTP_INTERCEPTORS** puede proveer distintas instancias de distintas clases, y esta es la forma de indicarselo a Angular.

```

import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { AuthInterceptor } from './auth.interceptor';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Después de añadir el token y el interceptor en el módulo de la aplicación, ya deberíamos de ver la cabecera de **Authorization** al inspeccionar la petición GET igual que hemos hecho anteriormente. Pero para verla nos tenemos que asegurar de que le hemos dado al botón de Login.

Name	Headers	Preview	Response	Initiator	Timing
todos	General	Request URL: http://jsonplaceholder.typicode.com/todos Request Method: GET Status Code: 200 OK Remote Address: 172.67.131.170:80 Referrer Policy: strict-origin-when-cross-origin			
filters.svg	Response Headers (25)	Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate Accept-Language: es-ES,es;q=0.9,en;q=0.8 Authorization: 3813025417197351 Cache-Control: no-cache Connection: keep-alive Host: jsonplaceholder.typicode.com Origin: http://localhost:4200 Pragma: no-cache Referer: http://localhost:4200/ Sec-GPC: 1			

Si pulsamos sobre el botón de Logout e inspeccionamos de nuevo la petición, esta vez no deberíamos de ver la cabecera, ya que se ha eliminado el token del localStorage.

Chapter 20. 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.

20.1. 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**.

20.2. 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.

20.3. 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.

20.4. 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.

20.5. 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.

20.6. 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.

20.7. 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**.

20.8. 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.

20.9. 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 clases que van a implementar una interfaz dependiendo del tipo de guard que queramos ejecutar:

- **CanActivate**: se ejecuta antes de entrar a una ruta.
- **CanDeactivate**: se ejecuta al salir de una ruta.

Estas clases tendrán un método `canActivate` o `canDeactivate` que tendremos que implementar, los cuales tienen 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.
- ...

20.10. 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 },
]
```

Ahora le vamos a pasar este array de rutas de la aplicación al método **forRoot** del módulo **RouterModule** de Angular para que este sepa cuales son las rutas principales de la aplicación.

/angular-router-lab/src/app/app.routes.ts

```
import { RouterModule, 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 },
]

export const RoutingModule = RouterModule.forRoot(APP_ROUTES)
```

Con esto ya tenemos el módulo de rutas configurado y solo falta importarlo dentro del módulo raíz de la aplicación para que Angular sepa que vamos a usar el routing, y que esas rutas que hemos definido son las que se van a utilizar por ahora.

/angular-router-lab/src/app/app.module.ts

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

import { AppComponent } from './app.component';
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 { Error404Component } from './error404/error404.component';
import { RoutingModule } from './app.routes';

@NgModule({
  declarations: [
    AppComponent,
    UsuariosComponent,
    NuevoUsuarioComponent,
    InfoUsuarioComponent,
    EditarUsuarioComponent,
    Error404Component
  ],
  imports: [
    BrowserModule,
    RoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Ya que estamos con el módulo de la aplicación, vamos a aprovechar a importar también el módulo de las peticiones HTTP que usaremos más adelante en el servicio que ya hemos creado.

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

import { AppComponent } from './app.component';
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 { Error404Component } from './error404/error404.component';
import { RoutingModule } from './app.routes';

@NgModule({
  declarations: [
    AppComponent,
    UsuariosComponent,
    NuevoUsuarioComponent,
    InfoUsuarioComponent,
    EditarUsuarioComponent,
    Error404Component
  ],
  imports: [
    BrowserModule,
    RoutingModule,
    HttpClientModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

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.

```
<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.

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

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

  constructor(private router: Router) { }

  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',
  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',
  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>
  <li *ngFor="let usuario of listaUsuarios">
    {{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

/angular-router-lab/src/app/usuarios/usuarios.component.html

```
<h2>Usuarios</h2>

<ul>
  <li *ngFor="let usuario of listaUsuarios">
    {{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 propiedad **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**.

```
import { RouterModule, Routes } from "@angular/router";
import { NuevoUsuarioComponent } from "./nuevo-usuario/nuevo-usuario.component";
import { UsuariosComponent } from "./usuarios/usuarios.component";

const APP_ROUTES: Routes = [
  { path: 'usuarios', component: UsuariosComponent },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: 'usuarios', pathMatch: 'full' },
]

export const RoutingModule = RouterModule.forRoot(APP_ROUTES)
```

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 { RouterModule, Routes } from "@angular/router";
import { EditarUsuarioComponent } from "./editar-usuario/editar-usuario.component";
import { InfoUsuarioComponent } from "./info-usuario/info-usuario.component";
import { NuevoUsuarioComponent } from "./nuevo-usuario/nuevo-usuario.component";
import { UsuariosComponent } from "./usuarios/usuarios.component";

const USUARIOS_ROUTES: Routes = [
  { path: ':id/info', component: InfoUsuarioComponent },
  { path: ':id/editar', component: EditarUsuarioComponent },
]

const APP_ROUTES: Routes = [
  { path: 'usuarios', component: UsuariosComponent },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: 'usuarios', pathMatch: 'full' },
]

export const RoutingModule = RouterModule.forRoot(APP_ROUTES)
```

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 { RouterModule, Routes } from "@angular/router";
import { EditarUsuarioComponent } from "./editar-usuario/editar-usuario.component";
import { InfoUsuarioComponent } from "./info-usuario/info-usuario.component";
import { NuevoUsuarioComponent } from "./nuevo-usuario/nuevo-usuario.component";
import { UsuariosComponent } from "./usuarios/usuarios.component";

const USUARIOS_ROUTES: Routes = [
  { path: ':id/info', component: InfoUsuarioComponent },
  { path: ':id/editar', component: EditarUsuarioComponent },
]

const APP_ROUTES: Routes = [
  { path: 'usuarios', component: UsuariosComponent, children: USUARIOS_ROUTES },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: 'usuarios', pathMatch: 'full' },
]

export const RoutingModule = RouterModule.forRoot(APP_ROUTES)
```

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**.

/angular-router-lab/src/app/usuarios/usuarios.component.html

```
<h2>Usuarios</h2>

<ul>
  <li *ngFor="let usuario of listaUsuarios">
    {{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',
  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.

```
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',
  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.

```
<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 { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot, UrlTree } from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class InfoUsuarioGuard implements CanActivate {
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    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 { RouterModule, Routes } from "@angular/router";
import { EditarUsuarioComponent } from "./editar-usuario/editar-usuario.component";
import { InfoUsuarioComponent } from "./info-usuario/info-usuario.component";
import { InfoUsuarioGuard } from "./info-usuario/info-usuario.guard";
import { NuevoUsuarioComponent } from "./nuevo-usuario/nuevo-usuario.component";
import { UsuariosComponent } from "./usuarios/usuarios.component";

const USUARIOS_ROUTES: Routes = [
  { path: ':id/info', component: InfoUsuarioComponent, canActivate: [InfoUsuarioGuard] },
  { path: ':id/editar', component: EditarUsuarioComponent },
]

const APP_ROUTES: Routes = [
  { path: 'usuarios', component: UsuariosComponent, children: USUARIOS_ROUTES },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: 'usuarios', pathMatch: 'full' },
]

export const RoutingModule = RouterModule.forRoot(APP_ROUTES)
```

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.

/angular-router-lab/src/app/editar-usuario/editar-usuario.component.ts

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

@Component({
  selector: 'app-editar-usuario',
  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**.

/angular-router-lab/src/app/editar-usuario/editar-usuario.component.html

```
<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 { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanDeactivate, RouterStateSnapshot, UrlTree } from '@angular/router';
import { Observable } from 'rxjs';

export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean
}

@Injectable({
  providedIn: 'root'
})
export class EditarUsuarioGuard implements CanDeactivate<CanComponentDeactivate> {
  canDeactivate(
    component: CanComponentDeactivate,
    currentRoute: ActivatedRouteSnapshot,
    currentState: RouterStateSnapshot,
    nextState?: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    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 { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanDeactivate, RouterStateSnapshot, UrlTree } from '@angular/router';
import { Observable } from 'rxjs';

export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean
}

@Injectable({
  providedIn: 'root'
})
export class EditarUsuarioGuard implements CanDeactivate<CanComponentDeactivate> {
  canDeactivate(
    component: CanComponentDeactivate,
    currentRoute: ActivatedRouteSnapshot,
    currentState: RouterStateSnapshot,
    nextState?: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    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',
  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.

```
import { RouterModule, Routes } from "@angular/router";
import { EditarUsuarioComponent } from "./editar-usuario/editar-usuario.component";
import { EditarUsuarioGuard } from "./editar-usuario/editar-usuario.guard";
import { InfoUsuarioComponent } from "./info-usuario/info-usuario.component";
import { InfoUsuarioGuard } from "./info-usuario/info-usuario.guard";
import { NuevoUsuarioComponent } from "./nuevo-usuario/nuevo-usuario.component";
import { UsuariosComponent } from "./usuarios/usuarios.component";

const USUARIOS_ROUTES: Routes = [
  { path: ':id/info', component: InfoUsuarioComponent, canActivate: [InfoUsuarioGuard] },
  { path: ':id/editar', component: EditarUsuarioComponent, canDeactivate: [EditarUsuarioGuard] },
]

const APP_ROUTES: Routes = [
  { path: 'usuarios', component: UsuariosComponent, children: USUARIOS_ROUTES },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: 'usuarios', pathMatch: 'full' },
]

export const RoutingModule = RouterModule.forRoot(APP_ROUTES)
```

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.

```
<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.

```
import { RouterModule, Routes } from "@angular/router";
import { EditarUsuarioComponent } from "./editar-usuario/editar-usuario.component";
import { EditarUsuarioGuard } from "./editar-usuario/editar-usuario.guard";
import { Error404Component } from "./error404/error404.component";
import { InfoUsuarioComponent } from "./info-usuario/info-usuario.component";
import { InfoUsuarioGuard } from "./info-usuario/info-usuario.guard";
import { NuevoUsuarioComponent } from "./nuevo-usuario/nuevo-usuario.component";
import { UsuariosComponent } from "./usuarios/usuarios.component";

const USUARIOS_ROUTES: Routes = [
  { path: ':id/info', component: InfoUsuarioComponent, canActivate: [InfoUsuarioGuard] },
  { path: ':id/editar', component: EditarUsuarioComponent, canDeactivate: [EditarUsuarioGuard] },
]

const APP_ROUTES: Routes = [
  { path: 'usuarios', component: UsuariosComponent, children: USUARIOS_ROUTES },
  { path: 'nuevo-usuario', component: NuevoUsuarioComponent },
  { path: '', redirectTo: 'usuarios', pathMatch: 'full' },
  { path: '**', component: Error404Component },
]

export const RoutingModule = RouterModule.forRoot(APP_ROUTES)
```

Si probamos a entrar en <http://localhost:4200/no-existe> veremos que se muestra la página de error.

Chapter 21. Módulos

Los módulos de Angular proveen de un mecanismo para crear bloques de funcionalidad que se pueden combinar para crear una aplicación. Estos módulos agrupan componentes, directivas, pipes y servicios que están relacionados.

Para crear los módulos hay que hacer uso del decorador **NgModule**. Este decorador necesita al menos las siguientes propiedades: **imports**, **declarations** y **bootstrap**.

app/app.module.ts

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

@NgModule({
  imports: [],
  declarations: [],
  bootstrap: []
})

export class AppModule;
```

Creamos un componente y lo vamos a añadir al módulo anterior, que quedaría de la siguiente forma:

app/app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
    <h1>Hola mundo</h1>
  `
})
export class AppComponent { }
```

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

@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    AppComponent
  ],
  bootstrap: [AppComponent]
})

export class AppModule;
```

El **BrowserModule** es un módulo que exporta directivas básicas, servicios y pipes. Y al importarlo en nuestro módulo, vamos a tener acceso a toda esa funcionalidad que nos proporciona.

Hay dos tipos de módulos:

- **Módulos raíz**: solo hay uno por aplicación.
- **Módulos de funciones**: puede haber varios módulos de este tipo o ninguno.

21.1. Añadir elementos a un módulo

En el anterior apartado hemos visto como es un módulo. Pero los módulos pueden contener componentes, servicios y pipes, así que vamos a extender la funcionalidad del módulo que hemos creado.

Creamos un componente que va a mostrar un mensaje con un número de tarjeta de crédito. El número de la tarjeta aparecerá oculto excepto por los últimos cuatro dígitos que serán visibles.

app/credit-card.component.ts

```
import { Component, OnInit } from '@angular/core';
import { CreditCardService } from './credit-card.service';

@Component({
  selector: 'app-credit-card',
  template: `<p>La tarjeta de credito es: {{ numeroTarjetaCredito | ocultarDigitosTarjeta }}</p>`
})
export class CreditCardComponent implements OnInit {
  numeroTarjetaCredito: string;

  constructor(private creditCardService: CreditCardService) { }

  ngOnInit() {
    this.numeroTarjetaCredito = this.creditCardService.getTarjetaCredito();
  }
}
```

Ahora creamos el servicio que devolverá el número de la tarjeta de crédito al componente.

app/credit-card.service.ts

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

@Injectable()
export class CreditCardService {
  getTarjetaCredito(): string {
    return '6372 7434 1239 2499';
  }
}
```

Y por último creamos el pipe que se encargará de ocultar los dígitos de la tarjeta.

app/ocultar-digitos-tarjeta.pipe.ts

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

@Pipe({
  name: 'ocultarDigitosTarjeta'
})
export class OcultarDigitosTarjetaPipe implements PipeTransform {
  transform(numeroTarjetaCredito: string): string {
    const numDigitosVisibles = 4;
    let seccionOculta = numeroTarjetaCredito.slice(0, -numDigitosVisibles);
    let seccionVisible = numeroTarjetaCredito.slice(-numDigitosVisibles);
    return seccionOculta.replace(/\./g, '*') + seccionVisible;
  }
}
```

Ya solo queda usar este nuevo componente en el componente raíz.

app/app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
    <h1>Hola mundo</h1>
    <app-credit-card></app-credit-card>
  `
})
export class AppComponent { }
```

Para que esto funcione también tenemos que añadir lo que hemos creado en el archivo de módulo.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { CreditCardComponent } from './credit-card.component';
import { CreditCardService } from './credit-card.service';
import { OcultarDigitosTarjetaPipe } from '.ocultar-digitos-tarjeta.pipe';

@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    AppComponent,
    CreditCardComponent,
    OcultarDigitosTarjetaPipe
  ],
  providers: [CreditCardService],
  bootstrap: [AppComponent]
})

export class AppModule;
```

21.2. Creando un módulo de funcionalidad

Ahora vamos a crear un módulo de funcionalidad donde vamos a meter la parte de la tarjeta de crédito que hemos creado. De esta forma tendremos dos módulos, el módulo raíz, y el módulo de la tarjeta de crédito.

Empezamos por crear una carpeta **credit-card** donde meteremos la funcionalidad que hemos hecho hasta ahora correspondiente a la tarjeta de crédito y vamos a crear un módulo **credit-card.module.ts**.

```
$ ng g module credit-card
```

Y una vez creado el módulo, procedemos a llenar sus propiedades. Esta vez si que usaremos el **exports** para indicar que queremos exportar el componente de la tarjeta de crédito para poder usarlo en el componente raíz que se encuentra en el módulo raíz.

credit-card/credit-card.module.ts

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

import { CreditCardComponent } from './credit-card.component';
import { OcultarDigitosTarjetaPipe } from './credit-card.pipe';
import { CreditCardService } from './credit-card.service';

@NgModule({
  imports: [
    CommonModule
  ],
  providers: [CreditCardService],
  declarations: [
    CreditCardComponent,
    OcultarDigitosTarjetaPipe
  ],
  exports: [CreditCardComponent]
})
export class CreditCardModule { }
```

En este caso importa el **CommonModule** en lugar de importar el **BrowserModule** como se hace en el módulo raíz. Esto se debe a que este módulo no es el encargado de mostrar el componente, entonces no se necesita las propiedades que tiene de más el **BrowserModule** frente a este. El **CommonModule** nos da acceso a directivas, servicios y pipes básicos.

Solo nos queda usar este módulo dentro del módulo raíz, por lo que en el **app.module.ts** vamos a importar este módulo y podremos usar el componente de la tarjeta de crédito en el componente raíz.

app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';
import { CreditCardModule } from '../credit-card/credit-card.module';

import { AppComponent } from './app.component';

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

app/app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
    <h1>Tarjeta de crédito</h1>
    <app-credit-card></app-credit-card>
  `
})
export class AppComponent { }
```

Chapter 22. Transclusión

En Angular podemos utilizar la **transclusión**, que consiste en poder pasarle a un componente un código HTML que pueden ser etiquetas de HTML u otros componentes de Angular para utilizarlo dentro del componente.

Normalmente esto lo usamos cuando un mismo componente vamos a querer que muestre distintas cosas internamente, es decir, que inicialmente a la hora de construirlo no sabemos exactamente que van a querer mostrar aquellos usuarios que usen este componente.

Un ejemplo podría ser un carrusel en el que una vez podemos mostrar imágenes, otra vez podemos usarlo para mostrar componentes Card, otra lo usamos para mostrar textos...

Para poder usar la transclusión, lo único que tenemos que hacer es pasar el contenido a mostrar dentro del componente entre las etiquetas de este (la de apertura y la de cierre). Y luego indicaremos en qué parte del HTML del componente se tiene que inyectar este contenido con la etiqueta **ng-content**.

Además, se nos permite pasar más de un elemento, y como tal, podemos indicar distintas posiciones para ellos utilizando un **ng-content** principal, y luego otros secundarios.

Los secundarios llevan un atributo **select** con un selector CSS que apunta a qué parte de los elementos que se inyectan se tienen que poner en la posición de esta etiqueta.

22.1. Lab: Transclusión

En este laboratorio vamos a ver como crear un componente Acordeon en el que vamos a proyectar el contenido a mostrar desde el exterior.

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

```
$ ng new angular-transclusion-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 componente y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-transclusion-lab  
$ ng g c acordeon  
$ ng s
```

Empezaremos creando la estructura del componente acordeon.

En la plantilla vamos a tener dos secciones, una para la cabecera y la otra para el cuerpo del acordeón.

/angular-transclusion-lab/src/app/acordeon/acordeon.component.html

```
<div>  
  <div>  
    <h3>Aquí va el título</h3>  
  </div>  
  <div>  
    Aquí va el contenido  
  </div>  
</div>
```

El título lo vamos a recibir desde el componente superior con un **@Input**, por lo que vamos a añadir esta propiedad en el TypeScript del componente **Acordeon**.

/angular-transclusion-lab/src/app/acordeon/acordeon.component.ts

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

@Component({
  selector: 'app-acordeon',
  templateUrl: './acordeon.component.html',
  styleUrls: ['./acordeon.component.css']
})
export class AcordeonComponent implements OnInit {
  @Input() titulo: string = 'Título'

  constructor() { }

  ngOnInit(): void {
  }

}
```

La idea de este acordeón es que al pulsar sobre la cabecera se despliegue el contenido o se colapse, por lo que vamos a necesitar otra propiedad para controlar si hay que mostrarlo abierto o cerrado.

/angular-transclusion-lab/src/app/acordeon/acordeon.component.ts

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

@Component({
  selector: 'app-acordeon',
  templateUrl: './acordeon.component.html',
  styleUrls: ['./acordeon.component.css']
})
export class AcordeonComponent implements OnInit {
  @Input() titulo: string = 'Título'
  estaCerrado: boolean = false

  constructor() { }

  ngOnInit(): void {
  }

}
```

Vamos a añadir una función para ir cambiando el valor de **estaCerrado** y que así se pueda desplegar y colapsar el contenido.

/angular-transclusion-lab/src/app/acordeon/acordeon.component.ts

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

@Component({
  selector: 'app-acordeon',
  templateUrl: './acordeon.component.html',
  styleUrls: ['./acordeon.component.css']
})
export class AcordeonComponent implements OnInit {
  @Input() titulo: string = 'Título'
  estaCerrado: boolean = false

  constructor() { }

  ngOnInit(): void {
  }

  toggleAcordeon() {
    this.estaCerrado = !this.estaCerrado
  }
}
```

Esta función que acabamos de añadir se va a ejecutar cuando pulsemos sobre la cabecera del Acordeon, por lo que vamos a añadirle el evento **click** en la plantilla.

También cambiaremos el contenido de la cabecera para mostrar el título que vamos a recibir desde el componente superior.

/angular-transclusion-lab/src/app/acordeon/acordeon.component.html

```
<div>
  <div (click)="toggleAcordeon()">
    <h3>{{titulo}}</h3>
  </div>
  <div>
    Aquí va el contenido
  </div>
</div>
```

El siguiente paso es añadir la etiqueta **ng-content** dentro del **div** que mostrará el contenido. De esta forma, vamos a poder proyectar el contenido que queremos mostrar desde fuera, haciendo que lo que se muestre no tenga la misma estructura siempre.

/angular-transclusion-lab/src/app/acordeon/acordeon.component.html

```
<div>
  <div (click)="toggleAcordeon()">
    <h3>{{titulo}}</h3>
  </div>
  <div>
    <ng-content></ng-content>
  </div>
</div>
```

Hasta aquí, ya podemos ir a probar el componente, a ver si se muestran distintos elementos dentro de él.

En el componente **App** vamos a utilizar este componente, una vez para mostrar una lista de datos, y otra para mostrar un párrafo.

Estos dos elementos se los vamos a pasar entre las etiquetas de apertura y cierre de **app-acordeon**, ya que es lo que se va a mostrar donde hemos puesto el **ng-content**.

/angular-transclusion-lab/src/app/app.component.html

```
<app-acordeon titulo="Lista de productos">
  <ul>
    <li>Albahaca</li>
    <li>Queso parmesano</li>
    <li>Pechuga de pollo</li>
    <li>Tomates</li>
  </ul>
</app-acordeon>

<app-acordeon titulo="Frase de Big Bang Theory">
  <p>¿El amor está en el aire? Erróneo. El nitrógeno, el oxígeno y el dióxido de carbono están en el aire.</p>
</app-acordeon>
```

Ya deberíamos de poder ver el contenido de los dos acordeones.

Ahora vamos a añadirle unas clases y estilos al componente para que mejore su apariencia y que se oculte o muestre el contenido al pulsar sobre la cabecera de estos.

Vamos a añadirle las siguientes clases fijas al Acordeón.

/angular-transclusion-lab/src/app/acordeon/acordeon.component.html

```
<div class="acordeon">
  <div class="acordeon-heading" (click)="toggleAcordeon()">
    <h3>{{titulo}}</h3>
  </div>
  <div class="acordeon-content">
    <ng-content></ng-content>
  </div>
</div>
```

Y también le vamos a añadir unas clases **cerrado** y **abierto**, pero esta vez con la directiva **ngClass**, ya que estas se van a activar o desactivar dependiendo del valor de la propiedad **estaCerrado**.

/angular-transclusion-lab/src/app/acordeon/acordeon.component.html

```
<div class="acordeon">
  <div class="acordeon-heading" (click)="toggleAcordeon()">
    <h3>{{titulo}}</h3>
  </div>
  <div class="acordeon-content" [ngClass]="{cerrado: estaCerrado, abierto: !estaCerrado}">
    <ng-content></ng-content>
  </div>
</div>
```

Ahora nos vamos al archivo del CSS para añadir los estilos de todas estas clases.

/angular-transclusion-lab/src/app/acordeon/acordeon.component.css

```
.acordeon {  
  border: 1px solid black;  
  border-radius: 5px;  
}  
  
.acordeon-heading {  
  text-align: center;  
  border-bottom: 2px solid black;  
  cursor: pointer;  
}  
  
.acordeon-content {  
  overflow: hidden;  
}  
  
.acordeon-content.cerrado {  
  height: 0;  
}  
  
.acordeon-content.abierto {  
  height: auto;  
  padding: 20px;  
}
```

Después de hacer esto, ya deberíamos de poder desplegar y colapsar los acordeones.

Imaginemos que ahora queremos proyectar otro tipo de contenido dentro del acordeón, pero en otro lugar distinto.

Vamos a hacerlo metiendo este contenido entre las etiquetas de apertura y cierre, al igual que antes, pero esta vez le vamos a poner algo identificativo como una clase.

Vamos a añadir otro párrafo con el autor de la frase del segundo acordeón.

/angular-transclusion-lab/src/app/app.component.html

```
<app-acordeon titulo="Lista de productos">  
  <ul>  
    <li>Albahaca</li>  
    <li>Queso parmesano</li>  
    <li>Pechuga de pollo</li>  
    <li>Tomates</li>  
  </ul>  
</app-acordeon>  
  
<app-acordeon titulo="Frase de Big Bang Theory">  
  <p>¿El amor está en el aire? Erróneo. El nitrógeno, el oxígeno y el dióxido de carbono están en el aire.</p>  
  <p class="ac-footer">Sheldon Cooper</p>  
</app-acordeon>
```

Para mostrar esto nuevo en alguna parte de la plantilla del Acordeon, vamos a añadirle otra caja dentro del contenido y dentro de esta pondremos otro **ng-content**, pero esta vez con una propiedad **select** cuyo valor será el selector del elemento que se va a proyectar y que queremos pintar en este otro **ng-content**.

/angular-transclusion-lab/src/app/acordeon/acordeon.component.html

```
<div class="acordeon">
  <div class="acordeon-heading" (click)="toggleAcordeon()">
    <h3>{{titulo}}</h3>
  </div>
  <div class="acordeon-content" [ngClass]="{cerrado: estaCerrado, abierto: !estaCerrado}">
    <ng-content></ng-content>
    <div class="acordeon-footer">
      <ng-content select=".ac-footer"></ng-content>
    </div>
  </div>
</div>
```

Por último, vamos a añadirle los estilos para esta nueva clase que hemos añadido.

```
.acordeon {  
  border: 1px solid black;  
  border-radius: 5px;  
}  
  
.acordeon-heading {  
  text-align: center;  
  border-bottom: 2px solid black;  
  cursor: pointer;  
}  
  
.acordeon-content {  
  overflow: hidden;  
}  
  
.acordeon-content.cerrado {  
  height: 0;  
}  
  
.acordeon-content.abierto {  
  height: auto;  
  padding: 20px;  
}  
  
.acordeon-footer {  
  text-align: right;  
  font-style: italic;  
}
```

Ya deberíamos ver este footer dentro del segundo acordeón, y con los estilos aplicados.

Chapter 23. Decorador ViewChild

El decorador ViewChild es un decorador de propiedad, por lo que se le asigna a las propiedades de los componentes.

Con este decorador podemos acceder al primer elemento o directiva que hace match con el selector que se le pasa como parámetro.

El decorador acepta como selectores principalmente los siguientes, aunque hay alguno más:

- Clases de componentes y directivas: **@ViewChild(MiComponente) prop1: MiComponente.**
- Referencias o variables de plantilla: **@ViewChild('miRef') prop2: any**

Los valores a los que apunta este decorador se asignan después del método del ciclo de vida **ngAfterViewInit**, por lo que si necesitamos acceder a las propiedades que devuelve al cargar el componente, tendremos que hacerlo dentro de este método en lugar de hacerlo en el **ngOnInit**.

23.1. Lab: ViewChild

En este laboratorio vamos a ver como crear un modal y acceder a sus propiedades y métodos desde el exterior con el decorador `@ViewChild`.

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

```
$ ng new angular-viewchild-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 componente y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-viewchild-lab  
$ ng g c modal  
$ ng s
```

Vamos a empezar por implementar el modal.

Dentro de la plantilla, vamos a tener dos **divs**, uno que hará de backdrop (la sombra que aparece por detrás de la caja del modal) y el otro para la propia caja que va a tener el contenido.

/angular-viewchild-lab/src/app/modal/modal.component.html

```
<div class="backdrop">  
  <div id="modal">  
  
    </div>  
  </div>
```

Dentro del modal, vamos a añadir un botón para cerrarlo y la etiqueta **ng-content** con la que inyectaremos el contenido a mostrar desde el exterior, usando la transclusión.

/angular-viewchild-lab/src/app/modal/modal.component.html

```
<div class="backdrop">  
  <div id="modal">  
    <ng-content></ng-content>  
    <button type="button">Cerrar</button>  
  </div>  
</div>
```

Ahora vamos a añadir en el TypeScript del modal una propiedad **hidden** con la que vamos a controlar si se tiene que mostrar o no el modal.

También vamos a añadir dos métodos, **abrir** y **cerrar** para cambiar el valor de la propiedad anterior.

/angular-viewchild-lab/src/app/modal/modal.component.ts

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

@Component({
  selector: 'app-modal',
  templateUrl: './modal.component.html',
  styleUrls: ['./modal.component.css']
})
export class ModalComponent implements OnInit {
  hidden: boolean = true
  constructor() { }

  ngOnInit(): void {
  }

  abrir(): void {
    this.hidden = false
  }

  cerrar(): void {
    this.hidden = true
  }
}
```

Ahora pondremos dos eventos click para llamar a **cerrar**, uno en el backdrop y otro en el botón.

/angular-viewchild-lab/src/app/modal/modal.component.html

```
<div class="backdrop" (click)="cerrar()">
  <div id="modal">
    <ng-content></ng-content>
    <button type="button" (click)="cerrar()">Cerrar</button>
  </div>
</div>
```

También vamos a añadir la directiva **ngIf** sobre el backdrop para eliminar todo el contenido cuando se tiene que ocultar el modal.

```
/angular-viewchild-lab/src/app/modal/modal.component.html
```

```
<div class="backdrop" (click)="cerrar()" *ngIf="!hidden">
  <div id="modal">
    <ng-content></ng-content>
    <button type="button" (click)="cerrar()>Cerrar</button>
  </div>
</div>
```

Vamos a añadir unos estilos para el modal y el backdrop en su CSS.

```
/angular-viewchild-lab/src/app/modal/modal.component.css
```

```
div.backdrop {
  position: fixed;
  background-color: rgba(0, 0, 0, 0.8);
  width: 100vw;
  height: 100vh;
  display: flex;
  justify-content: center;
  align-items: center;
}

div#modal {
  width: 40%;
  margin: 0 auto;
  background-color: white;
  padding: 20px;
  border-radius: 10px;
}
```

Pues ahora nos vamos a ir al componente App, en el que vamos a pintar este modal, con algo de contenido dentro, el cual tenemos que pasar entre las dos etiquetas de **app-modal** para inyectarlo en el **ng-content** que hemos puesto dentro.

```
/angular-viewchild-lab/src/app/app.component.html
```

```
<app-modal>
  <h4>Hakuna matata, vive y se feliz</h4>
</app-modal>
```

También añadiremos un botón para poder abrir el modal.

/angular-viewchild-lab/src/app/app.component.html

```
<app-modal>
  <h4>Hakuna matata, vive y se feliz</h4>
</app-modal>

<button type="button" (click)="abrirModal()">Abrir modal</button>
```

Dentro de su TypeScript tenemos que añadir la función **abrirModal**.

/angular-viewchild-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { ModalComponent } from './modal/modal.component';

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

  abrirModal() {
    }
}
```

Para poder abrir el modal desde esta función, necesitamos poder acceder al componente del modal, y para ello vamos a utilizar el decorador **ViewChild**.

Este decorador necesita recibir como parámetro el nombre de una variable de plantilla que tenemos que poner en la etiqueta del componente, o la otra opción es pasarle el nombre de la clase del componente.

Vamos a usar la de la variable de plantilla, por lo que se la vamos a añadir en la etiqueta **app-modal**.

/angular-viewchild-lab/src/app/app.component.html

```
<app-modal #modal>
  <h4>Hakuna matata, vive y se feliz</h4>
</app-modal>

<button type="button" (click)="abrirModal()">Abrir modal</button>
```

Una vez tenemos la variable de plantilla, vamos a añadir el decorador y le pasaremos como parámetro **modal**, y con el declararemos una propiedad con el mismo nombre.

/angular-viewchild-lab/src/app/app.component.ts

```
import { Component, ViewChild } from '@angular/core';
import { ModalComponent } from './modal/modal.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  @ViewChild('modal') modal!: ModalComponent;

  abrirModal() {

  }
}
```

Solo nos queda llamar al método **abrir** del modal dentro del método **abrirModal** del componente App.

/angular-viewchild-lab/src/app/app.component.ts

```
import { Component, ViewChild } from '@angular/core';
import { ModalComponent } from './modal/modal.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  @ViewChild('modal') modal!: ModalComponent;

  abrirModal() {
    this.modal.abrir()
  }
}
```

Si entramos en la aplicación <http://localhost:4200/>, ya debemos de poder abrir y cerrar el modal.

Chapter 24. Componentes dinámicos

Cuando usamos componentes dentro de las aplicaciones de Angular, estos los ponemos en las plantillas de otros componentes, por lo tanto, tienen una posición fija, dada por el lugar en el que ponemos la etiqueta de estos.

Pues en Angular podemos utilizar otro tipo de componentes, estos son los **componentes dinámicos**, y la diferencia con los anteriores es que no tienen esa posición fija, es decir, no los vamos a añadir en nuestra aplicación poniendo su etiqueta dentro de la plantilla de otro componente.

Estos componentes se crean mediante código, y se van a pintar en un sitio u otro en tiempo de ejecución.

Por ejemplo, un ejemplo de este tipo de componentes son el contenido de los modales, ya que en lugar de crear distintos modales que pinten distintos componentes, podríamos crear un único modal en el que vamos a pintar los componentes de forma dinámica.

Para utilizar este tipo de componentes, lo primero que necesitamos es una forma de acceder a la posición donde se van a añadir. Para ello, lo normal es crear una directiva e injectarle **ViewContainerRef**.

El **ViewContainerRef** es una referencia a un contenedor en el que podremos añadir y eliminar estos componentes dinámicos.

Esta directiva tendremos que añadirla sobre un elemento (puede ser también un **ng-template**), y desde el componente al cual pertenezca la plantilla sobre la que se ha usado esta directiva, nos vamos a encargar de indicar que tiene que pintar.

Esto lo haremos accediendo al **host**, elemento en el que hemos puesto la directiva, y llamando a los métodos de la instancia de **ViewContainerRef**. Estos métodos son:

- **createComponent**: crea el componente que se le pasa como parámetro en la posición del host.
- **clear**: limpia el contenido que hay pintado en el host.

24.1. Lab: Componentes dinámicos

En este laboratorio vamos a mostrar una lista de audios y videos que se cargarán como componentes dinámicos dentro de un modal según pulsemos sobre ellos.

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

```
$ ng new angular-componentes-dinamicos-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 tres componentes, una directiva y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-componentes-dinamicos-lab  
$ ng g c modal  
$ ng g c video  
$ ng g c audio  
$ ng g d host  
$ ng s
```

Lo primero que vamos a hacer es crear en la carpeta de assets dos carpetas, una de videos y otra de audios, y después descargarnos 3 videos y 3 audios de las siguientes páginas para guardarlos en las carpetas:

- Audios: <https://licensing.jamendo.com/es/catalogo>
- Videos: <https://www.pexels.com/videos>

Ahora nos vamos a ir al TypeScript del componente App, donde vamos a crear una interfaz **IMedia** para definir el tipo de los videos y audios y que propiedades van a tener.

Estos items tendrán:

- src: la ruta hasta el archivo.
- titulo: el título que vamos a mostrar.
- tipo: un string audio o video para saber de qué tipo es el elemento que vamos a reproducir.

/angular-componentes-dinamicos-lab/src/app/app.component.ts

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

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

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

}
```

Ahora vamos a crear dos propiedades, una, **audios** con un array en el que vamos a añadir los datos para los audios que hemos descargado, y haremos lo mismo con **videos**.

/angular-componentes-dinamicos-lab/src/app/app.component.ts

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

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'assets/videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'assets/videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'assets/videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'assets/audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'assets/audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'assets/audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
}
```

En la plantilla vamos a mostrar estas dos listas y les pondremos un evento **click** a cada item para llamar a una función **mostrarMedia** a la que le pasaremos el objeto sobre el que hemos pulsado.

/angular-componentes-dinamicos-lab/src/app/app.component.html

```
<ul>
  <li *ngFor="let v of videos" (click)="mostrarMedia(v)">{{v.titulo}}</li>
</ul>

<ul>
  <li *ngFor="let a of audios" (click)="mostrarMedia(a)">{{a.titulo}}</li>
</ul>
```

Ahora vamos a añadir el componente del modal al que le pondremos una variable de plantilla para poder acceder después a él.

Y mediante transclusión le vamos a inyectar una plantilla con la directiva **HostDirective**.

/angular-componentes-dinamicos-lab/src/app/app.component.html

```
<app-modal #modal>
  <ng-template appHost></ng-template>
</app-modal>

<ul>
  <li *ngFor="let v of videos" (click)="mostrarMedia(v)">{{v.titulo}}</li>
</ul>

<ul>
  <li *ngFor="let a of audios" (click)="mostrarMedia(a)">{{a.titulo}}</li>
</ul>
```

Vamos a poner en el TypeScript la función de **mostrarMedia** y con el decorador **ViewChild** vamos a buscar tanto el modal como la directiva.

/angular-componentes-dinamicos-lab/src/app/app.component.ts

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'assets/videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'assets/videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'assets/videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'assets/audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'assets/audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'assets/audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
  @ViewChild('modal') modal!: ModalComponent;

  mostrarMedia(itemMedia: IMedia): void {
  }
}
```

Antes de continuar por aquí, vamos a inyectar el **ViewContainerRef** en la directiva.

/angular-componentes-dinamicos-lab/src/app/host.directive.ts

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

@Directive({
  selector: '[appHost]'
})
export class HostDirective {

  constructor(public viewContainerRef: ViewContainerRef) { }

}
```

También tenemos que cambiar el contenido de los componentes **AudioComponent** y **VideoComponent** a los que les declararemos una propiedad **src** que usaremos para mostrar estos items en las etiquetas **audio** y **video**.

/angular-componentes-dinamicos-lab/src/app/video/video.component.ts

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

@Component({
  selector: 'app-video',
  templateUrl: './video.component.html',
  styleUrls: ['./video.component.css']
})
export class VideoComponent implements OnInit {
  @Input() src: string = ''
  constructor() { }

  ngOnInit(): void {
  }

}
```

/angular-componentes-dinamicos-lab/src/app/video/video.component.html

```
<video [src]="src" controls width="200"></video>
```

/angular-componentes-dinamicos-lab/src/app/audio/audio.component.ts

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

@Component({
  selector: 'app-audio',
  templateUrl: './audio.component.html',
  styleUrls: ['./audio.component.css']
})
export class AudioComponent implements OnInit {
  @Input() src: string = ''

  constructor() { }

  ngOnInit(): void {
  }

}
```

/angular-componentes-dinamicos-lab/src/app/audio/audio.component.html

```
<audio [src]="src" controls></audio>
```

El siguiente paso es crear el modal. Dentro del TypeScript vamos a añadir la propiedad **hidden** que la usaremos para pintar o eliminar el modal y dos métodos para cambiar el valor de esta propiedad.

/angular-componentes-dinamicos-lab/src/app/modal/modal.component.ts

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

@Component({
  selector: 'app-modal',
  templateUrl: './modal.component.html',
  styleUrls: ['./modal.component.css']
})
export class ModalComponent implements OnInit {
  hidden: boolean = true
  constructor() { }

  ngOnInit(): void {
  }

  abrir(): void {
    this.hidden = false
  }

  cerrar(): void {
    this.hidden = true
  }
}
```

En la plantilla, pondremos dos divs, en el primero el **ngIf** con la propiedad **hidden** para controlar cuando mostrar el modal.

En el div interno vamos a añadir el **ng-content** para pintar el contenido que se le va a inyectar, y un botón con el que llamaremos al método que cierra el modal.

/angular-componentes-dinamicos-lab/src/app/modal/modal.component.html

```
<div class="backdrop" *ngIf="!hidden">
  <div id="modal">
    <ng-content></ng-content>
    <button type="button" (click)="cerrar()">Cerrar</button>
  </div>
</div>
```

Y ahora la ponemos unos estilos para que se vea mejor.

```
div.backdrop {  
  position: fixed;  
  background-color: rgba(0, 0, 0, 0.8);  
  width: 100vw;  
  height: 100vh;  
  display: flex;  
  justify-content: center;  
  align-items: center;  
}  
  
div#modal {  
  width: 40%;  
  margin: 0 auto;  
  background-color: white;  
  padding: 20px;  
  border-radius: 10px;  
}
```

Pues ya tenemos todos los elementos necesarios para añadir los videos y audios dentro del modal como componentes dinámicos.

Esto lo vamos a hacer en el método **mostrarMedia** que habíamos puesto en el componente App.

Vamos a empezar por acceder al **ViewContainerRef** de la directiva **Host** y llamaremos a **clear** para eliminar el contenido anterior si lo hubiese.

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'assets/videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'assets/videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'assets/videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'assets/audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'assets/audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'assets/audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
  @ViewChild('modal') modal!: ModalComponent;

  mostrarMedia(itemMedia: IMedia): void {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
  }
}
```

Ahora vamos a obtener el tipo del componente que queremos crear en el modal.

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';
import { AudioComponent } from './audio/audio.component';
import { VideoComponent } from './video/video.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'assets/videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'assets/videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'assets/videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'assets/audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'assets/audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'assets/audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
  @ViewChild('modal') modal!: ModalComponent;

  mostrarMedia(itemMedia: IMedia): void {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    const tipoCmp = itemMedia.tipo === 'video' ? VideoComponent : AudioComponent
  }
}
```

El siguiente paso es crear el componente, para lo que llamaremos al método **createComponent** del **viewContainerRef** pasándole como parámetro el tipo del componente que queremos crear.

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';
import { AudioComponent } from './audio/audio.component';
import { VideoComponent } from './video/video.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'assets/videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'assets/videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'assets/videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'assets/audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'assets/audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'assets/audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
  @ViewChild('modal') modal!: ModalComponent;

  mostrarMedia(itemMedia: IMedia): void {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    const tipoCmp = itemMedia.tipo === 'video' ? VideoComponent : AudioComponent
    const componente: any = viewContainerRef.createComponent(tipoCmp)
  }
}
```

Ahora accedemos a la propiedad **src** a través de la instancia del componente que hemos creado para asignarle la ruta donde se encuentra el audio o video que se va a reproducir.

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';
import { AudioComponent } from './audio/audio.component';
import { VideoComponent } from './video/video.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'assets/videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'assets/videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'assets/videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'assets/audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'assets/audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'assets/audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
  @ViewChild('modal') modal!: ModalComponent;

  mostrarMedia(itemMedia: IMedia): void {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    const tipoCmp = itemMedia.tipo === 'video' ? VideoComponent : AudioComponent
    const componente: any = viewContainerRef.createComponent(tipoCmp)
    componente.instance.src = itemMedia.src
  }
}
```

Y por último, solo nos queda abrir el modal. Como tenemos acceso a el por el **ViewChild**, vamos a llamar al método **abrir** que habíamos añadido dentro de el.

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from './host.directive';
import { ModalComponent } from './modal/modal.component';
import { AudioComponent } from './audio/audio.component';
import { VideoComponent } from './video/video.component';

interface IMedia {
  src: string,
  titulo: string,
  tipo: 'video' | 'audio'
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  videos: Array<IMedia> = [
    { src: 'assets/videos/video1.mp4', titulo: 'Puesta de sol silueta - Maksim Goncharenok', tipo: 'video' },
    { src: 'assets/videos/video2.mp4', titulo: 'Vela a la lluvia - Beytlik', tipo: 'video' },
    { src: 'assets/videos/video3.mp4', titulo: 'Flores al viento - Anastasia Ilina-Makarova', tipo: 'video' },
  ]
  audios: Array<IMedia> = [
    { src: 'assets/audios/audio1.mp3', titulo: 'Dance Endlessly - TimTaj', tipo: 'audio' },
    { src: 'assets/audios/audio2.mp3', titulo: 'Feel Good - MatFix', tipo: 'audio' },
    { src: 'assets/audios/audio3.mp3', titulo: 'Epic Trailer - Dmitry Taras', tipo: 'audio' },
  ]
  @ViewChild(HostDirective) host!: HostDirective
  @ViewChild('modal') modal!: ModalComponent;

  mostrarMedia(itemMedia: IMedia): void {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    const tipoCmp = itemMedia.tipo === 'video' ? VideoComponent : AudioComponent
    const componente: any = viewContainerRef.createComponent(tipoCmp)
    componente.instance.src = itemMedia.src
    this.modal.abrir()
  }
}
```

Y con esto ya deberíamos de poder abrir el modal y ver el video o escuchar un audio al pulsar sobre ellos en el listado que se está mostrando.

24.2. Lab: Componentes dinámicos

En este laboratorio vamos a ver como crear nuestro propio modulo de Routing de Angular usando los componentes dinámicos.

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

```
$ ng new angular-componentes-dinamicos-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 y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-componentes-dinamicos-router-lab  
$ ng g m mi-router  
$ ng g d mi-router/host  
$ ng g d mi-router/mi-router-link  
$ ng g s mi-router/mi-router  
$ ng g c mi-router/mi-router-outlet  
$ ng g c inicio  
$ ng g c fin
```

Vamos a empezar rellenando la directiva **host** en la que vamos a inyectar de forma pública **ViewContainerRef** que usaremos en nuestro componente **mi-router-outlet** después.

Con el **ViewContainerRef** podremos agregar los componentes dentro de las plantillas donde apliquemos la directiva.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/host.directive.ts

```
import { Directive, ViewContainerRef } from '@angular/core';  
  
@Directive({  
  selector: '[appHost]'  
)  
export class HostDirective {  
  
  constructor(public viewContainerRef: ViewContainerRef) {}  
  
}
```

Esta directiva la vamos a utilizar dentro de nuestro componente **mi-router-outlet**, en el que pondremos un **ng-template** con dicha directiva.

Esta plantilla será el lugar donde vamos a añadir los componentes dinámicos cuando cambiemos entre las distintas rutas de la aplicación, y lo haremos con el **viewContainerRef** que hemos creado

en la directiva **host**.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-outlet/mi-router-outlet.component.html

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

Ahora en el archivo de TypeScript del componente vamos a añadir la parte para agregar y quitar los componentes en el lugar donde se ha añadido la directiva **host**.

Empezamos por obtener dicha directiva usando el decorador **@ViewChild** de Angular.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-outlet/mi-router-outlet.component.ts

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from '../host.directive';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent {

  @ViewChild(HostDirective) host!: HostDirective

  constructor() { }

}
```

Ahora vamos a añadir un método **crearComponente** al que le vamos a pasar el componente que queremos renderizar.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-outlet/mi-router-outlet.component.ts

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from '../host.directive';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent {

  @ViewChild(HostDirective) host!: HostDirective

  constructor() { }

  crearComponente(cmp: any) {
    }
}
```

Dentro de esta función vamos a obtener primero la referencia al contenedor en el que queremos pintar el componente. Esto es lo que nos va a devolver el **viewContainerRef** que tenemos en la directiva **host**.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-outlet/mi-router-outlet.component.ts

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from '../host.directive';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent {

  @ViewChild(HostDirective) host!: HostDirective

  constructor() { }

  crearComponente(cmp: any) {
    const viewContainerRef = this.host.viewContainerRef

    }
}
```

Antes de añadir el componente a pintar, vamos a limpiar el contenedor, por si acaso ya se estaba

pintando algún componente dentro de este. Para esto vamos a llamar a la función **clear**.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-outlet/mi-router-outlet.component.ts

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from '../host.directive';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent {

  @ViewChild(HostDirective) host!: HostDirective

  constructor() { }

  crearComponente(cmp: any) {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()

  }
}
```

Y por último, vamos a utilizar la función **createComponent** pasándole como parámetro el componente que queremos pintar.

```
import { Component, ViewChild } from '@angular/core';
import { HostDirective } from '../host.directive';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent {

  @ViewChild(HostDirective) host!: HostDirective

  constructor() { }

  crearComponente(cmp: any) {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    viewContainerRef.createComponent(cmp)
  }
}
```

Nos toca llamar a esta función para probar que funciona correctamente.

Vamos a añadir el método del ciclo de vida **ngAfterViewInit** en el cual se ejecuta cuando TODO:

También vamos a importar el componente de Inicio para pintarlo.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-outlet/mi-router-outlet.component.ts

```
import { Component, ViewChild, AfterViewInit } from '@angular/core';
import { HostDirective } from '../host.directive';
import { InicioComponent } from '../../inicio/inicio.component';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent implements AfterViewInit {

  @ViewChild(HostDirective) host!: HostDirective

  constructor() { }

  ngAfterViewInit(): void {
    }

    crearComponente(cmp: any) {
      const viewContainerRef = this.host.viewContainerRef
      viewContainerRef.clear()
      viewContainerRef.createComponent(cmp)
    }
}
```

De momento vamos a hacer que se ejecute la función de **crearComponente** cuando pasen 2 segundos.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-outlet/mi-router-outlet.component.ts

```
import { Component, ViewChild, AfterViewInit } from '@angular/core';
import { HostDirective } from '../host.directive';
import { InicioComponent } from '../../inicio/inicio.component';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent implements AfterViewInit {

  @ViewChild(HostDirective) host!: HostDirective

  constructor() { }

  ngAfterViewInit(): void {
    setTimeout(() => {
      this.crearComponente(InicioComponent)
    }, 2000)
  }

  crearComponente(cmp: any) {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    viewContainerRef.createComponent(cmp)
  }
}
```

Vamos a utilizar el componente de **mi-router-outlet** en nuestra aplicación para comprobar que se muestra el componente dinámico de Inicio.

Para poder usarlo en App, tenemos que terminar de configurar nuestros módulos.

Primero vamos a comprobar que en el módulo de **MiRouterModule** tengamos el componente **mi-router-outlet** y las dos directivas que hemos creado dentro de **declarations**.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MiRouterLinkDirective } from './mi-router-link.directive';
import { MiRouterOutletComponent } from './mi-router-outlet/mi-router-outlet.component';
import { HostDirective } from './host.directive';

@NgModule({
  declarations: [
    MiRouterLinkDirective,
    MiRouterOutletComponent,
    HostDirective,
  ],
  imports: [
    CommonModule
  ]
})
export class MiRouterModule {}
```

Y vamos a hacer que se exporten tanto el componente como la directiva de **MiRouterLinkDirective**, que son los dos elementos que vamos a utilizar fuera del módulo del router.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MiRouterLinkDirective } from './mi-router-link.directive';
import { MiRouterOutletComponent } from './mi-router-outlet/mi-router-outlet.component';
import { HostDirective } from './host.directive';

@NgModule({
  declarations: [
    MiRouterLinkDirective,
    MiRouterOutletComponent,
    HostDirective,
  ],
  imports: [
    CommonModule
  ],
  exports: [
    MiRouterLinkDirective,
    MiRouterOutletComponent,
  ]
})
export class MiRouterModule {}
```

Ahora en el módulo raíz tenemos que importar el módulo de **MiRouterModule** añadiéndolo en el array de **imports**.

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

import { AppComponent } from './app.component';
import { InicioComponent } from './inicio/inicio.component';
import { FinComponent } from './fin/fin.component';
import { MiRouterModule } from './mi-router/mi-router.module';

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

Una vez importado nuestro módulo, ya podemos utilizar nuestro componente **MiRouterOutletComponent** en la plantilla de la aplicación para mostrar los componentes dinámicos.

```
<hr>
<app-mi-router-outlet></app-mi-router-outlet>
<hr>
```

Con esto ya tendríamos que ver el componente de Inicio en el navegador tras pasar los 2 segundos.

Pues ya hemos usado un componente dinámico, aunque la idea del laboratorio es ir pintando distintos componentes en tiempo de ejecución de la aplicación según cambiamos de rutas.

Vamos a continuar con el ejemplo para añadir unos enlaces y poder cambiar la ruta de la aplicación web.

Estos enlaces los añadiremos en la plantilla del componente App. Como ya vimos en el tema del Router de Angular, no podemos utilizar el atributo **href** de los enlaces ya que recargaría la página al pulsar sobre ellos, por lo que vamos a utilizar la directiva **MiRouterLinkDirective** que hemos creado antes, y le vamos a asignar la ruta a la que queremos ir.

/angular-componentes-dinamicos-router-lab/src/app/app.component.html

```
<a appMiRouterLink="/">Inicio</a>
<a appMiRouterLink="/fin">Fin</a>

<hr>
<app-mi-router-outlet></app-mi-router-outlet>
<hr>
```

Ahora que ya estamos aplicando la directiva, vamos a ir a implementarla.

Lo primero que vamos a hacer es añadirle unos estilos para que coja la apariencia de un enlace.

Para ello usaremos el decorador **HostBinding** y le asignaremos estos estilos a la propiedad **style** del elemento al que se añada esta directiva.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-link.directive.ts

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

@Directive({
  selector: '[appMiRouterLink]'
})
export class MiRouterLinkDirective {
  @HostBinding('style') styles = {
    textDecoration: 'underline',
    color: 'blue',
    cursor: 'pointer',
  }

  constructor() { }

}
```

El siguiente paso es recibir el path de la ruta a la que queremos cambiar desde el exterior. Por lo que vamos a añadir una propiedad **path** con el decorador **Input** y como alias el nombre de la directiva.

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

@Directive({
  selector: '[appMiRouterLink]'
})
export class MiRouterLinkDirective {
  @HostBinding('style') styles = {
    textDecoration: 'underline',
    color: 'blue',
    cursor: 'pointer',
  }
  @Input('appMiRouterLink') path: string = '/'

  constructor() { }

}
```

Ahora tenemos que detectar el evento **click** sobre el elemento que tiene la directiva.

Esta vez vamos a usar el decorador **HostListener** sobre una función en la que vamos a cambiar la ruta de la aplicación añadiéndola en la pila del historial de navegación con el método **window.history.pushState**.

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

@Directive({
  selector: '[appMiRouterLink]'
})
export class MiRouterLinkDirective {
  @HostBinding('style') styles = {
    textDecoration: 'underline',
    color: 'blue',
    cursor: 'pointer',
  }
  @Input('appMiRouterLink') path: string = '/'

  constructor() { }

  @HostListener('click') onClick() {
    window.history.pushState({}, '', this.path)
  }
}
```

Con esto ya podemos cambiar entre las dos rutas de la aplicación, podemos ver que al pulsar en los enlaces, la barra de navegación cambia entre las rutas:

- <http://localhost:4200>
- <http://localhost:4200/fin>

Ahora que ya podemos cambiar entre las rutas, vamos a sacar esta llamada al **pushState** al servicio de **MiRouterService** que habíamos creado, para poder realizar esta navegación no solo desde la directiva, sino desde cualquier otro lado.

Dentro del servicio vamos a crear un método **navigate** al cual le vamos a pasar el path, y pondremos lo que teníamos en el HostListener de la directiva anterior.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router.service.ts

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

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

  constructor() { }

  navigate(path: string) {
    window.history.pushState({}, '', path)
  }
}
```

Ahora en nuestra directiva tendremos que importar este servicio y ejecutar esta función pasándole el path.

```
import { Directive, HostBinding, Input, HostListener } from '@angular/core';
import { MiRouterService } from './mi-router.service';

@Directive({
  selector: '[appMiRouterLink]'
})
export class MiRouterLinkDirective {
  @HostBinding('style') styles = {
    textDecoration: 'underline',
    color: 'blue',
    cursor: 'pointer',
  }
  @Input('appMiRouterLink') path: string = '/'

  constructor(private miRouter: MiRouterService) { }

  @HostListener('click') onClick() {
    this.miRouter.navigate(this.path)
  }
}
```

El siguiente paso es comunicar este servicio con nuestro componente **MiRouterOutletComponent** para que cuando se ejecute el método **navigate** del servicio, podamos avisar al componente que tiene que cambiar el componente que se está mostrando.

Aquí utilizaremos observables, más concretamente el **BehaviorSubject** para poder enviar un valor por defecto a los nuevos suscriptores.

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

@Injectable({
  providedIn: 'root'
})
export class MiRouterService {
  pathChanged$ = new BehaviorSubject('/')

  constructor() { }

  navigate(path: string) {
    window.history.pushState({}, '', path)
  }
}
```

Y después de cambiar la ruta de la barra de navegación, vamos a emitir el nuevo path a nuestro

componente utilizando el método **next** del observable.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router.service.ts

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

@Injectable({
  providedIn: 'root'
})
export class MiRouterService {
  pathChanged$ = new BehaviorSubject('/')

  constructor() { }

  navigate(path: string) {
    window.history.pushState({}, '', path)
    this.pathChanged$.next(path)
  }
}
```

Desde nuestro componente **MiRouterOutlet** tenemos que suscribirnos a este observable para que cuando se cambie la ruta, podamos recibir el nuevo path y pintar el componente asociado a dicho path.

Vamos a empezar por injectar la instancia del servicio en el constructor del componente, y vamos a quitar el **setTimeout** que habíamos puesto.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-outlet/mi-router-outlet.component.ts

```
import { Component, ViewChild, AfterViewInit } from '@angular/core';
import { HostDirective } from '../host.directive';
import { InicioComponent } from '../../inicio/inicio.component';
import { MiRouterService } from '../mi-router.service';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent implements AfterViewInit {

  @ViewChild(HostDirective) host!: HostDirective

  constructor(private miRouter: MiRouterService) { }

  ngAfterViewInit(): void {
    }

  crearComponente(cmp: any) {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    viewContainerRef.createComponent(cmp)
  }
}
```

Dentro del **ngAfterViewInit** vamos a suscribirnos al observable del servicio, y por ahora pondremos un switch para indicar que componente se va a mostrar por cada una de las rutas.

También tenemos que importar el componente de **FinComponent** que creamos al inicio del laboratorio.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-outlet/mi-router-outlet.component.ts

```
import { Component, ViewChild, AfterViewInit } from '@angular/core';
import { HostDirective } from '../host.directive';
import { InicioComponent } from '../../inicio/inicio.component';
import { MiRouterService } from '../mi-router.service';
import { FinComponent } from '../../fin/fin.component';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent implements AfterViewInit {

  @ViewChild(HostDirective) host!: HostDirective

  constructor(private miRouter: MiRouterService) { }

  ngAfterViewInit(): void {
    this.miRouter.pathChanged$.subscribe(path => {
      switch(path) {
        case '/':
          this.crearComponente(InicioComponent)
          break;
        case '/fin':
          this.crearComponente(FinComponent)
          break;
        default:
          this.crearComponente(InicioComponent)
          break;
      }
    })
  }

  crearComponente(cmp: any) {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    viewContainerRef.createComponent(cmp)
  }
}
```

Con estos cambios ya tenemos una primera versión de nuestro router con componentes dinámicos.

Si probamos a pulsar los dos enlaces que habíamos puesto veremos que los componentes cambian.

Ahora vamos a hacer que sea más parecido al Router de Angular.

El siguiente paso es poder pasarle a nuestro módulo del routing el array de rutas en lugar de tenerlas definidas en el **switch**.

Nos vamos a crear un archivo **app.routes.ts** en el que vamos a inicializar nuestro array de rutas.

/angular-componentes-dinamicos-router-lab/src/app/app.routes.ts

```
const APP_ROUTES = [  
]
```

Dentro de este array vamos a poner las dos rutas que tenemos como valor de la clave **path** y los componentes como valor de la clave **component**.

/angular-componentes-dinamicos-router-lab/src/app/app.routes.ts

```
import { FinComponent } from "./fin/fin.component";  
import { InicioComponent } from "./inicio/inicio.component";  
  
const APP_ROUTES = [  
  { path: '', component: InicioComponent },  
  { path: 'fin', component: FinComponent },  
]
```

Una vez tenemos las rutas, llega la parte más compleja, que es hacerle llegar estas rutas a nuestro componente **MiRouterOutletComponent** a través del módulo **MiRouterModule**.

Para ello, vamos a tener que crear un **token de inyección** al que le vamos a asignar este array de rutas. Esto lo tendremos que hacer en el módulo del router.

Dentro de la clase del módulo vamos a crear un método estático **forRoot** al que le pasaremos las rutas como parámetro.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MiRouterLinkDirective } from './mi-router-link.directive';
import { MiRouterOutletComponent } from './mi-router-outlet/mi-router-outlet.component';
import { HostDirective } from './host.directive';

@NgModule({
  declarations: [
    MiRouterLinkDirective,
    MiRouterOutletComponent,
    HostDirective,
  ],
  imports: [
    CommonModule
  ],
  exports: [
    MiRouterLinkDirective,
    MiRouterOutletComponent,
  ]
})
export class MiRouterModule {
  static forRoot(routes: Array<any>) {
  }
}
```

Vamos a hacer que este método devuelva un objeto del tipo **ModuleWithProviders<T>** ya que es lo que tendremos que pasarle al **exports** del módulo raíz de la aplicación.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MiRouterLinkDirective } from './mi-router-link.directive';
import { MiRouterOutletComponent } from './mi-router-outlet/mi-router-outlet.component';
import { HostDirective } from './host.directive';

@NgModule({
  declarations: [
    MiRouterLinkDirective,
    MiRouterOutletComponent,
    HostDirective,
  ],
  imports: [
    CommonModule
  ],
  exports: [
    MiRouterLinkDirective,
    MiRouterOutletComponent,
  ]
})
export class MiRouterModule {
  static forRoot(routes: Array<any>): ModuleWithProviders<MiRouterModule> {
    return {
      ngModule: MiRouterModule,
      providers: []
    }
  }
}
```

Dentro de este array de **providers** es donde vamos a crear un token de inyección al que vamos a llamar **MisRutas**, y le vamos a indicar que cuando pidan este token les tiene que devolver el valor de las rutas que recibe la función como parámetro y que se han definido en el archivo de rutas.

```
import { NgModuleWithProviders, NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MiRouterLinkDirective } from './mi-router-link.directive';
import { MiRouterOutletComponent } from './mi-router-outlet/mi-router-outlet.component';
import { HostDirective } from './host.directive';

@NgModule({
  declarations: [
    MiRouterLinkDirective,
    MiRouterOutletComponent,
    HostDirective,
  ],
  imports: [
    CommonModule
  ],
  exports: [
    MiRouterLinkDirective,
    MiRouterOutletComponent,
  ]
})
export class MiRouterModule {
  static forRoot(routes: Array<any>): NgModuleWithProviders<MiRouterModule> {
    return {
      ngModule: MiRouterModule,
      providers: [
        {
          provide: 'MisRutas',
          useValue: routes,
        }
      ]
    }
  }
}
```

Ahora volvemos al archivo de rutas para llamar a esta función del **forRoot** y obtener el nuevo módulo.

/angular-componentes-dinamicos-router-lab/src/app/app.routes.ts

```
import { FinComponent } from './fin/fin.component';
import { InicioComponent } from './inicio/inicio.component';
import { MiRouterModule } from './mi-router/mi-router.module';

const APP_ROUTES = [
  { path: '', component: InicioComponent },
  { path: 'fin', component: FinComponent },
]

export const RoutingModule = MiRouterModule.forRoot(APP_ROUTES)
```

Este módulo que estamos exportando es el nuevo módulo que importaremos en el **app.module.ts** en lugar del **MiRouterModule** que estabamos importando.

/angular-componentes-dinamicos-router-lab/src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { InicioComponent } from './inicio/inicio.component';
import { FinComponent } from './fin/fin.component';
import { RoutingModule } from './app.routes';

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

Por último, tenemos que quitar del **MiRouterOutletComponent** el **switch** que controla los componentes que hay que mostrar, ya que ahora las rutas y componentes se encuentran definidas en el token de **MisRutas** que hemos añadido en el módulo y tenemos que sacar de ahí los datos.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-outlet/mi-router-outlet.component.ts

```
import { Component, ViewChild, AfterViewInit } from '@angular/core';
import { HostDirective } from '../host.directive';
import { InicioComponent } from '../../inicio/inicio.component';
import { MiRouterService } from '../mi-router.service';
import { FinComponent } from '../../fin/fin.component';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent implements AfterViewInit {

  @ViewChild(HostDirective) host!: HostDirective

  constructor(private miRouter: MiRouterService) { }

  ngAfterViewInit(): void {
    this.miRouter.pathChanged$.subscribe(path => {
      })
    }

  crearComponente(cmp: any) {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    viewContainerRef.createComponent(cmp)
  }
}
```

Ahora tenemos que inyectar el array de rutas del módulo, usando el decorador **Inject** y pasandole el token de inyección que tiene asociado el array de rutas.

```
import { Component, ViewChild, AfterViewInit, Inject } from '@angular/core';
import { HostDirective } from '../host.directive';
import { MiRouterService } from '../mi-router.service';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent implements AfterViewInit {

  @ViewChild(HostDirective) host!: HostDirective

  constructor(private miRouter: MiRouterService, @Inject('MisRutas') private misRutas: any) { }

  ngAfterViewInit(): void {
    this.miRouter.pathChanged$.subscribe(path => {

    })
  }

  crearComponente(cmp: any) {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    viewContainerRef.createComponent(cmp)
  }
}
```

Dentro de la suscripción vamos a buscar la ruta cuyo path coincida con el que se ha recibido en la suscripción.

/angular-componentes-dinamicos-router-lab/src/app/mi-router/mi-router-outlet/mi-router-outlet.component.ts

```
import { Component, ViewChild, AfterViewInit, Inject } from '@angular/core';
import { HostDirective } from '../host.directive';
import { MiRouterService } from '../mi-router.service';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent implements AfterViewInit {

  @ViewChild(HostDirective) host!: HostDirective

  constructor(private miRouter: MiRouterService, @Inject('MisRutas') private misRutas: any) { }

  ngAfterViewInit(): void {
    this.miRouter.pathChanged$.subscribe(path => {
      const route = this.misRutas.find((r: any) => (('/' + r.path) === path))
    })
  }

  crearComponente(cmp: any) {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    viewContainerRef.createComponent(cmp)
  }
}
```

Una vez encontrada la ruta, solo tenemos que llamar a la función de **crearComponente** pasándole como parámetro el componente que se encuentra dentro de la ruta.

```
import { Component, ViewChild, AfterViewInit, Inject } from '@angular/core';
import { HostDirective } from '../host.directive';
import { MiRouterService } from '../mi-router.service';

@Component({
  selector: 'app-mi-router-outlet',
  templateUrl: './mi-router-outlet.component.html',
  styleUrls: ['./mi-router-outlet.component.css']
})
export class MiRouterOutletComponent implements AfterViewInit {

  @ViewChild(HostDirective) host!: HostDirective

  constructor(private miRouter: MiRouterService, @Inject('MisRutas') private misRutas: any) { }

  ngAfterViewInit(): void {
    this.miRouter.pathChanged$.subscribe(path => {
      const route = this.misRutas.find((r: any) => (('/' + r.path) === path))
      this.crearComponente(route.component)
    })
  }

  crearComponente(cmp: any) {
    const viewContainerRef = this.host.viewContainerRef
    viewContainerRef.clear()
    viewContainerRef.createComponent(cmp)
  }
}
```

Si probamos a pulsar en los distintos enlaces debemos de ver como se va cambiando los componentes según los hayamos asociado a las rutas por las que vamos pasando.

Con esto ya tendríamos una primera versión de un router propio que funciona con componentes dinámicos y de forma muy similar al router de Angular.

Chapter 25. Progressive Web Apps (PWAs)

Las **Progressive Web Apps (PWA)** son aplicaciones web con funcionalidades extra que les permite comportarse como si fueran aplicaciones móviles. Lo que buscan es que la experiencia de usuario sea igual que con las aplicaciones nativas, y para ello tienen que cumplir las siguientes características:

- De confianza: la aplicación se tiene que cargar instantáneamente al igual que ocurre con las aplicaciones nativas, incluso cuando tengamos una mala conexión a internet o esta sea nula.
- Rápida: tiene que tardar poco en responder al usuario.
- Adictiva: tiene que parecer que es una aplicación nativa, que tenga características propias de las nativas.

Para convertir nuestras aplicaciones de Angular en PWAs tendremos que añadir en nuestro proyecto la librería de `@angular/pwa`.

25.1. Beneficios de usar una PWA

Lo bueno de crear las PWAs es que vamos a tener lo mejor de las páginas web y lo mejor de las aplicaciones a la vez.

Las PWAs nos van a permitir tener nuestra aplicación instalada en nuestros dispositivos móviles, por lo tanto ya no es necesario tener que entrar al navegador del móvil para entrar en ella, sino que tendremos un ícono en nuestro **homescreen** y podremos acceder desde el.

Este ícono no es un acceso directo a nuestra web, sino que al darle se abrirá como una aplicación nativa. Podemos verla en el listado de aplicaciones del dispositivo como si hubiéramos instalado el **apk**.

Cuando entramos en alguna aplicación nativa y no tenemos conexión, la aplicación se carga aunque luego no puedas usar todas las funcionalidades que tiene. Estas aplicaciones tienen que permitirnos usarlas tanto con internet como sin el.

La distribución de estas aplicaciones es mucho más rápida que las de las aplicaciones nativas, ya que no tenemos que esperar a que los marketplaces nos validen la aplicación para que los usuarios puedan empezar a descargarla. En este caso, son **URLs** por tanto con entrar en la página web ya podemos instalarla en nuestro dispositivo y a partir de ahí, la podremos usar como si de una aplicación nativa se tratase.

Son mucho más fáciles de descubrir que las aplicaciones nativas, ya que no es necesario entrar en los marketplaces para buscar algo que instalar, sino que podemos ir navegando por internet, y en el momento que entremos en una web y nos guste la podremos instalar, habiéndola podido probar antes, que esto no se puede hacer con las nativas.

Las webs suelen ocupar mucho menos que las aplicaciones nativas porque estas suelen llevar instaladas librerías que al final es muy probable que no se vayan a usar en la aplicación, mientras que en las PWAs solo vamos a tener las librerías que vamos a usar para construirla.

También podemos tener características propias de las aplicaciones nativas como pueden ser las notificaciones push o el funcionamiento de estas sin internet.

Todos estos beneficios, los vamos a conseguir añadiendo las siguientes funcionalidades a nuestra aplicación web:

- App Manifest
- Service Workers
- Notificaciones Push
- App Shell

25.2. App Manifest

Para poder instalar la aplicación en el dispositivo móvil, necesitamos el archivo de **App Manifest** que es el archivo donde se le da información extra al navegador y que permitirá que se instale la aplicación en los dispositivos móviles y se muestre como una app nativa.

En este archivo vamos a poder añadir las siguientes propiedades:

- **name**: indica el nombre de la aplicación que se va a mostrar por ejemplo en la pantalla de **splash**.
- **short_name**: aquí le indicamos el nombre de la aplicación que queremos mostrar junto al icono en el dispositivo una vez que la tengamos instalada.
- **description**: una descripción sobre la aplicación para mostrarla en sitios donde se pueda mostrar más información sobre la PWA.
- **start_url**: aquí le indicamos la URL de la pantalla que se tiene que abrir al abrir la aplicación.
- **display**: esta propiedad indica como se tiene que mostrar la aplicación una vez que la abramos. Los posibles valores son:
 - **standalone**: se muestra como una aplicación nativa.
 - **fullscreen**: pensado para juegos o aplicaciones que se tienen que ver en pantalla completa.
 - **browser**: se muestra como en el navegador.
- **orientation**: aquí le vamos a indicar la orientación en la que se tiene que mostrar la aplicación, y los posibles valores son **portrait** y **landscape**.
- **background_color**: indica el color que se va a aplicar a la pantalla de **splash**.
- **theme_color**: indica el color que se va a mostrar el toolbar, en la pantalla del gestor de aplicaciones...
- **icons**: esta propiedad recibe como valor un array de iconos que representan nuestra aplicación. Estos iconos son objetos JS en los que vamos a poner las siguientes propiedades:
 - **src**: ruta hasta el ícono.
 - **sizes**: tamaño del ícono (512x512).
 - **type**: el tipo de archivo (**image/png**).

Dentro de los proyectos de angular, este archivo es el **src/manifest.webmanifest** y se importa en el **index.html**.

25.3. Service Workers

Los **Services Workers** son archivos JS que se ejecutan en segundo plano, y no necesitan que la página web esté abierta para que sigan funcionando.

Al poder ejecutarlos sin necesidad de que la tengamos la web abierta, esto nos da la posibilidad de recibir notificaciones push, pero también nos permite hacer más cosas.

Los service workers actúan como proxy, y cada petición que se va a realizar para descargarnos algún archivo desde el servidor (al abrir la página web nos descargamos varios archivos) pasa por él.

Para que los service workers funcionen, la aplicación se tiene que servir usando el protocolo HTTPS para evitar que nos hagan un **man in the middle** y lean todos los paquetes que pasan por ahí. Existe una excepción, y es si utilizamos el **localhost**.

Los service workers interactúan con nuestra aplicación mediante eventos, pero no todos los tipos de eventos. Al estar ejecutándose en segundo plano, no puede acceder al DOM y por tanto no puede escuchar los eventos relacionados con él. Algunos de los eventos que vamos a poder detectar son:

- Fetch: se activa cuando se realiza una petición HTTP para pedir algún recurso al servidor.
- Interacciones con las notifications: se activan cuando interactuamos con las notificaciones que hemos recibido, por ejemplo al cerrarlas o al pulsar alguna de las opciones que nos muestran.
- Push Notifications: se activa cuando recibimos una mensaje Push.
- Eventos del ciclo de vida: se activa con algunos eventos del ciclo de vida de los service workers.



Los **service workers** no funcionan con el servidor de desarrollo de Angular, es decir el comando **ng s**, por lo que tenemos que utilizar alguna otra librería como **http-server**.

<https://angular.io/guide/service-worker-getting-started#serving-with-http-server>

Dentro de Angular, no nos tenemos que preocupar por tener que crear la lógica que hace uso de los service workers, sino que esto lo controlaremos desde un archivo de configuración JSON, haciendo que todo sea mucho más sencillo.

Este archivo se crea al añadir la librería de **@angular/pwa**, y lo podemos encontrar en **ngsw-config.json**.

Las opciones que podemos llenar son las siguientes:

- **index**: se especifica el archivo que se sirve como página inicial, es decir, nuestro **index.html**.
- **assetGroups**: assets o recursos estáticos que son parte de la aplicación. Se pueden cargar desde el mismo origen, algún CDN o URLs externas. Aquí definimos grupos de assets, cada uno con su propia configuración. Dentro de cada grupo de assets nos encontramos:
 - **name**: este nombre es muy importante ya que identifica el grupo de assets entre las distintas versiones de la configuración (por ejemplo a la hora de actualizar).

- **installMode:** indica como se tienen que cachear los assets inicialmente.
 - Si usamos **prefetch** le estamos indicando que cachee todos los recursos al arrancar la aplicación de tal forma que esto siempre estará disponible incluso cuando estemos sin conexión.
 - En caso de utilizar **lazy**, le estaríamos indicando que solo cachee los recursos cuando se pidan, de tal forma que si nos quedamos sin conexión y no se había pedido hasta ahora un recurso, no tendremos acceso a él hasta que volvamos a tener conexión.
- **updateMode:** tiene las mismas opciones que el `installMode`. Esta opción indica como se tienen que cachear los recursos cuando se descubre una nueva versión de la aplicación. El **prefetch** cachea inmediatamente todos los assets que hayan cambiado, y el **lazy** (solo funciona si en `installMode` hemos usado `lazy` también) espera a cachear las nuevas versiones de los recursos cuando se pidan estos otra vez.
- **resources:** tenemos dos opciones:
 - **files** para indicar que archivos del fichero de distribución queremos cachear.
 - **urls** para indicar que archivos que se encuentran en URLs externas queremos cachear.
- **dataGroups:** al contrario que los assets, los datos no están versionados, son items dinámicos como los datos que obtenemos de una API, así que tienen sus propias opciones a la hora de cachearlos. Las opciones que encontramos dentro de estos objetos son:
 - **name:** al igual que en los assets, aquí va el nombre con el que identificamos los datos cacheados aquí.
 - **urls:** es una lista de patrones de URLs a cachear. Cuando se realice una petición HTTP, si coincide con alguno de los patrones entonces se cachea.
 - **version:** es un número que solo cambiaremos cuando la API a la que pedimos los datos se haya actualizado y los cambios realizados sobre ella sean incompatibles con la antigua API.
 - **cacheConfig:** en este objeto de configuración es donde vamos a establecer las políticas de la caché:
 - **maxSize:** el número máximo de entradas en la caché.
 - **maxAge:** el tiempo máximo que se van a mantener las entradas en la caché. Utilizaremos sufijos para indicar la unidad de tiempo (d - días, h - horas, m - minutos, s - segundos y u - milisegundos). Por ejemplo, **6h30s**.
 - **timeout:** el tiempo máximo que el SW va a esperar a que se obtenga la respuesta a una petición. Si se termina este tiempo, entonces se devuelve la respuesta cacheada. Se usa el mismo formato que en **maxAge**.
 - **strategy:** el tipo de estrategia de cacheado a seguir:
 - **freshness:** pide el recurso a la API o servidor, y si no puede obtenerlo o se termina el tiempo de espera entonces devuelve el cacheado. Esta es una buena estrategia para esos datos que cambian frecuentemente, ya que siempre los tendremos lo más actualizados posible.
 - **performance:** devuelve lo de la cache, y si no está entonces pide el recurso al servidor. Se suele usar para recursos que no cambian demasiado a lo largo del tiempo.

25.4. Lab: Progressive Web Apps (PWAs)

En este laboratorio vamos a crear una pequeña aplicación para publicar y listar ofertas de trabajo que vamos a convertir en una PWA con cacheado de assets y datos.

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

```
$ ng new angular-pwa-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 unos componentes, un servicio y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-pwa-lab  
$ ng g c inicio  
$ ng g c nueva-oferta  
$ ng g c detalle-oferta  
$ ng g c oferta  
$ ng g s ofertas  
$ ng s
```

Vamos a empezar creando las rutas por las que vamos a poder navegar en la aplicación:

- / → Inicio
- /nueva-oferta → Formulario para crear una nueva oferta
- /ofertas/:id → Información completa de una oferta dado su identificador

Por tanto, vamos a crear un archivo **app.routes.ts** en la carpeta **app** en el que vamos a declarar estas rutas y vamos a configurar el **RouterModule** con ellas.

/angular-pwa-lab/src/app/app.routes.ts

```
import { RouterModule, Routes } from "@angular/router";  
import { DetalleOfertaComponent } from "./detalle-oferta/detalle-oferta.component";  
import { InicioComponent } from "./inicio/inicio.component";  
import { NuevaOfertaComponent } from "./nueva-oferta/nueva-oferta.component";  
  
const APP_ROUTES: Routes = [  
  { path: '', component: InicioComponent },  
  { path: 'nueva-oferta', component: NuevaOfertaComponent },  
  { path: 'ofertas/:id', component: DetalleOfertaComponent },  
]  
  
export const RoutingModule = RouterModule.forRoot(APP_ROUTES)
```

Ahora vamos a importar en el módulo de App el módulo del routing y el resto de módulos que vamos a utilizar:

- HttpClientModule
- RouterModule
- ReactiveFormsModule

/angular-pwa-lab/src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { InicioComponent } from './inicio/inicio.component';
import { NuevaOfertaComponent } from './nueva-oferta/nueva-oferta.component';
import { DetalleOfertaComponent } from './detalle-oferta/detalle-oferta.component';
import { OfertaComponent } from './oferta/oferta.component';
import { HttpClientModule } from '@angular/common/http';
import { RouterModule } from './app.routes';
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent,
    InicioComponent,
    NuevaOfertaComponent,
    DetalleOfertaComponent,
    OfertaComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    ReactiveFormsModule,
    RouterModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Con esto, ya podemos centrarnos en ir desarrollando los componentes y el servicio.

Vamos a empezar por añadir el **router-outlet** y los enlaces para navegar por las rutas en el componente App.

/angular-pwa-lab/src/app/app.component.html

```
<ul>
  <li>
    <a [routerLink]="/>Inicio</a>
  </li>
  <li>
    <a [routerLink]="/nueva-oferta">Crear oferta</a>
  </li>
</ul>

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

Ahora vamos a ir al servicio a crear los 3 métodos que vamos a necesitar:

- getOfertas: petición GET para obtener todas las ofertas de trabajo

/angular-pwa-lab/src/app/ofertas.service.ts

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

@Injectable({
  providedIn: 'root'
})
export class OfertasService {
  URL: string = 'https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas'

  constructor(private http: HttpClient) { }

  getOfertas(): Observable<any> {
    return this.http.get(`${this.URL}.json`)
      .pipe(
        map((objOfertas: any) => {
          const arrOfertas = []

          for (let id in objOfertas) {
            arrOfertas.push({...objOfertas[id], id})
          }

          return arrOfertas
        })
      )
  }
}
```

- getOferta: petición GET para obtener una oferta de trabajo dado su identificador

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

@Injectable({
  providedIn: 'root'
})
export class OfertasService {
  URL: string = 'https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas'

  constructor(private http: HttpClient) { }

  getOfertas(): Observable<any> {
    return this.http.get(`.${this.URL}.json`)
      .pipe(
        map((objOfertas: any) => {
          const arrOfertas = []

          for (let id in objOfertas) {
            arrOfertas.push({...objOfertas[id], id})
          }

          return arrOfertas
        })
      )
  }

  getOferta(id: string): Observable<any> {
    return this.http.get(`.${this.URL}/${id}.json`)
  }
}
```

- crearOferta: petición POST para crear una nueva oferta de trabajo

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

@Injectable({
  providedIn: 'root'
})
export class OfertasService {
  URL: string = 'https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas'

  constructor(private http: HttpClient) { }

  getOfertas(): Observable<any> {
    return this.http.get(`.${this.URL}.json`)
      .pipe(
        map((objOfertas: any) => {
          const arrOfertas = []

          for (let id in objOfertas) {
            arrOfertas.push({...objOfertas[id], id})
          }

          return arrOfertas
        })
      )
  }

  getOferta(id: string): Observable<any> {
    return this.http.get(`.${this.URL}/${id}.json`)
  }

  crearOferta(nuevaOferta: any): Observable<any> {
    return this.http.post(`.${this.URL}.json`, nuevaOferta)
  }
}
```

El siguiente paso es crear el componente **Oferta** que usaremos para listar en la página de **Inicio** cada una de las ofertas.

En este componente solo vamos a mostrar la imagen de la empresa y el título de esta.

El título lo envolveremos con la etiqueta **a** para poder navegar al detalle de la oferta al pulsar sobre el.

/angular-pwa-lab/src/app/oferta/oferta.component.html

```
<div>
  <img [src]="oferta.urlImagen" alt="Logo de {{oferta.empresa}}">
  <a [routerLink]=["/ofertas", oferta.id]>
    <h3>{{oferta.titulo}}</h3>
  </a>
</div>
```

Ese objeto **oferta** al cual estamos accediendo a la imagen, título, id y empresa lo obtendremos desde el exterior (desde el componente de **Inicio**), por lo que tenemos que declarar la propiedad con un **@Input**.

/angular-pwa-lab/src/app/oferta/oferta.component.ts

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

@Component({
  selector: 'app-oferta',
  templateUrl: './oferta.component.html',
  styleUrls: ['./oferta.component.css']
})
export class OfertaComponent implements OnInit {
  @Input() oferta: any = {}
  constructor() { }

  ngOnInit(): void {
  }

}
```

Ahora nos vamos a ir al componente de Inicio, donde vamos a declarar la lista de ofertas que pediremos desde el servicio que hemos creado anteriormente.

```
import { Component, OnInit } from '@angular/core';
import { OfertasService } from '../ofertas.service';

@Component({
  selector: 'app-inicio',
  templateUrl: './inicio.component.html',
  styleUrls: ['./inicio.component.css']
})
export class InicioComponent implements OnInit {
  listaOfertas: Array<any> = []

  constructor(private ofertas: OfertasService) { }

  ngOnInit(): void {
    this.ofertas.getOfertas()
      .subscribe((datos: Array<any>) => {
        this.listaOfertas = datos
      })
  }
}
```

Como nos estamos suscribiendo al **getOfertas**, vamos a crear una propiedad para guardar la suscripción, y nos vamos a desuscribir en el método **ngOnDestroy**.

/angular-pwa-lab/src/app/inicio/inicio.component.ts

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { Subscription } from 'rxjs';
import { OfertasService } from '../ofertas.service';

@Component({
  selector: 'app-inicio',
  templateUrl: './inicio.component.html',
  styleUrls: ['./inicio.component.css']
})
export class InicioComponent implements OnInit, OnDestroy {
  listaOfertas: Array<any> = []
  ofertasSubs: Subscription | null = null

  constructor(private ofertas: OfertasService) { }

  ngOnInit(): void {
    this.ofertasSubs = this.ofertas.getOfertas()
      .subscribe((datos: Array<any>) => {
        this.listaOfertas = datos
      })
  }

  ngOnDestroy(): void {
    this.ofertasSubs?.unsubscribe()
  }
}
```

Ya podemos iterar la **listaOfertas** en la plantilla y crear un componente **Oferta** por cada una de las que haya en la lista.

/angular-pwa-lab/src/app/inicio/inicio.component.html

```
<h2>Ofertas de trabajo</h2>

<app-oferta *ngFor="let oferta of listaOfertas" [oferta]="oferta"></app-oferta>
```

Ahora nos vamos a ir al componente **DetalleOferta** en el que vamos a mostrar toda la información de cada oferta.

/angular-pwa-lab/src/app/detalle-oferta/detalle-oferta.component.html

```
<h2>{{oferta.titulo}}</h2>
<img [src]="oferta.urlImagen" alt="Logo de {{oferta.empresa}}">
<p>{{oferta.descripcion}}</p>
<p>Empresa: {{oferta.empresa}}</p>
<p>Salario: {{oferta.salario | currency:'EUR'}}</p>
<p>Ciudad: {{oferta.ciudad | titlecase}}</p>
```

Ahora dentro del TypeScript tenemos que obtener la oferta, primero sacando el identificador del parámetro de la ruta usando el servicio de **ActivatedRoute**, y después teniendo ya el id, llamando al método **getOferta** del servicio de **Ofertas**.

/angular-pwa-lab/src/app/detalle-oferta/detalle-oferta.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { OfertasService } from '../ofertas.service';

@Component({
  selector: 'app-detalle-oferta',
  templateUrl: './detalle-oferta.component.html',
  styleUrls: ['./detalle-oferta.component.css']
})
export class DetalleOfertaComponent implements OnInit {
  oferta: any = {}

  constructor(private ofertas: OfertasService, private activatedRoute: ActivatedRoute) { }

  ngOnInit(): void {
    this.activatedRoute.paramMap
      .subscribe((params: ParamMap) => {
        const id = params.get('id')
        if (id) {
          this.ofertas.getOferta(id)
            .subscribe((datos: any) => {
              this.oferta = datos
            })
        }
      })
  }
}
```

Por último, vamos a abrir el TypeScript del componente de **NuevaOferta** donde vamos a definir un formulario con los campos para llenar una oferta de trabajo:

- título
- descripción
- empresa
- salario
- ciudad
- urlImagen

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';
import { Router } from '@angular/router';
import { OfertasService } from '../ofertas.service';

@Component({
  selector: 'app-nueva-oferta',
  templateUrl: './nueva-oferta.component.html',
  styleUrls: ['./nueva-oferta.component.css']
})
export class NuevaOfertaComponent implements OnInit {
  formOferta: FormGroup

  constructor(private ofertas: OfertasService, private router: Router) {
    this.formOferta = new FormGroup({
      titulo: new FormControl(''),
      descripcion: new FormControl(''),
      empresa: new FormControl(''),
      salario: new FormControl(0),
      ciudad: new FormControl(''),
      urlImagen: new FormControl('')
    })
  }

  ngOnInit(): void {
  }

}
```

Ahora vamos a añadir una función **guardar** en la que llamaremos al método **crearOferta** del servicio de **Ofertas** pasándole los datos obtenidos del formulario, y guardaremos la suscripción en una propiedad para desuscribirnos en el método **ngOnDestroy**.

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';
import { Router } from '@angular/router';
import { Subscription } from 'rxjs';
import { OfertasService } from '../ofertas.service';

@Component({
  selector: 'app-nueva-oferta',
  templateUrl: './nueva-oferta.component.html',
  styleUrls: ['./nueva-oferta.component.css']
})
export class NuevaOfertaComponent implements OnInit, OnDestroy {
  formOferta: FormGroup
  nuevaOfertaSubs: Subscription | null = null

  constructor(private ofertas: OfertasService, private router: Router) {
    this.formOferta = new FormGroup({
      titulo: new FormControl(''),
      descripcion: new FormControl(''),
      empresa: new FormControl(''),
      salario: new FormControl(0),
      ciudad: new FormControl(''),
      urlImagen: new FormControl('')
    })
  }

  ngOnInit(): void {
  }

  guardar(): void {
    this.nuevaOfertaSubs = this.ofertas.crearOferta(this.formOferta.value)
      .subscribe(() => {
        this.router.navigate(['/'])
      })
  }

  ngOnDestroy(): void {
    this.nuevaOfertaSubs?.unsubscribe()
  }
}
```

El siguiente paso es crear el formulario en la plantilla y enlazarlo con el que acabamos de crear en el TypeScript.

```
<h2>Nueva oferta de trabajo</h2>

<form [formGroup]="formOferta" (ngSubmit)="guardar()">
  <div>
    <label for="titulo">Título</label>
    <input type="text" id="titulo" formControlName="titulo">
  </div>
  <div>
    <label for="descripcion">Descripción</label>
    <input type="text" id="descripcion" formControlName="descripcion">
  </div>
  <div>
    <label for="empresa">Empresa</label>
    <input type="text" id="empresa" formControlName="empresa">
  </div>
  <div>
    <label for="salario">Salario</label>
    <input type="number" id="salario" formControlName="salario">
  </div>
  <div>
    <label for="ciudad">Ciudad</label>
    <select id="ciudad" formControlName="ciudad">
      <option value="madrid">Madrid</option>
      <option value="barcelona">Barcelona</option>
      <option value="sevilla">Sevilla</option>
      <option value="valencia">Valencia</option>
      <option value="full-remoto">Full Remoto</option>
    </select>
  </div>
  <div>
    <label for="urlImagen">Url imagen</label>
    <input type="text" id="urlImagen" formControlName="urlImagen">
  </div>
  <button type="submit">Guardar</button>
</form>
```

Con esto ya tenemos la aplicación, ahora nos toca transformarla en una PWA.

El primer paso es añadir la librería que se encarga de esto con el siguiente comando:

```
$ ng add @angular/pwa
```

```
The package @angular/pwa@13.2.1 will be installed and executed.
Would you like to proceed? Yes
```

Con este comando se ha generado el archivo del service worker (**ngsw-config.json**) y el archivo del manifest (**manifest.webmanifest**) que son los que nos interesan modificar.

Vamos a empezar cambiando alguna opción del archivo de **manifest** para modificar los estilos de la pantalla de splash.

El nombre de la aplicación lo vamos a cambiar a **Jobs, tu futuro en la palma de tu mano**. Este nombre es el que aparece al abrir la aplicación en un dispositivo móvil en la pantalla de splash.

También vamos a cambiar el **short_name** a **jobs**. Este otro nombre es el que aparece debajo del icono de la aplicación.

Y el último cambio que vamos a hacer es el de **theme_color** y **background_color** para cambiar los colores que se muestran en la pantalla de splash y

/angular-pwa-lab/src/manifest.webmanifest

```
{
  "name": "Jobs, tu futuro en la palma de tu mano",
  "short_name": "jobs",
  "theme_color": "#cccccc",
  "background_color": "#000000",
  "display": "standalone",
  "scope": "./",
  "start_url": "./",
  "icons": [
    {
      "src": "assets/icons/icon-72x72.png",
      "sizes": "72x72",
      "type": "image/png",
      "purpose": "maskable any"
    },
    {
      "src": "assets/icons/icon-96x96.png",
      "sizes": "96x96",
      "type": "image/png",
      "purpose": "maskable any"
    },
    {
      "src": "assets/icons/icon-128x128.png",
      "sizes": "128x128",
      "type": "image/png",
      "purpose": "maskable any"
    },
    {
      "src": "assets/icons/icon-144x144.png",
      "sizes": "144x144",
      "type": "image/png",
      "purpose": "maskable any"
    },
    {
      "src": "assets/icons/icon-152x152.png",
      "sizes": "152x152",
      "type": "image/png",
      "purpose": "maskable any"
    }
  ]
}
```

```
        "purpose": "maskable any"
    },
    {
        "src": "assets/icons/icon-192x192.png",
        "sizes": "192x192",
        "type": "image/png",
        "purpose": "maskable any"
    },
    {
        "src": "assets/icons/icon-384x384.png",
        "sizes": "384x384",
        "type": "image/png",
        "purpose": "maskable any"
    },
    {
        "src": "assets/icons/icon-512x512.png",
        "sizes": "512x512",
        "type": "image/png",
        "purpose": "maskable any"
    }
]
```

El otro cambio que vamos a realizar es el de cambiar los iconos por unos propios de nuestra aplicación.

Para ello, vamos a coger la siguiente imagen y vamos a subirla en la página <https://manifest-gen.netlify.app/>, la cual nos va a generar el archivo manifest con la carpeta de iconos.



A nosotros solo nos interesa la carpeta de iconos, por lo que vamos a copiar esa carpeta y la vamos a pegar dentro de **src/assets**, sobreescribiendo todos los iconos que venían por defecto.

Una vez hecho esto, vamos a comprobar que nuestra aplicación ya es una PWA y que podemos instalarla en nuestro dispositivo móvil.

El primer paso para poder instalarla en nuestro dispositivo móvil es generar el proyecto para producción con el siguiente comando:

```
$ ng build
```

Una vez construido el proyecto, vamos a entrar en la carpeta * y vamos a levantar un servidor utilizando la dependencia *http-serve*.

```
$ cd dist/angular-pwa-lab  
$ npx http-serve
```

Ya está levantada la aplicación en <http://localhost:8080>. Si accedemos a la página veremos nuestra aplicación.

Ahora queremos verla en nuestro dispositivo móvil para ver el mensaje que nos permitirá instalarla dentro de este como si fuese una aplicación.

Para ello vamos a acceder, desde Google Chrome, a <chrome://inspect/#devices> para poder conectar nuestro móvil al <http://localhost:8080> de nuestro ordenador y así poder ver la misma web en los dos sitios.

Tenemos que habilitar la **Depuración USB** del dispositivo móvil desde **Ajustes > Herramientas del desarrollador** y conectar el móvil al ordenador.

Al hacer los pasos anteriores, debe de salirnos un popup en nuestro móvil pidiendo que le demos acceso a nuestro ordenador para conectarse.

Depuración USB habilitada

Toca aquí para desactivar la depuración USB

¿Permitir depuración por USB?

La huella digital de tu clave RSA es:

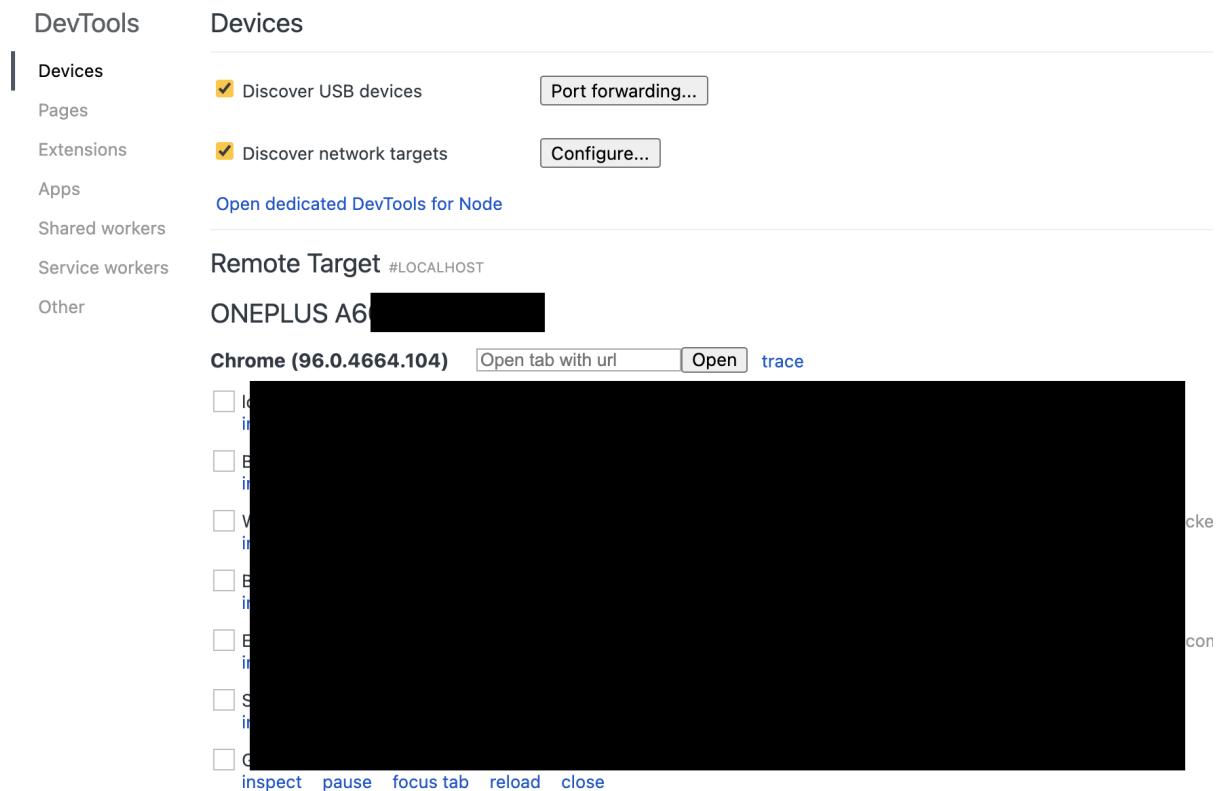
A: [REDACTED] B:
[REDACTED]

Permitir siempre desde este ordenador

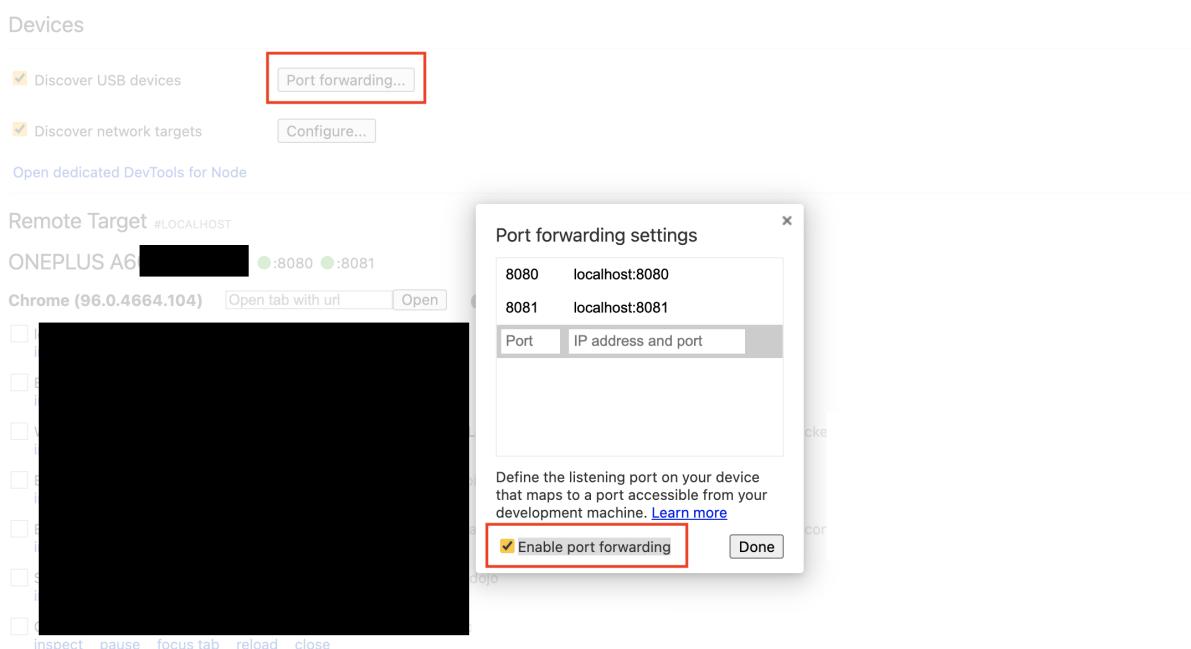
CANCELAR PERMITIR



Una vez aceptado el permiso, en la página de los dispositivos de Chrome (<chrome://inspect/#devices>), debería de salir el nombre de nuestro dispositivo y al entrar en el navegador de nuestro móvil, también deberíamos de poder ver la lista de páginas que tenemos abiertas en este.



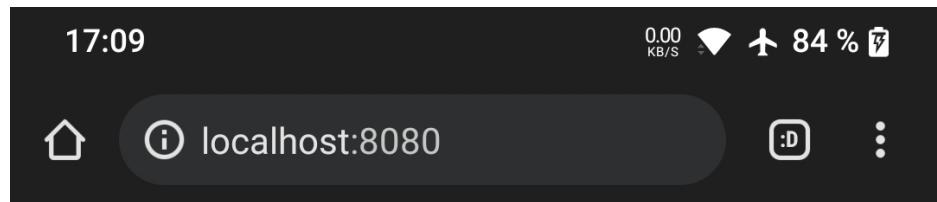
Debemos de asegurarnos de que tenemos activado el **Port forwarding** pulsando sobre dicho botón y seleccionando la opción de **Enable port forwarding**.



Ya podemos entrar en el navegador del dispositivo móvil en la URL <http://localhost:8080/>, donde debería de salirnos la aplicación que se está sirviendo en ese mismo puerto dentro del ordenador.

Deberíamos de poder ver la aplicación en el móvil, y justo por la parte de abajo debería de haber

salido una alerta para añadir la aplicación en nuestro dispositivo.

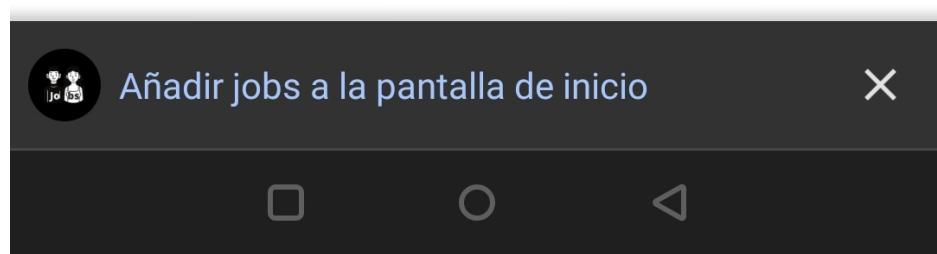


- [Inicio](#)
- [Crear oferta](#)

Ofertas de trabajo

Empresa 1

Diseñador/a Gráfico/a in Madrid



Al pulsar sobre la alerta se abre un popup del sistema y nos pregunta si queremos instalar la aplicación. Vamos a darle a **Instalar**.



- [Inicio](#)
- [Crear oferta](#)

Ofertas de trabajo

Empresa 1

[Diseñador/a Gráfico/a in Madrid](#)

Instalar aplicación



Jobs, tu futuro en la palma de
tu mano

<http://localhost:8080>

[Cancelar](#)

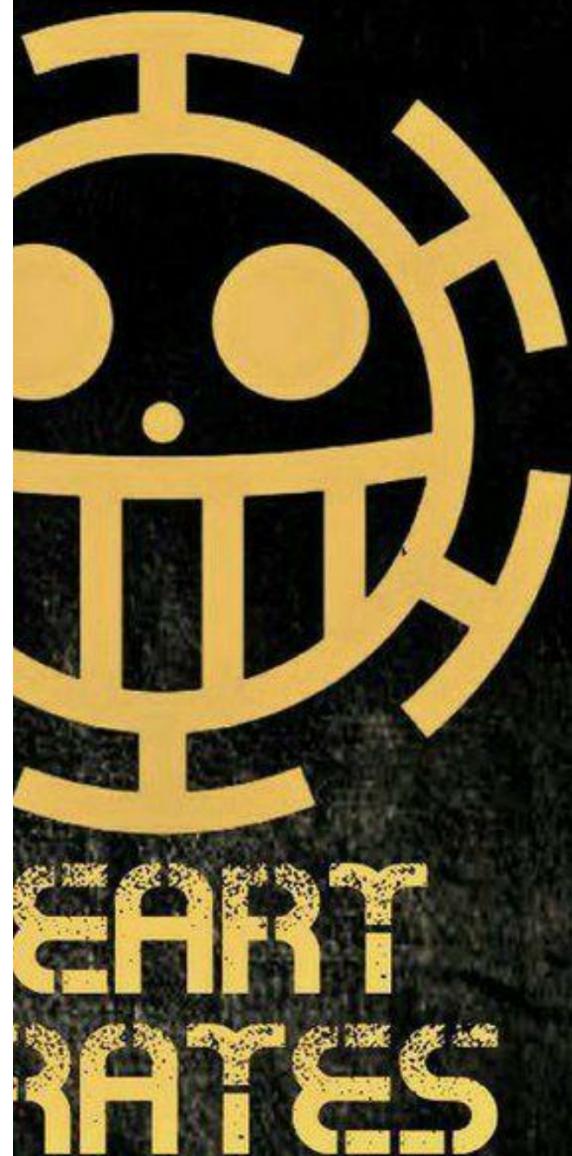
[Instalar](#)



Una vez termina de instalarse, ya la podremos ver en nuestro **HomeScreen**.

15:47

34 %



jobs

^



Al abrirla, saldrá primero la pantalla de splash con el icono, nombre y colores que habíamos definido en el archivo de **manifest**.

15:47

– 🔔 ✈ 34 % 🔋



Jobs, tu futuro en la palma de tu mano



Pues ya tenemos algunas de las características de las aplicaciones nativas solo añadiendo el paquete de las PWAs en nuestro proyecto:

- Aplicación instalable
- Pantalla de splash

Vamos a añadir una pequeña mejora en nuestro archivo **package.json**. En la sección de scripts, vamos a crear un nuevo script **start:pwa** que se va a encargar de lanzar los 3 comandos que hemos lanzado anteriormente para levantar la aplicación.

```
{  
  "name": "angular-pwa-lab",  
  "version": "0.0.0",  
  "scripts": {  
    "ng": "ng",  
    "start": "ng serve",  
    "build": "ng build",  
    "watch": "ng build --watch --configuration development",  
    "test": "ng test",  
    "start:pwa": "ng build && cd dist/angular-pwa-lab && npx http-serve"  
  },  
  "private": true,  
  "dependencies": {  
    "@angular/animations": "~13.2.0",  
    "@angular/common": "~13.2.0",  
    "@angular/compiler": "~13.2.0",  
    "@angular/core": "~13.2.0",  
    "@angular/forms": "~13.2.0",  
    "@angular/platform-browser": "~13.2.0",  
    "@angular/platform-browser-dynamic": "~13.2.0",  
    "@angular/router": "~13.2.0",  
    "@angular/service-worker": "~13.2.0",  
    "rxjs": "~7.5.0",  
    "tslib": "^2.3.0",  
    "zone.js": "~0.11.4"  
  },  
  "devDependencies": {  
    "@angular-devkit/build-angular": "~13.2.1",  
    "@angular/cli": "~13.2.1",  
    "@angular/compiler-cli": "~13.2.0",  
    "@types/jasmine": "~3.10.0",  
    "@types/node": "^12.11.1",  
    "jasmine-core": "~4.0.0",  
    "karma": "~6.3.0",  
    "karma-chrome-launcher": "~3.1.0",  
    "karma-coverage": "~2.1.0",  
    "karma-jasmine": "~4.0.0",  
    "karma-jasmine-html-reporter": "~1.7.0",  
    "typescript": "~4.5.2"  
  }  
}
```

A partir de ahora, cuando queramos construir la aplicación y servirla lanzaremos el siguiente comando, en lugar de lanzar los 3 comandos que habíamos lanzado antes:

```
$ npm run start:pwa
```

Vamos a ver ahora la parte de como cachear los distintos elementos de la aplicación como los estilos, los scripts, las imágenes y los datos.

Vamos a empezar por añadir por ejemplo la librería de bootstrap directamente en el **index.html**, por lo que lo vamos a abrir y añadiremos el enlace `<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">`.

/angular-pwa-lab/src/index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularPwaLab</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
  <link rel="manifest" href="manifest.webmanifest">
  <meta name="theme-color" content="#1976d2">
</head>
<body>
  <app-root></app-root>
  <noscript>Please enable JavaScript to continue using this application.</noscript>
</body>
</html>
```

La idea es ver que los archivos principales de la aplicación ya se están cacheando según está definido en el archivo de **ngsw-config.json**, pero los estilos de Bootstrap no.

Vamos al navegador y vamos a cargar de nuevo la página. Una vez hecho esto, deberíamos de ver la aplicación con los estilos de Bootstrap aplicados.

Ahora vamos a deshabilitar la conexión a internet desde las herramientas del desarrollador, en la pestaña de **Application > Service Workers > Offline**.

Al volver a refrescar la página, veremos que no se han cargado los datos de las ofertas de trabajo.

También veremos que los estilos de Bootstrap tampoco los ha cacheado el service worker, sino que este solo ha dejado en la caché aquellos archivos que tenemos en **ngsw-config.json**, es decir, la página de HTML, el manifest, los scripts y css de la aplicación, los archivos de las carpetas de assets...

Vamos a añadir los estilos de Bootstrap en la caché, y como estos estilos se descargan desde un CDN, entonces tendremos que meterlo dentro de **resources.urls**. Al ser parte de los estilos de la aplicación, lo meteremos en el assetGroup que tiene como nombre **app**.

```
{  
  "$schema": "./node_modules/@angular/service-worker/config/schema.json",  
  "index": "/index.html",  
  "assetGroups": [  
    {  
      "name": "app",  
      "installMode": "prefetch",  
      "resources": {  
        "files": [  
          "/favicon.ico",  
          "/index.html",  
          "/manifest.webmanifest",  
          "/*.css",  
          "/*.js"  
        ],  
        "urls": [  
          "https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"  
        ]  
      }  
    },  
    {  
      "name": "assets",  
      "installMode": "lazy",  
      "updateMode": "prefetch",  
      "resources": {  
        "files": [  
          "/assets/**",  
          "/*.(svg|cur|jpg|jpeg|png|apng|webp|avif|gif|otf|ttf|woff|woff2)"  
        ]  
      }  
    }  
  ]  
}
```

Ahora, al realizar los mismos pasos que antes, deberíamos de ver que los estilos de Bootstrap si que se aplican, ya que los hemos dejado cacheados con el **service worker**.

Ahora nos toca cachear los datos de las ofertas de trabajo que pedimos a nuestro backend (en este caso a Firebase). Es decir, la petición GET.

En este caso, al ser datos dinámicos que van a cambiar, tendremos que añadirlos en la sección de **dataGroups**.

Vamos a añadir un objeto con el nombre de **datos-ofertas** y la url <https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas>.

La estrategia a utilizar esta vez es **freshness**, ya que queremos obtener siempre la versión más reciente de estos datos. Además, le vamos a poner que mantenga estos datos cacheados durante 2

minutos añadiendole **1m** como valor de **maxAge**, de esta forma podemos probarlo sin esperar una eternidad a que se eliminen.

```
{  
  "$schema": "./node_modules/@angular/service-worker/config/schema.json",  
  "index": "/index.html",  
  "assetGroups": [  
    {  
      "name": "app",  
      "installMode": "prefetch",  
      "resources": {  
        "files": [  
          "/favicon.ico",  
          "/index.html",  
          "/manifest.webmanifest",  
          "/*.css",  
          "/*.js"  
        ],  
        "urls": [  
          "https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"  
        ]  
      }  
    },  
    {  
      "name": "assets",  
      "installMode": "lazy",  
      "updateMode": "prefetch",  
      "resources": {  
        "files": [  
          "/assets/**",  
          "/*.(svg|cur|jpg|jpeg|png|apng|webp|avif|gif|otf|ttf|woff|woff2)"  
        ]  
      }  
    }  
  ],  
  "dataGroups": [  
    {  
      "name": "datos-ofertas",  
      "urls": [  
        "https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas"  
      ],  
      "cacheConfig": {  
        "strategy": "freshness",  
        "maxAge": "2m",  
        "maxSize": 5  
      }  
    }  
  ]  
}
```

Si volvemos a seguir los pasos anteriores, veremos que esta vez tras refrescar la página sin internet,

ya se carga el listado de ofertas.

Pero no se cargan los logos de las empresas, ya que todavía no le hemos dicho a Angular que cachee estas imágenes.

Para cachear las imágenes que se descargan al pintar la lista de ofertas, en este caso vamos a crear otro **dataGroup** con el nombre de **imagenes-ofertas**, vamos a pasarle la URL * y **esta vez usaremos la estrategia de *performance**.

/angular-pwa-lab/ngsw-config.json

```
{
  "$schema": "./node_modules/@angular/service-worker/config/schema.json",
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "resources": {
        "files": [
          "/favicon.ico",
          "/index.html",
          "/manifest.webmanifest",
          "/*.css",
          "/*.js"
        ],
        "urls": [
          "https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
        ]
      }
    },
    {
      "name": "assets",
      "installMode": "lazy",
      "updateMode": "prefetch",
      "resources": {
        "files": [
          "/assets/**",
          "/*.svg|cur|jpg|jpeg|png|apng|webp|avif|gif|otf|ttf|woff|woff2"
        ]
      }
    }
  ],
  "dataGroups": [
    {
      "name": "datos-ofertas",
      "urls": [
        "https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas"
      ],
      "cacheConfig": {
        "strategy": "freshness",
        "maxSize": 100,
        "maxAge": "1d"
      }
    }
  ]
}
```

```
        "maxAge": "2m",
        "maxSize": 5
    },
},
{
    "name": "imagenes-ofertas",
    "urls": [
        "https://s3.eu-west-1.amazonaws.com/www.jobfluent.com/company_logos/**/*.png"
    ],
    "cacheConfig": {
        "strategy": "performance",
        "maxAge": "20d",
        "maxSize": 5
    }
}
]
```

Con esto ya deberíamos de ver también los logos cuando entramos a la página sin internet.

25.5. Notificaciones Push

Las **Notificaciones Push** son una de las novedades más importantes de las PWAs ya que es una de las principales características que tienen las aplicaciones nativas, y que no tenían las aplicaciones web hasta que se han implementado los service workers.

Estas notificaciones son aquellas que aparecen en las aplicaciones nativas, navegadores, aplicaciones de escritorio y que avisan a los usuarios de que algo interesante ha ocurrido en la aplicación, como por ejemplo, un aviso de que te ha llegado un email nuevo, alguien te ha mencionado en un tweet...

Para recibir las alertas no es necesario tener la aplicación abierta, ni siquiera el navegador, y esto es gracias a los service workers, que se quedan en segundo plano esperando a recibir estos mensajes.

Las notificaciones push se componen de dos estándares de los navegadores:

- **Notification API:** es una API que se encarga de mostrar las notificaciones en los dispositivos. Estas notificaciones son las nativas de cada dispositivo.
- **Push API:** es una API que permite enviar mensajes desde un servidor a los navegadores.

A la hora de configurar la apariencia que va a tener la notificación, podemos usar las siguientes opciones:

- **body:** es el cuerpo de la notificación.
- **image:** la ruta hasta una imagen que se mostrará en el cuerpo de la notificación.
- **icon:** la ruta hasta un ícono que aparecerá en la notificación.
- **badge:** aquí se le pasa la ruta al ícono que queremos usar para que aparezca en la barra de notificaciones del dispositivo. No hace falta que sea un ícono en blanco y negro porque en caso de no serlo, Android se ocupa de modificarlo.
- **tag:** etiqueta que le ponemos a la notificación para que pueda agrupar por tags las notificaciones que tenemos.
- **renotify:** con esto le indicamos si queremos que el dispositivo suene cada vez que nos llegue una notificación o si por el contrario solo queremos que suene la primera vez.
- **actions:** una lista de acciones que nos saldrán junto a la notificación y nos permiten hacer algo al pulsar sobre ellas. Esta lista contendrá un objeto por acción con las siguientes propiedades:
 - **action:** nombre que se le da a la acción para poder identificarla y hacer algo con ella.
 - **title:** el texto que aparece sobre la acción.

Para poder utilizar las notificaciones Push necesitamos usar un sistema de llaves públicas/privadas llamadas Vapid Keys. Estas claves las podemos generar con alguna librería de NPM como **web-push**.

Una vez generadas estas claves, tendremos que guardar en el cliente la clave pública y en el servidor tanto la pública como la privada, ya que estas claves son las que permitirán que el navegador reciba los mensajes para levantar las notificaciones.

En el **servidor** usaremos **setVapidDetails** de la librería **web-push** para setear las claves Vapid pública y privada, junto a un email o url de contacto.

A la hora de enviar una notificación, con esta misma librería vamos a utilizar el método **sendNotification** al que le vamos a pasar la suscripción del cliente al que enviar la notificación push, y un payload con los datos que queremos mostrar en la notificación.

El **cliente** tiene que generar un objeto con una serie de datos que representan una suscripción a los mensajes push de uno de los servidores de notificaciones push que usan los navegadores, como por ejemplo Google Chrome que usa el servicio de mensajería de Firebase.

En la aplicación de Angular usaremos el servicio de **SwPush** para pedir los permisos de la aplicación con el método **requestSubscription** que recibe como parámetro un objeto con **serverPublicKey: <la-clave-vapid-publica>**.

Esta función devuelve una promesa en la que obtendremos los datos de la suscripción que tenemos que guardar en el servidor.



Es posible que las notificaciones no estén soportadas en todos los navegadores. Por ejemplo, Brave parece ser que todavía no las soporta correctamente.

25.6. Lab: Notificaciones Push en PWAs

En este laboratorio vamos a ver como enviar notificaciones Push a los dispositivos que están usando nuestra aplicación al crear una nueva oferta de trabajo.

Para este laboratorio vamos a utilizar el proyecto de Angular que hemos creado en el laboratorio **Lab: Progressive Web Apps (PWAs)**.



Vamos a empezar creando una carpeta donde crearemos la parte del backend y donde vamos a copiar el proyecto de angular.

```
$ mkdir angular-pwa-notificaciones-push-lab  
$ cd angular-pwa-notificaciones-push-lab  
$ mkdir server-notificaciones-push
```

Dentro de la carpeta **angular-pwa-notificaciones-push-lab** vamos a copiar el proyecto **angular-pwa-lab** del laboratorio anterior.

Después de esto, tenemos que tener una estructura como la siguiente:

```
|- angular-pwa-notificaciones-push-lab  
  |- server-notificaciones-push  
  |- angular-pwa-lab
```

Una vez tenemos la estructura, vamos a inicializar el proyecto del servidor.

```
$ cd server-notificaciones-push  
$ npm init -y
```

Ahora vamos a instalar las dependencias que vamos a necesitar, con los siguientes comandos:

- express: framework para crear el servidor
- cors: habilita el cors
- axios: realizar peticiones http
- web-push: se encarga de las notificaciones push
- dotenv: carga variables de entorno de un archivo .env
- nodemon: watcher para levantar el servidor con cada cambio

```
$ npm install --save express cors web-push dotenv axios  
$ npm install --save-dev nodemon
```

Con esto, ya deberíamos de tener un archivo **package.json** con la información y dependencias del proyecto.

```
{  
  "name": "server-notificaciones-push",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "axios": "^0.25.0",  
    "cors": "^2.8.5",  
    "dotenv": "^16.0.0",  
    "express": "^4.17.2",  
    "web-push": "^3.4.5"  
  },  
  "devDependencies": {  
    "nodemon": "^2.0.15"  
  }  
}
```

Vamos a sustituir el script **test** por uno de **start** que va a arrancar la aplicación con **nodemon**.

```
{  
  "name": "server-notificaciones-push",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "nodemon src/app.js"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "axios": "^0.25.0",  
    "cors": "^2.8.5",  
    "dotenv": "^16.0.0",  
    "express": "^4.17.2",  
    "web-push": "^3.4.5"  
  },  
  "devDependencies": {  
    "nodemon": "^2.0.15"  
  }  
}
```

Ahora vamos a crear el archivo **app.js** dentro de una carpeta **src** en este proyecto.

Dentro de este archivo vamos a crear un servidor con **express** y lo pondremos a escuchar peticiones en el puerto **3000**.

```
const express = require('express')  
  
const app = express()  
  
app.listen(3000, () => {  
  console.log('Listening on http://localhost:3000')  
})
```

Ahora vamos a utilizar los middlewares de **cors** y **express.json**, el primero para poder recibir peticiones desde otros dominios y el segundo para guardar el cuerpo de las peticiones POST, PUT... en una propiedad **body** de la petición.

```
/angular-pwa-notificaciones-push-lab/server-notificaciones-push/src/app.js
```

```
const express = require('express')
const cors = require('cors')

const app = express()

app.use(cors())
app.use(express.json())

app.listen(3000, () => {
  console.log('Listening on http://localhost:3000')
})
```

Al principio del todo, vamos a hacer que se carguen las variables de entorno con la dependencia **dotenv**.

```
/angular-pwa-notificaciones-push-lab/server-notificaciones-push/src/app.js
```

```
require('dotenv').config()
const express = require('express')
const cors = require('cors')

const app = express()

app.use(cors())
app.use(express.json())

app.listen(3000, () => {
  console.log('Listening on http://localhost:3000')
})
```

Ahora vamos a generar nuestras Vapid Keys para poder enviar notificaciones push. Para generarlas necesitamos instalar de forma global la dependencia **web-push** y con ella generarlas.

```
$ npm install -g web-push
$ web-push generate-vapid-keys --json

{"publicKey": "BA1MZ0EcfvkMVhiK0qG6K1Z5W1XeS5kDk- SA37HT37nRPr2NeKtKA_te7eSAqcwwaSC8sIkV14b0NdcApScrVf3", "privateKey": "89DfSIcqj5R7ngBkJV11FB0jdNliFZLptmEqz1X0mik"}
```

Vamos a crear un archivo **.env** dentro de la carpeta **server-notificaciones-push** y vamos a añadir dos variables con esos dos valores que hemos obtenido al generar las claves.

```
/angular-pwa-notificaciones-push-lab/server-notificaciones-push/.env
```

```
VAPID_PUBLIC_KEY=BA1MZ0EcfvkMVhiK0qG6K1Z5W1XeS5kDk- SA37HT37nRPr2NeKtKA_te7eSAqcwwaSC8sIkV14b0NdcApScrVf3
VAPID_PRIVATE_KEY=89DfSIcqj5R7ngBkJV11FB0jdNliFZLptmEqz1X0mik
```

Ahora vamos a setear para la librería webpush estas claves Vapid, para ello, al ejecutar nuestro **app.js** vamos a llamar a la función **webpush.setVapidDetails** pasandole como parámetros una url o email de contacto, la clave pública y la privada.

/angular-pwa-notificaciones-push-lab/server-notificaciones-push/src/app.js

```
require('dotenv').config()
const express = require('express')
const cors = require('cors')
const webpush = require('web-push')

const app = express()

webpush.setVapidDetails('mailto:contact@jobs.com', process.env.VAPID_PUBLIC_KEY, process.env.VAPID_PRIVATE_KEY)

app.use(cors())
app.use(express.json())

app.listen(3000, () => {
  console.log('Listening on http://localhost:3000')
})
```

Ahora vamos a crear nuestro modelo de ofertas para poder crear este tipo de objetos y guardarlas en Firebase, donde las estabamos guardando desde el Front en el anterior laboratorio.

Aquí usaremos **axios** para realizar la petición POST.

/angular-pwa-notificaciones-push-lab/server-notificaciones-push/src/models/oferta.model.js

```
const axios = require('axios')

class Oferta {
  constructor(id, titulo, descripcion, empresa, salario, ciudad, urlImagen) {
    this.id = id
    this.titulo = titulo
    this.descripcion = descripcion
    this.empresa = empresa
    this.salario = salario
    this.ciudad = ciudad
    this.urlImagen = urlImagen
  }

  save() {
    return axios.post('https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas.json', this)
  }
}

module.exports = Oferta
```

Ahora vamos a crear un controlador para guardar las suscripciones a las notificaciones push de los clientes que aceptan los permisos para recibirlas.

Dentro del controlador vamos a crear una función **saveSuscripcionPush** en la que vamos a obtener los datos de la suscripción del cuerpo de la petición. Estos datos tendriamos que guardarlos

en alguna BBDD, pero nosotros lo vamos a hacer en un archivo JSON en **src/suscripciones/suscripciones-db.json**.

/angular-pwa-notificaciones-push-lab/server-notificaciones-push/src/controllers/push.controller.js

```
const path = require('path')
const fs = require('fs').promises

exports.saveSuscripcionPush = (req, res) => {
  const suscripcion = req.body
  const pathSuscripciones = path.join(__dirname, '..', 'suscripciones', 'suscripciones-db.json')

}
```

Ahora vamos a leer el archivo donde se guardan las suscripciones para no perder las que ya teníamos guardadas. Una vez obtenidas, vamos a añadir al final esta nueva y las volveremos a guardar en el mismo archivo.

/angular-pwa-notificaciones-push-lab/server-notificaciones-push/src/controllers/push.controller.js

```
const path = require('path')
const fs = require('fs').promises

exports.saveSuscripcionPush = (req, res) => {
  const suscripcion = req.body
  const pathSuscripciones = path.join(__dirname, '..', 'suscripciones', 'suscripciones-db.json')

  fs.readFile(pathSuscripciones)
    .then(data => {
      const suscripciones = JSON.parse(data)
      suscripciones.push(suscripcion)
      return fs.writeFile(pathSuscripciones, JSON.stringify(suscripciones, null, 2))
    })
    .then(() => {
      return res.json({msg: 'suscripción guardada'})
    })
}
```

Ahora vamos a crear otro controlador para las ofertas, y en él vamos a crear un método **createOfertaTrabajo**.

Dentro de este método, primero vamos a obtener los datos que llegan en la petición POST y vamos a crear un objeto Oferta desde el que vamos a llamar al método **save** del modelo que hemos creado antes.

/angular-pwa-notificaciones-push-lab/server-notificaciones-push/src/controllers/ofertas.controller.js

```
const path = require('path')
const fs = require('fs').promises
const webpush = require('web-push')
const Oferta = require("../models/oferta.model")

exports.createOfertaTrabajo = (req, res) => {
  const { titulo, empresa, ciudad, salario, urlImagen, descripcion } = req.body
  const oferta = new Oferta(null, titulo, descripcion, empresa, salario, ciudad, urlImagen)

  oferta.save()
    .then(resp => {

      })
}
```

Una vez obtenemos la respuesta de la petición POST vamos a definir el objeto de payload con el que vamos a indicar que como se tiene que mostrar la notificación en los dispositivos.

/angular-pwa-notificaciones-push-lab/server-notificaciones-push/src/controllers/ofertas.controller.js

```
const path = require('path')
const fs = require('fs').promises
const webpush = require('web-push')
const Oferta = require("../models/oferta.model")

exports.createOfertaTrabajo = (req, res) => {
  const { titulo, empresa, ciudad, salario, urlImagen, descripcion } = req.body
  const oferta = new Oferta(null, titulo, descripcion, empresa, salario, ciudad, urlImagen)

  oferta.save()
    .then(resp => {

      const payload = {
        notification: {
          title: 'Nueva oferta de trabajo',
          body: `${titulo} - ${salario}`,
          image: urlImagen,
          vibrate: [100, 50, 150],
        }
      }

    })
}
```

Ahora vamos a leer nuestro archivo de suscripciones para recorrerlas e ir enviando una notificación push a cada una de ellas con el método **webpush.sendNotification** al que le vamos a pasar la suscripción (objeto que indica a que dispositivo hay que enviar la notificación y a través de que servicio de mensajería hacerlo), y un JSON con el payload.

```

const path = require('path')
const fs = require('fs').promises
const webpush = require('web-push')
const Oferta = require("../models/oferta.model")

exports.createOfertaTrabajo = (req, res) => {
    const { titulo, empresa, ciudad, salario, urlImagen, descripcion } = req.body
    const oferta = new Oferta(null, titulo, descripcion, empresa, salario, ciudad, urlImagen)

    oferta.save()
        .then(resp => {

            const payload = {
                notification: {
                    title: 'Nueva oferta de trabajo',
                    body: `${titulo} - ${salario}`,
                    image: urlImagen,
                    vibrate: [100, 50, 150],
                }
            }

            fs.readFile(path.join(__dirname, '..', 'suscripciones', 'suscripciones-db.json'))
                .then(data => {
                    const suscripciones = JSON.parse(data)
                    suscripciones.forEach((suscripcion) => {
                        webpush.sendNotification(suscripcion, JSON.stringify(payload))
                            .then(() => {
                                console.log('Notificación PUSH enviada')
                            })
                            .catch(err => {
                                console.log(err)
                            })
                    })
                    return res.json({name: resp.data.name})
                })
        })
    })
}

```

Ahora vamos a crear un archivo de rutas para poner las rutas y que método de los controladores tienen que ejecutar.

/angular-pwa-notificaciones-push-lab/server-notificaciones-push/src/routes/app.routes.js

```
const express = require('express')
const OfertasController = require('../controllers/ofertas.controller')
const PushController = require('../controllers/push.controller')

const router = express.Router()

router.post('/ofertas', OfertasController.createOfertaTrabajo)
router.post('/suscribir-push', PushController.saveSuscripcionPush)

module.exports = router
```

Ahora vamos a añadir estas rutas en un **app.use** dentro del archivo principal.

/angular-pwa-notificaciones-push-lab/server-notificaciones-push/src/app.js

```
require('dotenv').config()
const express = require('express')
const cors = require('cors')
const webpush = require('web-push')
const appRoutes = require('./routes/app.routes')

const app = express()

webpush.setVapidDetails('mailto:contact@jobs.com', process.env.VAPID_PUBLIC_KEY, process.env.VAPID_PRIVATE_KEY)

app.use(cors())
app.use(express.json())
app.use(appRoutes)

app.listen(3000, () => {
  console.log('Listening on http://localhost:3000')
})
```

Con esto ya tenemos la parte del backend. Podemos levantar el servidor con el siguiente comando:

```
$ npm start
```

Ahora nos toca modificar nuestra aplicación de Angular para poder pedir el permiso para recibir notificaciones y así enviar los datos de la suscripción al servidor y que este los guarde en el archivo JSON que habíamos generado.

Esto lo vamos a hacer en el componente App, en el que tendremos que injectar el servicio **SwPush** con el que llamaremos al método **requestSubscription** que recibe como parámetro un objeto con la Vapid Key pública que habíamos generado para el servidor.

```
import { Component } from '@angular/core';
import { SwPush } from '@angular/service-worker';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private swPush: SwPush) {
    this.pedirPermisoNotificaciones()
  }

  private pedirPermisoNotificaciones() {
    this.swPush.requestSubscription({
      serverPublicKey: 'BA1MZ0EcFvkMVhiK0qG6K1Z5W1XeS5kDk-SA37HT37nRPr2NeKtKA_te7eSAqcwwaSC8sIkV14b0NdcApSCrVf3'
    })
    .then((suscripcion: PushSubscription) => {

    })
  }
}
```

Esta suscripción es la que tenemos que guardar en el servidor para que luego nos pueda enviar notificaciones Push, por lo que vamos a crear un servicio nuevo que se va a encargar de realizar la petición POST a nuestro servidor.

Navegamos dentro del proyecto de Angular desde un terminal y lanzamos el siguiente comando:

```
$ cd angular-pwa-notificaciones-push-lab/angular-pwa-lab
$ ng g s notificaciones-push
```

Ahora dentro del servicio vamos a inyectar el servicio **HttpClient** y vamos a crear un método para realizar la petición POST.

/angular-pwa-notificaciones-push-lab/angular-pwa-lab/src/app/notificaciones-push.service.ts

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

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

  constructor(private http: HttpClient) { }

  saveSuscripcion(suscripcion: PushSubscription): Observable<any> {
    return this.http.post('http://localhost:3000/suscribir-push', suscripcion)
  }
}
```

Ahora que tenemos el servicio, vamos de nuevo al componente App, donde vamos a inyectar este servicio y llamar al método que hemos creado pasandole como parámetro los datos de la suscripción.

/angular-pwa-notificaciones-push-lab/angular-pwa-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { SwPush } from '@angular/service-worker';
import { NotificacionesPushService } from './notificaciones-push.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private swPush: SwPush, private notificacionesPush: NotificacionesPushService) {
    this.pedirPermisoNotificaciones()
  }

  private pedirPermisoNotificaciones() {
    this.swPush.requestSubscription({
      serverPublicKey: 'BA1MZ0EcfvkMVhiK0qG6K1Z5W1XeS5kDk-SA37HT37nRPr2NeKtKA_te7eSAqcwwaSC8sIkV14b0NdApSCrVf3'
    })
      .then((suscripcion: PushSubscription) => {
        this.notificacionesPush.saveSuscripcion(suscripcion)
          .subscribe({
            next: (resp: any) => {
              console.log(resp)
            },
            error: (err: any) => {
              console.log(err)
            }
          })
      })
  }
}
```

Con esto ya podemos pedir el permiso en el navegador para poder recibir notificaciones push.

Ahora tenemos que realizar otro cambio, y es que cuando guardamos una nueva oferta de trabajo, estamos haciendo la petición a Firebase, pero ahora tenemos que hacerla a nuestro servidor con express para que este se encargue de enviar la notificación Push.

Por tanto en el servicio de Ofertas, tenemos que cambiar la URL a la que se está haciendo la petición POST.

/angular-pwa-notificaciones-push-lab/angular-pwa-lab/src/app/ofertas.service.ts

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

@Injectable({
  providedIn: 'root'
})
export class OfertasService {
  URL: string = 'https://fluentjobs-fa22e.firebaseio.com/ejemplo-pwa/ofertas'

  constructor(private http: HttpClient) { }

  getOfertas(): Observable<any> {
    return this.http.get(`${this.URL}.json`)
      .pipe(
        map((objOfertas: any) => {
          const arrOfertas = []

          for (let id in objOfertas) {
            arrOfertas.push({...objOfertas[id], id})
          }

          return arrOfertas
        })
      )
  }

  getOferta(id: string): Observable<any> {
    return this.http.get(`${this.URL}/${id}.json`)
  }

  crearOferta(nuevaOferta: any): Observable<any> {
    return this.http.post('http://localhost:3000/ofertas', nuevaOferta)
  }
}
```

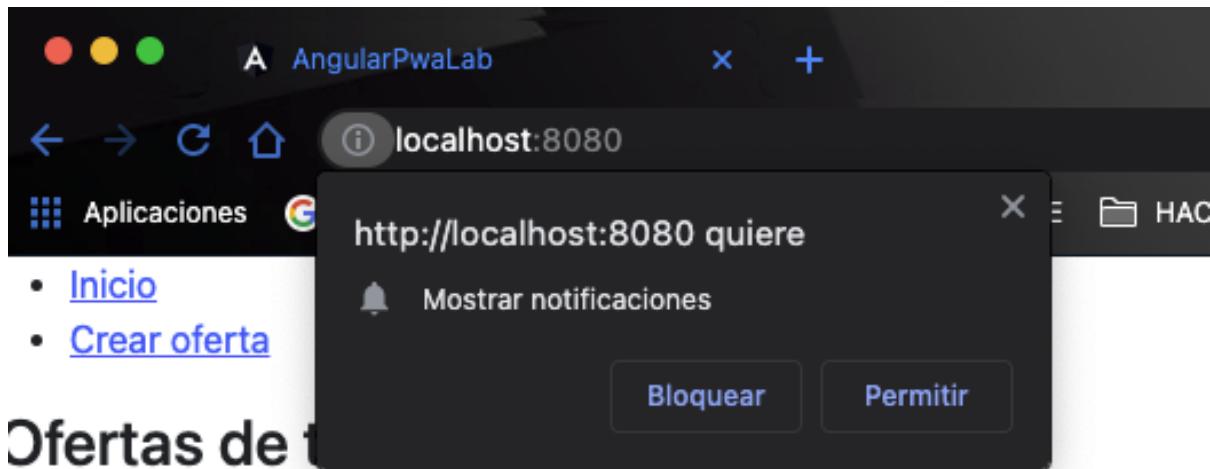
Y con esto ya tenemos nuestra aplicación de angular lista para recibir notificaciones push.

Vamos a comprobar que todo esto funciona.

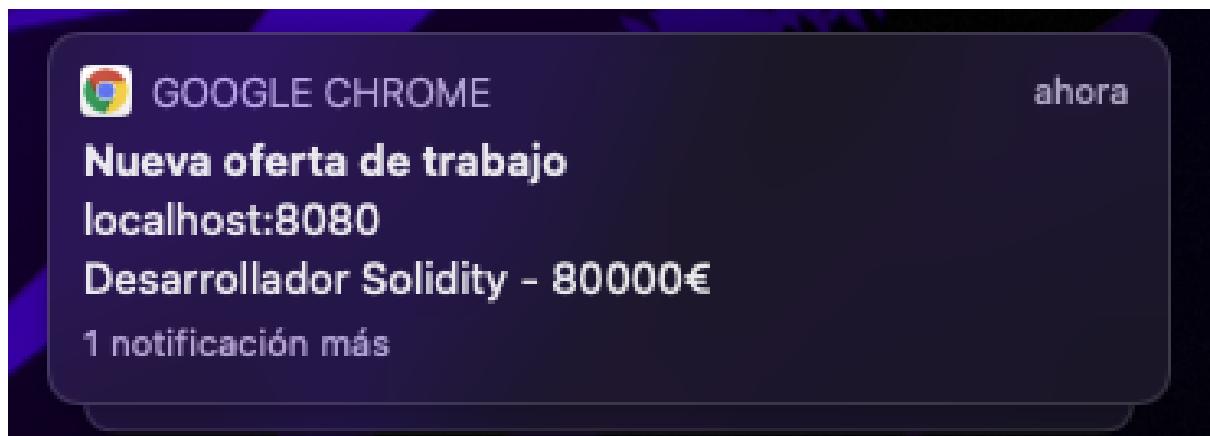
Ya teníamos levantado el servidor de express, y ahora vamos a lanzar el siguiente comando dentro del proyecto de angular para levantar esta aplicación.

```
$ npm run start:pwa
```

Una vez levantados los dos servidores, vamos a entrar en el navegador en <http://localhost:8080/> y debería de salirnos un popup pidiéndonos permiso para recibir las notificaciones.

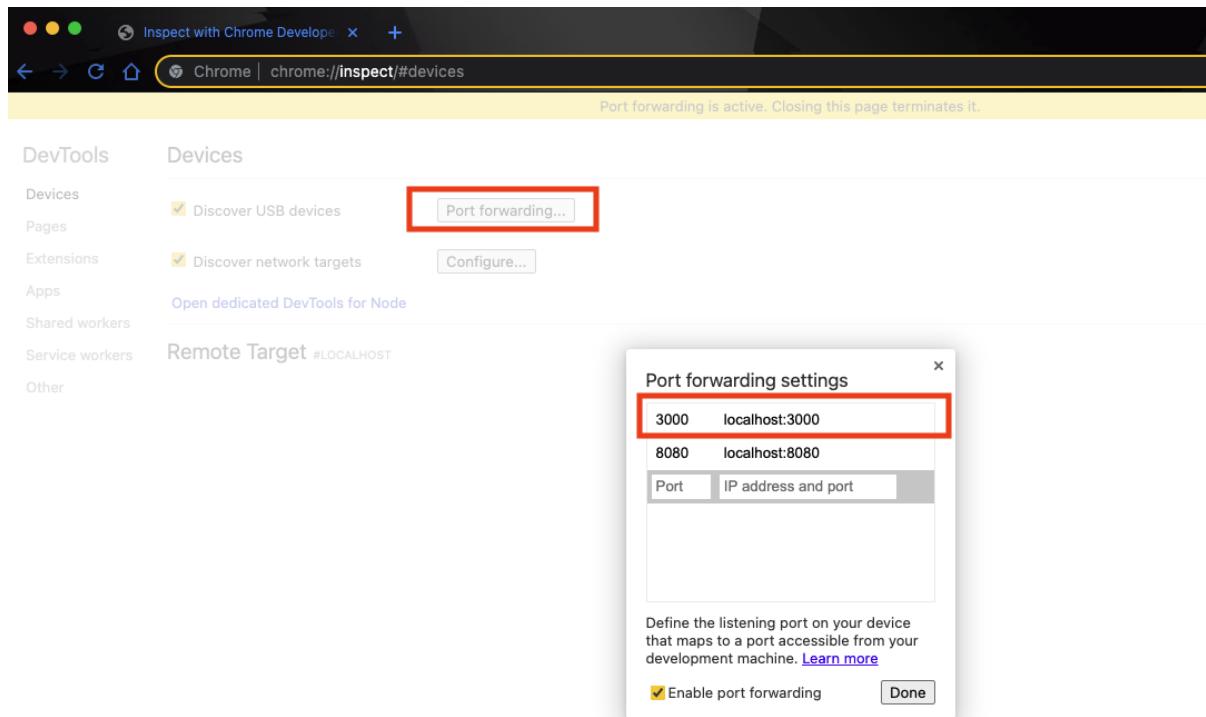


Ahora ya podemos crear una nueva oferta de trabajo y debería de salirnos una notificación en nuestro dispositivo.



Esto funciona de la misma forma en los dispositivos móviles, hay que conectar el móvil (con la depuración USB activada) al ordenador.

En el ordenador entramos en <chrome://inspect/#devices> para comprobar que aparece el dispositivo conectado. Dentro de esta pestaña tenemos que habilitar el **port forwarding** para el puerto **3000** a parte del **8080**, ya que haremos peticiones al servidor de express desde el móvil.



Una vez activado, al entrar en <http://localhost:8080/> dentro del dispositivo móvil, debería de salirnos también la petición de permisos para recibir notificaciones.

23:56

15.1 KB/S 72 %



localhost:8080



- [Inicio](#)
- [Crear oferta](#)

Ofertas de trabajo

Empresa 1

[Diseñador/a Gráfico/a in Madrid](#)



http://localhost:8080 quiere enviarte
notificaciones

Bloquear

Permitir



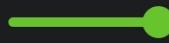
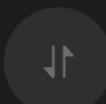
Al guardar una oferta nueva también nos tiene que salir la notificación en el dispositivo, y al igual que en las aplicaciones nativas se añade un circulito sobre el icono de la aplicación.

23:57

72 %

Jue., 3 feb.

0.06
KB/S



Modo avión



Notificaciones

Sistema Android

Depuración USB habilitada

Toca aquí para desactivar la depuración USB

jobs • localhost:8080 • ahora



Nueva oferta de trabajo

Desarrollador Solidity - 80000€



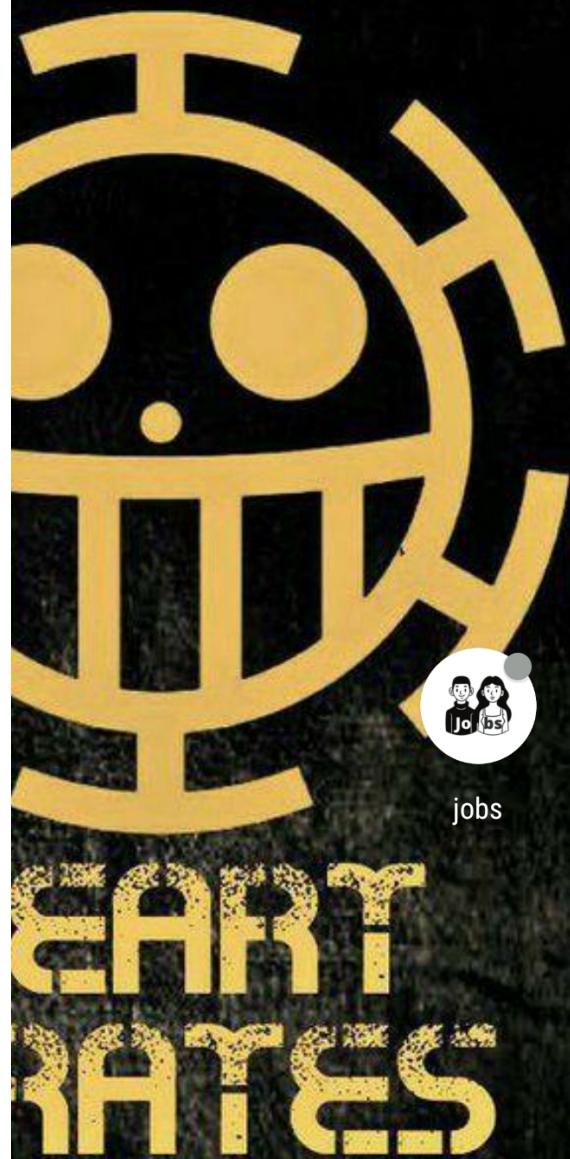
Gestionar

Borrar todo



23:57

2.84 KB/S 72 %



jobs



Pues con esto ya hemos visto como añadir notificaciones push en nuestra aplicación.

Chapter 26. Angular Universal

Angular Universal es una herramienta que permite hacer el **Server Side Rendering** (SSR) y **pre-rendering** de las aplicaciones de Angular.

26.1. Server Side Rendering

El **SSR** consiste en realizar el primer renderizado de la aplicación en el servidor, con lo que conseguimos que la primera vez que llega la página al cliente no venga vacía como ocurre de forma normal con las SPAs.

Gracias al SSR conseguimos que la primera página de la aplicación se muestre muy rápido al cliente, algo que es muy beneficioso, ya que según la regla de los 3 segundos de la UX, si una página tarda en cargar más de 3 segundos estas perdiendo el interés de los usuarios por tu aplicación.

Otro de los beneficios del SSR es que ayuda a que nuestras aplicaciones también puedan funcionar en dispositivos con menos recursos, sobre todo si estamos realizando un trabajo intenso con los scripts, ya que es este lo tendrán que hacer en estos dispositivos. Al menos con esto, le hemos quitado parte del trabajo inicial de pintar la primera página mediante scripts en el cliente.

Y si nos interesa que los buscadores nos tengan en cuenta a la hora de posicionarnos, esto es un gran paso, ya que no es lo mismo que estos cuando van indexando las páginas se encuentren con una página en blanco, a que se la encuentren con contenido.

Los comandos que debemos utilizar para construir la aplicación con SSR y servirla son los siguientes:

```
$ npm run build:ssr  
$ npm run serve:ssr
```

En el caso de querer que se vaya compilando automáticamente con cada cambio que realicemos sobre los archivos y que estos cambios se refresquen en el navegador, entonces usaremos este otro comando:

```
$ npm run dev:ssr
```

26.2. Lab: Server Side Rendering

En este laboratorio vamos a ver la diferencia entre cargar una aplicación de Angular con SSR y sin el.

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

```
$ ng new angular-universal-ssr-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 añadiremos la librería de angular universal:

```
$ cd angular-universal-ssr-lab  
$ ng add @nguniversal/express-engine  
  
The package @nguniversal/express-engine@13.0.2 will be installed and executed.  
Would you like to proceed? Yes
```

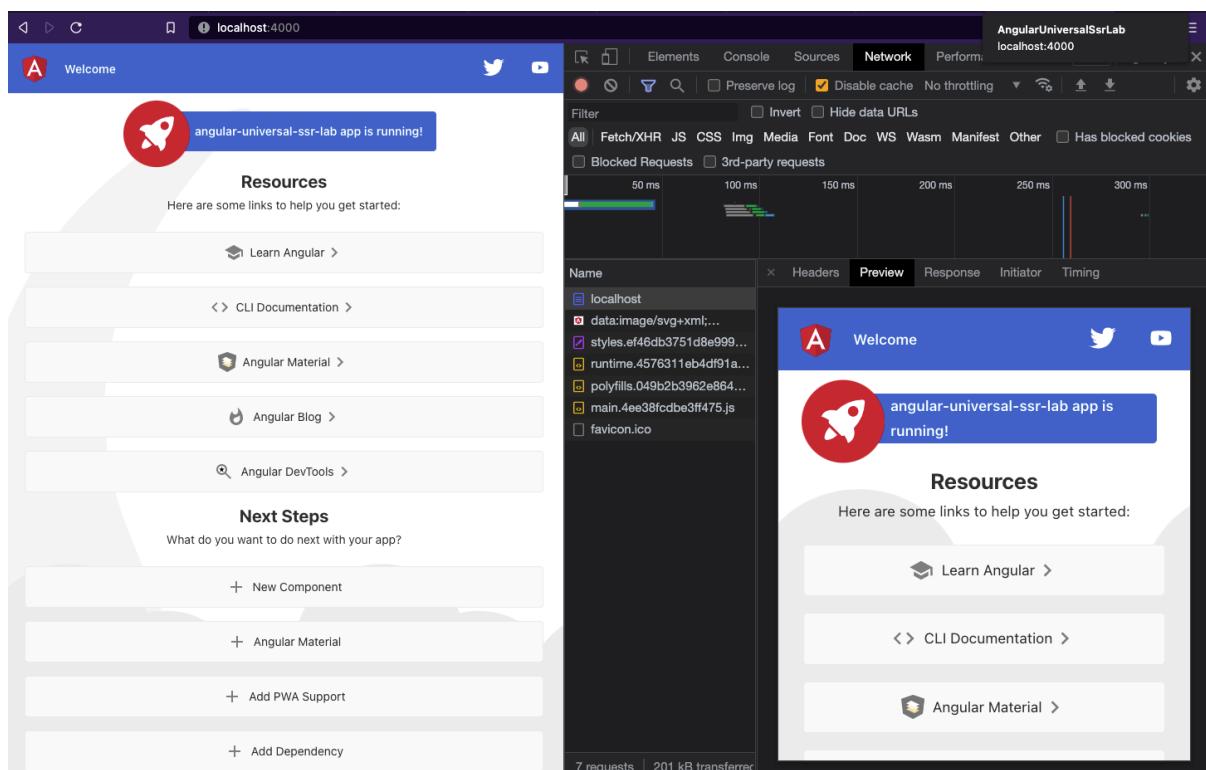
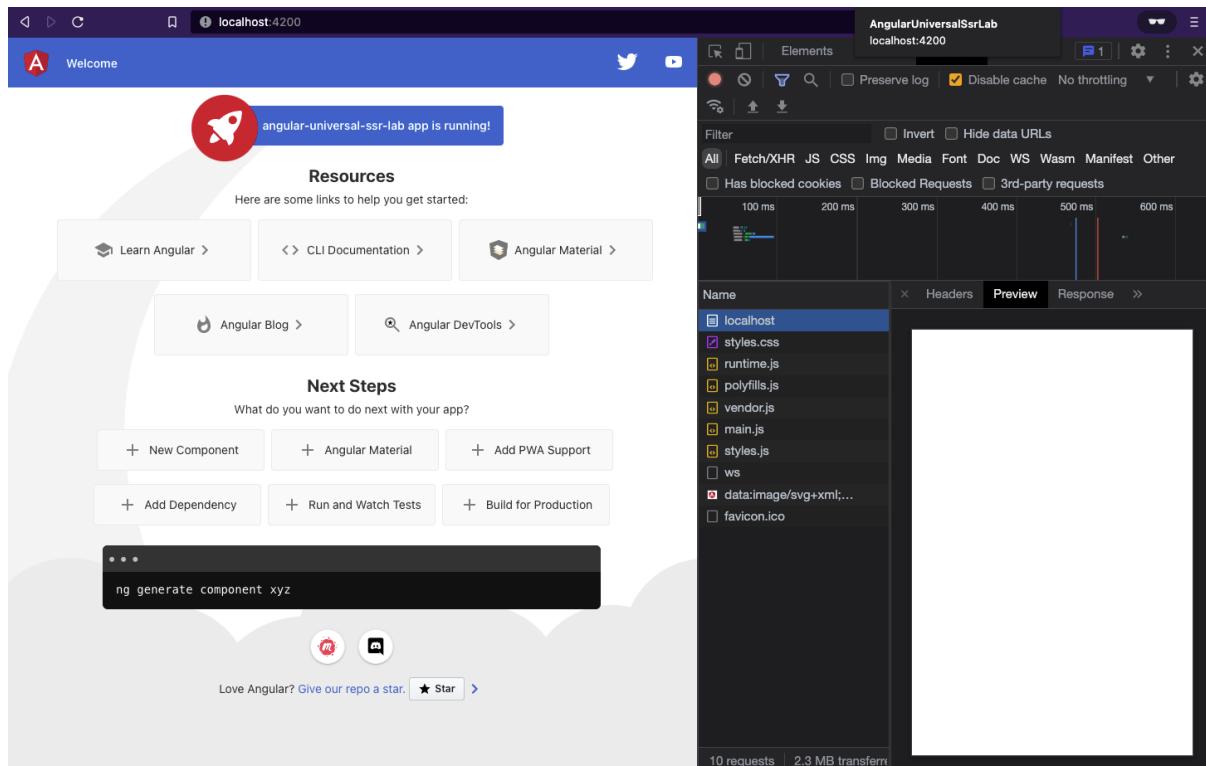
Una vez añadida, vamos a levantar dos servidores, con los siguientes comandos:

```
$ ng s  
$ npm run build:ssr && npm run serve:ssr
```

Con el primero se levanta la aplicación en <http://localhost:4200/>, sin SSR.

Mientras que con el segundo, se levanta en <http://localhost:4000/>, con SSR.

Si entramos en ambas direcciones y vamos a las herramientas del desarrollador, en la pestaña de **Network** buscamos la petición de **localhost** y entramos en la **preview** veremos que cuando la usamos sin SSR la página está en blanco, mientras que cuando usamos el SSR, la página tiene contenido.



26.3. Prerendering

El **prerendering** consiste en la generación de páginas HTML estáticas a partir de las páginas dinámicas.

Esto puede sernos útil si tenemos aplicaciones como un blog en el que el contenido es estático y no cambia muy a menudo.

Esto se debe a que con cada cambio que queramos realizar sobre el proyecto, tendremos que volver a prerenderizar las páginas para que se generen de nuevo con los últimos cambios.

Dentro del archivo de **angular.json** se genera una propiedad (por el final) **prerender** donde podemos configurar que rutas de la aplicación queremos prerenderizar.

Indicaremos que rutas son con **options.routes**, al que le pasamos un array de strings con las rutas. Aunque también podemos utilizar **options.routesFile** al que se le asigna como valor la ruta a un archivo en el que se han definido las rutas.

Para generar las páginas estáticas hay que lanzar el siguiente comando, habiéndole indicado previamente en el archivo **angular.json** que rutas queremos generar:

```
$ npm run prerender
```

Una vez prerenderizadas, podemos ver el resultado en la carpeta **dist/<nombre-proyecto>/browser**.

26.4. Lab: Prerendering

En este laboratorio vamos a prerenderizar una wiki con los datos de los digimon que vamos a extraer de la API <https://digimon-api.vercel.app/>.

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

```
$ ng new angular-universal-prerendering-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, generaremos unos componentes y un servicio:

```
$ cd angular-universal-prerendering-lab  
$ ng g c lista-digimon  
$ ng g c digimon  
$ ng g s digimon-api
```

Ahora vamos a añadir Angular Universal al proyecto y después levantaremos el servidor de desarrollo para ir construyendo la aplicación:

```
$ ng add @nguniversal/express-engine  
The package @nguniversal/express-engine@13.0.2 will be installed and executed.  
Would you like to proceed? Yes  
  
$ ng s
```

Vamos a crear una clase **Digimon** en una carpeta **models** para crear el modelo.

La API nos devuelve por cada digimon solo 3 propiedades:

- El nombre: name
- La imagen: img
- El nivel: level

/angular-universal-prerendering-lab/src/app/models/digimon.model.ts

```
export class Digimon {  
  constructor(public name: string, public img: string, public level: string) {}  
}
```

Ahora vamos a crear un archivo de rutas **app.routes.ts** en la carpeta **app**.

Dentro de este archivo vamos a definir 3 rutas:

- / → Lista el nombre de todos los digimons
- /digimon/:name → Muestra los datos de un digimon dado su nombre
- /digimon/level/:level → Lista el nombre de todos los digimons filtrados por el nivel

/angular-universal-prerendering-lab/src/app/app.routes.ts

```
import { RouterModule, Routes } from '@angular/router'
import { DigimonComponent } from './digimon/digimon.component'
import { ListaDigimonComponent } from './lista-digimon/lista-digimon.component'

const APP_ROUTES: Routes = [
  { path: '', component: ListaDigimonComponent },
  { path: 'digimon/:name', component: DigimonComponent },
  { path: 'digimon/level/:level', component: ListaDigimonComponent },
]

export const RoutingModule = RouterModule.forRoot(APP_ROUTES)
```

En el módulo de la aplicación vamos a importar el **RoutingModule** y el **HttpClientModule**.

/angular-universal-prerendering-lab/src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { ListaDigimonComponent } from './lista-digimon/lista-digimon.component';
import { DigimonComponent } from './digimon/digimon.component';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent,
    ListaDigimonComponent,
    DigimonComponent
  ],
  imports: [
    BrowserModule.withServerTransition({ appId: 'serverApp' }),
    RouterModule,
    HttpClientModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Dentro del TS del componente App vamos a declarar un array con los niveles de los digimon.

/angular-universal-prerendering-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 {
  levels: Array<string> = [
    'in training',
    'fresh',
    'training',
    'rookie',
    'champion',
    'ultimate',
    'mega',
  ]
}
```

En la plantilla vamos a poner un enlace para poder navegar hasta el inicio, y después crearemos un enlace por cada nivel del array.

Además de poner los enlaces, vamos a añadir el componente **router-outlet** para mostrar ahí los componentes asociados a las rutas.

/angular-universal-prerendering-lab/src/app/app.component.html

```
<h1>Digimon Wiki</h1>

<ul>
  <li>
    <a [routerLink]="/">Inicio</a>
  </li>
  <li *ngFor="let level of levels">
    <a [routerLink]="/digimon", 'level', level]>{{level | titlecase}}</a>
  </li>
</ul>

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

El siguiente paso es añadir las 3 peticiones GET en el servicio para poder obtener los datos de la API:

- getDigimons: obtiene todos los digimons
- getDigimonByName: obtiene el digimon dado su nombre

- `getDigimonByLevel`: obtiene todos los digimons del nivel dado

/angular-universal-prerendering-lab/src/app/digimon-api.service.ts

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { map, Observable } from 'rxjs';
import { Digimon } from './models/digimon.model';

@Injectable({
  providedIn: 'root'
})
export class Digimon ApiService {
  URL: string = 'https://digimon-api.vercel.app/api/digimon'

  constructor(private http: HttpClient) { }

  getDigimons(): Observable<Array<Digimon>> {
    return this.http.get<Array<Digimon>>(this.URL)
  }

  getDigimonByName(name: string): Observable<Digimon> {
    return this.http.get<Array<Digimon>>(` ${this.URL}/name/${name}`)
      .pipe(
        map((digimons: Array<Digimon>) => digimons[0])
      )
  }

  getDigimonByLevel(level: string): Observable<Array<Digimon>> {
    return this.http.get<Array<Digimon>>(` ${this.URL}/level/${level}`)
  }
}
```

Pues ya que tenemos las peticiones, vamos a añadir el código del componente que muestra los datos de un digimon.

En el componente **DigimonComponent** vamos a injectar el servicio que hemos creado además del **ActivatedRoute** para poder obtener el nombre de la URL.

/angular-universal-prerendering-lab/src/app/digimon/digimon.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { map } from 'rxjs';
import { Digimon ApiService } from '../digimon-api.service';
import { Digimon } from '../models/digimon.model';

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

  constructor(private activatedRoute: ActivatedRoute, private digimonApi: Digimon ApiService) { }

  ngOnInit(): void {
  }
}
```

En el método **ngOnInit** vamos a pedir el parámetro de la ruta con el **activatedRoute.snapshot.paramMap** ya que esta vez cada vez que cambiemos de URL se elimina y crea el componente.

Y este nombre se lo vamos a pasar a la función **getDigimonByName** de nuestro servicio.

/angular-universal-prerendering-lab/src/app/digimon/digimon.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { map } from 'rxjs';
import { Digimon ApiService } from '../digimon-api.service';
import { Digimon } from '../models/digimon.model';

@Component({
  selector: 'app-digimon',
  templateUrl: './digimon.component.html',
  styleUrls: ['./digimon.component.css']
})
export class DigimonComponent implements OnInit {
  digimon: Digimon = new Digimon('', '', '')

  constructor(private activatedRoute: ActivatedRoute, private digimonApi: Digimon ApiService) { }

  ngOnInit(): void {
    const name = this.activatedRoute.snapshot.paramMap.get('name')
    this.digimonApi.getDigimonByName(name!)
  }
}
```

Por último, nos suscribimos al observable y vamos a guardar los datos del digimon en una propiedad del componente.

/angular-universal-prerendering-lab/src/app/digimon/digimon.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { map } from 'rxjs';
import { Digimon ApiService } from '../digimon-api.service';
import { Digimon } from '../models/digimon.model';

@Component({
  selector: 'app-digimon',
  templateUrl: './digimon.component.html',
  styleUrls: ['./digimon.component.css']
})
export class DigimonComponent implements OnInit {
  digimon: Digimon = new Digimon('', '', '')

  constructor(private activatedRoute: ActivatedRoute, private digimonApi: Digimon ApiService) { }

  ngOnInit(): void {
    const name = this.activatedRoute.snapshot.paramMap.get('name')
    this.digimonApi.getDigimonByName(name!)
      .subscribe((digimon: Digimon) => {
        this.digimon = digimon
      })
  }
}
```

En la plantilla vamos a mostrar la imagen, el nombre y pondremos un enlace que nos va a mandar a la página que lista todos los digimon del mismo nivel.

/angular-universal-prerendering-lab/src/app/digimon/digimon.component.html

```
<div>
  <img [src]="digimon.img" alt="Imagen de {{digimon.name}}" width="150">
  <h2>{{digimon.name}}</h2>
  <p>Level: <a [routerLink]=['/digimon', 'level', digimon.level]">{{digimon.level}}</a></p>
</div>
```

En el componente **ListaDigimonComponent** vamos a pintar una lista de digimons, y para ello vamos a crearnos una propiedad para guardarlos.

Inyectaremos en el constructor nuestro servicio y el **ActivatedRoute**, al igual que en el componente anterior.

Y también vamos a crear un método **getLista** con el que vamos a devolver un observable.

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { Observable } from 'rxjs';
import { Digimon ApiService } from '../digimon-api.service';
import { Digimon } from '../models/digimon.model';

@Component({
  selector: 'app-lista-digimon',
  templateUrl: './lista-digimon.component.html',
  styleUrls: ['./lista-digimon.component.css']
})
export class ListaDigimonComponent implements OnInit {
  digimons: Array<Digimon> = []

  constructor(private activatedRoute: ActivatedRoute, private digimonApi: Digimon ApiService) { }

  ngOnInit(): void {
  }

  getLista(): Observable<Array<Digimon>> {
  }
}
```

Dentro de **getLista**, vamos a comprobar si la ruta tiene un parámetro **level** o no.

Si tiene el parámetro, vamos a llamar al método **getDigimonByLevel** y si no llamaremos a **getDigimons**. Vamos a devolver el observable que devuelven estos dos métodos.

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { Observable } from 'rxjs';
import { Digimon ApiService } from '../digimon-api.service';
import { Digimon } from '../models/digimon.model';

@Component({
  selector: 'app-lista-digimon',
  templateUrl: './lista-digimon.component.html',
  styleUrls: ['./lista-digimon.component.css']
})
export class ListaDigimonComponent implements OnInit {
  digimons: Array<Digimon> = []

  constructor(private activatedRoute: ActivatedRoute, private digimonApi: Digimon ApiService) { }

  ngOnInit(): void {
  }

  getLista(): Observable<Array<Digimon>> {
    const params: ParamMap = this.activatedRoute.snapshot.paramMap
    if (params.has('level')) {
      const level = params.get('level')!
      return this.digimonApi.getDigimonByLevel(level)
    }

    return this.digimonApi.getDigimons()
  }
}
```

Por último, en el **ngOnInit** nos vamos a suscribir al observable que retorne la función **getLista** y vamos a asignarle la lista de digimons a la propiedad que habíamos declarado antes.

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { Observable } from 'rxjs';
import { Digimon ApiService } from '../digimon-api.service';
import { Digimon } from '../models/digimon.model';

@Component({
  selector: 'app-lista-digimon',
  templateUrl: './lista-digimon.component.html',
  styleUrls: ['./lista-digimon.component.css']
})
export class ListaDigimonComponent implements OnInit {
  digimons: Array<Digimon> = []

  constructor(private activatedRoute: ActivatedRoute, private digimonApi: Digimon ApiService) { }

  ngOnInit(): void {
    this.getLista()
      .subscribe((digimons: Array<Digimon>) => {
        this.digimons = digimons
      })
  }

  getLista(): Observable<Array<Digimon>> {
    const params: ParamMap = this.activatedRoute.snapshot.paramMap
    if (params.has('level')) {
      const level = params.get('level')!
      return this.digimonApi.getDigimonByLevel(level)
    }

    return this.digimonApi.getDigimons()
  }
}
```

Una vez obtenida la lista de digimons, vamos a pintar los nombres de estos dentro de un enlace que nos llevará a su información.

```
<ul>
  <li *ngFor="let digimon of digimons; let i = index">
    {{i}}: <a [routerLink]=["/digimon", digimon.name]">{{digimon.name}}</a>
  </li>
</ul>
```

Ya tenemos la aplicación, vamos a configurar el prerender para indicarle que rutas tenemos y de las cuales tiene que generar una página de HTML estática.

Vamos a ir al archivo **angular.json** y buscaremos por el final una propiedad **prerender.options**, en

la que tenemos un array **routes** donde tendremos que añadir las rutas de la aplicación a prerenderizar.

Dentro de este array, además de la ruta de inicio /, vamos a añadir las distintas rutas de los niveles de los digimon **/digimon/level/:level**, cambiando el parámetro por sus posibles valores.

/angular-universal-prerendering-lab/angular.json

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "angular-universal-prerendering-lab": {
      "projectType": "application",
      "schematics": {
        "@schematics/angular:application": {
          "strict": true
        }
      },
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/angular-universal-prerendering-lab/browser",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
            "assets": [
              "src/favicon.ico",
              "src/assets"
            ],
            "styles": [
              "src/styles.css"
            ],
            "scripts": []
          },
          "configurations": {
            "production": {
              "budgets": [
                {
                  "type": "initial",
                  "maximumWarning": "500kb",
                  "maximumError": "1mb"
                },
                {
                  "type": "anyComponentStyle",
                  "maximumWarning": "1mb",
                  "maximumError": "5mb"
                }
              ]
            }
          }
        }
      }
    }
  }
}
```

```

        "maximumWarning": "2kb",
        "maximumError": "4kb"
    }
],
"fileReplacements": [
{
    "replace": "src/environments/environment.ts",
    "with": "src/environments/environment.prod.ts"
}
],
"outputHashing": "all"
},
"development": {
    "buildOptimizer": false,
    "optimization": false,
    "vendorChunk": true,
    "extractLicenses": false,
    "sourceMap": true,
    "namedChunks": true
}
},
"defaultConfiguration": "production"
},
"serve": {
    "builder": "@angular-devkit/build-angular:dev-server",
    "configurations": {
        "production": {
            "browserTarget": "angular-universal-prerendering-lab:build:production"
        },
        "development": {
            "browserTarget": "angular-universal-prerendering-lab:build:development"
        }
    },
    "defaultConfiguration": "development"
},
"extract-i18n": {
    "builder": "@angular-devkit/build-angular:extract-i18n",
    "options": {
        "browserTarget": "angular-universal-prerendering-lab:build"
    }
},
"test": {
    "builder": "@angular-devkit/build-angular:karma",
    "options": {
        "main": "src/test.ts",
        "polyfills": "src/polyfills.ts",
        "tsConfig": "tsconfig.spec.json",
        "karmaConfig": "karma.conf.js",
        "assets": [
            "src/favicon.ico",
            "src/assets"
        ]
    }
}
]
}

```

```

        ],
        "styles": [
          "src/styles.css"
        ],
        "scripts": []
      }
    },
    "server": {
      "builder": "@angular-devkit/build-angular:server",
      "options": {
        "outputPath": "dist/angular-universal-prerendering-lab/server",
        "main": "server.ts",
        "tsConfig": "tsconfig.server.json"
      },
      "configurations": {
        "production": {
          "outputHashing": "media",
          "fileReplacements": [
            {
              "replace": "src/environments/environment.ts",
              "with": "src/environments/environment.prod.ts"
            }
          ]
        },
        "development": {
          "optimization": false,
          "sourceMap": true,
          "extractLicenses": false
        }
      },
      "defaultConfiguration": "production"
    },
    "serve-ssr": {
      "builder": "@nguniversal/builders:ssr-dev-server",
      "configurations": {
        "development": {
          "browserTarget": "angular-universal-prerendering-lab:build:development",
          "serverTarget": "angular-universal-prerendering-lab:server:development"
        },
        "production": {
          "browserTarget": "angular-universal-prerendering-lab:build:production",
          "serverTarget": "angular-universal-prerendering-lab:server:production"
        }
      },
      "defaultConfiguration": "development"
    },
    "prerender": {
      "builder": "@nguniversal/builders:prerender",
      "options": {
        "routes": [
          "/"
        ],
        "dest": "dist/prerendered"
      }
    }
  }
}

```

```

        "/digimon/level/in-training",
        "/digimon/level/fresh",
        "/digimon/level/training",
        "/digimon/level/rookie",
        "/digimon/level/champion",
        "/digimon/level/ultimate",
        "/digimon/level/mega"
    ],
},
"configurations": {
    "production": {
        "browserTarget": "angular-universal-prerendering-lab:build:production",
        "serverTarget": "angular-universal-prerendering-lab:server:production"
    },
    "development": {
        "browserTarget": "angular-universal-prerendering-lab:build:development",
        "serverTarget": "angular-universal-prerendering-lab:server:development"
    }
},
"defaultConfiguration": "production"
}
}
}
},
"defaultProject": "angular-universal-prerendering-lab"
}

```

También queremos generar las distintas páginas de HTML de cada uno de los digimons, pero son tantos que escribirlas todas a mano nos llevaría bastante tiempo, por lo que vamos a generar un script con Node que lo hará por nosotros.

El prerender nos permite pasarle también una propiedad **routesFile** cuyo valor será un archivo en el que se encontrarán todas las rutas a prerenderizar.

Vamos a añadirle esta propiedad debajo de **routes** y le asignaremos como valor **digimon-routes.txt**, que generaremos después con el script de Node.

/angular-universal-prerendering-lab/angular.json

```
{
"$schema": "./node_modules/@angular/cli/lib/config/schema.json",
"version": 1,
"newProjectRoot": "projects",
"projects": {
    "angular-universal-prerendering-lab": {
        "projectType": "application",
        "schematics": {
            "@schematics/angular:application": {
                "strict": true
            }
        },
        ...
    }
},
"architect": {
    "build": {
        "builder": "ng-build:prod"
    }
}
}
```

```
"root": "",
"sourceRoot": "src",
"prefix": "app",
"architect": {
  "build": {
    "builder": "@angular-devkit/build-angular:browser",
    "options": {
      "outputPath": "dist/angular-universal-prerendering-lab/browser",
      "index": "src/index.html",
      "main": "src/main.ts",
      "polyfills": "src/polyfills.ts",
      "tsConfig": "tsconfig.app.json",
      "assets": [
        "src/favicon.ico",
        "src/assets"
      ],
      "styles": [
        "src/styles.css"
      ],
      "scripts": []
    },
    "configurations": {
      "production": {
        "budgets": [
          {
            "type": "initial",
            "maximumWarning": "500kb",
            "maximumError": "1mb"
          },
          {
            "type": "anyComponentStyle",
            "maximumWarning": "2kb",
            "maximumError": "4kb"
          }
        ],
        "fileReplacements": [
          {
            "replace": "src/environments/environment.ts",
            "with": "src/environments/environment.prod.ts"
          }
        ],
        "outputHashing": "all"
      },
      "development": {
        "buildOptimizer": false,
        "optimization": false,
        "vendorChunk": true,
        "extractLicenses": false,
        "sourceMap": true,
        "namedChunks": true
      }
    }
  }
}
```

```
  },
  "defaultConfiguration": "production"
},
"serve": {
  "builder": "@angular-devkit/build-angular:dev-server",
  "configurations": {
    "production": {
      "browserTarget": "angular-universal-prerendering-lab:build:production"
    },
    "development": {
      "browserTarget": "angular-universal-prerendering-lab:build:development"
    }
  },
  "defaultConfiguration": "development"
},
"extract-i18n": {
  "builder": "@angular-devkit/build-angular:extract-i18n",
  "options": {
    "browserTarget": "angular-universal-prerendering-lab:build"
  }
},
"test": {
  "builder": "@angular-devkit/build-angular:karma",
  "options": {
    "main": "src/test.ts",
    "polyfills": "src/polyfills.ts",
    "tsConfig": "tsconfig.spec.json",
    "karmaConfig": "karma.conf.js",
    "assets": [
      "src/favicon.ico",
      "src/assets"
    ],
    "styles": [
      "src/styles.css"
    ],
    "scripts": []
  }
},
"server": {
  "builder": "@angular-devkit/build-angular:server",
  "options": {
    "outputPath": "dist/angular-universal-prerendering-lab/server",
    "main": "server.ts",
    "tsConfig": "tsconfig.server.json"
  },
  "configurations": {
    "production": {
      "outputHashing": "media",
      "fileReplacements": [
        {
          "replace": "src/environments/environment.ts",

```

```
        "with": "src/environments/environment.prod.ts"
    }
]
},
"development": {
    "optimization": false,
    "sourceMap": true,
    "extractLicenses": false
}
},
"defaultConfiguration": "production"
},
"serve-ssr": {
    "builder": "@nguniversal/builders:ssr-dev-server",
    "configurations": {
        "development": {
            "browserTarget": "angular-universal-prerendering-lab:build:development",
            "serverTarget": "angular-universal-prerendering-lab:server:development"
        },
        "production": {
            "browserTarget": "angular-universal-prerendering-lab:build:production",
            "serverTarget": "angular-universal-prerendering-lab:server:production"
        }
    },
    "defaultConfiguration": "development"
},
"prerender": {
    "builder": "@nguniversal/builders:prerender",
    "options": {
        "routes": [
            "/",
            "/digimon/level/in training",
            "/digimon/level/fresh",
            "/digimon/level/training",
            "/digimon/level/rookie",
            "/digimon/level/champion",
            "/digimon/level/ultimate",
            "/digimon/level/mega"
        ],
        "routesFile": "digimon-routes.txt"
    },
    "configurations": {
        "production": {
            "browserTarget": "angular-universal-prerendering-lab:build:production",
            "serverTarget": "angular-universal-prerendering-lab:server:production"
        },
        "development": {
            "browserTarget": "angular-universal-prerendering-lab:build:development",
            "serverTarget": "angular-universal-prerendering-lab:server:development"
        }
    }
},
```

```

        "defaultConfiguration": "production"
    }
}
},
{
  "defaultProject": "angular-universal-prerendering-lab"
}

```

Pues vamos a crear en la raíz del proyecto un nuevo archivo que llamaremos **generate-routes-file.js**.

Necesitaremos instalar la dependencia de axios para obtener todos los digimons y de estos sacar los nombres para generar las rutas. Por lo que vamos a lanzar el siguiente comando dentro de la carpeta del proyecto:

```
$ npm install axios
```

Ahora dentro del archivo **generate-routes-file.js**, vamos a empezar por importar las dependencias que usaremos:

- axios: para hacer una petición GET a la API
- fs: para crear el archivo digimon-routes.txt con las rutas
- path: para crear la ruta donde generar el archivo TXT

/angular-universal-prerendering-lab/generate-routes-file.js

```

const path = require('path')
const fs = require('fs').promises
const axios = require('axios')

```

Vamos a empezar haciendo una petición GET con axios a la api para traer todos los digimons.

Una vez los tengamos, vamos a generar a partir de la respuesta un string con las rutas **/digimon/<nombre-digimon>** separadas por saltos de línea.

/angular-universal-prerendering-lab/generate-routes-file.js

```

const path = require('path')
const fs = require('fs').promises
const axios = require('axios')

axios.get('https://digimon-api.vercel.app/api/digimon')
  .then((resp) => {
    const rutas = resp.data.map(d => `/digimon/${d.name}`).join('\n')
  })

```

Ahora tenemos que escribir estas rutas en el archivo **digimon-routes.txt**, y para ello usaremos el módulo de fs (con promesas) y el de path.

/angular-universal-prerendering-lab/generate-routes-file.js

```
const path = require('path')
const fs = require('fs').promises
const axios = require('axios')

axios.get('https://digimon-api.vercel.app/api/digimon')
  .then((resp) => {
    const rutas = resp.data.map(d => `/digimon/${d.name}`).join('\n')
    return fs.writeFile(path.join(__dirname, 'digimon-routes.txt'), rutas)
  })
  .then(() => {
    console.log('Archivo de rutas generado correctamente :)')
  })
  .catch(err => {
    console.log('Error al generar el archivo de rutas', err)
  })
}
```

Ya tenemos el script, ahora solo falta comprobar que todo esto funciona y nos genera los archivos estáticos.

Vamos a lanzar desde la carpeta raíz del proyecto los siguientes comandos:

```
$ node generate-routes-file.js
$ npm run prerender
```

Cuando termina la ejecución del segundo comando, deberíamos de tener todos las páginas HTML dentro de **dist/angular-universal-prerender-lab/browser**, entonces podemos servirlas lanzando el siguiente comando:

```
$ npx http-serve dist/angular-universal-prerendering-lab/browser
```

Al entrar en <http://localhost:8080> ya podemos ver la página web.

Para mejorar el proceso de prerenderizar las páginas podemos crear un script que lance primero el script para generar el archivo de rutas, y después haga el prerenderizado.

Dentro del **package.json**, vamos a añadir un nuevo script **genfile:prerender**.

/angular-universal-prerendering-lab/package.json

```
{
  "name": "angular-universal-prerendering-lab",
  "version": "0.0.0",
  "scripts": {
```

```

    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "watch": "ng build --watch --configuration development",
    "test": "ng test",
    "dev:ssr": "ng run angular-universal-prerendering-lab:serve-ssr",
    "serve:ssr": "node dist/angular-universal-prerendering-lab/server/main.js",
    "build:ssr": "ng build && ng run angular-universal-prerendering-lab:server",
    "prerender": "ng run angular-universal-prerendering-lab:prerender",
    "genfile:prerender": "node generate-routes-file.js && npm run prerender"
},
"private": true,
"dependencies": {
    "@angular/animations": "~13.2.0",
    "@angular/common": "~13.2.0",
    "@angular/compiler": "~13.2.0",
    "@angular/core": "~13.2.0",
    "@angular/forms": "~13.2.0",
    "@angular/platform-browser": "~13.2.0",
    "@angular/platform-browser-dynamic": "~13.2.0",
    "@angular/platform-server": "~13.2.0",
    "@angular/router": "~13.2.0",
    "@nguniversal/express-engine": "^13.0.2",
    "axios": "^0.25.0",
    "express": "^4.15.2",
    "rxjs": "~7.5.0",
    "tslib": "^2.3.0",
    "zone.js": "~0.11.4"
},
"devDependencies": {
    "@angular-devkit/build-angular": "~13.2.1",
    "@angular/cli": "~13.2.1",
    "@angular/compiler-cli": "~13.2.0",
    "@nguniversal/builders": "^13.0.2",
    "@types/express": "^4.17.0",
    "@types/jasmine": "~3.10.0",
    "@types/node": "^12.11.1",
    "jasmine-core": "~4.0.0",
    "karma": "~6.3.0",
    "karma-chrome-launcher": "~3.1.0",
    "karma-coverage": "~2.1.0",
    "karma-jasmine": "~4.0.0",
    "karma-jasmine-html-reporter": "~1.7.0",
    "typescript": "~4.5.2"
}
}

```

Ahora en lugar de lanzar dos comandos para realizar el prerenderizado, podemos lanzar el siguiente:

```
$ npm run genfile:prerender
```

Chapter 27. Angular Elements

En Angular 7 añadieron el paquete de `@angular/elements` al proyecto.

Hasta este momento los componentes de Angular solo se podían utilizar dentro de proyectos de Angular. Pero con **Angular Elements** vamos a poder crear **web components nativos** a partir de los componentes de Angular.

Esto nos permite crear componentes de Angular y poder transformarlos después en componentes web que se pueden utilizar en cualquier proyecto, incluso los que no son de Angular. Por ejemplo, podríamos utilizarlos en:

- Aplicaciones hechas con HTML, CSS y JS sin ninguna librería o framework.
- Aplicaciones hechas con Angular.
- Aplicaciones hechas con React, Vue o cualquier otra librería del frontend.
- ...

En fin, en cualquier aplicación web sin importar como está montada.

Esto se debe a que los componentes web nativos son una de las novedades que venían con HTML 5 y que nos permite crear nuestras propias etiquetas, evitando así tener que utilizar muchos divs que hacen el código ilegible o lidiar con problemas de conflictos de CSS entre otras cosas.

27.1. Lab: Angular Elements

En este laboratorio vamos a ver como crear un componente web nativo a partir de los componentes de Angular utilizando la dependencia de @angular/elements.

Vamos a crear un proyecto nuevo de Angular lanzando el siguiente comando:

```
$ ng new angular-elements-lab
```

Ahora se nos hace unas preguntas a las que contestaremos con la respuesta que viene por defecto pulsando **Intro**.

```
? 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 se instalen las dependencias vamos a entrar al proyecto y añadir dos dependencias que necesitaremos:

- @angular/elements: se encarga de crear un web componente a partir de un componente de Angular.
- ngx-build-plus: esta nos permite construir el proyecto del web component de una forma muy sencilla.

```
$ cd angular-elements-lab  
$ ng add @angular/elements  
$ ng add ngx-build-plus
```

Al añadir las librerías ya podemos levantar el servidor de desarrollo con el siguiente comando.

```
ng serve
```

Ya podemos empezar a crear nuestro componente.

Para este laboratorio vamos a crear un componente **Acordeon** que expande y colapsa su contenido al pulsar sobre la cabecera de este.

Lanzamos el siguiente comando para crear el componente desde otra terminal distinta:

```
$ ng g c acordeon
```

Entramos al archivo de **acordeon.component.html** para añadir su estructura:

/angular-elements-lab/src/app/acordeon/acordeon.component.html

```
<div>
  <div>
    <h3>Título</h3>
  </div>
  <div>
    <p>Contenido</p>
  </div>
</div>
```

Este componente necesitará recibir tanto el título como el contenido que tiene que mostrar. El título lo va a recibir como una propiedad externa del componente, mientras que el contenido se va a proyectar desde el exterior ya que no es algo que vayamos a conocer de antemano, sino que una vez pueden querer mostrar una lista de datos y otra pueden querer mostrar un párrafo con información.

Por lo tanto, para el título vamos a utilizar el **string interpolation**, y para el contenido la etiqueta **ng-content**.

/angular-elements-lab/src/app/acordeon/acordeon.component.html

```
<div>
  <div>
    <h3>{{titulo}}</h3>
  </div>
  <div>
    <ng-content></ng-content>
  </div>
</div>
```

Nos vamos al archivo de typescript para añadir las propiedades que necesita para funcionar correctamente.

Tendremos que añadir el título que recibe desde el exterior (con el decorador **Input**), y también una variable que controle si está expandido o cerrado.

/angular-elements-lab/src/app/acordeon/acordeon.component.ts

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

@Component({
  selector: 'app-acordeon',
  templateUrl: './acordeon.component.html',
  styleUrls: ['./acordeon.component.css']
})
export class AcordeonComponent implements OnInit {
  @Input() titulo = '';
  estaCerrado = true;

  constructor() { }

  ngOnInit(): void {
  }
}
```

También vamos a crear un método para cambiar el valor de **estaCerrado**.

/angular-elements-lab/src/app/acordeon/acordeon.component.ts

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

@Component({
  selector: 'app-acordeon',
  templateUrl: './acordeon.component.html',
  styleUrls: ['./acordeon.component.css']
})
export class AcordeonComponent implements OnInit {
  @Input() titulo = '';
  estaCerrado = true;

  constructor() { }

  ngOnInit(): void {
  }

  toggleEstaCerrado() {
    this.estacerrado = !this.estacerrado;
  }
}
```

Ahora añadiremos sobre el div de la cabecera un evento **click** que llamará a la función que acabamos de crear.

/angular-elements-lab/src/app/acordeon/acordeon.component.html

```
<div>
  <div (click)="toggleEstaCerrado()">
    <h3>{{titulo}}</h3>
  </div>
  <div>
    <ng-content></ng-content>
  </div>
</div>
```

Por último vamos a añadirle estilos a nuestro componente para que no quede tan cutre. Nos vamos al archivo de CSS y añadimos las siguientes reglas:

/angular-elements-lab/src/app/acordeon/acordeon.component.css

```
.acordeon {
  border: 1px solid black;
  border-radius: 5px;
}

.acordeon-heading {
  text-align: center;
  border-bottom: 2px solid black;
  cursor: pointer;
}

.acordeon-content {
  overflow: hidden;
}

.acordeon-content.cerrado {
  height: 0;
}

.acordeon-content.abierto {
  height: 100%;
  padding: 20px;
}
```

Ahora vamos a añadir las clases sobre nuestro componente.

Todas ellas las añadiremos como una clase normal a excepción de las que vamos a poner en el div del contenido ya que estás cambian dependiendo del valor de **estaCerrado**. Para ellas usaremos la directiva **ngClass**.

/angular-elements-lab/src/app/acordeon/acordeon.component.html

```
<div class="acordeon">
  <div class="acordeon-heading" (click)="toggleEstaCerrado()">
    <h3>{{titulo}}</h3>
  </div>
  <div [ngClass]="{{'acordeon-content': true, abierto: !estaCerrado, cerrado: estaCerrado}}">
    <ng-content></ng-content>
  </div>
</div>
```

Ya tendríamos nuestro componente de Angular creado. Vamos a añadirlo en el componente principal de la aplicación para ver que funciona como se espera.

/angular-elements-lab/src/app/app.component.html

```
<app-acordeon titulo="Pollo al limón: ingredientes">
  <ul>
    <li>200g pechuga de pollo</li>
    <li>1 zanahoria</li>
    <li>1 cebolla</li>
    <li>1 pimiento morrón</li>
    <li>1 limón</li>
    <li>30g queso parmesano rallado</li>
    <li>1 cdta. mezcla de hierbas italianas</li>
  </ul>
</app-acordeon>
```

Si vamos a <http://localhost:4200/> podremos ver nuestra lista de ingredientes y podremos hacer que se colapse o se expanda pulsando sobre la cabecera.

Una vez que tenemos el componente toca configurarlo para poder convertirlo en un web componente nativo que podamos utilizar en cualquier aplicación sin necesidad de que esta sea una aplicación de Angular.

Lo primero que vamos a hacer es encapsular el contenido de nuestro componente en un **shadow DOM** para que los estilos que tengamos en el componente no conflictan con aquellos que tengamos en los proyectos donde lo vamos a utilizar.

Para esto, en el archivo de typescript vamos a añadir el tipo de encapsulación que vamos a utilizar dentro del decorador de la clase.

/angular-elements-lab/src/app/acordeon/acordeon.component.ts

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

@Component({
  selector: 'app-acordeon',
  templateUrl: './acordeon.component.html',
  styleUrls: ['./acordeon.component.css'],
  encapsulation: ViewEncapsulation.ShadowDom
})
export class AcordeonComponent implements OnInit {
  @Input() titulo = '';
  estaCerrado = true;

  constructor() { }

  ngOnInit(): void {
  }

  toggleEstaCerrado() {
    this.estaCerrado = !this.estaCerrado;
  }
}
```

Los siguientes cambios los vamos a realizar sobre el módulo de la aplicación.

Para empezar vamos a quitar las referencias al componente App porque este no lo vamos a necesitar para nada. También lo quitamos del array de **bootstrap** ya que no queremos que se cargue ningún componente de Angular.

/angular-elements-lab/src/app/app.module.ts

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

import { AcordeonComponent } from './acordeon/acordeon.component';

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

Ahora vamos a inyectar en el constructor del módulo el **Injector** de Angular que se encarga de

gestionar la inyección de dependencias en Angular.

/angular-elements-lab/src/app/app.module.ts

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

import { AcordeonComponent } from './acordeon/acordeon.component';

@NgModule({
  declarations: [
    AcordeonComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: []
})
export class AppModule {
  constructor(private injector: Injector) { }
}
```

Ahora vamos a añadir el método **ngDoBootstrap** que se ejecuta al cargar este módulo. En este método es donde vamos a indicar que conviertan nuestro componente en un web component nativo.

Aquí usaremos la función de **createCustomElement** de **@angular/elements** a la que le vamos a pasar nuestro componente y como segundo parámetro un objeto con el injector de Angular. Esta función nos devuelve el componente web.

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

import { AcordeonComponent } from './acordeon/acordeon.component';
import { createCustomElement } from '@angular/elements';

@NgModule({
  declarations: [
    AcordeonComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: []
})
export class AppModule {
  constructor(private injector: Injector) { }

  ngDoBootstrap() {
    const AcordeonElement = createCustomElement(AcordeonComponent, { injector: this.injector })
  }
}
```

Por último, vamos a registrar el web component en la página utilizando la API de customElements de los navegadores.

Vamos a llamar a la función **define** y le pasaremos como parámetros el nombre de la etiqueta para mostrar nuestro componente y también el componente web que nos ha generado angular elements.

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

import { AcordeonComponent } from './acordeon/acordeon.component';
import { createCustomElement } from '@angular/elements';

@NgModule({
  declarations: [
    AcordeonComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: []
})
export class AppModule {
  constructor(private injector: Injector) { }

  ngDoBootstrap() {
    const AcordeonElement = createCustomElement(AcordeonComponent, { injector: this.injector })
    customElements.define('mi-acordeon', AcordeonElement)
  }
}
```

Con esto ya tendríamos nuestro web component. Ahora tenemos que construir el proyecto para poder empezar a utilizar este componente en otras páginas HTML.

Para no tener que aprendernos el comando, vamos a añadir un nuevo script de NPM (**build:wc**) en el **package.json** del proyecto.

```
{  
  "name": "angular-elements-lab",  
  "version": "0.0.0",  
  "scripts": {  
    "ng": "ng",  
    "start": "ng serve",  
    "build": "ng build",  
    "test": "ng test",  
    "lint": "ng lint",  
    "e2e": "ng e2e",  
    "build:wc": "ng build --prod --single-bundle true --output-hashing none"  
  },  
  "private": true,  
  "dependencies": {  
    "@angular/animations": "~11.1.2",  
    "@angular/common": "~11.1.2",  
    "@angular/compiler": "~11.1.2",  
    "@angular/core": "~11.1.2",  
    "@angular/forms": "~11.1.2",  
    "@angular/platform-browser": "~11.1.2",  
    "@angular/platform-browser-dynamic": "~11.1.2",  
    "@angular/router": "~11.1.2",  
    "rxjs": "~6.6.0",  
    "tslib": "^2.0.0",  
    "zone.js": "~0.11.3"  
  },  
  "devDependencies": {  
    "@angular-devkit/build-angular": "~0.1101.4",  
    "@angular/cli": "~11.1.4",  
    "@angular/compiler-cli": "~11.1.2",  
    "@types/jasmine": "~3.6.0",  
    "@types/node": "^12.11.1",  
    "codelyzer": "^6.0.0",  
    "jasmine-core": "~3.6.0",  
    "jasmine-spec-reporter": "~5.0.0",  
    "karma": "~5.2.0",  
    "karma-chrome-launcher": "~3.1.0",  
    "karma-coverage": "~2.0.3",  
    "karma-jasmine": "~4.0.0",  
    "karma-jasmine-html-reporter": "^1.5.0",  
    "protractor": "~7.0.0",  
    "ts-node": "~8.3.0",  
    "tslint": "~6.1.0",  
    "typescript": "~4.1.2"  
  }  
}
```

Con este script le indicamos que construya el proyecto para producción, que no añada los hashes a

los archivos generados y que no genere múltiples archivos de bundles, sino que lo genere todo en un archivo.

Esta última parte del script es la que añade la dependencia de **ngx-build-plus** que añadimos al proyecto al principio. En caso de no utilizar esta herramienta, tendríamos que importar múltiples archivos en las páginas de HTML, o crearnos un script que concatene el contenido de estos en uno solo.

Solo tenemos que lanzar el siguiente comando para generar nuestro componente web:

```
$ npm run build:wc
```

Al terminar de construir el proyecto nos encontramos con una carpeta **dist/angular-elements-lab** en la que tendremos el código final. Vamos a entrar hasta esta carpeta desde el terminal y levantaremos un servidor para que nos sirva el **index.html**.

```
$ cd dist/angular-elements-lab
$ npx http-serve
```

Ahora podemos entrar a la aplicación desde <http://localhost:8080>.

Pero no se está mostrando nada, y es porque en nuestro **index.html** final no hemos puesto el componente todavía.

Vamos a sustituir la etiqueta **app-root** que no pinta nada, por la etiqueta **mi-acordeon** con el código que habíamos puesto en el componente **App**.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularElementsLab</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
<link rel="stylesheet" href="styles.css"></head>
<body>
  <mi-acordeon titulo="Pollo al limón: ingredientes">
    <ul>
      <li>200g pechuga de pollo</li>
      <li>1 zanahoria</li>
      <li>1 cebolla</li>
      <li>1 pimiento morrón</li>
      <li>1 limón</li>
      <li>30g queso parmesano rallado</li>
      <li>1 cdta. mezcla de hierbas italianas</li>
    </ul>
  </mi-acordeon>
  <script src="polyfills.js" defer></script>
  <script src="main.js" defer></script></body>
</html>
```

Ahora deberíamos ver nuestro acordeón con la lista de ingredientes al igual que antes cuando lo hemos utilizado en el componente App. La diferencia es que ahora podemos coger el código generado para este componente y utilizarlo en cualquier aplicación web sin importar si se usa Angular en el proyecto o cualquier otra librería.

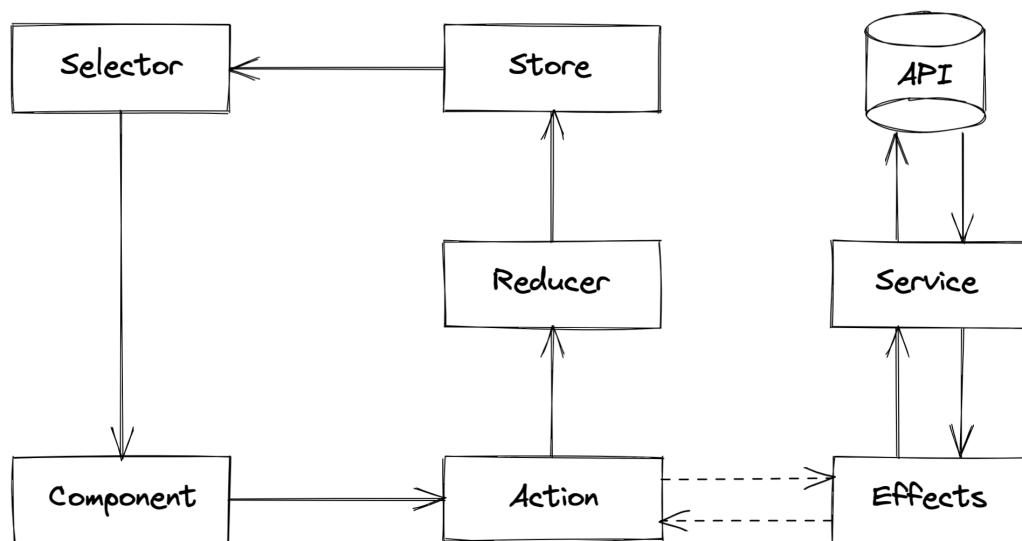
Chapter 28. Ngrx

Ngrx es la implementación del patrón Flux para las aplicaciones de Angular.

Con Ngrx podemos **gestionar el estado** de las aplicaciones de Angular, dejando así los componentes mucho más limpios ya que vamos a poder quitar toda la lógica que haya en ellos. De lo único que se van a encargar es pintar cosas e indicar que cambios hay que realizar en el estado.

Dentro de Ngrx nos encontramos con varios elementos que tendremos que utilizar:

- Actions
- Reducers
- Selectors
- Store
- Effects



Ngrx sigue 3 principios:

- Los cambios en el estado se definen a través de las **actions** que son eventos que se despachan desde los componentes y servicios.
- Los **reducers** son los encargados de realizar los cambios en el estado.
- Los **selectors** son las funciones puras con las que pedimos una parte del estado.

28.1. Actions

Las **actions** son eventos únicos que vamos a emitir desde los componentes, servicios... y que van a pasar a través de la aplicación para indicar que cambios hay que realizar sobre el estado.

Un action es un objeto que tiene una propiedad **type** con la que indicamos que cambio se va a realizar en el estado, y opcionalmente se le puede pasar un payload con datos extra necesarios para realizar dicho cambio.

Las acciones se crean con la función **createAction** a la que le pasamos como primer parámetro el tipo de la acción, y se le puede pasar un segundo parámetro que sería una llamada a **props** en la que se le indica como tipo genérico el tipo de datos que vamos a enviar en el payload.

Las acciones se despachan desde el servicio del **Store** con el método **dispatch** al que le pasamos como parámetro la acción que devuelve la función **createAction** cuando se ejecuta.

28.2. Reducers

Los **reducers** son funciones puras que reciben el **estado actual** y un **action** y los usan para crear el **nuevo estado** de la aplicación. Este nuevo estado se inyecta en los componentes para refrescar la interfaz de usuario con los nuevos datos.

Los reducers los creamos con la función **createReducer** que recibe como parámetros el estado inicial y una lista de handlers (función **on**) para gestionar el estado de distintas formas en función de la acción despachada.

La función **on** recibe como parámetros el tipo de la acción como primer parámetro y una función de callback con el estado y la acción despachada. Estos handlers tienen que retornar el nuevo estado.

28.3. Selectors

Los selectores son funciones puras que se usan para obtener una parte del estado del store de Ngrx.

Creamos los selectores con la función **createSelector**. Esta función recibe varias funciones de callback que reciben el estado global (o anterior si ya hay una función previa) y nos permite devolver la parte del estado que necesitamos obtener (normalmente en la vista).

Estos datos se piden desde la función **select** del servicio del Store, pasándole como parámetro la referencia a la función selectora que nos va a devolver lo que queremos.

28.4. Lab: Contador con Ngrx

En este laboratorio vamos a ver como usar las acciones, reducers, selectores y store de Ngrx para gestionar el estado de un contador.

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

```
$ ng new angular-ngrx-contador-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 dentro de ella vamos a empezar creando el pipe filtro y después levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-ngrx-contador-lab  
$ ng g c contador
```

Además de crear el componente, vamos a instalar la librería de **@ngrx/store** con el siguiente comando:

```
$ ng add @ngrx/store  
The package @ngrx/store@13.0.2 will be installed and executed.  
Would you like to proceed? Yes
```

Y ahora levantaremos el servidor de desarrollo:

```
$ ng s
```

Antes de ponernos manos a la obra a escribir código, vamos a crear la estructura de carpetas y archivos pertenecientes a Ngrx. Dentro de la carpeta **app**, vamos a crear el siguiente árbol de carpetas y archivos:

```
|- app  
| |- actions  
| | |- contador.actions.ts  
| | |- actions.types.ts  
| |- reducers  
| | |- contador.reducer.ts  
| |- selectors  
| | |- contador.selectors.ts  
| |- app.state.ts
```

Una vez creados todos esos archivos y carpetas, vamos a empezar por declarar unas constantes con el tipo de las acciones que vamos a enviar desde nuestro componente. Tendremos 3 acciones:

- Incrementar la cuenta
- Decrementar la cuenta
- Cambiar la cuenta

Por tanto en el archivo de **action.types.ts** vamos a declarar 3 constantes que vamos a exportar para luego usarlas al definir las acciones.

/angular-ngrx-contador-lab/src/app/state/actions/action.types.ts

```
export const INCREMENTAR = 'Incrementar'  
export const DECREMENTAR = 'Decrementar'  
export const CAMBIAR CUENTA = 'Cambiar cuenta'
```

Ahora que tenemos los tipos, vamos a crear las funciones que van a generar las acciones.

Dentro del archivo de **contador.actions.ts**, vamos a crear una acción **incrementar** usando la función **createAction** que viene de Ngrx, y le vamos a pasar el tipo de la acción, es decir la constante que hemos añadido en el archivo de antes.

/angular-ngrx-contador-lab/src/app/state/actions/contador.actions.ts

```
import { createAction, props } from "@ngrx/store";  
import * as ActionTypes from './action.types'  
  
export const incrementar = createAction(  
  ActionTypes.INCREMENTAR  
)
```

Vamos a hacer lo mismo para la acción de **decrementar**.

/angular-ngrx-contador-lab/src/app/state/actions/contador.actions.ts

```
import { createAction, props } from "@ngrx/store";  
import * as ActionTypes from './action.types'  
  
export const incrementar = createAction(  
  ActionTypes.INCREMENTAR  
)  
  
export const decrementar = createAction(  
  ActionTypes.DECREMENTAR  
)
```

Y por último, la acción de **cambiarCuenta** es un poco distinta, ya que a parte del tipo de la acción, vamos a necesitar pasarle una propiedad (un payload) **cuenta** para poder cambiar el valor del

estado con unos datos que se le van a dar. Para ello, le pasamos como segundo parámetro la llamada a la función **props** en la que vamos a indicar el tipo de dato del payload.

/angular-ngrx-contador-lab/src/app/state/actions/contador.actions.ts

```
import { createAction, props } from '@ngrx/store';
import * as ActionTypes from './action.types'

export const incrementar = createAction(
  ActionTypes.INCREMENTAR
)

export const decrementar = createAction(
  ActionTypes.DECREMENTAR
)

export const cambiarCuenta = createAction(
  ActionTypes.CAMBIAR CUENTA,
  props<{cuenta: number}>()
)
```

Con esto ya tenemos las acciones, ahora toca rellenar el reducer al que le van a llegar estas acciones y en el que cambiaremos el estado de la aplicación.

Dentro del archivo de **contador.reducer.ts** vamos a empezar declarando un estado inicial.

/angular-ngrx-contador-lab/src/app/state/reducers/contador.reducer.ts

```
export interface IContadorState {
  cuenta: number
}

const initialState: IContadorState = {
  cuenta: 0
}
```

Ahora vamos a crear el reducer con la función **createReducer** a la que le vamos a pasar como primer parámetro el estado inicial que hemos creado antes.

/angular-ngrx-contador-lab/src/app/state/reducers/contador.reducer.ts

```
import { createReducer } from "@ngrx/store"

export interface IContadorState {
  cuenta: number
}

const initialState: IContadorState = {
  cuenta: 0
}

export const contadorReducer = createReducer(
  initialState
)
```

También le vamos a pasar otros 3 parámetros más para indicarle como tiene que cambiar el estado en función de cada acción. Esto lo haremos con la función **on** en la que le pasamos como primer parámetro la acción, y como segundo parámetro una función de callback en la que recibimos el estado y la acción y en la que tenemos que retornar el nuevo estado.

/angular-ngrx-contador-lab/src/app/state/reducers/contador.reducer.ts

```
import { createReducer, on } from "@ngrx/store"
import { decrementar, incrementar, cambiarCuenta } from '../actions/contador.actions';

export interface IContadorState {
  cuenta: number
}

const initialState: IContadorState = {
  cuenta: 0
}

export const contadorReducer = createReducer(
  initialState,
  on(incrementar, (state) => {
    return { cuenta: state.cuenta + 1 }
}),
  on(decrementar, (state) => {
    return { cuenta: state.cuenta - 1 }
}),
  on(cambiarCuenta, (state, action) => {
    return { cuenta: action.cuenta }
}),
)
```

Ahora vamos a registrar nuestro reducer, y para ello, en el archivo **app.state.ts** vamos a exportar un objeto con todos los reducers de nuestra aplicación.

/angular-ngrx-contador-lab/src/app/state/app.state.ts

```
import { ActionReducerMap } from "@ngrx/store";
import { contadorReducer, IContadorState } from "./reducers/contador.reducer";

export interface IAppState {
  contadorState: IContadorState
}

export const AppReducers: ActionReducerMap<IAppState> = {
  contadorState: contadorReducer
}
```

Y este objeto **AppReducers** lo tenemos que pasar como primer parámetro a la función **forRoot** del módulo de Ngrx que se ha importado en el módulo de la aplicación.

/angular-ngrx-contador-lab/src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { StoreModule } from '@ngrx/store';
import { ContadorComponent } from './contador/contador.component';
import { AppReducers } from './state/app.state';

@NgModule({
  declarations: [
    AppComponent,
    ContadorComponent
  ],
  imports: [
    BrowserModule,
    StoreModule.forRoot(AppReducers, {})
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Ahora vamos a ir a mostrar el componente del contador en App.

/angular-ngrx-contador-lab/src/app/app.component.html

```
<app-contador></app-contador>
```

Dentro del contador, vamos a añadir dos botones (para incrementar y decrementar), la cuenta y un input para cambiar la cuenta.

/angular-ngrx-contador-lab/src/app/contador/contador.component.html

```
<p>Cuenta: {{cuenta}}</p>
<button type="button" (click)="decrementar()"-></button>
<button type="button" (click)="incrementar()" +></button>
<input type="number" (change)="cambiarCuenta($event)">
```

En el archivo de TypeScript vamos a añadir estos método y la cuenta.

/angular-ngrx-contador-lab/src/app/contador/contador.component.ts

```
import { Component, OnInit } from '@angular/core';
import { incrementar, cambiarCuenta } from '../state/actions/contador.actions';

@Component({
  selector: 'app-contador',
  templateUrl: './contador.component.html',
  styleUrls: ['./contador.component.css']
})
export class ContadorComponent implements OnInit {
  cuenta: number = 0

  constructor() { }

  ngOnInit(): void {
  }

  incrementar() {

  }

  decrementar() {

  }

  cambiarCuenta(event: any) {
  }
}
```

Desde estas 3 funciones vamos a despachar las acciones que ya habíamos creado anteriormente, para poder hacer esto necesitamos inyectar el **store** en el componente.

```
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { incrementar, cambiarCuenta } from '../state/actions/contador.actions';
import { IAppState } from '../state/app.state';

@Component({
  selector: 'app-contador',
  templateUrl: './contador.component.html',
  styleUrls: ['./contador.component.css']
})
export class ContadorComponent implements OnInit {
  cuenta: number = 0

  constructor(private store: Store<IAppState>) { }

  ngOnInit(): void {
  }

  incrementar() {

  }

  decrementar() {

  }

  cambiarCuenta(event: any) {
    }
}
```

Ahora podemos usar el método **dispatch** al cual le vamos a pasar la llamada a las acciones, y en el caso de la función de **cambiarCuenta** también le tendremos que pasar la nueva cuenta como parámetro de la función creadora de la acción.

```
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { incrementar, cambiarCuenta, decrementar } from '../state/actions/contador.actions';
import { IAppState } from '../state/app.state';

@Component({
  selector: 'app-contador',
  templateUrl: './contador.component.html',
  styleUrls: ['./contador.component.css']
})
export class ContadorComponent implements OnInit {
  cuenta: number = 0

  constructor(private store: Store<IAppState>) { }

  ngOnInit(): void {
  }

  incrementar() {
    this.store.dispatch(incrementar())
  }

  decrementar() {
    this.store.dispatch(decrementar())
  }

  cambiarCuenta(event: any) {
    this.store.dispatch(cambiarCuenta({ cuenta: Number(event.target.value) }))
  }
}
```

Ahora nos falta obtener la cuenta del Store para mostrarla, por lo que primero vamos a crear el selector que nos la va a retornar.

Dentro del archivo de **contador.selectors.ts** vamos a empezar creando una función que va a recibir el estado global y del cual vamos a sacar el estado perteneciente al contador.

```
import { IAppState } from "../app.state";

const selectContador = (state: IAppState) => state.contadorState
```

Ahora vamos a crear el selector con la función **createSelector** al que le vamos a pasar el **selectContador** que acabamos de añadir como primera parámetro y después una función similar en la que recibiremos el estado del **contadorState** y sacaremos los datos que vamos a usar en el componente. En este caso, la cuenta.

```
import { createSelector } from "@ngrx/store";
import { IAppState } from "../app.state";
import { IContadorState } from "../reducers/contador.reducer";

const selectContador = (state: IAppState) => state.contadorState

export const selectCuenta = createSelector(
  selectContador,
  (state: IContadorState) => state.cuenta
)
```

Ahora podemos volver al componente del Contador en el que vamos a pedir con la función **store.select** el selector que nos devuelve la cuenta.

Esto nos devuelve un observable por lo que nos vamos a suscribir para obtener la cuenta y asignarla a la propiedad que tenemos en el componente.

```
import { Component, OnInit } from '@angular/core';
import { Store } from '@ngrx/store';
import { incrementar, cambiarCuenta, decrementar } from '../state/actions/contador.actions';
import { IAppState } from '../state/app.state';
import { selectCuenta } from '../state/selectors/contador.selectors';

@Component({
  selector: 'app-contador',
  templateUrl: './contador.component.html',
  styleUrls: ['./contador.component.css']
})
export class ContadorComponent implements OnInit {
  cuenta: number = 0

  constructor(private store: Store<IAppState>) { }

  ngOnInit(): void {
    this.store.select(selectCuenta)
      .subscribe((cuenta: number) => {
        this.cuenta = cuenta
      })
  }

  incrementar() {
    this.store.dispatch(incrementar())
  }

  decrementar() {
    this.store.dispatch(decrementar())
  }

  cambiarCuenta(event: any) {
    this.store.dispatch(cambiarCuenta({ cuenta: Number(event.target.value) }))
  }
}
```

Ahora ya deberíamos de poder cambiar la cuenta pulsando sobre los botones y escribiendo un número sobre el campo numérico.

28.5. Effects

Los **effects** son una forma de poder separar la comunicación con los servicios de los componentes, dejando que sean los efectos los que interactúen con estos servicios cuando ciertas acciones se van despachando en la aplicación.

Los efectos se suelen utilizar para gestionar tareas como pedir datos, interactuar con otros elementos externos a los componentes...

Chapter 29. Animaciones

Las animaciones nos van a permitir realizar la transición de unos estilos CSS a otros cuando ocurre cierta acción. De esta forma vamos a poder hacer que los cambios de estilos ocurran de una forma más suave.

Las animaciones en Angular funcionan mediante **cambios de estado** que ocurren en los componentes. Cuando uno de los estados definidos cambia, es cuando la animación se dispara y entonces se aplican los estilos definidos para esa animación en el nuevo estado.

Para utilizar las animaciones de Angular, hay que importar el módulo **BrowserAnimationsModule** que se importa desde `@angular/platform-browser/animations` en el módulo de la aplicación.

Vamos a ver las distintas funciones que vamos a usar para crear las animaciones en nuestros componentes.

Estas funciones se van a usar dentro de la propiedad **animations** del decorador del componente para definir las distintas animaciones, y son las siguientes:

- **trigger**
- **state**
- **style**
- **animate**
- **transition**
- **keyframes**

29.1. Trigger

La función **trigger** es aquella en la que se va a definir la animación (estados y transiciones) y que se va a ejecutar cada vez que haya un cambio en alguno de los estados.

El primer parámetro que recibe esta función es el nombre que le vamos a dar a la animación que se está definiendo, y como segundo parámetro recibe un array con los estados y las transiciones que vamos a utilizar.

29.2. State

El **state** representa un estado de la animación y con esta función vamos a definir los estilos que se tienen que aplicar al elemento cuando este estado se encuentra activo.

Como primer parámetro recibe el nombre del estado, y como segundo parámetro la función de **style** donde se definen los estilos a aplicar.

Para mantener el estado de la animación, tenemos que guardarla en una propiedad del componente que luego hay que asignar a la directiva (@nombreAnimación) en el componente o etiqueta que queremos animar.

El cambio de estado se realizará cambiando el valor de la propiedad que mantiene el estado, y esta solo tiene que guardar un string con el nombre del estado.

29.3. Style

La función **style** recibe como parámetro un objeto de JavaScript con las propiedades CSS que queremos aplicar en la animación.

29.4. Transition

La función **transition** es la que define cuento tiempo tiene que durar la animación cuando pasa de un estado a otro, y es donde se indica esta secuencia de estados.

Secuencia	Descripción
std1 ⇒ std2	Se ejecutará la función animate cuando se pase del estado std1 al std2.
std1 ⇌ std2	Se ejecutará la función animate cuando se pase del estado std1 al std2 o viceversa.
std1 ⇒ std2, std2 ⇒ std3	Se ejecutará la función animate cuando se pase del estado std1 al std2 , o del std2 al std3 .
* ⇒ *	Se ejecutará la función animate cuando se pase de un estado cualquiera a otro cualquiera.
void ⇒ *	Se ejecutará la función animate cuando el elemento que tiene la animación se haya cargado y pase a cualquier estado. Este se suele usar para las animaciones que se ejecutan cuando se carga una página.

29.5. Animate

La función **animate** nos permite definir cuento tiempo tiene que durar una transición entre los estilos de dos estados o la duración de la animación definida con la función de **keyframes**.

Este tiempo se le indica como primer parámetro además de que podemos añadir el tiempo de **delay** y el **easing**. El formato de este parámetro es el siguiente: '**1s 0.5s ease-out**' (duration delay easing).

Los posibles valores para **easing** son: **ease**, **ease-in**, **ease-out**, **ease-in-out** y **cubic-bezier()**.

Podemos ver distintos tipos de valores para el **cubic-bezier** en la siguiente página:
<https://easings.net/>.

29.6. Keyframes

La función **keyframes** se usa como segundo parámetro de la función **animate** y nos permite pasarle un array de estilos como parámetro y en que momento de la animación (**offset**) queremos que se apliquen.

El valor del **offset** va de 0 a 1, donde 1 representa el 100% del tiempo que se ha dado a la función **animate** dentro de la cual se ha puesto el **keyframe**. En caso de no poner esta propiedad, su valor se calcula automáticamente, siendo el mismo para cada uno de los estilos definidos dentro del array de la función **keyframes**.

29.7. Inicio y fin de la animación

Al empezar a ejecutarse las animaciones es posible que queramos ejecutar algo de código.

Pues Angular nos permite hacerlo detectando los eventos de `@nombre-animacion.start` y `@nombre-animacion.done`.

29.8. Lab: Animaciones

En este laboratorio vamos a ver como crear las siguientes animaciones:

- Una con el efecto de zoom.
- Una con el efecto de shake.

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

```
$ ng new angular-animaciones-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 unos componentes y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-animaciones-lab  
$ ng g c cmp-zoom  
$ ng g c cmp-shake  
$ ng s
```

Dentro de nuestro **app.component.html** vamos a mostrar todos estos componentes que hemos creado.

/angular-animaciones-lab/src/app/app.component.html

```
<app-cmp-zoom></app-cmp-zoom>  
<app-cmp-shake></app-cmp-shake>
```

Vamos a empezar añadiendo una animación que haga **zoom** sobre una imagen al pasar el ratón por encima de ella.

Para ello, empezamos por añadir dentro del decorador del componente **CmpZoomComponent** el array de **animations** con el **trigger** y el nombre de la animación.

```
import { state, style, transition, trigger, animate } from '@angular/animations';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-cmp-zoom',
  templateUrl: './cmp-zoom.component.html',
  styleUrls: ['./cmp-zoom.component.css'],
  animations: [
    trigger('zoom', [
      ])
  ]
})
export class CmpZoomComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }
}
```

Para esta animación vamos a declarar dos estados, uno con el valor de **normal** en el que el tamaño de la imagen será el normal, 1. Y el otro, al que vamos a llamar **zoom** donde el tamaño de la imagen será por ejemplo **2.3**.

```
import { state, style, transition, trigger, animate } from '@angular/animations';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-cmp-zoom',
  templateUrl: './cmp-zoom.component.html',
  styleUrls: ['./cmp-zoom.component.css'],
  animations: [
    trigger('zoom', [
      state('normal', style({
        transform: 'scale(1)'
      })),
      state('zoom', style({
        transform: 'scale(2.3)'
      })),
    ])
  ]
})
export class CmpZoomComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }
}
```

Ahora vamos a añadir una propiedad para guardar el valor del estado actual de la animación. Inicialmente le vamos a poner como valor **normal**.

También tenemos que crear una función para cambiar de un estado a otro.

/angular-animaciones-lab/src/app/cmp-zoom/cmp-zoom.component.ts

```
import { state, style, transition, trigger, animate } from '@angular/animations';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-cmp-zoom',
  templateUrl: './cmp-zoom.component.html',
  styleUrls: ['./cmp-zoom.component.css'],
  animations: [
    trigger('zoom', [
      state('normal', style({
        transform: 'scale(1)'
      })),
      state('zoom', style({
        transform: 'scale(2.3)'
      })),
    ])
  ]
})
export class CmpZoomComponent implements OnInit {
  zoomState: string = 'normal'

  constructor() { }

  ngOnInit(): void {
  }

  changeZoomState(): void {
    this.zoomState = this.zoomState === 'normal' ? 'zoom' : 'normal'
  }
}
```

Nos vamos a ir a la plantilla, donde vamos a añadir un div con una imagen.

/angular-animaciones-lab/src/app/cmp-zoom/cmp-zoom.component.html

```
<div>
  
```

Sobre el div vamos a añadir unos estilos, mientras que sobre la imagen, vamos a añadirle la animación **zoom** asignándole la propiedad que guardará el estado de esta.

Además, vamos a añadirle los eventos de **mouseenter** y **mouseleave** para cambiar el estado de la animación al pasar el ratón por encima de la imagen y al quitarlo de ella.

```
<div [ngStyle]="{overflow: 'hidden', width: '150px', height: '150px', border: '1px solid black'}">
  
</div>
```

Pues con esto ya deberíamos de ver un cambio del tamaño de la imagen al pasar el ratón por encima, pero este cambio es demasiado brusco.

Para hacerlo mas fluido, vamos a añadirle a la animación la propiedad **transition** con la que le vamos a indicar que pasando de un estado al otro, tiene que tardar 600ms en pasar de un estado al otro, con un delay inicial de 100ms y siendo el efecto del tipo **ease-in-out**, es decir, que empieza lento, se acelera y termina lento otra vez.

```
import { state, style, transition, trigger, animate } from '@angular/animations';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-cmp-zoom',
  templateUrl: './cmp-zoom.component.html',
  styleUrls: ['./cmp-zoom.component.css'],
  animations: [
    trigger('zoom', [
      state('normal', style({
        transform: 'scale(1)'
      })),
      state('zoom', style({
        transform: 'scale(2.3)'
      })),
      transition('normal <=> zoom', animate('600ms 100ms ease-in-out'))
    ])
]
})
export class CmpZoomComponent implements OnInit {
  zoomState: string = 'normal'

  constructor() { }

  ngOnInit(): void {
  }

  changeZoomState(): void {
    this.zoomState = this.zoomState === 'normal' ? 'zoom' : 'normal'
  }
}
```

Con esto ya tenemos la primera animación.

Ahora vamos a ver como realizar la animación de **shake**, en la que usaremos la función de **keyframes**.

Al igual que antes, vamos a empezar añadiendo la animación en el componente **CmpShakeComponent**.

/angular-animaciones-lab/src/app/cmp-shake/cmp-shake.component.ts

```
import { trigger, transition, animate, keyframes, style } from '@angular/animations';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-cmp-shake',
  templateUrl: './cmp-shake.component.html',
  styleUrls: ['./cmp-shake.component.css'],
  animations: [
    trigger('shake', [
      ])
    ]
})
export class CmpShakeComponent implements OnInit {
  constructor() { }

  ngOnInit(): void {
  }
}
```

Ahora vamos a añadir una transición bidireccional entre los dos estados que tendremos, el **in** y el **out**.

/angular-animaciones-lab/src/app/cmp-shake/cmp-shake.component.ts

```
import { trigger, transition, animate, keyframes, style } from '@angular/animations';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-cmp-shake',
  templateUrl: './cmp-shake.component.html',
  styleUrls: ['./cmp-shake.component.css'],
  animations: [
    trigger('shake', [
      transition('out <=> in', [
        ])
      ])
    ]
})
export class CmpShakeComponent implements OnInit {
  constructor() { }

  ngOnInit(): void {
  }
}
```

Dentro de la transición vamos a indicar que la animación tiene que durar 800ms, y le pasaremos

como segundo parámetro la función de **keyframes**.

/angular-animaciones-lab/src/app/cmp-shake/cmp-shake.component.ts

```
import { trigger, transition, animate, keyframes, style } from '@angular/animations';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-cmp-shake',
  templateUrl: './cmp-shake.component.html',
  styleUrls: ['./cmp-shake.component.css'],
  animations: [
    trigger('shake', [
      transition('out <=> in', [
        animate(800, keyframes([
          style({ transform: 'translateX(0)' })
        ]))
      ])
    ])
  ]
})
export class CmpShakeComponent implements OnInit {
  constructor() { }

  ngOnInit(): void {
  }
}
```

Dentro de la función de keyframes le vamos a pasar los distintos cambios en los estilos que tiene que hacer según va pasando esos 800ms.

En este caso vamos a ir rotando la imagen, y estableceremos el momento de las distintas rotaciones que tiene que hacer con la propiedad **offset**. Este coge como valor un número entre el 0 y el 1 donde representan el 0% del tiempo (inicio de la animación) y el 100% (fin de la animación).

```
import { trigger, transition, animate, keyframes, style } from '@angular/animations';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-cmp-shake',
  templateUrl: './cmp-shake.component.html',
  styleUrls: ['./cmp-shake.component.css'],
  animations: [
    trigger('shake', [
      transition('out <=> in', [
        animate(800, keyframes([
          style({ transform: 'rotate(0deg)', offset: 0 }),
          style({ transform: 'rotate(90deg)', offset: 0.2 }),
          style({ transform: 'rotate(-45deg)', offset: 0.4 }),
          style({ transform: 'rotate(45deg)', offset: 0.6 }),
          style({ transform: 'rotate(-90deg)', offset: 0.8 }),
          style({ transform: 'rotate(0deg)', offset: 1 })
        ]))
      ])
    ])
  ]
})
export class CmpShakeComponent implements OnInit {
  constructor() { }

  ngOnInit(): void {
  }
}
```

Por último, vamos a añadir una propiedad para guardar el estado en el que se encuentra la animación, y añadimos una función para cambiar el estado.

/angular-animaciones-lab/src/app/cmp-shake/cmp-shake.component.ts

```
import { trigger, transition, animate, keyframes, style } from '@angular/animations';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-cmp-shake',
  templateUrl: './cmp-shake.component.html',
  styleUrls: ['./cmp-shake.component.css'],
  animations: [
    trigger('shake', [
      transition('out <=> in', [
        animate(800, keyframes([
          style({ transform: 'rotate(0deg)', offset: 0 }),
          style({ transform: 'rotate(90deg)', offset: 0.2 }),
          style({ transform: 'rotate(-45deg)', offset: 0.4 }),
          style({ transform: 'rotate(45deg)', offset: 0.6 }),
          style({ transform: 'rotate(-90deg)', offset: 0.8 }),
          style({ transform: 'rotate(0deg)', offset: 1 })
        ]))
      ])
    ])
  ]
})
export class CmpShakeComponent implements OnInit {
  shakeState: string = 'out'

  constructor() { }

  ngOnInit(): void {}

  changeShakeState(): void {
    this.shakeState = this.shakeState === 'out' ? 'in' : 'out'
  }
}
```

Ahora en la plantilla del componente, vamos a poner un div con una imagen, al igual que en el componente del zoom.

/angular-animaciones-lab/src/app/cmp-shake/cmp-shake.component.html

```
<div [ngStyle]="{overflow: 'hidden', width: '150px', height: '150px', border: '1px solid black'}">
  
```

Le añadimos la animación y le asignamos la propiedad que gestiona el estado, y añadiremos la

llamada a la función que cambia el estado al pasar el ratón por encima de la imagen.

/angular-animaciones-lab/src/app/cmp-shake/cmp-shake.component.html

```
<div [ngStyle]="{overflow: 'hidden', width: '150px', height: '150px', border: '1px solid black'}">
  
</div>
```

Con esto ya tenemos la segunda animación.

Chapter 30. Test unitarios

Los test unitarios consisten en probar que las unidades que hemos ido creando (componentes, clases, servicios, pipes...) funcionan correctamente, sin llegar a probar la interacción que realizan con otros elementos.

Angular utiliza Jasmine y Karma para la creación y ejecución de los test unitarios.

Algunos de los métodos que nos proporciona Jasmine para realizar las comprobaciones necesarias son:

Matcher	Descripción
toBe(any)	Hace una comprobación ===.
toEqual(any)	Hace una comprobación en profundidad. Para comprobar la igualdad entre objetos y también arrays.
toMatch(RegExp)	Comprueba que hace match con una expresión regular.
toBeTruthy()	Comprueba que es true.
toBeFalsy()	Comprueba que el valor es falsy.
toBeDefined()	Comprueba que es undefined.
BeNull()	Comprueba que es null.
toContain(string)	Comprueba que el valor dado está contenido en el valor esperado. Funciona con arrays y strings.
toBeLessThan(number)	Comprueba que el valor esperado es menor que el valor dado.
toBeGreaterThan(number)	Comprueba que el valor esperado es mayor o igual que el valor dado.
not.<matcher>	Niega el siguiente matcher.

Podemos encontrar los matchers en <https://jasmine.github.io/api/4.0/matchers>.

Estos tests los vamos a reconocer por que terminan con la extensión `.spec.ts`

Para ejecutar los test unitarios hay que lanzar el siguiente comando:

```
$ ng test
```

30.1. Testing componentes

Para testear los componentes, podemos crear una instancia de estos y probar la funcionalidad de la clase al igual que se haría con cualquier otra clase, mirando el valor de las propiedades y ejecutando los métodos de la clase.

Pero luego tenemos otra forma más correcta de hacer estos tests que veremos después, usando la clase **TestBed**.

30.2. Lab: Testing componentes

En este laboratorio vamos a ver como testear un componente contador.

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

```
$ ng new angular-testing-unitarios-componente-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 un componente y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-testing-unitarios-componente-lab  
$ ng g c contador
```

Vamos a empezar por implementar el componente. Dentro del TS vamos a crear una propiedad **cuenta** que inicializaremos a 0, y un método incrementar que va a ir sumándole 1 a la cuenta.

/angular-testing-unitarios-componentes-lab/src/app/contador/contador.component.ts

```
import { Component, OnInit } from '@angular/core';  
  
@Component({  
  selector: 'app-contador',  
  templateUrl: './contador.component.html',  
  styleUrls: ['./contador.component.css']  
})  
export class ContadorComponent implements OnInit {  
  cuenta: number = 0  
  
  constructor() {}  
  
  ngOnInit(): void {}  
  
  incrementar(): void {  
    this.cuenta += 1  
  }  
}
```

En la plantilla vamos a mostrar la cuenta y pondremos un botón para llamar al método de incrementar.

/angular-testing-unitarios-componentes-lab/src/app/contador/contador.component.html

```
<p>Cuenta: {{cuenta}}</p>
<button type="button" (click)="incrementar()">+1</button>
```

Ahora nos vamos a abrir el archivo de test del componente, y aunque viene que alguna configuración inicial, vamos a quitarla y vamos a testearlo sin ella.

/angular-testing-unitarios-componentes-lab/src/app/contador/contador.component.spec.ts

```
import { ContadorComponent } from './contador.component';

describe('ContadorComponent', () => {
  let component: ContadorComponent;

  beforeEach(() => {

  })

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

Dentro del **beforeEach** vamos a crear la instancia del componente, y con ella podremos acceder a las propiedades y métodos que hemos declarado dentro.

/angular-testing-unitarios-componentes-lab/src/app/contador/contador.component.spec.ts

```
import { ContadorComponent } from './contador.component';

describe('ContadorComponent', () => {
  let component: ContadorComponent;

  beforeEach(() => {
    component = new ContadorComponent()
  })

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

Ahora que tenemos la instancia, vamos a ejecutar el test inicial que venía a ver si se pasa correctamente.

Para ejecutar los tests vamos a lanzar el comando:

```
$ ng test
```

Al lanzar el comando se abre un navegador con la ejecución de los tests, si aparece alguno en rojo será porque ese test no se está pasando correctamente.

Ahora vamos a añadir otro test, en el que vamos a comprobar que la cuenta es inicialmente 0.

/angular-testing-unitarios-componentes-lab/src/app/contador/contador.component.spec.ts

```
import { ContadorComponent } from './contador.component';

describe('ContadorComponent', () => {
  let component: ContadorComponent;

  beforeEach(() => {
    component = new ContadorComponent()
  })

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería de tener cuenta a 0 inicialmente', () => {
    expect(component.cuenta).toBe(0)
  });
});
```

Y por último vamos a comprobar que si llamamos a **incrementar**, la cuenta aumenta en 1.

```
import { ContadorComponent } from './contador.component';

describe('ContadorComponent', () => {
  let component: ContadorComponent;

  beforeEach(() => {
    component = new ContadorComponent()
  })

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería de tener cuenta a 0 inicialmente', () => {
    expect(component.cuenta).toBe(0)
  });

  it('debería de incrementar la cuenta en 1 al ejecutar el método incrementar', () => {
    component.incrementar()
    expect(component.cuenta).toBe(1);
  });
});
```

30.3. Testing pipes

Testear un pipe es de lo más sencillo que hay a la hora de testear cualquier elemento en Angular.

Lo único que tenemos que hacer es crear la instancia del Pipe y llamar al método **transform** de este para realizar las comprobaciones que queramos.

30.4. Lab: Testing pipes

En este laboratorio vamos a ver como testear un pipe.

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

```
$ ng new angular-testing-unitarios-pipes-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 un pipe y levantaremos el servidor de desarrollo con los siguientes comandos:

```
$ cd angular-testing-unitarios-pipes-lab  
$ ng g p ocultar-palabras  
$ ng s
```

Vamos a ir al pipe a implementar el método **transform**.

Este pipe tiene que coger un string y un array de palabras y sustituir las palabras que se encuentre dentro del string por el símbolo * (un asterisco por cada letra que tenga la palabra).

Vamos a hacer que el pipe tenga un tercer parámetro para poder cambiar el * por cualquier otro símbolo que le pasemos.

/angular-testing-unitarios-pipes-lab/src/app/ocultar-palabras.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';  
  
@Pipe({  
  name: 'ocultarPalabras'  
)  
export class OcultarPalabrasPipe implements PipeTransform {  
  
  transform(value: string, palabras: Array<string>, sustituto: string = '*'): string {  
  
    palabras.forEach((palabra: string) => {  
      const regExp = new RegExp(palabra, 'gi')  
      value = value.replace(regExp, sustituto.repeat(palabra.length))  
    })  
  
    return value;  
  }  
}
```

Una vez tenemos el pipe, vamos a ir al archivo de tests, donde ya viene uno creado por defecto que

comprueba que la instancia del pipe se ha creado.

/angular-testing-unitarios-pipes-lab/src/app/ocultar-palabras.pipe.spec.ts

```
import { OcultarPalabrasPipe } from './ocultar-palabras.pipe';

describe('OcultarPalabrasPipe', () => {
  it('create an instance', () => {
    const pipe = new OcultarPalabrasPipe();
    expect(pipe).toBeTruthy();
  });
});
```

Lanzamos los tests con el siguiente comando:

```
$ ng test
```

Vamos a añadir un test en el que comprobaremos que cambia las palabras **ipsum**, **potato** y **pepete** por asteriscos.

Empezamos por crear la instancia del pipe, después vamos a añadir el texto, y llamaremos a la función **transform** del pipe pasándole como parámetros el texto y el array de palabras.

/angular-testing-unitarios-pipes-lab/src/app/ocultar-palabras.pipe.spec.ts

```
import { OcultarPalabrasPipe } from './ocultar-palabras.pipe';

describe('OcultarPalabrasPipe', () => {
  it('create an instance', () => {
    const pipe = new OcultarPalabrasPipe();
    expect(pipe).toBeTruthy();
  });

  it('debería de ocultar la palabra mundo con *', () => {
    const pipe = new OcultarPalabrasPipe();

    const texto = 'Minions ipsum tank yuuu! Hana dul sae hahaha. Bee do bee do bee do potatoooo aaaaaah daa. Me want
bananaaa! poulet tikka masala baboiii ti aamoo! La bodaaa belloo! Belloo! Jeje ti aamoo! Wiiiip pepete. Pepete uuuhhh
tank yuuu!'

    const valor = pipe.transform(texto, ['ipsum', 'potato', 'pepete'])
  })
});
```

Ahora toca poner el **expect** y le vamos a pasar el valor que retorna el pipe, y usaremos el matcher **toEqual** de jasmine para compararlo con el mismo texto pero con los asteriscos.

/angular-testing-unitarios-pipes-lab/src/app/ocultar-palabras.pipe.spec.ts

```
import { OcultarPalabrasPipe } from './ocultar-palabras.pipe';

describe('OcultarPalabrasPipe', () => {
  it('create an instance', () => {
    const pipe = new OcultarPalabrasPipe();
    expect(pipe).toBeTruthy();
  });

  it('debería de ocultar la palabra mundo con *', () => {
    const pipe = new OcultarPalabrasPipe();

    const texto = 'Minions ipsum tank yuuu! Hana dul sae hahaha. Bee do bee do bee do potatoooo aaaaaah daa. Me want bananaaa! poulet tikka masala baboiii ti aamoo! La bodaaa belloo! Belloo! Jeje ti aamoo! Wiii pepete. Pepete uuuhhh tank yuuu!'

    const valor = pipe.transform(texto, ['ipsum', 'potato', 'pepete'])

    expect(valor).toEqual('Minions ***** tank yuuu! Hana dul sae hahaha. Bee do bee do bee do *****ooo aaaaaah daa. Me want bananaaa! poulet tikka masala baboiii ti aamoo! La bodaaa belloo! Belloo! Jeje ti aamoo! Wiii *****. ***** uuuhhh tank yuuu!')
  });
});
```

Ahora vamos a añadir otro texto, pero como vamos a repetir unas cuantas instrucciones, vamos a sacarlas al hook **beforeEach**.

/angular-testing-unitarios-pipes-lab/src/app/ocultar-palabras.pipe.spec.ts

```
import { OcultarPalabrasPipe } from './ocultar-palabras.pipe';

describe('OcultarPalabrasPipe', () => {
  let texto: string
  let palabras: Array<string>
  let pipe: OcultarPalabrasPipe;

  beforeEach(() => {
    texto = 'Minions ipsum tank yuuu! Hana dul sae hahaha. Bee do bee do bee do potatoooo aaaaaah daa. Me want bananaaa! poulet tikka masala baboiii ti aamoo! La bodaaa belloo! Belloo! Jeje ti aamoo! Wiii pepete. Pepete uuuhhh tank yuuu!'
    palabras = ['ipsum', 'potato', 'pepete']
    pipe = new OcultarPalabrasPipe();
  })

  it('create an instance', () => {
    expect(pipe).toBeTruthy();
  });

  it('debería de ocultar las palabras con *', () => {
    const valor = pipe.transform(texto, palabras)

    expect(valor).toEqual('Minions ***** tank yuuu! Hana dul sae hahaha. Bee do bee do bee do *****ooo aaaaaah daa. Me want bananaaa! poulet tikka masala baboiii ti aamoo! La bodaaa belloo! Belloo! Jeje ti aamoo! Wiii *****. ***** uuuhhh tank yuuu!')
  });
});
```

Ahora vamos a crear otro test en el que vamos a comprobar que si le pasamos un - como tercer parámetro, en lugar de poner asteriscos pone -.

```
import { OcultarPalabrasPipe } from './ocultar-palabras.pipe';

describe('OcultarPalabrasPipe', () => {
  let texto: string
  let palabras: Array<string>
  let pipe: OcultarPalabrasPipe;

  beforeEach(() => {
    texto = 'Minions ipsum tank yuuu! Hana dul sae hahaha. Bee do bee do bee do potatoooo aaaaaah daa. Me want bananaaaa!
poulet tikka masala baboiii ti aamoo! La bodaaa belloo! Belloo! Jeje ti aamoo! Wiiiip pepete. Pepete uuuhhh tank yuuu!'
    palabras = ['ipsum', 'potato', 'pepete']
    pipe = new OcultarPalabrasPipe();
  })

  it('create an instance', () => {
    expect(pipe).toBeTruthy();
  });

  it('debería de ocultar las palabras con *', () => {
    const valor = pipe.transform(texto, palabras)
    expect(valor).toEqual('Minions ***** tank yuuu! Hana dul sae hahaha. Bee do bee do bee do *****ooo aaaaaah daa. Me
want bananaaaa! poulet tikka masala baboiii ti aamoo! La bodaaa belloo! Belloo! Jeje ti aamoo! Wiiiip *****. *****
uuuhhh tank yuuu!')
  })

  it('debería de ocultar la palabra mundo con -', () => {
    const valor = pipe.transform(texto, palabras, '-')
    expect(valor).toEqual('Minions ---- tank yuuu! Hana dul sae hahaha. Bee do bee do bee do -----ooo aaaaaah daa. Me
want bananaaaa! poulet tikka masala baboiii ti aamoo! La bodaaa belloo! Belloo! Jeje ti aamoo! Wiiiip -----. -----
uuuhhh tank yuuu!')
  })
});
```

30.5. TestBed

La clase **TestBed** va a permitirnos configurar un entorno de testing en el que ejecutar nuestros tests, además de proporcionarnos unos métodos para crear componentes y servicios.

Para configurar el entorno de testing usaremos el método **configureTestingModule** al que le pasaremos un objeto con las propiedades de los módulos de las que vayamos a hacer uso.

Por ejemplo, si vamos a testear el componente A, tendremos que añadirlo dentro del array de declaraciones del módulo de testing.

Otra función que vamos a estar utilizando constantemente es la de **createComponent** con la que vamos a obtener un **fixture** del componente.

Un **fixture** es un wrapper sobre el componente que nos permite acceder a sus propiedades y métodos, además de otros elementos que nos ayudarán a testear el componente:

- **componentInstance**: nos da la instancia del componente para acceder a sus propiedades y métodos.
- **detectChanges**: función que se encarga de realizar la detección de cambios de Angular, ya que en el entorno de testing no se detectan automáticamente.
- **nativeElement**: nos devuelve el elemento nativo del DOM y por tanto podríamos utilizar métodos de JS como **querySelector** para acceder a las etiquetas internas del componente.
- **debugElement**: es un wrapper de Angular que pone sobre el **nativeElement** y nos da otra serie de métodos para interactuar con el DOM:
 - Búsquedas por el DOM con **query(By.<método>)**, **queryAll(By.<método>)**.
 - Emitir eventos con **triggerEventHandler(nombreEvento, objEvento)**.
 - Acceder al inyector de dependencias con **injector**.
 - ...

La clase **By** de **@angular/platform-browser** nos proporciona métodos para buscar los elementos de diferentes formas:



- **By.css('.clase1')**: busca por un selector de CSS.
- **By.directive(Directiva1)**: busca por la clase de una directiva.
- **By.all**: devuelve todos los elementos.

30.6. Lab: HTML de los componentes

En este laboratorio vamos a ver como testear que el HTML de un componente es el correcto, usando la clase TestBed para configurar el entorno de testing.

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

```
$ ng new angular-testing-unitarios-testbed-componentes-html-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 crearemos un componente con los siguientes comandos:

```
$ cd angular-testing-unitarios-testbed-componentes-html-lab  
$ ng g c sugus
```

Vamos a implementar este componente, y para ello, empezamos con el TS en el que vamos a incluir dos propiedades cuyos valores pueden venir desde fuera del componente.

/angular-testing-unitarios-testbed-componentes-html-lab/src/app/sugus/sugus.component.ts

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

En la plantilla vamos a mostrar el sabor, y le pondremos como color de fondo el valor de la propiedad color, además de asignarle una clase para añadirle unos estilos después.

/angular-testing-unitarios-testbed-componentes-html-lab/src/app/sugus/sugus.component.html

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

Y por último le vamos a añadir unos estilos en su archivo de CSS.

/angular-testing-unitarios-testbed-componentes-html-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;
}
```

Pues con esto ya tenemos nuestro componente Sugus completo.

Ahora vamos a testearlo, y para ello, vamos al test donde ya nos encontramos con un primer test.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { SugusComponent } from './sugus.component';

describe('SugusComponent', () => {
  let component: SugusComponent;
  let fixture: ComponentFixture<SugusComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ SugusComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(SugusComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

En este archivo vemos un primer **beforeEach** en el que se está configurando el entorno para testing con la clase **TestBed**.

Vemos que al módulo se le está pasando un array de **declarations** con la clase del sugus. Esto se hace para poder crear el componente, ya que es lo que vamos a testear.

Si este componente necesitara más dependencias, podríamos añadir en este array esas dependencias, como otros componentes, directivas, pipes... Aunque también podríamos querer mockear estas dependencias.

Luego se llama a **compileComponents**, que se encarga de compilar los componentes, por si acaso estos tests se ejecutan en un entorno no gráfico.

En el otro **beforeEach**, nos encontramos con que se está creando el **fixture** del componente, es decir, un wrapper que usaremos para interactuar con nuestro componente de una forma adecuada, y luego obtenemos la instancia de este componente con **componentInstance**.

Además se llama a la función **detectChanges** que se encarga de lanzar la detección de cambios de Angular por si tiene que actualizar la vista con los cambios que haya que realizar sobre el componente.

Pues entendido el código, vamos a crear un nuevo test en el que vamos a comprobar que por

defecto se pinta el sugus de limón.

/angular-testing-unitarios-testbed-componentes-html-lab/src/app/sugus/sugus.component.spec.ts

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { SugusComponent } from './sugus.component';

describe('SugusComponent', () => {
  let component: SugusComponent;
  let fixture: ComponentFixture<SugusComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ SugusComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(SugusComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('el color inicial debería de ser amarillo y el sabor limón', () => {
    expect(component.color).toBe('#FDE23A')
    expect(component.sabor).toBe('limón')
  })
});
```

Ahora vamos a comprobar que el HTML es correcto cuando se pinta el componente, es decir, que en el párrafo aparece limón, y el color de fondo es el amarillo.

Para acceder al HTML vamos a utilizar la propiedad **fixture.nativeElement** con el que accederemos a los elementos nativos del DOM, y para buscar el párrafo y el div, vamos a utilizar **querySelector**.

Una vez buscados, ya podemos acceder a **textContent** para acceder al texto del párrafo.

Y a **style.backgroundColor** para buscar el color de fondo en los estilos del div.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { SugusComponent } from './sugus.component';

describe('SugusComponent', () => {
  let component: SugusComponent;
  let fixture: ComponentFixture<SugusComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ SugusComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(SugusComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('el color inicial debería de ser amarillo y el sabor limón', () => {
    expect(component.color).toBe('#FDE23A')
    expect(component.sabor).toBe('limón')
  })

  it('el párrafo debería de mostrar limón por defecto, y el backgroundColor debería de ser amarillo', () => {
    const txtParrafo = fixture.nativeElement.querySelector('p').textContent
    const bgColor = fixture.nativeElement.querySelector('div.sugus').style.backgroundColor
    expect(txtParrafo).toBe('limón')
    expect(bgColor).toBe('rgb(253, 226, 58)')
  })
});
```

Además de utilizar el **nativeElement**, también podemos usar **debugElement** para acceder a estos elementos, pero esta vez a través de un wrapper que además nos da más opciones para hacer lo mismo.

Vamos a crear el mismo test, pero usando el **debugElement**.

Esta vez, para buscar los elementos vamos a utilizar la función **query**, pasándole como parámetro **By.css** y el selector con el cual encontrar los dos elementos a los que queremos acceder. Una vez obtenidos, usamos **nativeElement** y pedimos los atributos de los cuales queremos obtener los valores que usaremos en el **expect**.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

import { SugusComponent } from './sugus.component';

describe('SugusComponent', () => {
  let component: SugusComponent;
  let fixture: ComponentFixture<SugusComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ SugusComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(SugusComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('el color inicial debería de ser amarillo y el sabor limón', () => {
    expect(component.color).toBe('#FDE23A')
    expect(component.sabor).toBe('limón')
  })

  it('el párrafo debería de mostrar limón por defecto, y el backgroundColor debería de ser amarillo', () => {
    const txtParrafo = fixture.nativeElement.querySelector('p').textContent
    const bgColor = fixture.nativeElement.querySelector('div.sugus').style.backgroundColor
    expect(txtParrafo).toBe('limón')
    expect(bgColor).toBe('rgb(253, 226, 58)')
  })

  it('el párrafo debería de mostrar limón por defecto, y el backgroundColor debería de ser amarillo (debugElement)', () => {
    const txtParrafo = fixture.debugElement.query(By.css('p')).nativeElement.textContent
    const bgColor = fixture.debugElement.query(By.css('.sugus')).nativeElement.style.backgroundColor
    expect(txtParrafo).toBe('limón')
    expect(bgColor).toBe('rgb(253, 226, 58)')
  })
});
```

Vemos que son prácticamente iguales, y es así, solo que el **debugElement** nos da acceso a otro tipo de cosas como el inyector de dependencias o la instancia del componente.

En el siguiente test vamos a comprobar que si le cambiamos las propiedades de color y sabor, pinta un sugus con estos nuevos datos.

Para ello, vamos a empezar por asignarle los nuevos valores a estas propiedades.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

import { SugusComponent } from './sugus.component';

describe('SugusComponent', () => {
  let component: SugusComponent;
  let fixture: ComponentFixture<SugusComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ SugusComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(SugusComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('el color inicial debería de ser amarillo y el sabor limón', () => {
    expect(component.color).toBe('#FDE23A')
    expect(component.sabor).toBe('limón')
  })

  it('el párrafo debería de mostrar limón por defecto, y el backgroundColor debería de ser amarillo', () => {
    const txtParrafo = fixture.nativeElement.querySelector('p').textContent
    const bgColor = fixture.nativeElement.querySelector('div.sugus').style.backgroundColor
    expect(txtParrafo).toBe('limón')
    expect(bgColor).toBe('rgb(253, 226, 58)')
  })

  it('el párrafo debería de mostrar limón por defecto, y el backgroundColor debería de ser amarillo (debugElement)', () => {
    const txtParrafo = fixture.debugElement.query(By.css('p')).nativeElement.textContent
    const bgColor = fixture.debugElement.query(By.css('.sugus')).nativeElement.style.backgroundColor
    expect(txtParrafo).toBe('limón')
    expect(bgColor).toBe('rgb(253, 226, 58)')
  })

  it('debería pintar el sugus con el color azul y el sabor piña cuando se le asignan estos valores a las propiedades color y sabor', () => {
    component.color = '#227BBE'
    component.sabor = 'piña'

    const txtParrafo = fixture.nativeElement.querySelector('p').textContent
    const bgColor = fixture.debugElement.query(By.css('.sugus')).nativeElement.style.backgroundColor
    expect(txtParrafo).toContain('piña')
    expect(bgColor).toBe('rgb(34, 123, 190)')
  })
});
```

Si nos fijamos en el navegador de los resultados, veremos que nos da dos errores en los que nos indica que los valores del sugus siguen siendo los valores iniciales y no los que le hemos asignado.

SugusComponent > debería pintar el sugus con el color azul y el sabor piña cuando se le asignan estos valores a las propiedades color y sabor

Expected 'limón' to contain 'piña'.

```
Error: Expected 'limón' to contain 'piña'.
at <Jasmine>
at UserContext.<anonymous> (http://localhost:9876/_karma_webpack_/webpack:/src/app/sugus/sugus.component.spec.ts:5:1)
at ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:372:1)
at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone-testi
```

Expected 'rgb(253, 226, 58)' to be 'rgb(34, 123, 190)'.

```
Error: Expected 'rgb(253, 226, 58)' to be 'rgb(34, 123, 190)'.
at <Jasmine>
at UserContext.<anonymous> (http://localhost:9876/_karma_webpack_/webpack:/src/app/sugus/sugus.component.spec.ts:5:1)
at ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:372:1)
at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone-testi
```

Esto se debe a la detección de cambios, que en los entornos de testing no está habilitada y por tanto somos nosotros quienes le tenemos que indicar que la aplique para que actualice los componentes. Para esto, solo tenemos que llamar a la función **fixture.detectChanges** cuando realicemos algún cambio sobre las propiedades, o sepamos que este cambio va a ocurrir.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

import { SugusComponent } from './sugus.component';

describe('SugusComponent', () => {
  let component: SugusComponent;
  let fixture: ComponentFixture<SugusComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ SugusComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(SugusComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('el color inicial debería de ser amarillo y el sabor limón', () => {
    expect(component.color).toBe('#FDE23A')
    expect(component.sabor).toBe('limón')
  })

  it('el párrafo debería de mostrar limón por defecto, y el backgroundColor debería de ser amarillo', () => {
    const txtParrafo = fixture.nativeElement.querySelector('p').textContent
    const bgColor = fixture.nativeElement.querySelector('div.sugus').style.backgroundColor
    expect(txtParrafo).toBe('limón')
    expect(bgColor).toBe('rgb(253, 226, 58)')
  })

  it('el párrafo debería de mostrar limón por defecto, y el backgroundColor debería de ser amarillo (debugElement)', () => {
    const txtParrafo = fixture.debugElement.query(By.css('p')).nativeElement.textContent
    const bgColor = fixture.debugElement.query(By.css('.sugus')).nativeElement.style.backgroundColor
    expect(txtParrafo).toBe('limón')
    expect(bgColor).toBe('rgb(253, 226, 58)')
  })

  it('debería pintar el sugus con el color azul y el sabor piña cuando se le asignan estos valores a las propiedades color y sabor', () => {
    component.color = '#227BBE'
    component.sabor = 'piña'

    fixture.detectChanges()

    const txtParrafo = fixture.nativeElement.querySelector('p').textContent
    const bgColor = fixture.debugElement.query(By.css('.sugus')).nativeElement.style.backgroundColor
    expect(txtParrafo).toContain('piña')
    expect(bgColor).toBe('rgb(34, 123, 190)')
  })
});
```

Con este cambio deberían de pasarse correctamente estas comprobaciones.

30.7. Lab: Lanzar eventos dentro de componentes

En este laboratorio vamos a ver como testear un componente en el que podemos lanzar un evento para que cambie una propiedad.

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

```
$ ng new angular-testing-unitarios-testbed-componentes-con-eventos-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 crearemos un componente con los siguientes comandos:

```
$ cd angular-testing-unitarios-testbed-componentes-con-eventos-lab  
$ ng g c tema-app
```

Vamos a testear que al pulsar sobre un botón, cambia un botón que nos permite activar el modo oscuro o el modo claro de la aplicación dependiendo de una propiedad **darkMode**.

Para ello, en el componente que hemos creado, vamos a añadir esta propiedad y una función **cambiarTemaApp** para cambiar su valor.

/angular-testing-unitarios-testbed-componentes-con-eventos-lab/src/app/tema-app/tema-app.component.ts

```
import { Component, OnInit } from '@angular/core';  
  
@Component({  
  selector: 'app-tema-app',  
  templateUrl: './tema-app.component.html',  
  styleUrls: ['./tema-app.component.css']  
})  
export class TemaAppComponent implements OnInit {  
  darkMode: boolean = true  
  
  constructor() { }  
  
  ngOnInit(): void {  
  }  
  
  cambiarTemaApp(activar: boolean): void {  
    this.darkMode = activar  
  }  
}
```

En la plantilla vamos a añadir dos botones que llamarán a la función de **cambiarTemaApp**, uno

con un true y el otro con un false, y les añadiremos el **ngIf** para mostrar el que toca dependiendo de la propiedad **darkMode**.

/angular-testing-unitarios-testbed-componentes-con-eventos-lab/src/app/tema-app/tema-app.component.html

```
<button type="button" (click)="cambiarTemaApp(false)" *ngIf="darkMode; else darkModeBtn">Activar light mode </button>  
  
<ng-template #darkModeBtn>  
  <button type="button" (click)="cambiarTemaApp(true)">Activar dark mode </button>  
</ng-template>
```

Ahora nos vamos al archivo de test.

Vamos a crear un primer test en el que empezaremos buscando el botón que se está mostrando y cuando lo tengamos, vamos a lanzar un evento **click** con el método **triggerEventHandler**.

/angular-testing-unitarios-testbed-componentes-con-eventos-lab/src/app/tema-app/tema-app.component.spec.ts

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

import { TemaAppComponent } from './tema-app.component';

describe('TemaAppComponent', () => {
  let component: TemaAppComponent;
  let fixture: ComponentFixture<TemaAppComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ TemaAppComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(TemaAppComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería mostrar el botón con el ☀ si el modo oscuro está activado', () => {
    const button = fixture.debugElement.query(By.css('button'))
    button.triggerEventHandler('click', {})
  })
});
```

Una vez lanzado el evento, ya podemos comprobar que el texto del botón contiene el emoji del sol.

/angular-testing-unitarios-testbed-componentes-con-eventos-lab/src/app/tema-app/tema-app.component.spec.ts

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

import { TemaAppComponent } from './tema-app.component';

describe('TemaAppComponent', () => {
  let component: TemaAppComponent;
  let fixture: ComponentFixture<TemaAppComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ TemaAppComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(TemaAppComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería mostrar el botón con el ☀ si el modo oscuro está activado', () => {
    const button = fixture.debugElement.query(By.css('button'))
    button.triggerEventHandler('click', {})
    expect(button.nativeElement.textContent).toContain('☀')
  })
});
```

Lanzamos los tests con el comando:

```
$ ng test
```

Ahora vamos a hacer el caso contrario, vamos a cambiar el valor de **darkMode** a false para que se muestre el botón del sol inicialmente. Para que se actualicen los cambios, llamamos al **detectChanges**.

/angular-testing-unitarios-testbed-componentes-con-eventos-lab/src/app/tema-app/tema-app.component.spec.ts

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

import { TemaAppComponent } from './tema-app.component';

describe('TemaAppComponent', () => {
  let component: TemaAppComponent;
  let fixture: ComponentFixture<TemaAppComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ TemaAppComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(TemaAppComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería mostrar el botón con el ☽ si el modo oscuro está activado', () => {
    const button = fixture.debugElement.query(By.css('button'))
    button.triggerEventHandler('click', {})
    expect(button.nativeElement.textContent).toContain('☽')
  })

  it('debería mostrar el botón con la ☾ si el modo oscuro está desactivado', () => {
    component.darkMode = false
    fixture.detectChanges()
  })
});
```

Ahora solo tenemos que hacer lo que hemos hecho en el test anterior, buscar el botón, lanzar el evento **click** sobre él y comprobar que en el texto del botón aparece el emoji de la luna.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

import { TemaAppComponent } from './tema-app.component';

describe('TemaAppComponent', () => {
  let component: TemaAppComponent;
  let fixture: ComponentFixture<TemaAppComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ TemaAppComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(TemaAppComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería mostrar el botón con el ☰ si el modo oscuro está activado', () => {
    const button = fixture.debugElement.query(By.css('button'))
    button.triggerEventHandler('click', {})
    expect(button.nativeElement.textContent).toContain('☰')
  })

  it('debería mostrar el botón con la ☰ si el modo oscuro está desactivado', () => {
    component.darkMode = false
    fixture.detectChanges()
    const button = fixture.debugElement.query(By.css('button'))
    button.triggerEventHandler('click', {})
    expect(button.nativeElement.textContent).toContain('☰')
  })
});
```

30.8. Testing servicios

Para testear servicios tenemos que crear la instancia del servicio y una vez la tenemos podemos llamar a los métodos que hayamos implementado para comprobar que funcionan correctamente.

Cuando estos servicios interactúan con APIs externas, entonces podremos crear mocks de estas APIs para no tener dependencias que no podemos controlar.

30.9. Lab: Testing servicios

En este laboratorio vamos a ver como testear un servicio.

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

```
$ ng new angular-testing-unitarios-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, crearemos un servicio y una clase:

```
$ cd angular-testing-unitarios-servicios-lab  
$ ng g s tareas  
$ ng g cl tarea
```

Vamos a empezar por poner en el constructor de la clase que esta tendrá un título, un identificador y un booleano completada.

/angular-testing-unitarios-servicios-lab/src/app/tarea.ts

```
export class Tarea {  
  constructor(  
    public id: string,  
    public titulo: string,  
    public completada: boolean) {}  
}
```

Dentro del servicio, vamos a declarar un array de tareas vacio, y tres métodos, uno para añadir tareas, otro para eliminarlas y el último para completarlas.

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

@Injectable({
  providedIn: 'root'
})
export class TareasService {
  tareas: Array<Tarea> = []

  constructor() { }

  addTarea(titulo: string): void {
    const id = Math.random().toString().slice(2)
    const tarea = new Tarea(id, titulo, false)
    this.tareas = [...this.tareas, tarea]
  }

  completeTarea(id: string): void {
    this.tareas = this.tareas.map((t: Tarea) => {
      if (t.id === id) {
        return new Tarea(t.id, t.titulo, !t.completada)
      }
      return t
    })
  }

  delTarea(id: string): void {
    this.tareas = this.tareas.filter((t: Tarea) => t.id !== id)
  }
}
```

Una vez que tenemos el servicio implementado, nos vamos a ir al archivo de testing de este servicio donde nos encontraremos un primer test con el **beforeEach** en el que se pide la instancia del servicio.

```
import { TestBed } from '@angular/core/testing';

import { TareasService } from './tareas.service';

describe('TareasService', () => {
  let service: TareasService;

  beforeEach(() => {
    TestBed.configureTestingModule({});  
    service = TestBed.inject(TareasService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });
});
```

Lanzamos los tests con el siguiente comando:

```
$ ng test
```

Vamos a crear un primer test en el que comprobaremos que inicialmente no hay tareas en el servicio.

```
import { TestBed } from '@angular/core/testing';
import { TareasService } from './tareas.service';

describe('TareasService', () => {
  let service: TareasService;

  beforeEach(() => {
    TestBed.configureTestingModule({});  

    service = TestBed.inject(TareasService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('inicialmente no debería de haber ninguna tarea', () => {
    expect(service.tareas.length).toBe(0)
  })
});
```

En el siguiente test vamos a añadir una tarea y comprobar que la longitud del array de tareas es 1.

```
import { TestBed } from '@angular/core/testing';
import { TareasService } from './tareas.service';

describe('TareasService', () => {
  let service: TareasService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(TareasService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('inicialmente no debería de haber ninguna tarea', () => {
    expect(service.tareas.length).toBe(0)
  })

  it('debería de añadir una tarea al llamar a addTarea', () => {
    service.addTarea('Tarea 1')
    expect(service.tareas.length).toBe(1)
  })
});
```

En el siguiente test vamos a comprobar que se pueden eliminar. Para ello vamos a empezar añadiendo 3 tareas.

```
import { TestBed } from '@angular/core/testing';
import { TareasService } from './tareas.service';

describe('TareasService', () => {
  let service: TareasService;

  beforeEach(() => {
    TestBed.configureTestingModule({});  

    service = TestBed.inject(TareasService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('inicialmente no debería de haber ninguna tarea', () => {
    expect(service.tareas.length).toBe(0)
  })

  it('debería de añadir una tarea al llamar a addTarea', () => {
    service.addTarea('Tarea 1')
    expect(service.tareas.length).toBe(1)
  })

  it('debería de eliminar la tarea al llamar a delTarea', () => {
    service.addTarea('Tarea 1')
    service.addTarea('Tarea 2')
    service.addTarea('Tarea 3')

  })
});

});
```

Ahora vamos a obtener el identificador de la segunda tarea y se lo vamos a pasar a la función **delTarea** del servicio.

```
import { TestBed } from '@angular/core/testing';
import { TareasService } from './tareas.service';

describe('TareasService', () => {
  let service: TareasService;

  beforeEach(() => {
    TestBed.configureTestingModule({});  

    service = TestBed.inject(TareasService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('inicialmente no debería de haber ninguna tarea', () => {
    expect(service.tareas.length).toBe(0)
  })

  it('debería de añadir una tarea al llamar a addTarea', () => {
    service.addTarea('Tarea 1')
    expect(service.tareas.length).toBe(1)
  })

  it('debería de eliminar la tarea al llamar a delTarea', () => {
    service.addTarea('Tarea 1')
    service.addTarea('Tarea 2')
    service.addTarea('Tarea 3')
    const id = service.tareas[1].id
    service.delTarea(id)
  })
})
```

Ahora vamos a comprobar que la longitud del array de tareas es de 2.

Además comprobaremos que esa tarea no existe dentro del array de tareas, sacando los identificadores de todas ellas a un array, y comprobando que el id eliminado no está contenido en dicho array.

```
import { TestBed } from '@angular/core/testing';
import { TareasService } from './tareas.service';

describe('TareasService', () => {
  let service: TareasService;

  beforeEach(() => {
    TestBed.configureTestingModule({});  

    service = TestBed.inject(TareasService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('inicialmente no debería de haber ninguna tarea', () => {
    expect(service.tareas.length).toBe(0)
  })

  it('debería de añadir una tarea al llamar a addTarea', () => {
    service.addTarea('Tarea 1')
    expect(service.tareas.length).toBe(1)
  })

  it('debería de eliminar la tarea al llamar a delTarea', () => {
    service.addTarea('Tarea 1')
    service.addTarea('Tarea 2')
    service.addTarea('Tarea 3')
    const id = service.tareas[1].id
    service.delTarea(id)
    expect(service.tareas.length).toBe(2)
    expect(service.tareas.map(t => t.id)).not.toContain(id)
  })
});
```

Por último vamos a comprobar que podemos completar las tareas, y para ello vamos a crear una tarea, obtendremos su identificador y vamos a pasarselo a la función **completeTarea**.

```
import { TestBed } from '@angular/core/testing';
import { TareasService } from './tareas.service';

describe('TareasService', () => {
  let service: TareasService;

  beforeEach(() => {
    TestBed.configureTestingModule({});  

    service = TestBed.inject(TareasService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('inicialmente no debería de haber ninguna tarea', () => {
    expect(service.tareas.length).toBe(0)
  })

  it('debería de añadir una tarea al llamar a addTarea', () => {
    service.addTarea('Tarea 1')
    expect(service.tareas.length).toBe(1)
  })

  it('debería de eliminar la tarea al llamar a delTarea', () => {
    service.addTarea('Tarea 1')
    service.addTarea('Tarea 2')
    service.addTarea('Tarea 3')
    const id = service.tareas[1].id
    service.delTarea(id)
    expect(service.tareas.length).toBe(2)
    expect(service.tareas.map(t => t.id)).not.toContain(id)
  })

  it('debería de completar la tarea al llamar a completeTarea', () => {
    service.addTarea('Tarea 1')
    const id = service.tareas[0].id
    service.completeTarea(id)

  })
});
```

Ahora vamos a comprobar que la propiedad **completada** es true.

```
import { TestBed } from '@angular/core/testing';
import { TareasService } from './tareas.service';

describe('TareasService', () => {
  let service: TareasService;

  beforeEach(() => {
    TestBed.configureTestingModule({});  

    service = TestBed.inject(TareasService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('inicialmente no debería de haber ninguna tarea', () => {
    expect(service.tareas.length).toBe(0)
  })

  it('debería de añadir una tarea al llamar a addTarea', () => {
    service.addTarea('Tarea 1')
    expect(service.tareas.length).toBe(1)
  })

  it('debería de eliminar la tarea al llamar a delTarea', () => {
    service.addTarea('Tarea 1')
    service.addTarea('Tarea 2')
    service.addTarea('Tarea 3')
    const id = service.tareas[1].id
    service.delTarea(id)
    expect(service.tareas.length).toBe(2)
    expect(service.tareas.map(t => t.id)).not.toContain(id)
  })

  it('debería de completar la tarea al llamar a completeTarea', () => {
    service.addTarea('Tarea 1')
    const id = service.tareas[0].id
    service.completeTarea(id)

    expect(service.tareas[0].completada).toBeTruthy()
  })
});
```

30.10. Lab: Mocks

En este laboratorio vamos a ver como utilizar los mocks para testear los componentes sin utilizar sus dependencias reales.

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

```
$ ng new angular-testing-unitarios-mocks-lab  
? Would you like to add Angular routing? Y  
? 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 unos componentes y un servicio con los siguientes comandos:

```
$ cd angular-testing-unitarios-mocks-lab  
$ ng g c tarea  
$ ng g c lista-tareas  
$ ng g c nueva-tarea  
$ ng g s tareas
```

Vamos a ir rellenando estos elementos primero, y después veremos como hacer los tests.

Empezamos por el servicio, añadiendo un método para obtener la lista de tareas, otro para obtener una tarea dada su identificador, y uno más para guardar nuevas tareas.

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

@Injectable({
  providedIn: 'root'
})
export class TareasService {
  private URL: string = 'https://ejemplos-dc1c1.firebaseio.com/tareas'
  constructor(private http: HttpClient) { }

  getTareas(): Observable<any> {
    return this.http.get(`.${this.URL}.json`)
      .pipe(
        map((tareas: any) => {
          const arrTareas: Array<any> = []
          for (let id in tareas) {
            const tarea = {...tareas[id], id}
            arrTareas.push(tarea)
          }
          return arrTareas
        })
      )
  }

  getTarea(id: string): Observable<any> {
    return this.http.get(`.${this.URL}/${id}.json`)
      .pipe(
        map((tarea: any) => {
          tarea.id = id
          return tarea
        })
      )
  }

  createTarea(nuevaTarea: any): Observable<any> {
    return this.http.post(`.${this.URL}.json`, nuevaTarea)
  }
}
```

En la plantilla del componente App vamos a añadir un par de enlaces y el **router-outlet** para pintar las distintas páginas de la aplicación.

/angular-testing-unitarios-mocks-lab/src/app/app.component.html

```
<a [routerLink]=["'/'"]>Inicio</a>
<a [routerLink]=["'/nueva-tarea']>Nueva tarea</a>

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

En el módulo de rutas que ha generado Angular, vamos a añadir nuestras tres rutas.

/angular-testing-unitarios-mocks-lab/src/app/app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { NuevaTareaComponent } from './nueva-tarea/nueva-tarea.component';
import { TareaComponent } from './tarea/tarea.component';
import { ListaTareasComponent } from './lista-tareas/lista-tareas.component';

const routes: Routes = [
  { path: '', component: ListaTareasComponent },
  { path: 'nueva-tarea', component: NuevaTareaComponent },
  { path: 'tareas/:id', component: TareaComponent },
];

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

En el componente **ListaTareas** vamos a pedir las tareas y las vamos a almacenar en una propiedad para luego mostrarlas en la plantilla.

/angular-testing-unitarios-mocks-lab/src/app/lista-tareas/lista-tareas.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
import { TareasService } from '../tareas.service';

@Component({
  selector: 'app-lista-tareas',
  templateUrl: './lista-tareas.component.html',
  styleUrls: ['./lista-tareas.component.css']
})
export class ListaTareasComponent implements OnInit {
  listaTareas: Array<any> = []

  constructor(private tareas: TareasService) {}

  ngOnInit(): void {
    this.tareas.getTareas()
      .subscribe(tareas => {
        this.listaTareas = tareas
      })
  }
}
```

Y en la plantilla vamos a mostrar las tareas que hemos obtenido en una lista, añadiendo un enlace para ir a ver la información completa de la tarea.

/angular-testing-unitarios-mocks-lab/src/app/lista-tareas/lista-tareas.component.html

```
<h2>Listado de tareas</h2>

<ul>
  <li *ngFor="let t of listaTareas">
    <a [routerLink]=["/tareas", t.id]">{{t.titulo}}</a>
  </li>
</ul>
```

En el componente de **TareaComponent** vamos a sacar el identificador de la tarea de la URL, y pediremos los datos de esta tarea para guardarlos en una propiedad.

/angular-testing-unitarios-mocks-lab/src/app/tarea/tarea.component.ts

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

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

  constructor(private activatedRoute: ActivatedRoute, private tareas: TareasService) { }

  ngOnInit(): void {
    const id = this.activatedRoute.snapshot.paramMap.get('id')!
    this.tareas.getTarea(id)
      .subscribe((tarea: any) => {
        this.tarea = tarea
      })
  }
}
```

En la plantilla vamos a mostrar los datos de la tarea.

/angular-testing-unitarios-mocks-lab/src/app/tarea/tarea.component.html

```
<h2>Tarea {{tarea.id}}</h2>

<p>Título: {{tarea.titulo}}</p>

<label for="completada">
  Completada: <input type="checkbox" id="completada" [checked]="tarea.completada">
</label>
```

En el componente de **NuevaTarea** vamos a crear un formulario reactivo con un campo **título**, y uno **completada**, y añadimos una función **guardar** para obtener el valor del formulario y guardar la tarea haciendo la petición POST del servicio.

/angular-testing-unitarios-mocks-lab/src/app/nueva-tarea/nueva-tarea.component.ts

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { TareasService } from '../tareas.service';

@Component({
  selector: 'app-nueva-tarea',
  templateUrl: './nueva-tarea.component.html',
  styleUrls: ['./nueva-tarea.component.css']
})
export class NuevaTareaComponent implements OnInit {
  formTarea: FormGroup

  constructor(private tareas: TareasService, private router: Router) {
    this.formTarea = new FormGroup({
      titulo: new FormControl('', Validators.required),
      completada: new FormControl(false)
    })
  }

  ngOnInit(): void {
  }

  guardar() {
    this.tareas.createTarea(this.formTarea.value)
      .subscribe(() => {
        this.router.navigate(['/'])
      })
  }
}
```

Y en la plantilla creamos la estructura del formulario.

/angular-testing-unitarios-mocks-lab/src/app/nueva-tarea/nueva-tarea.component.html

```
<h2>Nueva tarea</h2>

<form [formGroup]="formTarea" (ngSubmit)="guardar()">
  <div>
    <label for="titulo">Nueva tarea</label>
    <input type="text" id="titulo" formControlName="titulo">
  </div>
  <button type="submit" [disabled]="formTarea.invalid">Guardar</button>
</form>
```

Por último, en el módulo de la aplicación vamos a importar los módulos de Http y de los formularios reactivos.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http'
import { ReactiveFormsModule } from '@angular/forms';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { NuevaTareaComponent } from './nueva-tarea/nueva-tarea.component';
import { TareaComponent } from './tarea/tarea.component';
import { ListaTareasComponent } from './lista-tareas/lista-tareas.component';

@NgModule({
  declarations: [
    AppComponent,
    NuevaTareaComponent,
    TareaComponent,
    ListaTareasComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    ReactiveFormsModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Pues ya tenemos la aplicación a testear, ahora vamos a realizar los tests.

Vamos a empezar por eliminar el último **it** del archivo de testing de App porque ese texto que se comprueba que aparece en la página lo hemos eliminado.

```
import { TestBed } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';

fdescribe('AppComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [
        RouterTestingModule
      ],
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  });

  it('should create the app', () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    expect(app).toBeTruthy();
  });

  it(`should have as title 'angular-testing-unitarios-mocks-lab'`, () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    expect(app.title).toEqual('angular-testing-unitarios-mocks-lab');
  });
})
```

Vamos a ir al test del componente de **ListaTareas** y empezaremos por comprobar que por cada tarea que hay en el array se pinta un elemento **li**.

Para ello, vamos a crear un **it** más.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { ListaTareasComponent } from './lista-tareas.component';

describe('ListaTareasComponent', () => {
  let component: ListaTareasComponent;
  let fixture: ComponentFixture<ListaTareasComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ ListaTareasComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(ListaTareasComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería pintar un li por cada tarea del array', () => {
    })
});
```

En este test vamos a utilizar un servicio externo, el de TareasService, y como estamos haciendo test unitarios, estos no tendrían que utilizar estas dependencias. Por lo que vamos a mockear este servicio con **jasmine.createSpyObj**, donde le diremos que tiene que crear un objeto de **TareasService** con un método **getTareas** que no van a hacer nada.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { ListaTareasComponent } from './lista-tareas.component';
import { TareasService } from '../tareas.service';

describe('ListaTareasComponent', () => {
  let component: ListaTareasComponent;
  let fixture: ComponentFixture<ListaTareasComponent>;
  let mockTareasService: any;

  beforeEach(async () => {
    mockTareasService = jasmine.createSpyObj(TareasService, ['getTareas'])

    await TestBed.configureTestingModule({
      declarations: [ ListaTareasComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(ListaTareasComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería pintar un li por cada tarea del array', () => {
  })
});
```

Ahora le vamos a indicar en la propiedad de **providers** que cuando vayan a utilizar el TareasService, en su lugar utilicen el mockTareasService como valor, de esta forma dejamos de utilizar el servicio real por el falso.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { ListaTareasComponent } from './lista-tareas.component';
import { TareasService } from '../tareas.service';

describe('ListaTareasComponent', () => {
  let component: ListaTareasComponent;
  let fixture: ComponentFixture<ListaTareasComponent>;
  let mockTareasService: any;

  beforeEach(async () => {
    mockTareasService = jasmine.createSpyObj(TareasService, ['getTareas'])

    await TestBed.configureTestingModule({
      declarations: [ ListaTareasComponent ],
      providers: [
        { provide: TareasService, useValue: mockTareasService }
      ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(ListaTareasComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería pintar un li por cada tarea del array', () => {
  })
});
```

Otra cosa que tenemos que hacer es añadir el módulo de HttpClient preparado para el testing, es decir el **HttpClientTestingModule**. Mockear el módulo de HttpClientModule es complejo, y este otro módulo nos facilitará este trabajo.

Este módulo lo tenemos que añadir en el array de **imports**.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ListaTareasComponent } from './lista-tareas.component';
import { TareasService } from '../tareas.service';

fdescribe('ListaTareasComponent', () => {
  let component: ListaTareasComponent;
  let fixture: ComponentFixture<ListaTareasComponent>;
  let mockTareasService: any;

  beforeEach(async () => {
    mockTareasService = jasmine.createSpyObj(TareasService, ['getTareas'])

    await TestBed.configureTestingModule({
      declarations: [ ListaTareasComponent ],
      imports: [
        HttpClientTestingModule
      ],
      providers: [
        { provide: TareasService, useValue: mockTareasService }
      ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(ListaTareasComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería pintar un li por cada tarea del array', () => {
    })
});
```

Con esto ya podemos centrarnos en implementar este primer test.

Como cuando se inicializa el componente de la lista de tareas se realiza una petición GET para obtener las tareas (desde el servicio de tareas), pues vamos a indicarle que cuando se detecte que se va a llamar al método **getTareas** del servicio (que ya hemos mockeado), vamos a devolver unos datos que podemos controlar nosotros.

Devolveremos un observable de un array con dos tareas utilizando el **returnValue** del mock.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ListaTareasComponent } from './lista-tareas.component';
import { TareasService } from '../tareas.service';
import { of } from 'rxjs';

fdescribe('ListaTareasComponent', () => {
  let component: ListaTareasComponent;
  let fixture: ComponentFixture<ListaTareasComponent>;
  let mockTareasService: any;

  beforeEach(async () => {
    mockTareasService = jasmine.createSpyObj(TareasService, ['getTareas'])

    await TestBed.configureTestingModule({
      declarations: [ ListaTareasComponent ],
      imports: [
        HttpClientTestingModule
      ],
      providers: [
        { provide: TareasService, useValue: mockTareasService }
      ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(ListaTareasComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería pintar un li por cada tarea del array', () => {
    const mockTareas = [{titulo: 'T1', completada: false, id: 1}, {titulo: 'T2', completada: true, id: 2},]
    mockTareasService.getTareas.and.returnValue(of(mockTareas))

  });
});
```

Como esto va a modificar las propiedades del componente, tenemos que detectar los cambios, por lo que vamos a llamar al método **detectChanges**.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ListaTareasComponent } from './lista-tareas.component';
import { TareasService } from '../tareas.service';
import { of } from 'rxjs';

fdescribe('ListaTareasComponent', () => {
  let component: ListaTareasComponent;
  let fixture: ComponentFixture<ListaTareasComponent>;
  let mockTareasService: any;

  beforeEach(async () => {
    mockTareasService = jasmine.createSpyObj(TareasService, ['getTareas'])

    await TestBed.configureTestingModule({
      declarations: [ ListaTareasComponent ],
      imports: [
        HttpClientTestingModule
      ],
      providers: [
        { provide: TareasService, useValue: mockTareasService }
      ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(ListaTareasComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería pintar un li por cada tarea del array', () => {
    const mockTareas = [{titulo: 'T1', completada: false, id: 1}, {titulo: 'T2', completada: true, id: 2},]
    mockTareasService.getTareas.and.returnValue(of(mockTareas))

    fixture.detectChanges()
  })
});
```

Y justo después de esto, ya podemos comprobar en la plantilla aparecen 2 elementos li.

Para ello vamos a buscar estos elementos con **debugElement.queryAll** y usando **By.css**, y una vez obtengamos la lista de elementos comprobamos que tienen una longitud de 2.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ListaTareasComponent } from './lista-tareas.component';
import { TareasService } from '../tareas.service';
import { of } from 'rxjs';
import { By } from '@angular/platform-browser';

fdescribe('ListaTareasComponent', () => {
  let component: ListaTareasComponent;
  let fixture: ComponentFixture<ListaTareasComponent>;
  let mockTareasService: any;

  beforeEach(async () => {
    mockTareasService = jasmine.createSpyObj(TareasService, ['getTareas'])

    await TestBed.configureTestingModule({
      declarations: [ ListaTareasComponent ],
      imports: [
        HttpClientTestingModule
      ],
      providers: [
        { provide: TareasService, useValue: mockTareasService }
      ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(ListaTareasComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería pintar un li por cada tarea del array', () => {
    const mockTareas = [{titulo: 'T1', completada: false, id: 1}, {titulo: 'T2', completada: true, id: 2,}]
    mockTareasService.getTareas.and.returnValue(of(mockTareas))

    fixture.detectChanges()

    const listItems = fixture.debugElement.queryAll(By.css('li'))
    expect(listItems).toHaveLength(2)
  });
});
```

Ya podemos lanzar los tests con el siguiente comando:

```
$ ng test
```

Al ejecutar los tests, vemos que da un error en el que se indica que no se pueden leer las propiedades de **undefined**.

Esto lo corregimos quitando el **detectChanges** del **beforeEach**, ya que se está ejecutando la detección de cambios antes de que le hayamos indicado que devolver con el mock de las tareas.

/angular-testing-unitarios-mocks-lab/src/app/lista-tareas/lista-tareas.component.spec.ts

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ListaTareasComponent } from './lista-tareas.component';
import { TareasService } from '../tareas.service';
import { of } from 'rxjs';
import { By } from '@angular/platform-browser';

fdescribe('ListaTareasComponent', () => {
  let component: ListaTareasComponent;
  let fixture: ComponentFixture<ListaTareasComponent>;
  let mockTareasService: any;

  beforeEach(async () => {
    mockTareasService = jasmine.createSpyObj(TareasService, ['getTareas'])

    await TestBed.configureTestingModule({
      declarations: [ ListaTareasComponent ],
      imports: [
        HttpClientTestingModule
      ],
      providers: [
        { provide: TareasService, useValue: mockTareasService }
      ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(ListaTareasComponent);
    component = fixture.componentInstance;
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería pintar un li por cada tarea del array', () => {
    const mockTareas = [{titulo: 'T1', completada: false, id: 1}, {titulo: 'T2', completada: true, id: 2,}]
    mockTareasService.getTareas.and.returnValue(of(mockTareas))

    fixture.detectChanges()

    const listItems = fixture.debugElement.queryAll(By.css('li'))
    expect(listItems).toHaveLength(2)
  });
});
```

Después de este cambio, ya debería de pasarse el test correctamente.

Vamos a crear otro test, esta vez para el componente de **NuevaTarea**.

Este componente depende de los formularios reactivos, del router y del http, por lo que vamos a importar estos módulos dentro del **imports**.

```
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { RouterTestingModule } from '@angular/router/testing';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

Al igual que antes, vamos empezar añadiendo el nuevo **it** con el que vamos a comprobar que el botón debería de estar deshabilitado cuando no se rellena el campo del formulario.

```
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { RouterTestingModule } from '@angular/router/testing';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
  })
});
```

Ahora vamos a buscar el botón por CSS, accederemos al elemento nativo y comprobaremos que la propiedad **disabled** es true.

```
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { RouterTestingModule } from '@angular/router/testing';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
    const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
    expect(btn.nativeElement.disabled).toBeTrue()
  })
});
```

Vamos a añadir otro test en el que comprobaremos el caso contrario, es decir que si rellenamos el campo del formulario, el botón aparece habilitado.

```
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { RouterTestingModule } from '@angular/router/testing';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
    const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
    expect(btn.nativeElement.disabled).toBeTrue()
  })

  it('debería de dejar habilitado el botón de enviar si rellenamos el campo de título', () => {
  })
});
```

Ahora lo primero que vamos a hacer es cambiar el valor del input.

```
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { RouterTestingModule } from '@angular/router/testing';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
    const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
    expect(btn.nativeElement.disabled).toBeTrue()
  })

  it('debería de dejar habilitado el botón de enviar si rellenamos el campo de título', () => {
    component.formTarea.get('titulo')?.setValue('Tarea 1')
    fixture.detectChanges()

  })
});
```

Una vez cambiado, vamos a buscar de nuevo el botón y comprobaremos que esta vez es false la propiedad disabled.

```

import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { RouterTestingModule } from '@angular/router/testing';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
    const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
    expect(btn.nativeElement.disabled).toBeTrue()
  })

  it('debería de dejar habilitado el botón de enviar si rellenamos el campo de título', () => {
    component.formTarea.get('titulo')?.setValue('Tarea 1')

    fixture.detectChanges()

    let btn = fixture.debugElement.query(By.css('button[type="submit"]'))
    expect(btn.nativeElement.disabled).toBeFalsy()
  })
});

```

Con el siguiente test vamos a comprobar que cuando se llama a `createTarea` del servicio, se le llama con un objeto cuyo título es el que hemos rellenado en el campo, y completada a false.

Para este caso vamos a necesitar mockear estas dependencias, pero esta vez, vamos a hacerlo de otra forma distinta a como lo hemos hecho en el de **ListaTareas**. Esta vez usaremos **spyOn**.

Para ello vamos a empezar por obtener el servicio de tareas y el del router con **TestBed.inject** en el **beforeEach**.

```
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { Router } from '@angular/router';
import { RouterTestingModule } from '@angular/router/testing';
import { TareasService } from '../tareas.service';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;
  let tareasService: TareasService
  let router: Router

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
    component = fixture.componentInstance;
    tareasService = TestBed.inject(TareasService)
    router = TestBed.inject(Router)
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
    const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
    expect(btn.nativeElement.disabled).toBeTrue()
  })

  it('debería de dejar habilitado el botón de enviar si rellenamos el campo de título', () => {
    component.formTarea.get('titulo')?.setValue('Tarea 1')

    fixture.detectChanges()

    let btn = fixture.debugElement.query(By.css('button[type="submit"]'))
    expect(btn.nativeElement.disabled).toBeFalsy()
  })
});
```

Ahora vamos a añadir el nuevo it, y esta vez, vamos a hacer que devuelva un observable de cualquier cosa para que no falle el subscribe. El valor a devolver nos da igual.

/angular-testing-unitarios-mocks-lab/src/app/nueva-tarea/nueva-tarea.component.spec.ts

```
import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { Router } from '@angular/router';
import { RouterTestingModule } from '@angular/router/testing';
import { of } from 'rxjs';
import { TareasService } from '../tareas.service';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;
  let tareasService: TareasService
  let router: Router

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
    component = fixture.componentInstance;
    tareasService = TestBed.inject(TareasService)
    router = TestBed.inject(Router)
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
    const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
    expect(btn.nativeElement.disabled).toBeTrue()
  })

  it('debería de dejar habilitado el botón de enviar si rellenamos el campo de título', () => {
    component.formTarea.get('titulo')?.setValue('Tarea 1')

    fixture.detectChanges()

    let btn = fixture.debugElement.query(By.css('button[type="submit"]'))
  })
})
```

```

    expect(btn.nativeElement.disabled).toBeFalsy()
  })

it('debería de llamar a createTarea con el título escrito y completada a false', () => {
  spyOn(tareasService, 'createTarea').and.returnValue(of([]))

})
});

```

El siguiente paso es setear el valor del formulario y detectar los cambios.

/angular-testing-unitarios-mocks-lab/src/app/nueva-tarea/nueva-tarea.component.spec.ts

```

import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { Router } from '@angular/router';
import { RouterTestingModule } from '@angular/router/testing';
import { of } from 'rxjs';
import { TareasService } from '../tareas.service';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;
  let tareasService: TareasService
  let router: Router

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
    component = fixture.componentInstance;
    tareasService = TestBed.inject(TareasService)
    router = TestBed.inject(Router)
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
    const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
    expect(btn.nativeElement.disabled).toBeTruthy()
  })
});

```

```

})
it('debería de dejar habilitado el botón de enviar si rellenamos el campo de título', () => {
  component.formTarea.get('titulo')?.setValue('Tarea 1')

  fixture.detectChanges()

  let btn = fixture.debugElement.query(By.css('button[type="submit"]'))
  expect(btn.nativeElement.disabled).toBeFalsy()
})

it('debería de llamar a createTarea con el título escrito y completada a false', () => {
  spyOn(tareasService, 'createTarea').and.returnValue(of({}))

  component.formTarea.get('titulo')?.setValue('Tarea 1')
  fixture.detectChanges()

})
});

}
);

```

Ahora llamamos a la función **guardar** del componente para que se llame **createTarea** que hemos mockeado.

/angular-testing-unitarios-mocks-lab/src/app/nueva-tarea/nueva-tarea.component.spec.ts

```

import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { Router } from '@angular/router';
import { RouterTestingModule } from '@angular/router/testing';
import { of } from 'rxjs';
import { TareasService } from '../tareas.service';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;
  let tareasService: TareasService
  let router: Router

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
  });

```

```

component = fixture.componentInstance;
tareasService = TestBed.inject(TareasService)
router = TestBed.inject(Router)
fixture.detectChanges();
});

it('should create', () => {
  expect(component).toBeTruthy();
});

it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
  const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
  expect(btn.nativeElement.disabled).toBeTrue()
})

it('debería de dejar habilitado el botón de enviar si rellenamos el campo de título', () => {
  component.formTarea.get('titulo')?.setValue('Tarea 1')

  fixture.detectChanges()

  let btn = fixture.debugElement.query(By.css('button[type="submit"]'))
  expect(btn.nativeElement.disabled).toBeFalsy()
})

it('debería de llamar a createTarea con el titulo escrito y completada a false', () => {
  spyOn(tareasService, 'createTarea').and.returnValue(of({}))

  component.formTarea.get('titulo')?.setValue('Tarea 1')
  fixture.detectChanges()

  component.guardar()
})
});

```

Y por último vamos a comprobar que la función de **createTarea** se ha llamado una vez con el objeto correcto.

/angular-testing-unitarios-mocks-lab/src/app/nueva-tarea/nueva-tarea.component.spec.ts

```

import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { Router } from '@angular/router';
import { RouterTestingModule } from '@angular/router/testing';
import { of } from 'rxjs';
import { TareasService } from '../tareas.service';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;
  let tareasService: TareasService
  let router: Router

  beforeEach(async () => {

```

```

await TestBed.configureTestingModule({
  declarations: [ NuevaTareaComponent ],
  imports: [
    HttpClientTestingModule,
    RouterTestingModule,
    ReactiveFormsModule,
  ],
})
.compileComponents();
});

beforeEach(() => {
  fixture = TestBed.createComponent(NuevaTareaComponent);
  component = fixture.componentInstance;
  tareasService = TestBed.inject(TareasService)
  router = TestBed.inject(Router)
  fixture.detectChanges();
});

it('should create', () => {
  expect(component).toBeTruthy();
});

it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
  const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
  expect(btn.nativeElement.disabled).toBeTrue()
})

it('debería de dejar habilitado el botón de enviar si rellenamos el campo de título', () => {
  component.formTarea.get('titulo')?.setValue('Tarea 1')

  fixture.detectChanges()

  let btn = fixture.debugElement.query(By.css('button[type="submit"]'))
  expect(btn.nativeElement.disabled).toBeFalsy()
})

it('debería de llamar a createTarea con el titulo escrito y completada a false', () => {
  spyOn(tareasService, 'createTarea').and.returnValue(of({}))

  component.formTarea.get('titulo')?.setValue('Tarea 1')
  fixture.detectChanges()

  component.guardar()

  expect(tareasService.createTarea).toHaveBeenCalledWith({titulo: 'Tarea 1', completada: false})
})
});

```

Con esto debería de seguir pasando los tests.

Un último test sería comprobar que después de guarda la tarea, se navega a la página de inicio. Por lo que vamos a crear el it, con el mismo spyOn del test anterior y llamaremos al **guardar**, al igual que antes.

/angular-testing-unitarios-mocks-lab/src/app/nueva-tarea/nueva-tarea.component.spec.ts

```

import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { Router } from '@angular/router';
import { RouterTestingModule } from '@angular/router/testing';
import { of } from 'rxjs';
import { TareasService } from '../tareas.service';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;
  let tareasService: TareasService
  let router: Router

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
    component = fixture.componentInstance;
    tareasService = TestBed.inject(TareasService)
    router = TestBed.inject(Router)
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
    const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
    expect(btn.nativeElement.disabled).toBeTrue()
  })

  it('debería de dejar habilitado el botón de enviar si rellenamos el campo de título', () => {
    component.formTarea.get('titulo')?.setValue('Tarea 1')

    fixture.detectChanges()

    let btn = fixture.debugElement.query(By.css('button[type="submit"]'))
    expect(btn.nativeElement.disabled).toBeFalsy()
  })

  it('debería de llamar a createTarea con el titulo escrito y completada a false', () => {
    spyOn(tareasService, 'createTarea').and.returnValue(of({}))
  })
})

```

```

    component.formTarea.get('titulo')?.setValue('Tarea 1')
    fixture.detectChanges()

    component.guardar()

    expect(tareasService.createTarea).toHaveBeenCalledWith({titulo: 'Tarea 1', completada: false})
  })
});

it('debería de cambiar a / si se guarda la tarea', () => {
  spyOn(tareasService, 'createTarea').and.returnValue(of({}))

  component.guardar()
})

```

Esta vez queremos comprobar algo sobre el router, por lo que vamos a llamar a **spyOn**, sobre el **router** y el método **navigate**, y esta vez como no devuelve nada y solo queremos saber con que parámetros se está llamando al método, no hace falta que pongamos que retorne ningún valor.

/angular-testing-unitarios-mocks-lab/src/app/nueva-tarea/nueva-tarea.component.spec.ts

```

import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { Router } from '@angular/router';
import { RouterTestingModule } from '@angular/router/testing';
import { of } from 'rxjs';
import { TareasService } from '../tareas.service';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;
  let tareasService: TareasService
  let router: Router

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ NuevaTareaComponent ],
      imports: [
        HttpClientTestingModule,
        RouterTestingModule,
        ReactiveFormsModule,
      ],
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(NuevaTareaComponent);
    component = fixture.componentInstance;
    tareasService = TestBed.inject(TareasService)
    router = TestBed.inject(Router)
    fixture.detectChanges();
  });

```

```

it('should create', () => {
  expect(component).toBeTruthy();
});

it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
  const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
  expect(btn.nativeElement.disabled).toBeTrue()
})

it('debería de dejar habilitado el botón de enviar si rellenamos el campo de título', () => {
  component.formTarea.get('titulo')?.setValue('Tarea 1')

  fixture.detectChanges()

  let btn = fixture.debugElement.query(By.css('button[type="submit"]'))
  expect(btn.nativeElement.disabled).toBeFalsy()
})

it('debería de llamar a createTarea con el titulo escrito y completada a false', () => {
  spyOn(tareasService, 'createTarea').and.returnValue(of({}))

  component.formTarea.get('titulo')?.setValue('Tarea 1')
  fixture.detectChanges()

  component.guardar()

  expect(tareasService.createTarea).toHaveBeenCalledWith({titulo: 'Tarea 1', completada: false})
})

it('debería de cambiar a / si se guarda la tarea', () => {
  spyOn(tareasService, 'createTarea').and.returnValue(of({}))
  spyOn(router, 'navigate')

  component.guardar()
})

```

Y por último, añadimos el **expect** en el que comprobaremos que se ha llamado una vez con el parámetro `['/']`.

/angular-testing-unitarios-mocks-lab/src/app/nueva-tarea/nueva-tarea.component.spec.ts

```

import { HttpClientTestingModule } from '@angular/common/http/testing';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { By } from '@angular/platform-browser';
import { Router } from '@angular/router';
import { RouterTestingModule } from '@angular/router/testing';
import { of } from 'rxjs';
import { TareasService } from '../tareas.service';

import { NuevaTareaComponent } from './nueva-tarea.component';

fdescribe('NuevaTareaComponent', () => {
  let component: NuevaTareaComponent;
  let fixture: ComponentFixture<NuevaTareaComponent>;

```

```

let tareasService: TareasService
let router: Router

beforeEach(async () => {
  await TestBed.configureTestingModule({
    declarations: [ NuevaTareaComponent ],
    imports: [
      HttpClientTestingModule,
      RouterTestingModule,
      ReactiveFormsModule,
    ],
  })
  .compileComponents();
});

beforeEach(() => {
  fixture = TestBed.createComponent(NuevaTareaComponent);
  component = fixture.componentInstance;
  tareasService = TestBed.inject(TareasService)
  router = TestBed.inject(Router)
  fixture.detectChanges();
});

it('should create', () => {
  expect(component).toBeTruthy();
});

it('debería de dejar deshabilitado el botón de enviar si no rellenamos el campo de título', () => {
  const btn = fixture.debugElement.query(By.css('button[type="submit"]'))
  expect(btn.nativeElement.disabled).toBeTrue()
})

it('debería de dejar habilitado el botón de enviar si rellenamos el campo de título', () => {
  component.formTarea.get('titulo')?.setValue('Tarea 1')

  fixture.detectChanges()

  let btn = fixture.debugElement.query(By.css('button[type="submit"]'))
  expect(btn.nativeElement.disabled).toBeFalsy()
})

it('debería de llamar a createTarea con el titulo escrito y completada a false', () => {
  spyOn(tareasService, 'createTarea').and.returnValue(of({}))

  component.formTarea.get('titulo')?.setValue('Tarea 1')
  fixture.detectChanges()

  component.guardar()

  expect(tareasService.createTarea).toHaveBeenCalledWith({titulo: 'Tarea 1', completada: false})
})

it('debería de cambiar a / si se guarda la tarea', () => {
  spyOn(tareasService, 'createTarea').and.returnValue(of({}))
  spyOn(router, 'navigate')

  component.guardar()
})

```

```
    expect(router.navigate).toHaveBeenCalledWith(['/'])  
})
```

Y ya tenemos otro test más que se está pasando.

Chapter 31. Testing E2E con Cypress

Cypress es una herramienta que nos permite testear aplicaciones web modernas y que se ha pensado para solucionar los típicos problemas que los desarrolladores y personas de testing se encuentran en este tipo de aplicaciones, como por ejemplo el manejo de las esperas. También funciona perfectamente con aplicaciones que no sean SPAs o que no se hayan desarrollado con las librerías y frameworks más modernos.

Cypress está construido sobre Node por lo que solo podremos utilizar JavaScript como lenguaje para escribir nuestros tests.

Las principales características de Cypress son:

- **Viaje en el tiempo:** según se van ejecutando los tests, Cypress saca snapshots de la aplicación antes y después de las acciones para que podamos ir viendo que se ha ido haciendo en cada paso del test, pudiendo ver la ejecución de este paso a paso.
- **Debug:** nos permite debuggear los tests directamente desde el test runner y las herramientas del desarrollador. Muchos de los errores que nos muestra son de gran ayuda y se entienden fácilmente.
- **Esperas automáticas:** con Cypress no hace falta que pongamos instrucciones de espera para hacer que la ejecución del test se pare para dar tiempo a que los elementos con los que queremos interactuar se puedan encontrar.
- **Screenshots y videos:** cuando ejecutamos los tests con el CLI, se graba un video con la ejecución del test y se sacan pantallazos cada vez que falla algún test.
- **Resultados consistentes:** como los tests se ejecutan desde el mismo navegador, se consigue que los resultados sean mucho más fiables ya que no depende de que algún intermediario le diga al navegador como tiene que ejecutar algunas instrucciones.

Algunas de las desventajas que podemos encontrarnos al usar esta herramienta son:

- Los tests **se ejecutan directamente dentro del navegador**, por lo que no tiene soporte (por el momento) para utilizar **múltiples pestañas** o los **popups** del navegador, es decir, no podemos testear funcionalidades nativas de los navegadores.
- Tampoco tiene soporte para el **Shadow DOM**.
- Antes de la versión 4 de Cypress solo soportaba **Chrome** y **Electron**. Actualmente soporta algunos navegadores más.
- No se puede visitar distintos dominios en el mismo test.

31.1. Instalación

Para instalar Cypress necesitamos tener un proyecto de Angular ya creado, y dentro de este lanzar el siguiente comando:

```
$ ng add @cypress/schematic
```

Una vez lanzado este comando y contestadas las preguntas que se nos hacen:

- Se instalan las dependencias necesarias: cypress.
- Se añaden un par de scripts en el **package.json**: cypress:open y cypress:run.
- Se crea la carpeta **cypress** con las carpetas y archivos donde crearemos nuestros tests.
- Se crea el archivo de configuración de cypress: cypress.json.
- Se modifica la configuración del **angular.json** para poder lanzar los comandos de cypress con el **ng**.

31.2. Ejecución de tests

Para ejecutar los tests solo hay que abrir la interfaz de Cypress lanzando alguno de los siguientes comandos:

```
$ ng e2e  
$ npm run cypress:open
```

Una vez abierta, pulsar sobre el test que queremos lanzar.

También podemos lanzar los tests sin necesidad de abrir la interfaz de Cypress con el siguiente comando:

```
$ npm run cypress:run
```

31.3. Búsqueda de elementos

Para poder testear la aplicación y poder simular todas las acciones que realizaría un usuario mientras está en el navegador, es necesario poder acceder a los distintos elementos de las páginas.

Puede que a veces sea complicado obtener algún elemento, pero siempre existe alguna forma para conseguirlo.

A continuación veremos los distintos métodos que se pueden usar para obtener los elementos de la aplicación web.

Para acceder a los elementos de la aplicación web usaremos la variable **cy** que nos proporciona Cypress.

Desde la variable **cy** tenemos acceso a bastantes métodos, pero ahora vamos a centrarnos en aquellos que nos permiten buscar los distintos elementos web con los que queremos interactuar:

- **get**: recibe como parámetro el selector con el que apuntamos a los elementos que queremos obtener. Devuelve un elemento o varios.
- **find**: es como el get, con la diferencia de que este no se puede utilizar desde el objeto **cy**.
- **contains**: recibe como primer parámetro un selector y como segundo un texto completo o parcial, para devolvernos el primer elemento que encuentre que se acceda con el selector y contenga el texto pasado.

```
cy.get('#mensaje-error');

cy.get('#lista-enlaces')
  .find('.active')

cy.contains('a', 'Google');
```

31.4. Lab: Búsqueda de elementos

En este laboratorio vamos a ver como utilizar los métodos de Cypress para localizar elementos web de la aplicación que estamos testeando.

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

```
$ ng new angular-testing-e2e-cypress-busquedas-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 vamos a añadir Cypress en el proyecto:

```
$ cd angular-testing-e2e-cypress-busquedas-lab  
$ ng add @cypress/schematic  
  
? The package @cypress/schematic@1.6.0 will be installed and executed.  
Would you like to proceed? Yes  
? Would you like the default 'ng e2e' command to use Cypress? [ Protractor to Cypress Migration Guide:  
https://on.cypress.io/protractor-to-cypress?cli=true ] Yes
```

Una vez añadido Cypress, vamos a levantar el servidor de desarrollo.

```
$ ng s
```

Vamos a añadir dos listados en nuestro componente **App**, que utilizaremos para ver como buscar algunos elementos en nuestros tests.

/angular-testing-e2e-cypress-busquedas-lab/src/app/app.component.html

```
<h1 id="titulo">Listados</h1>  
  
<ul id="listaItems">  
  <li class="li1">Item 1</li>  
  <li class="li2">Item 2</li>  
  <li class="li3">Item 3</li>  
</ul>  
  
<ul id="listaCosas">  
  <li class="li1">Cosa 1</li>  
  <li class="li2">Cosa 2</li>  
  <li class="li3">Cosa 3</li>  
</ul>
```

Ahora nos vamos a ir a nuestro test, y vamos a cambiar los títulos de las funciones **describe** e **it**, además de poner la navegación inicial en el hook **beforeEach**.

/angular-testing-e2e-cypress-busquedas-lab/cypress/integration/spec.ts

```
describe('Búsquedas', () => {
  beforeEach(() => {
    cy.visit('/')
  })

  it('', () => {

  })
})
```

La primera búsqueda consiste en encontrar el título y comprobar que existe y tiene el texto correcto. Para ello lo buscaremos por su identificador y usaremos varias aserciones por lo que las vamos a encadenar con la función **and**.

/angular-testing-e2e-cypress-busquedas-lab/cypress/integration/spec.ts

```
describe('Búsquedas', () => {
  beforeEach(() => {
    cy.visit('/')
  })

  it('deberías encontrar el título Listados', () => {
    cy.get('#titulo')
      .should('exist')
      .and('have.text', 'Listados')
  })
})
```

Ya tenemos nuestro primer test, vamos a lanzar los test de cypress con el siguiente comando:

```
$ ng e2e
```

Al abrirse el panel de Cypress, pulsamos sobre el archivo **spec.ts**, y se empezará a ejecutar. Debería de pasarse correctamente este primer test.

Ahora vamos a buscar el elemento **Item 2** que se encuentra en la lista de cosas. Para este caso vamos a utilizar primero el **get** para acceder a la lista de cosas, y después sobre la lista utilizaremos el **find** para buscar el que tiene la clase **li2**.

```
describe('Búsquedas', () => {
  beforeEach(() => {
    cy.visit('/')
  })

  it('deberías encontrar el título Listados', () => {
    cy.get('#titulo')
      .should('exist')
      .and('have.text', 'Listados')
  })

  it('deberías encontrar el elemento 2 del segundo listado', () => {
    cy.get('#listaCosas')
      .find('.li2')
      .should('have.text', 'Cosa 2')
  })
})
```

La última búsqueda consiste en encontrar el elemento **Item 3**, pero esta vez utilizando el método **contains**. Podemos hacer esta búsqueda de bastantes formas. Vamos a buscar primero con el **get** la lista de items, y después buscaremos el elemento **li** que contiene el texto buscado.

```
describe('Búsquedas', () => {
  beforeEach(() => {
    cy.visit('/')
  })

  it('deberías encontrar el título Listados', () => {
    cy.get('#titulo')
      .should('exist')
      .and('have.text', 'Listados')
  })

  it('deberías encontrar el elemento 2 del segundo listado', () => {
    cy.get('#listaCosas')
      .find('.li2')
      .should('have.text', 'Cosa 2')
  })

  it('deberías encontrar el elemento Item 3 (sin get y sin find)', () => {
    cy.get('#listaItems')
      .contains('li', 'Item 3')
      .should('have.text', 'Item 3');
  })
})
```

Otra forma de haberlo hecho hubiese sido buscando directamente el elemento que contiene el texto **Item 3** con el **contains**.

```
describe('Búsquedas', () => {
  beforeEach(() => {
    cy.visit('/')
  })

  it('deberías encontrar el título Listados', () => {
    cy.get('#titulo')
      .should('exist')
      .and('have.text', 'Listados')
  })

  it('deberías encontrar el elemento 2 del segundo listado', () => {
    cy.get('#listaCosas')
      .find('.li2')
      .should('have.text', 'Cosa 2')
  })

  it('deberías encontrar el elemento Item 3 (sin get y sin find)', () => {
    // Busca el Item 3 dentro del #listaItems
    cy.get('#listaItems')
      .contains('li', 'Item 3')
      .should('have.text', 'Item 3');

    // Busca el Item 3
    cy.contains('Item 3')
      .should('have.text', 'Item 3');
  })
})
```

Y con esto ya tendríamos una idea de como utilizar estos 3 métodos de búsqueda de elementos que iremos usando durante el resto del curso.

31.5. Navegar por el DOM

Cuando queremos testear listas, tablas o elementos web que contienen varias etiquetas del mismo tipo repetidas en su interior, nos encontramos con que todos estos elementos tienen las mismas clases o atributos y puede ser un poco complicado acceder directamente a ellos utilizando selectores como el de los identificadores (a cada elemento de una lista no se le suele poner un identificador único).

En estos casos tendremos que empezar a utilizar comandos que se encargan de filtrar, recorrer listas de elementos, acceder a los elementos hijos... que nos facilitarán nuestra tarea.

Algunos de estos elementos son:

- **children**: devuelve los elementos hijos del elemento sobre el que se aplica este método.
- **parent**: devuelve el elemento padre del elemento sobre el que se aplica este método.
- **first**: devuelve el primer elemento de una lista de elementos.
- **last**: devuelve el último elemento de una lista de elementos.
- **eq**: devuelve un elemento dada su posición dentro de una lista de elementos.
- **each**: itera sobre un array de elementos y nos los va pasando de uno en uno (como elementos de jQuery) en la función de callback que recibe como parámetro.
- **filter**: permite filtrar una lista de elementos devolviéndonos aquellos que cumplen con el selector que se le pasa como parámetro.
- ...

31.6. Lab: Navegar por el DOM

En este laboratorio vamos a ver como utilizar algunos de los métodos que nos permiten recorrer el DOM y buscar elementos de la aplicación para realizar una serie de comprobaciones sobre una tabla de datos.

Empresa	Ciudad	Pelicula/Serie
Stark Industries	New York	Ironman
Wayne Enterprises	Gotham	Batman
Pied Piper	Silicon Valley	Silicon Valley
El banco de hierro	Braavos	Game of thrones
Evil Corp	New York	Mr. Robot
Galley-La Company	Water 7	One Piece

A continuación nos encontramos con los objetivos:

- Comprueba que la tabla existe
- Comprueba que tiene el número de filas correcto
- Comprueba que la última fila tiene el número de celdas correcto
- Comprueba que después de la quinta fila, hay dos filas más
- Comprueba que todas las celdas tienen contenido

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

```
$ ng new angular-testing-e2e-cypress-navegar-dom-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 vamos a añadir Cypress en el proyecto:

```
$ cd angular-testing-e2e-cypress-navegar-dom-lab  
$ ng add @cypress/schematic  
  
? The package @cypress/schematic@1.6.0 will be installed and executed.  
Would you like to proceed? Yes  
? Would you like the default 'ng e2e' command to use Cypress? [ Protractor to Cypress Migration Guide:  
https://on.cypress.io/protractor-to-cypress?cli=true ] Yes
```

Una vez añadido Cypress, vamos a levantar el servidor de desarrollo.

```
$ ng s
```

Vamos a definir los datos a pintar en la tabla en un array dentro del **app.component.ts**.

/angular-testing-e2e-cypress-navegar-dom-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 {
  datosTabla: Array<any> = [
    { empresa: 'Stark Industries', ciudad: 'New York', pelicula: 'Ironman' },
    { empresa: 'Wayne Enterprises', ciudad: 'Gotham', pelicula: 'Batman' },
    { empresa: 'Pied Piper', ciudad: 'Silicon Valley', pelicula: 'Silicon Valley' },
    { empresa: 'El banco de hierro', ciudad: 'Braavos', pelicula: 'Game of thrones' },
    { empresa: 'Evil Corp', ciudad: 'New York', pelicula: 'Mr. Robot' },
    { empresa: 'Galley-La Company', ciudad: 'Water 7', pelicula: 'One Piece' }
  ]
}
```

Ahora vamos a pintar estos datos en la plantilla del componente.

/angular-testing-e2e-cypress-navegar-dom-lab/src/app/app.component.html

```
<table id="tabla-empresas">
  <thead>
    <tr>
      <th>Empresa</th>
      <th>Ciudad</th>
      <th>Pelicula/Serie</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let fila of datosTabla">
      <td>{{fila.empresa}}</td>
      <td>{{fila.ciudad}}</td>
      <td>{{fila.pelicula}}</td>
    </tr>
  </tbody>
</table>
```

Ya tenemos la tabla, por lo que vamos a empezar a hacer los tests.

/angular-testing-e2e-cypress-navegar-dom-lab/cypress/integration/spec.ts

```
describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('/')

  })
})
```

Ahora vamos a buscar la tabla para comprobar que existe.

/angular-testing-e2e-cypress-navegar-dom-lab/cypress/integration/spec.ts

```
describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('/')

    cy.get('#tabla-empresas')
      .should('exist');

  })
})
```

Ahora vamos a buscar las filas que hay dentro de la tabla para comprobar que tienen el número correcto de filas. Primero pedimos la tabla, y después usamos el método **find** para buscar las filas que hay dentro de la tabla.

/angular-testing-e2e-cypress-navegar-dom-lab/cypress/integration/spec.ts

```
describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('/')

    cy.get('#tabla-empresas')
      .should('exist');

    cy.get('#tabla-empresas')
      .find('tr')
      .should('have.length', 7);

  })
})
```

La siguiente comprobación nos dice que miremos que la última fila tiene 3 celdas. Para este caso, vamos a buscar todas las filas, y vamos a utilizar el método **last** para acceder a la última. Dentro de este último **tr** vamos a buscar los elementos **td** que hay dentro usando el método **find** para ello.

```
describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('/')

    cy.get('#tabla-empresas')
      .should('exist');

    cy.get('#tabla-empresas')
      .find('tr')
      .should('have.length', 7);

    cy.get('#tabla-empresas tr')
      .last()
      .find('td')
      .should('have.length', 3);

  })
})
```

La siguiente comprobación se complica un poco, porque esta vez vamos a necesitar el método **each** para recorrer todas las filas de la tabla y así ir incrementando una variable por cada fila cuya posición en el array de filas sea superior a 4.

Después de terminar de recorrer el array de filas, vamos a añadir la aserción utilizando el método **then** y dentro el **expect** de Chai.

```
describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('/')

    cy.get('#tabla-empresas')
      .should('exist');

    cy.get('#tabla-empresas')
      .find('tr')
      .should('have.length', 7);

    cy.get('#tabla-empresas tr')
      .last()
      .find('td')
      .should('have.length', 3);

    let numFilas = 0;
    cy.get('#tabla-empresas tr')
      .each(($tr, index) => {
        if (index > 4) {
          numFilas += 1;
        }
      })
      .then(() => {
        expect(numFilas).to.be.equal(2);
      })
  })
})
```

Por último, la aserción que nos queda nos dice que comprobemos que todas las celdas tienen texto, por lo que vamos a volver a utilizar el método **each** pero esta vez para acceder a los elementos de jQuery, obtener el texto que contienen, y comprobar que no está vacío.

```
describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('/')

    cy.get('#tabla-empresas')
      .should('exist');

    cy.get('#tabla-empresas')
      .find('tr')
      .should('have.length', 7);

    cy.get('#tabla-empresas tr')
      .last()
      .find('td')
      .should('have.length', 3);

    let numFilas = 0;
    cy.get('#tabla-empresas tr')
      .each(($tr, index) => {
        if (index > 4) {
          numFilas += 1;
        }
      })
      .then(() => {
        expect(numFilas).to.be.equal(2);
      })

    cy.get('#tabla-empresas td')
      .each($td => {
        const textoCelda = $td.text();
        expect(textoCelda).not.to.be.empty;
      })
  })
})
```

Con esto ya tenemos todas las aserciones anteriores completadas, podemos probar a cambiar las condiciones que se comprueban para ver que fallan los tests y no tenemos falsos positivos.

Ahora vamos a mejorar un poco las aserciones anteriores y vamos a ver como podríamos encadenar unas cuantas de ellas para evitar tener que buscar varias veces los mismos elementos.

Vamos a crear otro bloque `it`, en el que encadenaremos las tres primeras aserciones.

```
describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('/')

    cy.get('#tabla-empresas')
      .should('exist');

    cy.get('#tabla-empresas')
      .find('tr')
      .should('have.length', 7);

    cy.get('#tabla-empresas tr')
      .last()
      .find('td')
      .should('have.length', 3);

    let numFilas = 0;
    cy.get('#tabla-empresas tr')
      .each($tr, index) => {
        if (index > 4) {
          numFilas += 1;
        }
      })
      .then(() => {
        expect(numFilas).to.be.equal(2);
      })

    cy.get('#tabla-empresas td')
      .each($td => {
        const textoCelda = $td.text();
        expect(textoCelda).not.to.be.empty;
      })
  })

  it('debería poder unir las tres primeras comprobaciones en una sola', () => {
    cy.visit('/')

    cy.get('#tabla-empresas')
      .should('exist')
      .find('tr')
      .should('have.length', 7)
      .last()
      .find('td')
      .should('have.length', 3);
  })
})
```

Hemos podido encadenarlas porque los comandos de Cypress van retornando el objeto buscado o

sobre el que se realizan acciones hacia los comandos siguientes.

También podemos encadenar las dos últimas aserciones, así que vamos a crear otro bloque it para ver como quedarían.

/angular-testing-e2e-cypress-navegar-dom-lab/cypress/integration/spec.ts

```
describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('/')

    cy.get('#tabla-empresas')
      .should('exist');

    cy.get('#tabla-empresas')
      .find('tr')
      .should('have.length', 7);

    cy.get('#tabla-empresas tr')
      .last()
      .find('td')
      .should('have.length', 3);

    let numFilas = 0;
    cy.get('#tabla-empresas tr')
      .each(($tr, index) => {
        if (index > 4) {
          numFilas += 1;
        }
      })
      .then(() => {
        expect(numFilas).to.be.equal(2);
      })

    cy.get('#tabla-empresas td')
      .each($td => {
        const textoCelda = $td.text();
        expect(textoCelda).not.to.be.empty;
      })
  })

  it('debería poder unir las tres primeras comprobaciones en una sola', () => {
    cy.visit('/')

    cy.get('#tabla-empresas')
      .should('exist')
      .find('tr')
      .should('have.length', 7)
      .last()
      .find('td')
      .should('have.length', 3);
  })
})
```

```
})

it('debería poder unir las dos últimas comprobaciones en una sola', () => {
  cy.visit('/')

  let numFilas = 0;
  cy.get('#tabla-empresas tr')
    .each(($tr, index) => {
      if (index > 4) {
        numFilas += 1;
      }
    })
    .then(() => {
      expect(numFilas).to.be.equal(2);
    })
    .find('td')
    .each($td => {
      const textoCelda = $td.text();
      expect(textoCelda).not.to.be.empty;
    })
  })

})
```

31.7. Retry ability

Cypress viene con la funcionalidad de que el va a reintentar hacer ciertas acciones por un máximo de 4 segundos (es el valor por defecto). Si en estos 4 segundos no ha conseguido realizar la acción que necesitaba entonces el test va a fallar.

Esta funcionalidad está muy bien porque muchas veces en las aplicaciones algunos elementos con los que necesitamos interactuar no se muestran en el mismo momento y tardan un poco en aparecer por distintos motivos como que estos elementos pinten datos que vienen de una petición a una API, y esta API tarde algo más de lo necesario en enviárnoslos.

Entonces, cuando buscamos un elemento, si este se encuentra pues se sigue con la ejecución del test, pero si no es así, intenta otra vez a buscar el elemento, y así varias veces hasta que se acaba el tiempo de reintento que es cuando falla el test.

Este tiempo de reintento se puede cambiar pasándole, a los comandos, la propiedad **timeout** con el nuevo valor en milisegundos dentro del objeto de opciones.

Otra forma de cambiarlo, y que afectaría a todos los tests es definirlo en el archivo de **cypress.json** con la propiedad **defaultCommandTimeout** y asignándole el nuevo tiempo en milisegundos como antes.

Esta funcionalidad solo se aplica a comandos de Cypress que se encargan de buscar elementos, es decir, **get**, **contains**, **find**... Esto se debe a que si reintenta un comando de interacción como un **type** o un **click**, puede afectar al comportamiento de la aplicación y tener efectos colaterales sobre el test.

31.8. Lab: Retry ability

En este laboratorio vamos a ver como Cypress es capaz de esperar un tiempo hasta que puede interactuar con el elemento buscado, además de como modificar este tiempo de espera.

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

```
$ ng new angular-testing-e2e-cypress-retry-ability-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 vamos a añadir Cypress en el proyecto:

```
$ cd angular-testing-e2e-cypress-retry-ability-lab  
$ ng add @cypress/schematic  
  
? The package @cypress/schematic@1.6.0 will be installed and executed.  
Would you like to proceed? Yes  
? Would you like the default 'ng e2e' command to use Cypress? [ Protractor to Cypress Migration Guide:  
https://on.cypress.io/protractor-to-cypress?cli=true ] Yes
```

Una vez añadido Cypress, vamos a levantar el servidor de desarrollo.

```
$ ng s
```

Dentro de nuestro archivo **app.component.html** vamos a poner dos botones que inicialmente no se van a mostrar.

/angular-testing-e2e-cypress-retry-ability-lab/src/app/app.component.html

```
<button type="button" id="btn-lazy-3500" *ngIf="mostrarBtn1">Soy un botón perezoso</button>  
<button type="button" id="btn-lazy-5500" *ngIf="mostrarBtn2">Soy un botón más perezoso todavía</button>
```

Ahora en el archivo de TypeScript vamos a declarar las dos propiedades que hemos puesto en la directiva **ngIf**.

/angular-testing-e2e-cypress-retry-ability-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 {
  mostrarBtn1: boolean = false
  mostrarBtn2: boolean = false
}
```

Además, vamos a hacer que estas dos propiedades cambien su valor a **true**, para el primer botón cuando hayan pasado 3.5 segundos y para el segundo cuando hayan pasado 5.5 segundos.

/angular-testing-e2e-cypress-retry-ability-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 {
  mostrarBtn1: boolean = false
  mostrarBtn2: boolean = false

  ngOnInit() {
    setTimeout(() => {
      this.mostrarBtn1 = true
    }, 3500)

    setTimeout(() => {
      this.mostrarBtn2 = true
    }, 5500)
  }
}
```

Una vez tenemos el componente funcionando, vamos a crear nuestros tests.

Dentro del archivo **cypress/integration/spec.ts** vamos a cambiar los títulos del **describe** e **it** para poner nuestro primer test con el que queremos buscar el primer botón y comprobar que el texto que muestra es el correcto.

```
/angular-testing-e2e-cypress-retry-ability-lab/cypress/integration/spec.ts
```

```
describe('Retry ability', () => {
  it('debería encontrar el primer botón aunque tarde 4 seg en aparecer', () => {
    })
  })
```

El primer paso es navegar a la página de la aplicación, y como vamos a crear otros dos tests más, vamos a poner la instrucción **visit** en el hook **beforeEach**.

```
/angular-testing-e2e-cypress-retry-ability-lab/cypress/integration/spec.ts
```

```
describe('Retry ability', () => {
  beforeEach(() => {
    cy.visit('/')
  })

  it('debería encontrar el primer botón aunque tarde 4 seg en aparecer', () => {
    })
  })
```

Ahora dentro del primer **it** vamos a buscar el botón, y comprobaremos que debería de existir, estar visible y contener el texto que le hemos puesto.

```
/angular-testing-e2e-cypress-retry-ability-lab/cypress/integration/spec.ts
```

```
describe('Retry ability', () => {
  beforeEach(() => {
    cy.visit('/')
  })

  it('debería encontrar el primer botón aunque tarde 4 seg en aparecer', () => {
    cy.get('#btn-lazy-3500')
      .should('exist')
      .and('be.visible')
      .and('have.text', 'Soy un botón perezoso')
    })
  })
```

Ya tenemos nuestro primer test, vamos a lanzar los test de cypress con el siguiente comando:

```
$ ng e2e
```

Al abrirse el panel de Cypress, pulsamos sobre el archivo **spec.ts**, y se empezará a ejecutar. Debería de pasarse correctamente este primer test.

El siguiente test que vamos a añadir es para comprobar que con el tiempo por defecto que usa Cypress para reintentar buscar el elemento no consigue encontrar el botón que tarda 5.5 segundos en crearse.

Creamos el nuevo bloque it y en el vamos a indicar que el botón de 5.5 segundos no existe. Para esperar a que pase el tiempo antes de comprobar la condición usaremos la función de **wait** con un tiempo que esté entre 4 segundos y los 5.5 segundos que tarda en aparecer el botón.

/angular-testing-e2e-cypress-retry-ability-lab/cypress/integration/spec.ts

```
describe('Retry ability', () => {
  beforeEach(() => {
    cy.visit('/')
  })

  it('debería encontrar el primer botón aunque tarde 4 seg en aparecer', () => {
    cy.get('#btn-lazy-3500')
      .should('exist')
      .and('be.visible')
      .and('have.text', 'Soy un botón perezoso')
  })

  it('no debería encontrar el segundo botón si tarda más de 4 seg en aparecer', () => {
    cy.wait(4500);

    cy.get('#btn-lazy-5500')
      .should('not.exist');
  })
})
```

Para finalizar, vamos a añadir un último test en el que vamos a cambiar el tiempo de reintento para que sea capaz de encontrar el segundo botón. Para modificar el tiempo de reintento solo tenemos que pasarle un objeto de opciones al get y utilizar la propiedad **timeout** con el nuevo tiempo en milisegundos.

```
describe('Retry ability', () => {
  beforeEach(() => {
    cy.visit('/')
  })

  it('debería encontrar el primer botón aunque tarde 4 seg en aparecer', () => {
    cy.get('#btn-lazy-3500')
      .should('exist')
      .and('be.visible')
      .and('have.text', 'Soy un botón perezoso')
  })

  it('no debería encontrar el segundo botón si tarda más de 4 seg en aparecer', () => {
    cy.wait(4500);

    cy.get('#btn-lazy-5500')
      .should('not.exist');
  })

  it('debería encontrar el botón si tarda más de 4 seg en aparecer (cambiando el timeout)', () => {
    cy.get('#btn-lazy-5500', { timeout: 6000 })
      .and('be.visible')
  })
})
```

Ya tendríamos nuestros tests, y deberían de pasarse correctamente.

31.9. Interacciones con elementos

Ahora vamos a ver algunos de los comandos que nos permiten interactuar con los diferentes elementos de la web como pueden ser las cajas de texto, los botones, los checkboxes, los desplegables...

Las funciones que nos podemos encontrar para interactuar con estos elementos son:

- click: pulsamos sobre un elemento.

```
cy.get('#miBtn')  
  .click();
```

- dblclick: hacemos doble click sobre un elemento.

```
cy.get('#miBtn')  
  .dblclick();
```

- type: escribe lo que se le pasa como parámetro en un campo de texto. También podemos pasarle entre {} el nombre de las teclas sobre las que podemos pulsar. Algunos nombres de las teclas que podemos utilizar son:

enter	backspace	del	esc
downarrow	leftarrow	rightarrow	uparrow
alt	shift	ctrl	cmd

```
cy.get('#inputDNI')  
  .type('00000000T');  
  
cy.get('#inputDNI')  
  .type('00000000T{enter}');
```

- clear: borra el contenido de los campos de texto.

```
cy.get('#inputDNI')  
  .clear()  
  .type('00000000T');
```

- submit: lanza el evento **submit** de los formularios.

```
cy.get('#formulario')  
  .submit();
```

- check: marca un input de tipo radio y checkbox.

```
cy.('[type="checkbox"]')
  .check(['check1', 'check2']);
```

- uncheck: desmarca un input de tipo radio y checkbox.

```
cy.('[type="checkbox"]')
  .uncheck('check4');
```

- select: selecciona una o varias opciones de un desplegable. Se puede pasar el valor de la opción a marcar o el texto de esta.

```
cy.get('#miDesplegable')
  .select('Opción 7');

cy.get('#miDesplegableMultiple')
  .select(['opcion1', 'opcion4']);
```

31.10. Screenshots

Con Cypress podemos hacer capturas de pantalla que luego nos pueden servir para ver cual es el motivo por el que un test está fallando, o para guardar alguna captura como prueba visual de que se muestra en la página.

Para sacar un screenshot con Cypress solo hay que llamar a la función **screenshot** pasándole el nombre con el que queremos guardar la imagen.

```
cy.screenshot('nombre-imagen');
```

Todos los pantallazos se van a guardar en una carpeta **screenshots** automáticamente que se genera sola.

Al sacar un screenshot por defecto se hace de toda la ventana, pero esto podemos cambiarlo llamando al comando **screenshot** desde el elemento del cual queremos obtener la imagen.

```
cy.get('#miPerfil')
  .screenshot('perfil');
```

Para evitar sacar información sensible en los pantallazos, como puede ser un número de cuenta, el dni de una persona, el email... Cypress da la opción de ocultar esta información utilizando la opción de **blackout**. Esta es una propiedad que se pone en un objeto que le pasamos a la instrucción **screenshot** como parámetro. Esta propiedad recibe como valor un array con los selectores que queremos ocultar, y Cypress pondrá una caja de color negro sobre estos elementos.

```
cy.get('#miPerfil')
  .screenshot('perfil', {
    blackout: ['#dni', '#numero-cuenta']
 });
```

Cuando lanzamos los tests con el comando de **cypress run**, por defecto se obtiene un pantallazo cada vez que alguno de los tests falla, para poder ver que se estaba mostrando en el instante del error. Esto no ocurre al ejecutar los tests desde la interfaz de Cypress ya que al ejecutar el test, estamos viendo que se muestra en la página y el posible error.

Esta opción se puede deshabilitar desde la configuración de Cypress (archivo **cypress.json**) añadiendo la opción de **screenshotOnRunFailure: false**.

31.11. Lab: Screenshots

En este laboratorio vamos a ver como podemos realizar capturas de pantalla y como ocultar datos que pueden ser sensibles.

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

```
$ ng new angular-testing-e2e-cypress-screenshots-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 vamos a añadir Cypress en el proyecto:

```
$ cd angular-testing-e2e-cypress-screenshots-lab  
$ ng add @cypress/schematic  
  
? The package @cypress/schematic@1.6.0 will be installed and executed.  
Would you like to proceed? Yes  
? Would you like the default 'ng e2e' command to use Cypress? [ Protractor to Cypress Migration Guide:  
https://on.cypress.io/protractor-to-cypress?cli=true ] Yes
```

Una vez añadido Cypress, vamos a levantar el servidor de desarrollo.

```
$ ng s
```

Dentro de la plantilla del componente App, vamos a poner los datos de una persona con las inversiones que ha realizado en distintas empresas.

/angular-testing-e2e-cypress-screenshots-lab/src/app/app.component.html

```
<div id="dashboard-screenshot">  
  <div>  
    <h3>Datos de Charles Falco</h3>  
    <p>Email: <span id="email">cfalco@gmail.com</span></p>  
    <p>DNI: <span id="dni">00000000T</span></p>  
    <p>Saldo invertido: <span id="saldo">300000€</span></p>  
  </div>  
  <ul id="inversiones">  
    <li>50% - Hooly, Inc</li>  
    <li>30% - Pied Piper, Inc</li>  
    <li>20% - Aviato, Inc</li>  
  </ul>  
</div>
```

Ahora vamos a ir a nuestro test donde vamos a poner un caso de prueba en el que vamos a sacar un pantallazo de la interfaz, sin mostrar el saldo de la cuenta, el dni y el email del usuario.

/angular-testing-e2e-cypress-screenshots-lab/cypress/integration/spec.ts

```
describe('Screenshots', () => {
  it('sacar un pantallazo ocultando los datos sensibles', () => {
    cy.visit('/')
  })
})
```

Para este test queremos solo un pantallazo de un cierto elemento de toda la página web, por tanto vamos a empezar por buscar este elemento por su identificador y vamos a utilizar la función de **screenshot** sobre el.

/angular-testing-e2e-cypress-screenshots-lab/cypress/integration/spec.ts

```
describe('Screenshots', () => {
  it('sacar un pantallazo ocultando los datos sensibles', () => {
    cy.visit('/')

    cy.get('#dashboard-screenshot')
      .screenshot('dashboard');
  })
})
```

El siguiente paso es hacer que no se muestren los datos sensibles, por lo que vamos a pasarle un objeto con la opción de **blackout** a la función de **screenshot**. El blackout espera recibir un array con los selectores de aquellos elementos que no queremos que se muestren en el pantallazo, por lo que le pasamos los selectores de los identificadores del dni, email y saldo.

/angular-testing-e2e-cypress-screenshots-lab/cypress/integration/spec.ts

```
describe('Screenshots', () => {
  it('sacar un pantallazo ocultando los datos sensibles', () => {
    cy.visit('/')

    cy.get('#dashboard-screenshot')
      .screenshot('dashboard', {
        blackout: ['#saldo', '#dni', '#email']
      });
  })
})
```

Al ejecutar el test, veremos que se genera una nueva imagen en la carpeta de **screenshots**. Esta imagen se verá de la siguiente forma:

Datos de Charles Falco

Email: [REDACTED]

DNI: [REDACTED]

Saldo invertido: [REDACTED]

- 50% - Hooly, Inc
- 30% - Pied Piper, Inc
- 20% - Aviato, Inc

31.12. Alias

Los **alias** de Cypress nos van a permitir crear referencias a ciertos elementos con los que queremos interactuar varias veces y así evitar tener que buscarlos en varias ocasiones teniendo código duplicado.

Para crear un alias usamos el comando **as(alias)** justo después de la instrucción que nos da el elemento al cual queremos la referencia.

```
cy.get('#mi-lista').as('lista');
cy.get('#mi-lista').first().as('primer-elemento-de-la-lista');
```

Una vez creados los alias, tenemos que acceder a ellos para poder usarlos. Solo hay que utilizar el comando **get** y pasarle el nombre del alias al que queremos acceder precedido del símbolo de **@**.

```
cy.get('lista'); // Obtenemos una etiqueta con el nombre de lista
cy.get('@lista'); // Obtenemos la referencia a la lista
```

Ahora ya podemos utilizar el resto de comandos que hemos visto hasta ahora para interactuar con estos elementos que nos devuelven los alias.

```
cy.get('@lista')
    .should(...)
```

Algunas veces necesitaremos acceder al propio objeto al que hace referencia un alias, y para ello podemos usar el comando **then** y pasarle una función de callback donde recibir el elemento como parámetro.

```
cy.get('@lista')
    .then($lista => { ... })
```

Los alias también nos permiten acceder a datos o elementos que se mencionan en los hooks que se ejecutan antes de los tests para evitar repetir el mismo código en todos los bloques **it**.

Otra de las cosas que podemos hacer con los alias es hacer que el test espere hasta que aquello a lo que se hace referencia se encuentre disponible, por ejemplo para cuando se hace una petición a una API y esta tarda en respondernos.

```
cy.wait('@peticionGetAPI');
```

31.13. Fixtures

En la mayor parte de los tests vamos a necesitar utilizar una serie de datos, como datos de usuarios para hacer un login o un registro, un listado de objetos para guardarlos en nuestra aplicación, una imagen para subirla al servidor...

Cypress nos da la funcionalidad de los **fixtures** para leer estos datos que necesitaremos de archivos JSON, o directamente los propios archivos (imagen, video...) a utilizar y así no tener que tenerlos hardcodeados por los diferentes tests pudiéndolos reutilizar en cualquier momento.

Estos archivos se guardan dentro de la carpeta **cypress/fixtures**.

Para leer estos datos dentro de un test utilizamos la función **fixture** y le pasamos el nombre del archivo que queremos cargar como primer parámetro. Podemos pasarle como segundo parámetro el tipo de encoding a utilizar al leer los archivos.

Los tipos de encoding que soporta Cypress son: ascii, base64, binary, hex, utf8, utf16le, ucs2 y latin1.

Cypress puede inferir sin problema el tipo de encoding a partir de la extensión del archivo, por lo que no tendremos que preocuparnos por ello.

```
cy.fixture('misDatos.json')
  .then(datos => {})
```

31.14. Lab: Fixtures

En este laboratorio vamos a ver como utilizar en nuestro test unos datos que ya tenemos guardados en un archivo JSON.

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

```
$ ng new angular-testing-e2e-cypress-fixtures-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 vamos a añadir Cypress en el proyecto:

```
$ cd angular-testing-e2e-cypress-fixtures-lab  
$ ng add @cypress/schematic  
  
? The package @cypress/schematic@1.6.0 will be installed and executed.  
Would you like to proceed? Yes  
? Would you like the default 'ng e2e' command to use Cypress? [ Protractor to Cypress Migration Guide:  
https://on.cypress.io/protractor-to-cypress?cli=true ] Yes
```

Una vez añadido Cypress, vamos a crear dos componentes y a levantar el servidor de desarrollo.

```
$ ng g c login  
$ ng g c home  
$ ng s
```

Vamos a empezar configurando nuestra aplicación para tener dos rutas, una para el formulario de login y otra para la página de inicio.

Creamos un archivo **app.routes.ts** en el que vamos a definir estas dos rutas y vamos a exportar el **RouterModule** ya configurado.

/angular-testing-e2e-cypress-fixtures-lab/src/app/app.routes.ts

```
import { RouterModule, Routes } from "@angular/router";  
import { LoginComponent } from "./login/login.component";  
import { HomeComponent } from './home/home.component';  
  
const APP_ROUTES: Routes = [  
  { path: 'login', component: LoginComponent },  
  { path: 'home', component: HomeComponent },  
]  
  
export const RoutingModule = RouterModule.forRoot(APP_ROUTES)
```

Ahora tenemos que importar este módulo en el módulo de la aplicación. Vamos a aprovechar también para importar el módulo de los formularios reactivos ya que lo vamos a necesitar ahora después.

/angular-testing-e2e-cypress-fixtures-lab/src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { LoginComponent } from './login/login.component';
import { HomeComponent } from './home/home.component';
import { RouterModule } from './app.routes';
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent,
    LoginComponent,
    HomeComponent
  ],
  imports: [
    BrowserModule,
    RouterModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

El siguiente paso es añadir el componente **router-outlet** en la plantilla del componente App para poder mostrar las dos páginas que vamos a tener.

/angular-testing-e2e-cypress-fixtures-lab/src/app/app.component.html

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

En el componente **Home** vamos a poner un mensaje de bienvenida.

/angular-testing-e2e-cypress-fixtures-lab/src/app/home/home.component.html

```
<h1>Bienvenido a la página</h1>
```

Y ahora en el componente de **Login** vamos a añadir un poco de lógica, ya que vamos a añadir un formulario que nos va a pedir el usuario y la contraseña.

Vamos a crear un formulario reactivo para ello, por lo que dentro del TypeScript empezaremos creando el objeto del formulario con los dos campos.

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

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

  constructor() {
    this.loginForm = new FormGroup({
      usuario: new FormControl(''),
      password: new FormControl('')
    })
  }

  ngOnInit(): void {
  }
}
```

Vamos a añadir el método **login** al que se va a llamar desde la plantilla, y de momento vamos a sacar el valor de los dos campos.

/angular-testing-e2e-cypress-fixtures-lab/src/app/login/login.component.ts

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

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

  constructor() {
    this.loginForm = new FormGroup({
      usuario: new FormControl(''),
      password: new FormControl('')
    })
  }

  ngOnInit(): void {}

  login() {
    const { usuario, password } = this.loginForm.value
  }
}
```

Ahora en la plantilla, vamos a crear el formulario y lo vamos a enlazar con el objeto **loginForm** que hemos creado.

/angular-testing-e2e-cypress-fixtures-lab/src/app/login/login.component.html

```
<h1>Login</h1>

<form [formGroup]="loginForm" (ngSubmit)="login()">
  <div>
    <label for="usuario">Usuario:</label>
    <input type="text" id="usuario" formControlName="usuario">
  </div>
  <div>
    <label for="password">Password:</label>
    <input type="text" id="password" formControlName="password">
  </div>
  <button type="submit">Login</button>
</form>
```

Con esto ya podemos llenar los campos y obtener los valores de ellos al pulsar el botón de **Login**.

Dentro de la función **login** vamos a comprobar si el login es correcto. Será correcto si el usuario es

cfalco y la contraseña es **1234**.

En caso de ser correctos los datos, vamos a enviar al cliente a la página de **home**. Por lo que necesitamos injectar el servicio de **Router** en el constructor.

/angular-testing-e2e-cypress-fixtures-lab/src/app/login/login.component.ts

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

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

  constructor(private router: Router) {
    this.loginForm = new FormGroup({
      usuario: new FormControl(''),
      password: new FormControl('')
    })
  }

  ngOnInit(): void {}

  login() {
    const { usuario, password } = this.loginForm.value
    if (usuario === 'cfalco' && password === '1234') {
      this.router.navigate(['/home'])
    }
  }
}
```

Este ejemplo se puede mejorar, pero con la funcionalidad que tiene ahora mismo, ya podemos empezar a crear nuestros tests.

Vamos a empezar por crear dentro de la carpeta de **cypress** nuestros datos de prueba en una carpeta **fixtures**.

/angular-testing-e2e-cypress-fixtures-lab/cypress/fixtures/datos-login.json

```
{  
  "usuarioOk": {  
    "usuario": "cfalco",  
    "password": "1234"  
  },  
  "usuarioKo": {  
    "usuario": "kozinski",  
    "password": "qwerty"  
  }  
}
```

Ahora dentro de nuestro test, vamos a empezar por añadir la navegación inicial en el hook **beforeEach**. Navegaremos directamente a la página de login.

/angular-testing-e2e-cypress-fixtures-lab/cypress/integration/spec.ts

```
describe('Fixtures', () => {  
  beforeEach(() => {  
    cy.visit('/login');  
  })  
})
```

Vamos a crear un test en el que vamos a loguearnos con un usuario válido para ver si conseguimos llegar a la página de inicio.

Utilizamos la función de fixture para cargar el archivo que contiene los datos de los usuarios y que hemos creado anteriormente.

/angular-testing-e2e-cypress-fixtures-lab/cypress/integration/spec.ts

```
describe('Fixtures', () => {  
  beforeEach(() => {  
    cy.visit('/login');  
  })  
  
  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {  
    cy.fixture('datosUsuarios.json')  
  })  
})
```

Los datos que nos devuelve la función del **fixture** los vamos a encontrar en la función de callback del **then**, y ahí mismo vamos a aprovechar para obtener el **usuarioOk** utilizando la desestructuración (operador spread).

Una vez que tenemos los datos, podemos llenar los campos del formulario y enviarlo.

```
describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datos-login.json')
      .then(({ usuarioOk }) => {
        cy.get('input#usuario')
          .type(usuarioOk.usuario);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();
      })
    })
  })
})
```

Si todo va bien, como debería de ocurrir, ahora deberíamos de estar en la página de inicio, y por tanto vamos a comprobar que el path de la URL es el raíz, y además para asegurarnos podemos comprobar también que aparece el título de la página dándonos la bienvenida.

Para obtener el path, usaremos el comando de **location** al que le vamos a pasar como parámetro, cual es el dato que queremos obtener, que es el **pathname**.

```
describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datos-login.json')
      .then(({ usuarioOk }) => {
        cy.get('input#usuario')
          .type(usuarioOk.usuario);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();
      })
    })
  })
})
```

Ya tenemos nuestro primer test, vamos a lanzar los test de cypress con el siguiente comando:

```
$ ng e2e
```

Ahora vamos a crear el segundo caso de prueba, en el que usaremos los datos del usuario que no es válido (**usuarioKo**) para comprobar que vuelve a la página del login y no consigue entrar al inicio.

/angular-testing-e2e-cypress-fixtures-lab/cypress/integration/spec.ts

```
describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datos-login.json')
      .then(({ usuarioOk }) => {
        cy.get('input#usuario')
          .type(usuarioOk.usuario);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/home');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
  })
})
```

Los primeros pasos del test son igual que antes, hay que leer el archivo de fixtures, pero esta vez, en la desestructuración nos vamos a quedar con los datos del otro usuario (**usuarioKo**).

```
describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datos-login.json')
      .then(({ usuarioOk }) => {
        cy.get('input#usuario')
          .type(usuarioOk.usuario);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/home');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
    cy.fixture('datos-login.json')
      .then(({ usuarioKo }) => {
        })
  })
})
```

Ahora rellenamos con estos datos el formulario y lo enviamos pulsando sobre el botón.

```
describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datos-login.json')
      .then(({ usuarioOk }) => {
        cy.get('input#usuario')
          .type(usuarioOk.usuario);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/home');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
    cy.fixture('datos-login.json')
      .then(({ usuarioKo }) => {
        cy.get('input#usuario')
          .type(usuarioKo.usuario);
        cy.get('input#password')
          .type(usuarioKo.password);
        cy.get('button')
          .click();
      })
  })
})
```

Y ya podemos comprobar que hemos vuelto a la página de login. Para ello, podemos mirar que esta vez el pathname es **/login**, y que el formulario existe en la página.

```
describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datos-login.json')
      .then(({ usuarioOk }) => {
        cy.get('input#usuario')
          .type(usuarioOk.usuario);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/home');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
    cy.fixture('datos-login.json')
      .then(({ usuarioKo }) => {
        cy.get('input#usuario')
          .type(usuarioKo.usuario);
        cy.get('input#password')
          .type(usuarioKo.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/login');

        cy.get('form')
          .should('exist');
      })
  })
})
```

Si nos fijamos, en ambos tests leemos el archivo de **datos-login.json** para acceder a los datos de los usuarios. Esta tarea la podemos realizar en el **beforeEach** al igual que la navegación a la página de login.

Vamos a poner esta instrucción en el hook, y para poder acceder a los datos que lee, vamos a añadirle un alias con el comando **as** y pasándole el nombre del alias con el cual pediremos estos

datos en los tests.

/angular-testing-e2e-cypress-fixtures-lab/cypress/integration/spec.ts

```
describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('/login');
    cy.fixture('datos-login.json').as('datosLogin');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datos-login.json')
      .then(({ usuarioOk }) => {
        cy.get('input#usuario')
          .type(usuarioOk.usuario);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/home');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
    cy.fixture('datos-login.json')
      .then(({ usuarioKo }) => {
        cy.get('input#usuario')
          .type(usuarioKo.usuario);
        cy.get('input#password')
          .type(usuarioKo.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/login');

        cy.get('form')
          .should('exist');
      })
  })
})
```

Ahora solo tenemos que cambiar la llamada al fixture de los tests por un **get(@datosLogin)** para poder utilizar el then y acceder a los datos que queramos del archivo JSON.

```
describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('/login');
    cy.fixture('datos-login.json').as('datosLogin');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.get('@datosLogin')
      .then(({ usuarioOk }) => {
        cy.get('input#usuario')
          .type(usuarioOk.usuario);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/home');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
    cy.get('@datosLogin')
      .then(({ usuarioKo }) => {
        cy.get('input#usuario')
          .type(usuarioKo.usuario);
        cy.get('input#password')
          .type(usuarioKo.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/login');

        cy.get('form')
          .should('exist');
      })
  })
})
```

Si probamos a lanzar los tests, estos deberían de seguir pasándose correctamente.

31.15. Mocking

Aunque utilizemos Cypress para crear tests end-to-end, hay algunos casos en los que necesitaremos crear mocks de alguna función o una funcionalidad que no nos devuelve algún resultado predecible para poder controlar los tests y que no fallen unas veces y otras salgan como correctos.

Los método que nos ayudarán con esta tarea son:

- **spy**: nos permite envolver una función con esta para poder registrar las llamadas y parámetros que se le envían.

```
cy.spy(serie, 'verCapitulo');
```

- **stub**: es como **spy**, pero además nos permite reemplazar una función por otra para poder controlar el funcionamiento de esta y retornar los valores que a nosotros nos interesen para el test.

```
cy.stub(serie, 'verCapitulo');

cy.stub(serie, 'verCapitulo', () => {
  return 5;
});

cy.visit('http://mi-pagina-web.com', {
  onBeforeLoad: (win) => {
    cy.stub(win, 'prompt').returns('Texto escrito en el popup del navegador.');
  }
});
```

- **intercept**: nos permite interceptar peticiones HTTP. El primer parámetro es la URL a interceptar, y le podemos pasar un segundo parámetro para evitar que la petición se realice y se devolverá dicho valor.

```
cy.intercept('http://mi-api.com/get-response').as('peticion');
cy.wait('@peticion');

cy.intercept('http://mi-api.com/get-response', { fixture: 'datos.json' })
```

31.16. Lab: Mocking con spy y stub

En este laboratorio vamos a ver como utilizar los métodos de spy y stub para comprobar que:

- Cuando pedimos la ubicación en la que nos encontramos solo se llama una vez a la función de `getCurrentPosition`.
- Cuando hacemos que la función que nos devuelve la ubicación nos de las coordenadas de Fargo, la ciudad que se muestra es la misma.
- Cuando hacemos que la función que nos devuelve la ubicación nos de las coordenadas de Harlan, la ciudad que se muestra es la misma.
- Cuando hacemos que la función que nos devuelve la ubicación nos de unas coordenadas vacías, la ciudad que se muestra es "Una ciudad cualquiera".



No funciona en el navegador Brave debido a la API de geolocation propia de los navegadores.

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

```
$ ng new angular-testing-e2e-cypress-mocking-spy-y-stub-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 vamos a añadir Cypress en el proyecto:

```
$ cd angular-testing-e2e-cypress-mocking-spy-y-stub-lab  
$ ng add @cypress/schematic  
  
? The package @cypress/schematic@1.6.0 will be installed and executed.  
Would you like to proceed? Yes  
? Would you like the default 'ng e2e' command to use Cypress? [ Protractor to Cypress Migration Guide:  
https://on.cypress.io/protractor-to-cypress?cli=true ] Yes
```

Una vez añadido Cypress, vamos a crear un servicio y levantaremos el servidor de desarrollo.

```
$ ng g s lugar  
$ ng s
```

En el servicio vamos a poner un método al cual pasandole unas coordenadas nos tiene que devolver el nombre de la ciudad a la que apuntan.

/angular-testing-e2e-cypress-mocking-spy-y-stub-lab/src/app/lugar.service.ts

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

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

  constructor() { }

  getCiudad(coords: any): string {
    const { latitude, longitude } = coords
    if (latitude === '46.874396' && longitude === '-96.835556') {
      return 'Fargo'
    } else if (latitude === '36.848044' && longitude === '-83.320589') {
      return 'Harlan'
    } else if (latitude === '52.485973' && longitude === '-1.890715') {
      return 'Birmingham'
    } else if (latitude === '35.110816' && longitude === '-106.668173') {
      return 'Albuquerque'
    } else {
      return 'Una ciudad cualquiera...'
    }
  }
}
```

Ahora en el componente, vamos a inyectar este servicio y crearemos un método en el que vamos a utilizar la api de **geolocation** para pedir nuestra ubicación actual.

/angular-testing-e2e-cypress-mocking-spy-y-stub-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { LugarService } from './lugar.service';

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

  constructor(private lugar: LugarService) {}

  localizar(): void {
    navigator.geolocation.getCurrentPosition((position: any) => {
      const { coords } = position
    })
  }
}
```

Ahora que tenemos las coordenadas, vamos a llamar al servicio para obtener la ciudad y guardarla en una propiedad del componente.

/angular-testing-e2e-cypress-mocking-spy-y-stub-lab/src/app/app.component.ts

```
import { Component } from '@angular/core';
import { LugarService } from './lugar.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  ciudad: string = ''

  constructor(private lugar: LugarService) {}

  localizar(): void {
    navigator.geolocation.getCurrentPosition((position: any) => {
      const { coords } = position
      this.ciudad = this.lugar.getCiudad(coords)
    })
  }
}
```

Por último, en la plantilla, vamos aadir un botón para llamar al método **localizar** y un párrafo en el que mostraremos la ciudad.

/angular-testing-e2e-cypress-mocking-spy-y-stub-lab/src/app/app.component.html

```
<button type="button" id="btn-ubicacion" (click)="localizar()">¿Dónde estoy?</button>
<p id="ciudad">Ubicación: {{ciudad}}</p>
```

Una vez tenemos la aplicación, vamos a lanzar los tests con el siguiente comando:

```
$ ng e2e
```

Ahora vamos a ir al archivo de testing para empezar a crear nuestros tests.

Vamos a empezar por indicar a que página tiene que entrar, y el comando **visit**, a parte de la URL también recibe un segundo parámetro de opciones donde podemos aadir una propiedad **onBeforeLoad** cuyo valor es una función en la que recibimos el objeto de la ventana y que podemos utilizar para crear un **spy** sobre el método **getCurrentPosition**.

Para ello, le vamos a pasar al método **spy** el objeto de la ventana y un string con el nombre de la función que queremos espiar. También le añadiremos un alias para poder realizar las comprobaciones necesarias más tarde.

```
describe('Geolocalización', () => {
  it('debería de llamar una sola vez a getCurrentPosition cuando se pulsa el botón', () => {
    cy.visit('/', {
      onBeforeLoad: (win) => {
        cy.spy(win.navigator.geolocation, 'getCurrentPosition').as('spyUbicacion');
      }
    });
  })
})
```

Ahora que tenemos creado el spy, ya podemos pulsar sobre el botón que pide nuestra ubicación al navegador, y justo después buscaremos el spy por el alias que le habíamos dado para comprobar que la función del **getCurrentPosition** solo se ha llamado una sola vez.

```
describe('Geolocalización', () => {
  it('debería de llamar una sola vez a getCurrentPosition cuando se pulsa el botón', () => {
    cy.visit('/', {
      onBeforeLoad: (win) => {
        cy.spy(win.navigator.geolocation, 'getCurrentPosition').as('spyUbicacion');
      }
    });

    cy.get('#btn-ubicacion')
      .click()

    cy.get('@spyUbicacion')
      .should('be.calledOnce')
  })
})
```

Ya tenemos nuestro primer test.

Vamos a por el siguiente con el que en lugar de crear un spy, vamos a crear un stub sobre la función que pide la ubicación de nuestro dispositivo. Al método **stub** le pasamos como primer parámetro el objeto donde se encuentra la función de la que queremos crear el stub, y como siguiente parámetro la función.

Después, sobre el objeto que devuelve este stub, vamos a añadir la función **callsFake** para indicarle que cuando vaya a ejecutar el callback que recibe el **getCurrentPosition**, los valores que queremos que nos devuelva son los que se han definido previamente.

Por último, también le vamos a añadir un alias.

```
describe('Geolocalización', () => {
  it('debería de llamar una sola vez a getCurrentPosition cuando se pulsa el botón', () => {
    cy.visit('/', {
      onBeforeLoad: (win) => {
        cy.spy(win.navigator.geolocation, 'getCurrentPosition').as('spyUbicacion');
      }
    });

    cy.get('#btn-ubicacion')
      .click()

    cy.get('@spyUbicacion')
      .should('be.calledOnce')
  })

  it('debería mostrar "Fargo" cuando se le pasan sus coordenadas', () => {
    const coords = {latitude: '46.874396', longitude: '-96.835556'};
    cy.visit('/', {
      onBeforeLoad: win => {
        cy.stub(win.navigator.geolocation, 'getCurrentPosition')
          .callsFake(callback => {
            return callback(coords)
          })
          .as('stubUbicacion')
      }
    })
  })
})
```

Una vez que tenemos el stub, pulsamos el botón, y vamos a comprobar que para estas coordenadas, la ciudad es **Fargo**, y que la función del **getCurrentPosition** solo se ha llamado una vez.

```
describe('Geolocalización', () => {
  it('debería de llamar una sola vez a getCurrentPosition cuando se pulsa el botón', () => {
    cy.visit('/ ', {
      onBeforeLoad: (win) => {
        cy.spy(win.navigator.geolocation, 'getCurrentPosition').as('spyUbicacion');
      }
    });

    cy.get('#btn-ubicacion')
      .click()

    cy.get('@spyUbicacion')
      .should('be.calledOnce')
  })

  it('debería mostrar "Fargo" cuando se le pasan sus coordenadas', () => {
    const coords = {latitude: '46.874396', longitude: '-96.835556'};
    cy.visit('/ ', {
      onBeforeLoad: win => {
        cy.stub(win.navigator.geolocation, 'getCurrentPosition')
          .callsFake(callback => {
            return callback({coords})
          })
          .as('stubUbicacion')
      }
    });

    cy.get('#btn-ubicacion')
      .click();

    cy.get('p')
      .should('have.text', 'Ubicación: Fargo');

    cy.get('@stubUbicacion')
      .should('be.calledOnce');
  })
})
```

En el siguiente test, vamos a realizar una comprobación similar a la anterior, solo que ahora comprobaremos que la ciudad que se muestra es **Harlan**.

```
describe('Geolocalización', () => {
  it('debería de llamar una sola vez a getCurrentPosition cuando se pulsa el botón', () => {
    cy.visit('/ ', {
      onBeforeLoad: (win) => {
        cy.spy(win.navigator.geolocation, 'getCurrentPosition').as('spyUbicacion');
      }
    });

    cy.get('#btn-ubicacion')
```

```

    cy.get('#btn-ubicacion')
      .click()

    cy.get('@spyUbicacion')
      .should('be.calledOnce')
  })

it('debería mostrar "Fargo" cuando se le pasan sus coordenadas', () => {
  const coords = {latitude: '46.874396', longitude: '-96.835556'};
  cy.visit('/', {
    onBeforeLoad: win => {
      cy.stub(win.navigator.geolocation, 'getCurrentPosition')
        .callsFake(callback => {
          return callback({coords})
        })
        .as('stubUbicacion')
    }
  })

  cy.get('#btn-ubicacion')
    .click();

  cy.get('p')
    .should('have.text', 'Ubicación: Fargo');

  cy.get('@stubUbicacion')
    .should('be.calledOnce');
})

it('debería mostrar "Harlan" cuando se le pasan sus coordenadas', () => {
  const coords = {latitude: '36.848044', longitude: '-83.320589'};
  cy.visit('/', {
    onBeforeLoad: win => {
      cy.stub(win.navigator.geolocation, 'getCurrentPosition')
        .callsFake(callback => {
          return callback({coords})
        })
        .as('stubUbicacion')
    }
  })

  cy.get('#btn-ubicacion')
    .click();

  cy.get('p')
    .should('have.text', 'Ubicación: Harlan');
})

})

```

Por último, comprobaremos también que cuando las coordenadas son vacias o aleatorias, la ciudad que se nos muestra es **Una ciudad cualquiera....**

```
describe('Geolocalización', () => {
  it('debería de llamar una sola vez a getCurrentPosition cuando se pulsa el botón', () => {
    cy.visit('/', {
      onBeforeLoad: (win) => {
        cy.spy(win.navigator.geolocation, 'getCurrentPosition').as('spyUbicacion');
      }
    });

    cy.get('#btn-ubicacion')
      .click()

    cy.get('@spyUbicacion')
      .should('be.calledOnce')
  })

  it('debería mostrar "Fargo" cuando se le pasan sus coordenadas', () => {
    const coords = {latitude: '46.874396', longitude: '-96.835556'};
    cy.visit('/', {
      onBeforeLoad: win => {
        cy.stub(win.navigator.geolocation, 'getCurrentPosition')
          .callsFake(callback => {
            return callback({coords})
          })
          .as('stubUbicacion')
      }
    })

    cy.get('#btn-ubicacion')
      .click();

    cy.get('p')
      .should('have.text', 'Ubicación: Fargo');

    cy.get('@stubUbicacion')
      .should('be.calledOnce');
  })

  it('debería mostrar "Harlan" cuando se le pasan sus coordenadas', () => {
    const coords = {latitude: '36.848044', longitude: '-83.320589'};
    cy.visit('/', {
      onBeforeLoad: win => {
        cy.stub(win.navigator.geolocation, 'getCurrentPosition')
          .callsFake(callback => {
            return callback({coords})
          })
          .as('stubUbicacion')
      }
    })

    cy.get('#btn-ubicacion')
      .click();
  })
})
```

```
cy.get('p')
  .should('have.text', 'Ubicación: Harlan');
})

it('debería mostrar la ciudad comodín cuando se le pasan sus coordenadas', () => {
  const coords = {latitude: '', longitude: ''};
  cy.visit('/', {
    onBeforeLoad: win => {
      cy.stub(win.navigator.geolocation, 'getCurrentPosition')
        .callsFake(callback => {
          return callback({coords})
        })
        .as('stubUbicacion')
    }
  })

  cy.get('#btn-ubicacion')
    .click();

  cy.get('p')
    .should('have.text', 'Ubicación: Una ciudad cualquiera...');
})
})
```

Y con esto ya tendríamos los tests que deberían de pasarse correctamente.

31.17. Lab: Mocking con intercept

En este laboratorio vamos a ver como interceptar una petición HTTP para devolver una respuesta mockeada cuando se quieren testear funcionalidades que no devuelven un valor previsible. El objetivo del laboratorio es probar que:

- Cuando una API del tiempo devuelve el texto 'soleado', se pinta el emoji del sol en la aplicación.
- Cuando una API del tiempo devuelve el texto 'nevado', se pinta el emoji de la nube con nieve en la aplicación.

Para este laboratorio vamos a necesitar dos proyectos, uno para la API, y otro de Angular.

Vamos a empezar por crear una carpeta **angular-testing-e2e-cypress-mocking-intercept-lab** y dentro de esta ambos proyectos con los siguientes comandos:

```
$ mkdir angular-testing-e2e-cypress-mocking-intercept-lab
$ cd angular-testing-e2e-cypress-mocking-intercept-lab
$ mkdir back
$ ng new front

? Would you like to add Angular routing? N
? Which stylesheet format would you like to use? CSS
```

Empezaremos con la parte del backend, vamos a entrar dentro de la carpeta **back** y vamos a lanzar los siguientes comandos para inicializar el proyecto e instalar las dependencias necesarias:

```
$ cd back
$ npm init -y
$ npm install express cors
```

Ahora añadimos el script que vamos a utilizar mas tarde, dentro del archivo **package.json**.

/angular-testing-e2e-cypress-mocking-intercept-lab/back/package.json

```
{  
  "name": "back",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "node src/server.js"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "express": "^4.17.2"  
  }  
}
```

Ahora añadimos el archivo **server.js** con la API a la que vamos a realizar las peticiones para obtener el tiempo y que nos va a enviar la página inicial de la aplicación.

/angular-testing-e2e-cypress-mocking-intercept-lab/back/src/server.js

```
const http = require('http');  
const express = require('express');  
const cors = require('cors');  
  
function getWeather() {  
  const weathers = ['soleado', 'parcialmente-nublado', 'nublado', 'lluvioso', 'tormenta', 'nevado'];  
  const pos = Math.floor(Math.random() * weathers.length);  
  return weathers[pos];  
}  
  
const app = express();  
  
app.use(cors())  
  
app.get('/get-weather', (req, res, next) => {  
  res.json({ weather: getWeather() });  
})  
  
const server = http.createServer(app);  
server.listen('8080');
```

Con esto ya tenemos la parte del back. Solo tenemos que levantar el servidor con el siguiente comando:

```
$ npm start
```

Al ejecutarse el comando, podemos entrar en <http://localhost:8080/get-weather> para ver que obtenemos un objeto con el tiempo que se va sacando de forma aleatoria.

Ahora desde otra terminal nos vamos a posicionar en la parte del front, en la que vamos a empezar, añadiendo cypress, creando un servicio con el que pediremos el tiempo a nuestra API y levantando el servidor con los siguientes comandos:

```
$ cd angular-testing-e2e-cypress-mocking-intercept-lab/front  
$ ng add @cypress/schematic  
  
? The package @cypress/schematic@1.6.0 will be installed and executed.  
Would you like to proceed? Yes  
? Would you like the default 'ng e2e' command to use Cypress? [ Protractor to Cypress Migration Guide:  
https://on.cypress.io/protractor-to-cypress?cli=true ] Yes  
  
$ ng g s tiempo  
$ ng s
```

Tenemos que importar en el módulo raíz de la aplicación el **HttpClientModule** para poder realizar peticiones HTTP.

/angular-testing-e2e-cypress-mocking-intercept-lab/front/src/app/app.module.ts

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

En el servicio que hemos creado vamos a añadir una petición GET para obtener los datos del tiempo de la API, y así poder devolver el emoji adecuado.

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

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

  constructor(private http: HttpClient) { }

  getEmojiDelTiempo(): Observable<string> {
    return this.http.get('http://localhost:8080/get-weather')
  }
}
```

Como queremos un emoji, vamos a utilizar el operador **map** de los observables para cambiar el texto que recibimos de la API por el emoji.

También hay que declarar un objeto de JavaScript con que emoji hay que devolver para cada uno de los strings que vamos a poder recibir.

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

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

  emojis: any = {
    soleado: '☀',
    'parcialmente-nublado': '🌤',
    nublado: '☁',
    lluvioso: '🌧',
    tormenta: '⛈',
    nevado: '❄',
  }

  constructor(private http: HttpClient) { }

  getEmojiDelTiempo(): Observable<string> {
    return this.http.get('http://localhost:8080/get-weather')
      .pipe(
        map((data: any) => {
          return this.emojis[data.weather]
        })
      )
  }
}
```

Ahora en nuestro componente App vamos a realizar esta petición cuando este se inicialice, y vamos a guardar el emoji en una propiedad **tiempoEmoji**.

/angular-testing-e2e-cypress-mocking-intercept-lab/front/src/app/app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { TiempoService } from './tiempo.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit{
  tiempoEmoji: string = ''

  constructor(private tiempo: TiempoService) {}

  ngOnInit(): void {
    this.tiempo.getEmojiDelTiempo()
      .subscribe((emoji: string) => {
        this.tiempoEmoji = emoji
      })
  }
}
```

Ahora en la plantilla vamos a mostrar el emoji.

/angular-testing-e2e-cypress-mocking-intercept-lab/front/src/app/app.component.html

```
<p>Tiempo para hoy: {{tiempoEmoji}}</p>
```

Ahora que tenemos la aplicación, vamos al archivo de testing **spec.ts** dentro de la carpeta **cypress/integration** y vamos a empezar por crear el bloque **describe** con nuestro primer test.

/angular-testing-e2e-cypress-mocking-intercept-lab/front/cypress/integration/spec.ts

```
describe('Tiempo', () => {
  it('debería pintar el sol si el tiempo es soleado', () => {
    })
})
```

Empezamos por navegar a la página inicial de la aplicación, y añadimos la comprobación que habíamos indicado al principio, es decir, que el span que hay dentro del párrafo muestre el emoji del sol cuando la API nos indique que el tiempo es soleado.

```
describe('Tiempo', () => {
  it('debería pintar el sol si el tiempo es soleado', () => {
    cy.visit('/')

    cy.get('p')
      .should('have.text', 'Tiempo para hoy: ☀')
  })
})
```

Vamos a ejecutar el test.

```
$ ng e2e
```

Si probamos a ejecutar varias veces este test, veremos que algunas se ejecuta correctamente y otras no. Esto se debe a que la API nos devuelve una respuesta que no es predecible en base a unos parámetros. Que nos devuelve el string **soleado** depende del tiempo que haga durante el día que se están ejecutando los tests, y un día puede estarlo, pero otro día puede estar nublado.

Entonces en este caso, para solucionar este problema, la solución es utilizar el método **intercept** para interceptar la llamada a la API, e indicarle la respuesta que nos viene bien que se retorne, es decir, estamos mockeando una respuesta de una API.

Al método intercept le vamos a pasar como parámetros el path de la API que tiene que interceptar y el objeto que queremos obtener como respuesta. Este método tiene que ejecutarse antes de llegar a entrar en la página.

```
describe('Tiempo', () => {
  it('debería pintar el sol si el tiempo es soleado', () => {
    cy.intercept('/get-weather', { weather: 'soleado' })
    cy.visit('/')

    cy.get('p')
      .should('have.text', 'Tiempo para hoy: ☀')
  })
})
```

Ahora ya debería de pasarse nuestro test correctamente.

El siguiente test es prácticamente igual, solo tenemos que cambiar la respuesta y la aserción, utilizando el texto **nevado** y el emoji de la nube con nieve.

```
describe('Tiempo', () => {
  it('debería pintar el sol si el tiempo es soleado', () => {
    cy.intercept('/get-weather', { weather: 'soleado' })
    cy.visit('/')

    cy.get('p')
      .should('have.text', 'Tiempo para hoy: ☀')
  })

  it('debería pintar la nube con nieve si el tiempo es nevado', () => {
    cy.intercept('/get-weather', { weather: 'nevado' })
    cy.visit('/')

    cy.get('p')
      .should('have.text', 'Tiempo para hoy: ❄')
  })
})
```

Ahora ya deberíamos de pasar las dos pruebas correctamente.

31.18. Tick y Clock

Cuando tenemos aplicaciones en las que el tiempo es algo importante, ya que la aplicación depende de los segundos, minutos... que vayan pasando, como por ejemplo aplicaciones de gestión de tiempo que usan técnicas como pomodoro, nos encontramos con que son difíciles de testear por el tiempo en sí, ya que cuando ejecutamos los tests, el tiempo también irá pasando y puede ser difícil cuadrar las aserciones.

Aquí Cypress nos viene a proporcionar dos funciones: **tick** y **clock**.

Con la función **clock** vamos a poder sobrescribir las funciones nativas del tiempo (`setTimeout`, `setInterval`...) para luego poder controlar síncronamente el tiempo que va a ir pasando durante el test.

Luego tenemos la función **tick**, que es la que nos va a permitir mover el tiempo pasandoselo como parámetro en milisegundos.

Esta función solo se puede llamar después que hayamos sobrescrito las funciones nativas del tiempo con el **clock**.

```
cy.clock()  
  
cy.tick(1000 * 4) // Adelantamos el tiempo 4 segundos  
cy.tick(1000 * 10 * 60) // Adelantamos el tiempo 10 minutos
```



La función **tick** es una función a la que no se le puede concatenar otros comandos de utilidad ni ningún tipo de aserción.

31.19. Tests visuales

Desde Cypress podemos comprobar que los elementos tienen las propiedades CSS correctas para que se pinten como se tienen que pintar. Pero a veces queremos saber si al actualizar alguna librería o cambiar los estilos de los elementos, se ha modificado algo que hace que la aplicación cambie su apariencia y no se vea como esperamos.

Una solución sería añadir aserciones con todas las propiedades CSS de los elementos, pero esto sería una tarea muy lenta de hacer y que no podemos permitirnos.

La otra solución es utilizar el **testing visual**, el cual consiste en comparar capturas de pantalla de nuestra aplicación o de los elementos sueltos.

Se saca un screenshot de la página en el momento en que la vemos que está perfecta, y este screenshot será la prueba visual con la que se compararán los siguientes screenshots que se obtengan al ejecutar los tests de la aplicación.

Cuando se modifique la web y sepamos que ya se muestra todo correctamente tendremos que actualizar el screenshot principal.

Esta tipo de pruebas son más lentas que las que hemos visto hasta ahora, ya que no es lo mismo realizar unas interacciones sobre la web y comprobar que un texto aparezca, que comparar dos imágenes.

Para realizar este tipo de pruebas podemos utilizar algunas herramientas como Applitools o Percy, o alguna dependencia que nos permita comparar imágenes.

Este tipo de pruebas las utilizan por ejemplo en Google.

31.20. Lab: Tests visuales

En este laboratorio vamos a ver como podemos realizar pruebas visuales de nuestras aplicaciones para comprobar que los elementos que se pintan en la aplicación no han cambiado.

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

```
$ ng new angular-testing-e2e-cypress-tests-visuales-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 vamos a añadir Cypress en el proyecto:

```
$ cd angular-testing-e2e-cypress-tests-visuales-lab  
$ ng add @cypress/schematic  
  
? The package @cypress/schematic@1.6.0 will be installed and executed.  
Would you like to proceed? Yes  
? Would you like the default 'ng e2e' command to use Cypress? [ Protractor to Cypress Migration Guide:  
https://on.cypress.io/protractor-to-cypress?cli=true ] Yes
```

Una vez añadido Cypress, vamos a levantar el servidor de desarrollo.

```
$ ng s
```

Dentro de nuestro archivo **app.component.html** vamos a poner una caja con el fondo de color.

/angular-testing-e2e-cypress-tests-visuales-lab/src/app/app.component.html

```
<div class="caja"></div>
```

Vamos a añadirle unos estilos a la caja.

/angular-testing-e2e-cypress-tests-visuales-lab/src/app/app.component.css

```
.caja {  
  width: 100px;  
  height: 100px;  
  background-color: gold;  
}
```

Una vez tenemos la caja de color, vamos a añadir nuestro caso de prueba en el que visitaremos la página de la aplicación.

```
/angular-testing-e2e-cypress-tests-visuales-lab/cypress/integration/spec.ts
```

```
describe('Caja de oro', () => {
  it('debería pintar una caja de color oro', () => {
    cy.visit('/')
  })
})
```

Podemos comprobar que el color de la caja es el correcto utilizando la aserción de **have.css**.

```
/angular-testing-e2e-cypress-tests-visuales-lab/cypress/integration/spec.ts
```

```
describe('Caja de oro', () => {
  it('debería pintar una caja de color oro', () => {
    cy.visit('/')

    cy.get('.caja')
      .should('have.css', 'background-color', 'rgb(255, 215, 0)');
  })
})
```

Pero si tuviéramos que realizar comprobaciones de muchas propiedades CSS, el test se puede volver demasiado complejo. Por eso, vamos a utilizar una prueba visual para pasar este test.

Estas pruebas consisten en sacar una imagen de la página que se va a ir comparando con las siguientes imágenes que se saquen cuando ejecutemos el test en el futuro. Así que necesitamos utilizar alguna herramienta para que se encargue de estas tareas, realizar las capturas y la comprobación entre imágenes.

Vamos a utilizar el paquete **cypress-image-snapshot**, por lo que vamos a instalarlo en el proyecto utilizando NPM:

```
$ npm install --save-dev cypress-image-snapshot @types/cypress-image-snapshot
```

Una vez instalado, vamos a seguir los pasos que se nos indican en la documentación de esta dependencia para poder utilizarla.

Vamos a añadir el plugin en la configuración de Cypress, por lo que dentro del archivo **cypress/plugins/index.ts** vamos a importar el plugin que viene con la dependencia y lo vamos a ejecutar dentro del módulo que se está exportando.

/angular-testing-e2e-cypress-tests-visuales-lab/cypress/plugins/index.ts

```
// ...
import { addMatchImageSnapshotPlugin } from 'cypress-image-snapshot/plugin';

export default (on: any, config: any) => {
  addMatchImageSnapshotPlugin(on, config)
}
```

Una vez añadido aquí el plugin, nos toca añadir el comando para que podemos utilizar la funcionalidad desde el objeto **cy**. Vamos a llamar a la función que se encarga de añadir el comando **matchImageSnapshot** dentro del archivo **cypress/support/commands.ts** además de configurarlo con el porcentaje de fallo que consideramos aceptable para que un test se pase correctamente.

/angular-testing-e2e-cypress-tests-visuales-lab/cypress/support/commands.ts

```
// ...
import { addMatchImageSnapshotCommand } from 'cypress-image-snapshot/command';

addMatchImageSnapshotCommand({
  failureThresholdType: 'percent'
})
```

Tenemos que añadir la importación de este último archivo en el **cypress/support/index.ts**, ya que si no lo hacemos, el objeto **cy** no tendrá acceso a la función **matchImageSnapshot**.

/angular-testing-e2e-cypress-tests-visuales-lab/cypress/support/index.ts

```
// ...
import './commands';
```

Ahora ya podemos utilizar el comando de Cypress **matchImageSnapshot** pasándole como parámetro el nombre que le queremos dar al snapshot.

/angular-testing-e2e-cypress-tests-visuales-lab/cypress/integration/spec.ts

```
describe('Caja de oro', () => {
  it('debería pintar una caja de color oro', () => {
    cy.visit('/')

    cy.get('.caja')
      .should('have.css', 'background-color', 'rgb(255, 215, 0)');
  })

  it('debería pintar una caja de color oro con el nuevo comando', () => {
    cy.visit('/')
    cy.matchImageSnapshot('caja-de-oro')
  })
})
```

Con esto ya tendríamos nuestro test visual. Vamos a ejecutarlo una vez para sacar el snapshot inicial.

Después de que se haya ejecutado, podremos ver una carpeta **cypress/snapshots/caja-gold.spec.js** con el snapshot que se ha sacado.

Vamos a cambiar los estilos de la caja.

/angular-testing-e2e-cypress-tests-visuales-lab/src/app/app.component.css

```
.caja {
  width: 120px;
  height: 120px;
  background-color: gold;
}
```

Ahora volvemos a ejecutar el test para comprobar que falla y nos genera la imagen con las diferencias entre el nuevo snapshot y el snapshot original.



Chapter 32. Internacionalización

La **internacionalización** consiste en preparar nuestra aplicación para que pueda utilizarse en cualquier parte del mundo, traduciendo el contenido de esta, incluso adaptando la presentación de ciertos datos a los formatos que se aplican en los distintos países.

Por ejemplo, los precios no se muestran igual en España que en EEUU. En España ponemos el símbolo de la moneda a la derecha del precio, mientras que en EEUU lo suelen poner a la izquierda.

Angular ya cuenta con su propio módulo de internacionalización, `@angular/localize`.

Para ir sacando las traducciones de las distintas etiquetas de la aplicación vamos a utilizar la directiva **i18n** sobre ellas. Con ello, el módulo de Angular conseguirá extraer a un archivo de traducciones todos estos textos, archivo en el que tendremos que poner cual es el lenguaje destino y cuales son las traducciones en dicho lenguaje.

Por otro lado, tendremos que configurar nuestra aplicación de Angular para que al lanzar el servidor de desarrollo, o al generar la aplicación para producción, esta pueda ser traducida.

32.1. Lab: internacionalización

En este laboratorio vamos a ver como añadir la parte de internacionalización a una aplicación para poder tenerla en distintos idiomas y dependiendo del idioma seleccionado que esta utilice unos formatos u otros por ejemplo para el pipe currency.

Lo primero que vamos a hacer es crearnos un proyecto nuevo de Angular lanzando el siguiente comando y seleccionaremos las opciones que mostramos a continuación:

```
$ ng new angular-internacionalizacion-lab  
? Would you like to add Angular routing? (y/N) N  
? Which stylesheet format would you like to use? (Use arrow keys)  
[ CSS  
SCSS [ https://sass-lang.com/documentation/syntax#scss ]  
Sass [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]  
Less [ http://lesscss.org ]
```

Una vez creado el proyecto entramos desde el terminal a la carpeta que se ha creado ya que tendremos que lanzar una serie de comandos en ella:

```
$ cd angular-internacionalizacion-lab
```

Ahora vamos a eliminar el contenido del html del componente raíz (`app.component.html`) para rellenarlo con el siguiente:

```
/angular-internacionalizacion-lab/src/app/app.component.html
```

```
<h1>Internationalization</h1>  
<p>Hello world</p>  
<p>{{329.9 | currency}}</p>
```

Estos elementos que acabamos de añadir son los que vamos a traducir a distintos idiomas.

El primer paso que hay que dar es añadir la librería de angular encargada de la internacionalización con el siguiente comando:

```
$ ng add @angular/localize  
The package @angular/localize@13.0.2 will be installed and executed.  
Would you like to proceed? (Y/n) Y
```

Una vez instalada la librería de **@angular/localize**, vamos a añadir las directivas **i18n** en aquellas etiquetas sobre las que haya que realizar las traducciones.

```
<h1 i18n>Internationalization</h1>
<p i18n>Hello world</p>
<p>{{329.9 | currency}}</p>
```

Ahora toca extraer los archivos de traducciones de nuestros componentes, y para ello tenemos que lanzar el siguiente comando:

```
$ ng extract-i18n --output-path src/locales
```

Ahí le estamos indicando que nos genere los archivos de traducciones dentro de la carpeta **src/locales**. Después de que el comando lanzado haya terminado de ejecutarse, podremos ver dicha carpeta.

En esta carpeta se ha debido de generar un archivo **messages.xlf**, el cual vamos a copiar por cada idioma al cual queramos traducir nuestra página. A todos ellos les pondremos el código de idioma en el nombre.

Nosotros realizaremos traducciones a inglés (por defecto), español y francés, por lo que tendremos los siguientes archivos de traducciones:

- src
 - locales
 - messages.es.xlf
 - messages.fr.xlf

En estos archivos tendremos que añadir una etiqueta **target** por cada bloque de traducción con el texto traducido, además de añadir un atributo **target-language** con el código de idioma del archivo de traducciones.

Nuestros archivos quedarán como podemos ver a continuación:

/angular-internacionalizacion-lab/src/locales/messages.es.xlf

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en-US" target-language="es" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="7373613501758200135" datatype="html">
        <source>Internationalization</source>
        <target>Internacionalización</target>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">1</context>
        </context-group>
      </trans-unit>
      <trans-unit id="2023484548631819319" datatype="html">
        <source>Hello world</source>
        <target>Hola mundo</target>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">2</context>
        </context-group>
      </trans-unit>
    </body>
  </file>
</xliff>
```

/angular-internacionalizacion-lab/src/locales/messages.fr.xlf

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en-US" target-language="fr" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="7373613501758200135" datatype="html">
        <source>Internationalization</source>
        <target>Internationalisation</target>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">1</context>
        </context-group>
      </trans-unit>
      <trans-unit id="2023484548631819319" datatype="html">
        <source>Hello world</source>
        <target>Salut monde</target>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">2</context>
        </context-group>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Ahora tenemos que ya tenemos nuestro componente y sus traducciones, toca modificar el archivo de **angular.json** para añadir estos idiomas en la configuración del proyecto.

El primer cambio que vamos a realizar en este archivo consiste en añadir la propiedad **i18n.locales** para indicar que idiomas vamos a permitir, y donde se encuentran sus archivos de traducciones.

/angular-internacionalizacion-lab/angular.json

```
{  
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",  
  "version": 1,  
  "newProjectRoot": "projects",  
  "projects": {  
    "angular-internacionalizacion-lab": {  
      "projectType": "application",  
      "schematics": { ... },  
      "root": "",  
      "sourceRoot": "src",  
      "prefix": "app",  
      "i18n": {  
        "locales": {  
          "es": "src/locales/messages.es.xlf",  
          "fr": "src/locales/messages.fr.xlf"  
        }  
      },  
      "architect": { ... }  
    }  
  },  
  "defaultProject": "angular-internacionalizacion-lab"  
}
```

Ahora dentro de **architect.build.configurations** tenemos que añadir la configuración del **locale** de Angular para cada uno de estos lenguajes que hemos añadido en el paso anterior. De esta forma cuando lancemos comandos de construcción del proyecto, podremos seleccionar para que idioma queremos traducirlo pasandole la opción de **--configuration=es**.

```
{  
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",  
  "version": 1,  
  "newProjectRoot": "projects",  
  "projects": {  
    "angular-internacionalizacion-lab": {  
      "projectType": "application",  
      "schematics": {  
        "@schematics/angular:application": {  
          "strict": true  
        },  
        "root": "",  
        "sourceRoot": "src",  
        "prefix": "app",  
        "i18n": {  
          "locales": {  
            "es": "src/locales/messages.es.xlf",  
            "fr": "src/locales/messages.fr.xlf"  
          }  
        },  
        "architect": {  
          "build": {  
            "builder": "@angular-devkit/build-angular:browser",  
            "options": { ... },  
            "configurations": {  
              "es": {  
                "localize": ["es"]  
              },  
              "fr": {  
                "localize": ["fr"]  
              },  
              "production": { ... },  
              "development": { ... }  
            },  
            "defaultConfiguration": "production"  
          },  
          "serve": { ... },  
          "extract-i18n": { ... },  
          "test": { ... }  
        }  
      }  
    }  
  },  
  "defaultProject": "angular-internacionalizacion-lab"  
}
```

Vamos a hacer exactamente lo mismo para poder levantar el proyecto con el servidor de desarrollo pero con la configuración de los distintos idiomas. Esta vez añadiremos los cambios en

`architect.serve.configurations` y haremos que coja la configuración añadida en el anterior paso.

`/angular-internacionalizacion-lab/angular.json`

```
{  
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",  
  "version": 1,  
  "newProjectRoot": "projects",  
  "projects": {  
    "angular-internacionalizacion-lab": {  
      "projectType": "application",  
      "schematics": {  
        "@schematics/angular:application": {  
          "strict": true  
        }  
      },  
      "root": "",  
      "sourceRoot": "src",  
      "prefix": "app",  
      "i18n": {  
        "locales": {  
          "es": "src/locales/messages.es.xlf",  
          "fr": "src/locales/messages.fr.xlf"  
        }  
      },  
      "architect": {  
        "build": {  
          "builder": "@angular-devkit/build-angular:browser",  
          "options": {  
            "outputPath": "dist/angular-internacionalizacion-lab",  
            "index": "src/index.html",  
            "main": "src/main.ts",  
            "polyfills": "src/polyfills.ts",  
            "tsConfig": "tsconfig.app.json",  
            "assets": [  
              "src/favicon.ico",  
              "src/assets"  
            ],  
            "styles": [  
              "src/styles.css"  
            ],  
            "scripts": []  
          },  
          "configurations": {  
            "es": {  
              "localize": ["es"]  
            },  
            "fr": {  
              "localize": ["fr"]  
            },  
            "production": { ... },  
            "development": { ... }  
          }  
        }  
      }  
    }  
  }  
}
```

```

    "development": { ... }
  },
  "defaultConfiguration": "production"
},
"serve": {
  "builder": "@angular-devkit/build-angular:dev-server",
  "configurations": {
    "es": {
      "browserTarget": "angular-internacionalizacion-lab:build:es"
    },
    "fr": {
      "browserTarget": "angular-internacionalizacion-lab:build:fr"
    },
    "production": {
      "browserTarget": "angular-internacionalizacion-lab:build:production"
    },
    "development": {
      "browserTarget": "angular-internacionalizacion-lab:build:development"
    }
  },
  "defaultConfiguration": "development"
},
"extract-i18n": { ... },
"test": { ... }
}
}
},
"defaultProject": "angular-internacionalizacion-lab"
}

```

Ahora ya podemos probar a levantar la aplicación en modo desarrollo con cualquiera de los 3 idiomas lanzando alguno de los siguientes comandos:

```

$ ng s
$ ng s --configuration=es
$ ng s --configuration=fr

```

Internationalization

Hello world

\$329.90

Internacionalización

Hola mundo

329,90 US\$

Internationalisation

Salut monde

329,90 \$US

Como podemos observar en las anteriores imágenes, las traducciones han salido bien, además de que el formato para los pipes se está cogiendo de forma correcta según el locale.

Por ejemplo, para español y francés podemos ver que el símbolo de las monedas aparece a la

derecha en lugar de hacerlo a la izquierda como ocurre con el inglés (el que se usa por defecto).

Por último vamos a generar los archivos estáticos de la aplicación junto a las traducciones, pero antes, necesitamos un menú con el que poder navegar desde la página en un idioma a otro cualquiera de los que hemos puesto.

Esto vamos a añadirlo en el componente App.

Empezamos añadiendo, en el archivo de typescript, una lista con los distintos código de los idiomas y el texto que queremos que aparezca en los enlaces.

/angular-internacionalizacion-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 {
  localesList = [
    { code: 'en-US', label: 'English' },
    { code: 'es', label: 'Spanish' },
    { code: 'fr', label: 'French' }
  ]
}
```

Ahora vamos a añadir la lista de enlaces al final del html que ya teníamos.

/angular-internacionalizacion-lab/src/app/app.component.html

```
<h1 i18n>Internationalization</h1>
<p i18n>Hello world</p>
<p>{{329.9 | currency}}</p>

<ul>
  <li *ngFor="let locale of localesList">
    <a href="/{{locale.code}}"/>
      {{locale.label}}
    </a>
  </li>
</ul>
```

Ahora vamos a generar los archivos estáticos finales para todos los idiomas configurados con el siguiente comando:

```
$ ng build --localize
```

Una vez que termina de ejecutarse este comando, veremos una carpeta **dist/angular-internacionalizacion-lab** dentro de la cual tendremos una carpeta con los códigos de cada idioma, y a su vez dentro de estas carpetas tendremos la aplicación con sus distintas traducciones.

Podemos levantar esta aplicación con un servidor http para ver que todo esté bien.

Vamos a navegar hasta esta carpeta desde el terminal y vamos a lanzar los siguientes comandos:

```
$ cd dist/angular-internacionalizacion-lab  
$ npx http-serve
```

Entramos en <http://localhost:8080/en-US/>, y al ir pulsando sobre los distintos enlaces podremos ver como vamos navegando entre las 3 aplicaciones que hemos creado, cada una en un idioma distinto.

32.2. Lab: Internacionalización con ngx-translate

En este laboratorio vamos a ver como internacionalizar una aplicación de Angular usando la librería de **ngx-translate**.

Lo primero que vamos a hacer es crearnos un proyecto nuevo de Angular lanzando el siguiente comando y seleccionaremos las opciones que mostramos a continuación:

```
$ ng new angular-internacionalizacion-ngx-translate-lab  
? Would you like to add Angular routing? (y/N) N  
? Which stylesheet format would you like to use? (Use arrow keys)  
   CSS  
    SCSS [ https://sass-lang.com/documentation/syntax#scss ]  
    Sass [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]  
   Less [ http://lesscss.org ]
```

Una vez creado el proyecto entramos desde el terminal a la carpeta que se ha creado ya que tendremos que lanzar una serie de comandos en ella:

```
$ cd angular-internacionalizacion-ngx-translate-lab
```

Tenemos que instalar las dependencias de **ngx-translate** y levantar el servidor con los siguientes comandos:

```
$ npm install --save @ngx-translate/core @ngx-translate/http-loader  
$ ng serve
```

Ahora vamos a crear la página a la que después le añadiremos las traducciones necesarias.

Empezaremos por declarar una serie de propiedades en el componente App.

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  lenguajes: Array<string> = ['es', 'en']
  lenguajeSeleccionado: string = 'es'

  nombre: string = 'Charly'
  fechaActual: Date = new Date()
  hamburguesas: Array<any> = [
    { nombre: 'Burger Barney', precio: 9.95 },
    { nombre: 'Burger Mando', precio: 10.95 },
    { nombre: 'Burger Groot', precio: 11.95 },
    { nombre: 'Burger Doble Smash', precio: 12.95 },
    { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
  ]
}
```

Como vamos a utilizar un desplegable para seleccionar el lenguaje a utilizar, vamos a añadir una función para actualizar el valor de la propiedad **lenguajeSeleccionado** al elegido.

/angular-internacionalizacion-ngx-translate-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 {
  lenguajes: Array<string> = ['es', 'en']
  lenguajeSeleccionado: string = 'es'

  nombre: string = 'Charly'
  fechaActual: Date = new Date()
  hamburguesas: Array<any> = [
    { nombre: 'Burger Barney', precio: 9.95 },
    { nombre: 'Burger Mando', precio: 10.95 },
    { nombre: 'Burger Groot', precio: 11.95 },
    { nombre: 'Burger Doble Smash', precio: 12.95 },
    { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
  ]

  cambiarLenguaje(event: any) {
    this.lenguajeSeleccionado = event.target.value
  }
}
```

Ahora vamos a añadir el siguiente contenido en la plantilla del componente App para mostrar la bienvenida al usuario, la fecha actual, el desplegable de idiomas y el listado de hamburguesas.

/angular-internacionalizacion-ngx-translate-lab/src/app/app.component.html

```
<h1>Hamburguesas La Nave</h1>

<p>{{fechaActual}}</p>

<select (change)="cambiarLenguaje($event)">
  <option *ngFor="let lenguaje of lenguajes" [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{lenguaje}}</option>
</select>

<h2>Bienvenid@ {{nombre}}</h2>

<div *ngFor="let hamburguesa of hamburguesas">
  <h3>{{hamburguesa.nombre}}</h3>
  <p>Precio: {{hamburguesa.precio}}</p>
</div>
```

Una vez tenemos la estructura de la página, vamos a añadir la configuración inicial para las traducciones.

Dentro del **AppModule** vamos a añadir el módulo de **TranslateModule** y le vamos a pasar un

objeto de configuración con el que le vamos a indicar que cargue las traducciones usando Http. Como se va a utilizar Http, tendremos que importar también el módulo de **HttpClientModule**.

Para indicarle que vamos a cargar las traducciones mediante Http, le tenemos que pasar una factoría con una instancia de **TranslateHttpLoader** el cual recibe el servicio de **HttpClient**.

/angular-internacionalizacion-ngx-translate-lab/src/app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { TranslateModule, TranslateLoader } from '@ngx-translate/core';
import { TranslateHttpLoader } from '@ngx-translate/http-loader';
import { HttpClient, HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';

export function httpLoaderFactory(http: HttpClient) {
  return new TranslateHttpLoader(http);
}

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    TranslateModule.forRoot({
      loader: {
        provide: TranslateLoader,
        useFactory: httpLoaderFactory,
        deps: [HttpClient]
      }
    })
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Le vamos a pasar una opción más, **defaultLanguage**, para indicarle el lenguaje que vamos a usar por defecto.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { TranslateModule, TranslateLoader } from '@ngx-translate/core';
import { TranslateHttpLoader } from '@ngx-translate/http-loader';
import { HttpClient, HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';

export function httpLoaderFactory(http: HttpClient) {
  return new TranslateHttpLoader(http);
}

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    TranslateModule.forRoot({
      loader: {
        provide: TranslateLoader,
        useFactory: httpLoaderFactory,
        deps: [HttpClient]
      },
      defaultLanguage: 'en',
    })
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Una vez tenemos el módulo, vamos a crear los archivos de traducciones. Por defecto se busca en **/assets/i18n/[lang].json** donde el **[lang]** sería el código del lenguaje que vamos a usar en la aplicación para guardar el lenguaje seleccionado en el que queremos las traducciones.

Por tanto, vamos a crearnos dos archivos:

- /assets/i18n/es.json
- /assets/i18n/en.json

En estos archivos tenemos que poner las traducciones asociándolas a unas claves JSON que luego usaremos en nuestras plantillas con un pipe **translate** que nos ofrece esta librería.

/angular-internacionalizacion-ngx-translate-lab/src/assets/i18n/en.json

```
{  
  "HOME": {  
    "title": "La Nave's Hamburgers",  
    "welcome": "Welcome {{name}}",  
    "LANGUAGE_DROPDOWN": {  
      "en": "English",  
      "es": "Spanish"  
    },  
    "price": "Price"  
  }  
}
```

/angular-internacionalizacion-ngx-translate-lab/src/assets/i18n/es.json

```
{  
  "HOME": {  
    "title": "Hamburguesas La Nave",  
    "welcome": "Bienvenid@ {{name}}",  
    "LANGUAGE_DROPDOWN": {  
      "en": "Inglés",  
      "es": "Español"  
    },  
    "price": "Precio"  
  }  
}
```

Una vez tenemos las traducciones, nos vamos al componente App, en el que vamos a añadir el constructor y le vamos a inyectar el servicio de traducciones, **TranslateService** para añadir los lenguajes e inicializar el **lenguajeSeleccionado** con el valor que habíamos puesto como valor por defecto en la configuración del módulo.

```
import { Component } from '@angular/core';
import { TranslateService } from '@ngx-translate/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  lenguajes: Array<string> = ['es', 'en']
  lenguajeSeleccionado: string = 'es'

  nombre: string = 'Charly'
  fechaActual: Date = new Date()
  hamburguesas: Array<any> = [
    { nombre: 'Burger Barney', precio: 9.95 },
    { nombre: 'Burger Mando', precio: 10.95 },
    { nombre: 'Burger Groot', precio: 11.95 },
    { nombre: 'Burger Doble Smash', precio: 12.95 },
    { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
  ]
}

constructor(private translateService: TranslateService) {
  this.translateService.addLangs(this.lenguajes);
  this.lenguajeSeleccionado = this.translateService.getDefaultLang();
}

cambiarLenguaje(event: any) {
  this.lenguajeSeleccionado = event.target.value
}
}
```

Ahora al cambiar de lenguaje, le vamos a indicar a la librería que tiene que usar las traducciones pertenecientes al lenguaje seleccionado.

```
import { Component } from '@angular/core';
import { TranslateService } from '@ngx-translate/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  lenguajes: Array<string> = ['es', 'en']
  lenguajeSeleccionado: string = 'es'

  nombre: string = 'Charly'
  fechaActual: Date = new Date()
  hamburguesas: Array<any> = [
    { nombre: 'Burger Barney', precio: 9.95 },
    { nombre: 'Burger Mando', precio: 10.95 },
    { nombre: 'Burger Groot', precio: 11.95 },
    { nombre: 'Burger Doble Smash', precio: 12.95 },
    { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
  ]
}

constructor(private translateService: TranslateService) {
  this.translateService.addLangs(this.lenguajes);
  this.lenguajeSeleccionado = this.translateService.getDefaultLang();
}

cambiarLenguaje(event: any) {
  this.lenguajeSeleccionado = event.target.value
  this.translateService.use(this.lenguajeSeleccionado)
}
}
```

Ahora vamos a utilizar el pipe **translate** en nuestra plantilla, allí donde queremos realizar las traducciones. Este pipe se usa sobre un string con la ruta del documento JSON hasta la traducción que queremos.

Vamos a empezar por cambiar el título, por lo que usaremos como clave **HOME.title** como se muestra a continuación.

/angular-internacionalizacion-ngx-translate-lab/src/app/app.component.html

```
<h1>{{ 'HOME.title' | translate }}</h1>

<p>{{fechaActual}}</p>

<select (change)="cambiarLenguaje($event)">
  <option *ngFor="let lenguaje of lenguajes" [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{lenguaje}}</option>
</select>

<h2>Bienvenid@ {{nombre}}</h2>

<div *ngFor="let hamburguesa of hamburguesas">
  <h3>{{hamburguesa.nombre}}</h3>
  <p>Precio: {{hamburguesa.precio}}</p>
</div>
```

Ahora si cambiamos el lenguaje desde el desplegable, deberíamos de ver como cambia el título de la aplicación.

Vamos a añadir las traducciones para el precio y el desplegable de la misma forma, pero cada una con su clave.

/angular-internacionalizacion-ngx-translate-lab/src/app/app.component.html

```
<h1>{{ 'HOME.title' | translate }}</h1>

<p>{{fechaActual}}</p>

<select (change)="cambiarLenguaje($event)">
  <option *ngFor="let lenguaje of lenguajes" [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{'HOME.LANGUAGE_DROPDOWN.' + lenguaje | translate }}</option>
</select>

<h2>Bienvenid@ {{nombre}}</h2>

<div *ngFor="let hamburguesa of hamburguesas">
  <h3>{{hamburguesa.nombre}}</h3>
  <p>{{ 'HOME.price' | translate }}: {{hamburguesa.precio}}</p>
</div>
```

Ahora queremos añadir la traducción para el título de bienvenida, pero en este caso, vamos a hacerlo de otra forma, ya que en la propia traducción le habíamos puesto una interpolación de string con la clave **name**.

Para pasarle el nombre y que se devuelva junto a la traducción, le tenemos que pasar al pipe un parámetro que es un objeto de JS, donde la clave será **name** y el valor la propiedad **nombre**.

```
<h1>{{ 'HOME.title' | translate }}</h1>

<p>{{fechaActual}}</p>

<select (change)="cambiarLenguaje($event)">
  <option *ngFor="let lenguaje of lenguajes" [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{ 'HOME.LANGUAGE_DROPDOWN.' + lenguaje | translate }}</option>
</select>

<h2>{{ 'HOME.welcome' | translate:{name: nombre} }}</h2>

<div *ngFor="let hamburguesa of hamburguesas">
  <h3>{{hamburguesa.nombre}}</h3>
  <p>{{ 'HOME.price' | translate }}: {{hamburguesa.precio}}</p>
</div>
```

Lo último que nos queda por hacer con lo que tenemos es el poder cambiar tanto el formato de la fecha y el del precio con los pipes de **date** y **currency** teniendo en cuenta el idioma seleccionado.

Para ello, vamos a empezar por registrar los distintos locales que vamos a usar en el constructor del módulo de App. Usaremos la función de **registerLocaleData** pasándole como parámetros el **locale** que hay que importar, y un identificador para dicho locale.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { TranslateModule, TranslateLoader } from '@ngx-translate/core';
import {TranslateHttpLoader} from '@ngx-translate/http-loader';
import { HttpClient, HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';

import { registerLocaleData } from '@angular/common';
import localeEn from '@angular/common/locales/en'
import localeEs from '@angular/common/locales/es'

export function httpLoaderFactory(http: HttpClient) {
  return new TranslateHttpLoader(http);
}

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    TranslateModule.forRoot({
      loader: {
        provide: TranslateLoader,
        useFactory: httpLoaderFactory,
        deps: [HttpClient]
      },
      defaultLanguage: 'en',
    })
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
  constructor() {
    registerLocaleData(localeEn, 'en')
    registerLocaleData(localeEs, 'es')
  }
}
```

Una vez hecho esto, ya podemos ir a aplicar los dos pipes comentados antes a los que les pasaremos como parámetro el lenguaje seleccionado. En el caso del **currency** es el cuarto, mientras que para el de **date** es el tercero.

```
<h1>{{ 'HOME.title' | translate }}</h1>

<p>{{fechaActual | date:'medium':undefined:lenguajeSeleccionado}}</p>

<select (change)="cambiarLenguaje($event)">
  <option *ngFor="let lenguaje of lenguajes" [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{ 'HOME.LANGUAGE_DROPDOWN.' + lenguaje | translate }}</option>
</select>

<h2>{{ 'HOME.welcome' | translate:{name: nombre} }}</h2>

<div *ngFor="let hamburguesa of hamburguesas">
  <h3>{{hamburguesa.nombre}}</h3>
  <p>{{ 'HOME.price' | translate }}: {{hamburguesa.precio | currency:'EUR':undefined:undefined:lenguajeSeleccionado}}</p>
</div>
```

Para ir actualizando la fecha automáticamente, vamos a usar el observable **interval** para actualizar su valor cada segundo que pasa.

```
import { Component } from '@angular/core';
import { TranslateService } from '@ngx-translate/core';
import { interval } from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  lenguajes: Array<string> = ['es', 'en']
  lenguajeSeleccionado: string = 'es'

  nombre: string = 'Charly'
  fechaActual: Date = new Date()
  hamburguesas: Array<any> = [
    { nombre: 'Burger Barney', precio: 9.95 },
    { nombre: 'Burger Mando', precio: 10.95 },
    { nombre: 'Burger Groot', precio: 11.95 },
    { nombre: 'Burger Doble Smash', precio: 12.95 },
    { nombre: 'Burger Michelangelo Zayken', precio: 12.95 },
  ]

  constructor(private translateService: TranslateService) {
    this.translateService.addLangs(this.lenguajes);
    this.lenguajeSeleccionado = this.translateService.getDefaultLang();

    interval(1000)
      .subscribe(() => {
        this.fechaActual = new Date()
      })
  }

  cambiarLenguaje(event: any) {
    this.lenguajeSeleccionado = event.target.value
    this.translateService.use(this.lenguajeSeleccionado)
  }
}
```

Por último, vamos a añadir una traducción de fallback para cuando se intente traducir algo que no tiene configurada su traducción aparezca un mensaje por defecto y podamos darnos cuenta de ello para agregarla.

Para ello, vamos a añadir una clave más en los archivos JSON de traducciones.

/angular-internacionalizacion-ngx-translate-lab/src/assets/i18n/en.json

```
{  
  "HOME": {  
    "title": "La Nave's Hamburgers",  
    "welcome": "Welcome {{name}}",  
    "LANGUAGE_DROPDOWN": {  
      "en": "English",  
      "es": "Spanish"  
    },  
    "price": "Price"  
  },  
  "NO_TRANSLATION": "We don't have the translation"  
}
```

/angular-internacionalizacion-ngx-translate-lab/src/assets/i18n/es.json

```
{  
  "HOME": {  
    "title": "Hamburguesas La Nave",  
    "welcome": "Bienvenid@ {{name}}",  
    "LANGUAGE_DROPDOWN": {  
      "en": "Inglés",  
      "es": "Español"  
    },  
    "price": "Precio"  
  },  
  "NO_TRANSLATION": "No tenemos la traducción"  
}
```

Ahora en el módulo **AppModule** vamos a añadir la clave **missingTranslationHandler** a la que le vamos a indicar que cuando se pida el token **MissingTranslationHandler** se use la clase **MyMissingTranslationHandler** que vamos a crear en un archivo **my-missing-translation-handler.ts**.

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { TranslateModule, TranslateLoader, MissingTranslationHandler } from '@ngx-translate/core';
import {TranslateHttpLoader} from '@ngx-translate/http-loader';
import { HttpClient, HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';

import { registerLocaleData } from '@angular/common';
import localeEn from '@angular/common/locales/en'
import localeEs from '@angular/common/locales/es'
import { MyMissingTranslationHandler } from './my-missing-translation-handler';

export function httpLoaderFactory(http: HttpClient) {
  return new TranslateHttpLoader(http);
}

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    TranslateModule.forRoot({
      loader: {
        provide: TranslateLoader,
        useFactory: httpLoaderFactory,
        deps: [HttpClient]
      },
      defaultLanguage: 'en',
      missingTranslationHandler: {
        provide: MissingTranslationHandler,
        useClass: MyMissingTranslationHandler
      }
    })
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
  constructor() {
    registerLocaleData(localeEn, 'en')
    registerLocaleData(localeEs, 'es')
  }
}

```

Ahora creamos el archivo con la clase **MyMissingTranslationHandler** que va a extender de **MissingTranslationHandler**. Tenemos que implementar un método **handle** y aquello que devuelva dicho método será lo que se va a mostrar como traducción.

Aquí lo que haremos es acceder al servicio de traducciones que se recibe en los **params** y vamos a pedir el texto de **NO_TRANSLATION** que hemos añadido en los archivos JSON.

/angular-internacionalizacion-ngx-translate-lab/src/app/my-missing-translation-handler.ts

```
import { MissingTranslationHandler, MissingTranslationHandlerParams } from "@ngx-translate/core";

export class MyMissingTranslationHandler extends MissingTranslationHandler {
  handle(params: MissingTranslationHandlerParams) {
    return params.translateService.get('NO_TRANSLATION')
  }
}
```

Ahora vamos a la plantilla y vamos a añadir un párrafo con una clave que no existe para ver que se muestra el mensaje que hemos puesto antes.

/angular-internacionalizacion-ngx-translate-lab/src/app/app.component.html

```
<h1>{{ 'HOME.title' | translate }}</h1>

<p>{{fechaActual | date:'medium':undefined:lenguajeSeleccionado}}</p>

<select (change)="cambiarLenguaje($event)">
  <option *ngFor="let lenguaje of lenguajes" [value]="lenguaje" [selected]="lenguaje === lenguajeSeleccionado">{{ 'HOME.LANGUAGE_DROPDOWN.' + lenguaje | translate }}</option>
</select>

<h2>{{ 'HOME.welcome' | translate:{name: nombre} }}</h2>

<div *ngFor="let hamburguesa of hamburguesas">
  <h3>{{hamburguesa.nombre}}</h3>
  <p>{{ 'HOME.price' | translate }}: {{hamburguesa.precio | currency:'EUR':undefined:undefined:lenguajeSeleccionado}}</p>
</div>

<p>{{ 'HOME.no-existe' | translate }}</p>
```

Chapter 33. Compodoc

Compodoc es una herramienta que nos permite generar documentación de nuestras aplicaciones de Angular. También se usa para documentar aplicaciones hechas con Nest y Stencil.

Para instalar Compodoc hay que lanzar el siguiente comando:

```
$ ng add @compodoc/compodoc
```

A partir de la estructura de nuestro proyecto se va a generar esta documentación y podremos ver como se relacionan los distintos elementos del proyecto entre si.

Pero nosotros podemos mejorar la documentación generada usando las etiquetas que soporta de **JSDoc**. Estas etiquetas son:

- **@param**: da información sobre un parámetro, y podemos indicar el tipo de este metiéndolo entre llaves después de la etiqueta.
- **@returns**: permite indicar que valor se va a devolver por una función, incluso podemos ponerle el tipo de retorno entre llaves después de esta etiqueta.
- **@example**: permite poner un ejemplo del código, como por ejemplo, la llamada a una función con sus parámetros.
- **@ignore**: ignora el siguiente elemento y no lo muestra en la documentación final.
- **@link**: muestra un enlace a otra página donde podemos encontrar más información.

También podemos cambiar el estilo de la documentación indicándole que tema queremos utilizar con la opción **--theme** a la hora de generarla. En Compodoc nos proporcionan 8 temas:

- Gitbook
- Postmark
- Vagrant
- Laravel
- Stripe
- Readthedocs
- Materialdesign

Al añadir la librería, se generan los siguientes scripts de NPM en el **package.json** para generar y servir la documentación de la aplicación:

- **npm run compodoc:build**: genera la documentación.
- **npm run compodoc:serve**: sirve en el puerto 8080 la documentación generada previamente.
- **npm run compodoc:build-and-serve**: genera y sirve la documentación.

33.1. Lab: Compodoc

En este laboratorio vamos a ver como generar la documentación de una aplicación de Angular usando Compodoc.

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

```
$ ng new angular-compodoc-lab  
? Would you like to add Angular routing? Y  
? 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-compodoc-lab  
$ ng g c inicio  
$ ng g c fin  
$ ng g s bienvenida
```

En otro terminal vamos a añadir la librería de Compodoc en el proyecto lanzando el siguiente comando:

```
$ ng add @compodoc/compodoc  
The package @compodoc/compodoc@1.1.18 will be installed and executed.  
Would you like to proceed? Yes
```

Vamos a empezar por rellenar los componentes y el servicio.

Lo primero que vamos a hacer es añadir el **router-outlet** dentro de la plantilla del componente App.

/angular-compodoc-lab/src/app/app.component.html

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

En el servicio de **Bienvenida** vamos a poner un método que nos va a devolver un string diciendo **hola** seguido del nombre si se lo pasamos como parámetro, o seguido de **mundo** si no le pasamos un nombre.

/angular-compodoc-lab/src/app/bienvenida.service.ts

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

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

  constructor() { }

  getMsgBienvenida(nombre: string = 'mundo'): string {
    return 'Hola ' + nombre
  }

}
```

Ahora en el componente de **Inicio** vamos a inyectar el servicio y vamos a guardar el mensaje de bienvenida en una propiedad.

/angular-compodoc-lab/src/app/inicio/inicio.component.ts

```
import { Component, OnInit } from '@angular/core';
import { BienvenidaService } from '../bienvenida.service';

@Component({
  selector: 'app-inicio',
  templateUrl: './inicio.component.html',
  styleUrls: ['./inicio.component.css']
})
export class InicioComponent implements OnInit {
  mensajeBienvenida: string = ''

  constructor(private bienvenida: BienvenidaService) { }

  ngOnInit(): void {
    this.mensajeBienvenida = this.bienvenida.getMsgBienvenida('Charles Falco')
  }
}
```

En la plantilla mostramos el mensaje además de añadir un enlace para ir a la página de fin.

/angular-compodoc-lab/src/app/inicio/inicio.component.html

```
<a routerLink="/fin">Fin</a>

<p>{{mensajeBienvenida}}</p>
```

Ahora vamos a la página de fin, donde vamos a poner otro enlace a la página de inicio.

/angular-compodoc-lab/src/app/inicio/inicio.component.html

```
<a routerLink="/">Inicio</a>  
  
<p>Fin</p>
```

En el módulo de **AppRouting** vamos a añadir las dos rutas que tenemos.

/angular-compodoc-lab/src/app/app-routing.module.ts

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

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

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

Y con esto ya tenemos una aplicación pequeñita que podemos documentar.

En el servicio, podemos indicar que la función de **getMsgBienvenida**, tiene un parámetro nombre que es a quien se le da la bienvenida, y que devuelve el mensaje de bienvenida. Esta información se la vamos a dar con las etiquetas de JSDoc **param** y **returns**.

/angular-compodoc-lab/src/app/bienvenida.service.ts

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

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

  constructor() { }

  /**
   * Función que retorna un mensaje de bienvenida para la página de inicio
   * @param nombre - nombre a quien le das la bienvenida
   * @returns mensaje de bienvenida
   */
  getMsgBienvenida(nombre: string = 'mundo'): string {
    return 'Hola ' + nombre
  }
}
```

El constructor lo vamos a ignorar con la etiqueta **ignore**.

/angular-compodoc-lab/src/app/bienvenida.service.ts

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

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

  /**
   * @ignore
   */
  constructor() { }

  /**
   * Función que retorna un mensaje de bienvenida para la página de inicio
   * @param nombre - nombre a quien le das la bienvenida
   * @returns mensaje de bienvenida
   */
  getMsgBienvenida(nombre: string = 'mundo'): string {
    return 'Hola ' + nombre
  }
}
```

Y por último, vamos a añadir una descripción del servicio justo antes del decorador.

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

/**
 * Servicio que genera mensajes de bienvenida
 */
@Injectable({
  providedIn: 'root'
})
export class BienvenidaService {
  /**
   * @ignore
   */
  constructor() { }

  /**
   * Función que retorna un mensaje de bienvenida para la página de inicio
   * @param nombre - nombre a quien le das la bienvenida
   * @returns mensaje de bienvenida
   */
  getMsgBienvenida(nombre: string = 'mundo'): string {
    return 'Hola ' + nombre
  }
}
```

Con esto ya tenemos el servicio documentado. A la hora de generar la documentación se va a tener en cuenta la información que se puede sacar del propio typescript como el tipo de los datos, los valores por defecto... y las etiquetas y comentarios de JSDoc que hayamos puesto nosotros.

Ahora nos vamos a ir al componente de Inicio en el que vamos a añadir una descripción a la propiedad **mensajeBienvenida**.

```
import { Component, OnInit } from '@angular/core';
import { BienvenidaService } from '../bienvenida.service';

@Component({
  selector: 'app-inicio',
  templateUrl: './inicio.component.html',
  styleUrls: ['./inicio.component.css']
})
export class InicioComponent implements OnInit {
  /**
   * Propiedad para almacenar el mensaje de bienvenida
   */
  mensajeBienvenida: string = ''

  constructor(private bienvenida: BienvenidaService) { }

  ngOnInit(): void {
    this.mensajeBienvenida = this.bienvenida.getMsgBienvenida('Charles Falco')
  }
}
```

Y tendríamos que seguir añadiendo este tipo de comentarios y etiquetas sobre todos los elementos del proyecto para dejarlo lo mejor documentado posible.

Ahora vamos a ver como generar la documentación, y para ello, en el **package.json** se han añadido unos scripts de NPM con los que podemos generar y servir la documentación.

Podemos lanzar el comando siguiente y abrir a continuación el <http://localhost:8080> para ver la documentación:

```
$ npm run compodoc:build-and-serve
```

También podemos personalizar un poco más esta página cambiándole el tema y mostrando el logo de la empresa o del proyecto.

Para ello, vamos a añadir a mano un script de NPM que va a generar la documentación con los flags de **--theme** y **--customLogo** y servirá la documentación generada.

Vamos a descargarnos un logo y lo vamos a dejar en la raíz del proyecto.

```
{  
  "name": "angular-compodoc-lab",  
  "version": "0.0.0",  
  "scripts": {  
    "ng": "ng",  
    "start": "ng serve",  
    "build": "ng build",  
    "watch": "ng build --watch --configuration development",  
    "test": "ng test",  
    "compodoc:build": "compodoc -p tsconfig.doc.json",  
    "compodoc:build-and-serve": "compodoc -p tsconfig.doc.json -s",  
    "compodoc:serve": "compodoc -s",  
    "compodoc:custom": "compodoc -p tsconfig.doc.json --theme Postmark --customLogo logo.png -s"  
  },  
  "private": true,  
  "dependencies": {  
    "@angular/animations": "~13.2.0",  
    "@angular/common": "~13.2.0",  
    "@angular/compiler": "~13.2.0",  
    "@angular/core": "~13.2.0",  
    "@angular/forms": "~13.2.0",  
    "@angular/platform-browser": "~13.2.0",  
    "@angular/platform-browser-dynamic": "~13.2.0",  
    "@angular/router": "~13.2.0",  
    "@compodoc/compodoc": "^1.1.18",  
    "rxjs": "~7.5.0",  
    "tslib": "^2.3.0",  
    "zone.js": "~0.11.4"  
  },  
  "devDependencies": {  
    "@angular-devkit/build-angular": "~13.2.1",  
    "@angular/cli": "~13.2.1",  
    "@angular/compiler-cli": "~13.2.0",  
    "@types/jasmine": "~3.10.0",  
    "@types/node": "^12.11.1",  
    "jasmine-core": "~4.0.0",  
    "karma": "~6.3.0",  
    "karma-chrome-launcher": "~3.1.0",  
    "karma-coverage": "~2.1.0",  
    "karma-jasmine": "~4.0.0",  
    "karma-jasmine-html-reporter": "~1.7.0",  
    "typescript": "~4.5.2"  
  }  
}
```

Y ahora vamos a lanzar el nuevo comando:

```
$ npm run compodoc:custom
```

Ahora ya deberíamos de ver la documentación que se ha generado con el tema elegido y el logo.

The screenshot shows a documentation page for an Angular project. At the top left is the Angular logo. The title "AngularCompodocLab" is displayed, followed by a subtitle "This project was generated with [Angular CLI](#) version 13.2.1." A yellow circular button with a refresh icon is in the top right corner.

The main content area has a dark background with white text. It includes a search bar at the top left labeled "Type to search". On the left side, there is a sidebar with a tree view of the project structure:

- Getting started
- Overview
- README**
- Dependencies
- Modules**
 - AppModule
 - AppComponent
 - FinComponent
 - InicioComponent
 - AppRoutingModule
 - Injectables
 - BienvenidoService
 - Miscellaneous
- Routes
- Documentation coverage

The main content area contains several sections with instructions:

- Development server**: Run `ng serve` for a dev server. Navigate to `http://localhost:4200/`. The app will automatically reload if you change any of the source files.
- Code scaffolding**: Run `ng generate component component-name` to generate a new component. You can also use `ng generate directive|pipe|service|class|guard|interface|enum|module`.
- Build**: Run `ng build` to build the project. The build artifacts will be stored in the `dist/` directory.
- Running unit tests**: Run `ng test` to execute the unit tests via [Karma](#).
- Running end-to-end tests**: Run `ng e2e` to execute the end-to-end tests via a platform of your choice. To use this command, you need to first add a package that implements end-to-end testing capabilities.
- Further help**: To get more help on the Angular CLI use `ng help` or go check out the [Angular CLI Overview](#) and [Command Reference](#) page.