

Máster en Full Stack Developer

Framework de Front End Angular

Índice

| | |
|--|-----------|
| Tema 18. Introducción a Angular | 6 |
| 18.1. Introducción y objetivos | 6 |
| 18.2. ¿Cómo funciona Angular? | 7 |
| 18.3. Características principales de Angular | 8 |
| 18.4. ¿Cómo trabajamos con Angular? | 9 |
| 18.5. Herramientas para trabajar con Angular | 10 |
| 18.6. ¿Cómo instalar Angular? | 13 |
| Tema 19. TypeScript | 22 |
| 19.1. Introducción y objetivos | 22 |
| 19.2. Instalación y uso TypeScript | 23 |
| 19.3. Tipado | 24 |
| 19.4. Interfaces | 28 |
| 19.5. Decoradores | 29 |
| Tema 20. Programas y módulos | 32 |
| 20.1. Introducción y objetivos | 32 |
| 20.2. Partes de una aplicación de Angular | 32 |
| 20.3. Módulo principal | 36 |
| 20.4. Arquitectura por módulos | 37 |
| 20.5. Creación de módulos | 38 |
| Tema 21. Componentes | 40 |
| 21.1. Introducción y objetivos | 40 |
| 21.2. ¿Qué es un componente? | 40 |
| 21.2. Partes de un componente | 41 |
| 21.3. Cómo crear un componente | 44 |
| 21.4. Componente principal y componentes hijos. ¿Cómo se enlazan? | 46 |
| 21.5. Funciones de ciclo de vida de un componente | 50 |
| Tema 22. Templating | 54 |
| 22.1. Introducción y objetivos | 54 |
| 22.2. Sintaxis de plantillas | 54 |

| | |
|---|------------|
| 22.3. Hojas de estilos de componente | 62 |
| Tema 23. Data Binding: One-way and Two-ways | 66 |
| 23.1. Introducción y objetivos | 66 |
| 23.2. Property Binding | 66 |
| 23.2. Two way data binding: FormsModule y [(ngModel)]. Doble comunicación | 70 |
| Tema 24. Event Binding | 73 |
| 24.1. Introducción y objetivos | 73 |
| 24.2. Creación de eventos en Angular | 73 |
| 24.3. \$event | 79 |
| Tema 25. Input y Output | 82 |
| 25.1. Introducción y objetivos | 82 |
| 25.2. Comunicación entre componentes | 82 |
| 25.3. Decorador @Input(). Del padre al hijo | 85 |
| 25.4. Decorador @Output(). Del hijo al padre | 88 |
| Tema 26. Directivas | 92 |
| 26.1. Introducción y objetivos | 92 |
| 26.2. Directivas: definición y tipos | 92 |
| 26.3. ngStyle | 94 |
| 26.4. ngClass | 95 |
| 26.5. ngIf | 97 |
| 26.6. ngFor | 100 |
| 26.7. ngSwitch | 103 |
| 26.8. Creación de tu propia directiva | 104 |
| Tema 27. Inyección de dependencias | 108 |
| 27.1. Introducción y objetivos | 108 |
| 27.2. ¿Qué es y para qué se usa? | 108 |
| 27.3. Ejemplo de inyección de dependencia | 110 |
| Tema 28. Formularios y validaciones | 115 |
| 28.1. Introducción y objetivos | 115 |
| 28.2. Formulario de tipo Template. FormsModule | 116 |
| 28.3. Formulario de tipo Model. Reactive Forms | 119 |

| | |
|--|------------|
| 28.4. Validaciones de Angular y validaciones personalizadas | 122 |
| Tema 29. Rutas y navegación | 129 |
| 29.1. Introducción y objetivos | 129 |
| 29.2. Configuración básica. Módulo de rutas | 130 |
| 29.3. RouterLink y RouterLinkActivate | 136 |
| 29.4. Parámetros de ruta: Activated Route | 138 |
| 29.5. Rutas Hijas | 139 |
| 29.6. LocalStorage y Guards | 141 |
| Tema 30. Servicios y Http | 145 |
| 30.1. Introducción y objetivos | 145 |
| 30.2. ¿Qué es un servicio? | 145 |
| 30.3. Comunicación del servicio con los componentes | 148 |
| 30.4. HttpClientModule | 154 |
| 30.5. Peticiones GET, POST, PUT, DELETE | 156 |
| 30.6. Promesas y observables | 156 |
| Tema 31. Pipes | 166 |
| 31.1. Introducción y objetivos | 166 |
| 31.2. ¿Qué es un Pipes? | 166 |
| 31.3. Pipes propios de Angular | 167 |
| 31.4. Creación de pipe propio | 169 |
| Tema 32. Bootstrap en Angular | 173 |
| 32.1. Introducción y objetivos | 173 |
| 32.2. Instalación y configuración | 173 |
| 32.3. Uso de Bootstrap en Angular | 175 |
| Tema 33. Angular y Firebase. Gestión de Librerías Externas con Angular. Google Maps | 177 |
| 33.1. Introducción y objetivos | 177 |
| 33.2. ¿Cómo publicar una aplicación de Angular? | 178 |
| 33.3. Instalación de librerías de terceros en Angular | 180 |

| | |
|--|-----|
| 33.4. Uso de Google Maps en Angular a través de la librería AGM Maps | 184 |
|--|-----|

| | |
|---------|-----|
| A fondo | 186 |
|---------|-----|

| | |
|------|-----|
| Test | 196 |
|------|-----|

Tema 18. Introducción a Angular

18.1. Introducción y objetivos

A partir de este tema vamos a empezar a descubrir cómo podemos generar aplicaciones web **front-end** a través de una de los frameworks JavaScript más usados del momento.

Angular se ha convertido en uno de los frameworks más populares para el desarrollo de SPA (Single Page Applications).

Existen dos versiones de Angular: **Angular 1** o como se conoce AngularJS y **Angular 2** y sucesivas versiones (actualmente nos encontramos en la 13, el momento de escribir estas líneas). Son mantenidos en gran parte por Google, aparte de una gran comunidad de desarrolladores que aportan sus modificaciones para mejorar el framework.

Nosotros vamos a basar el contenido de este módulo en Angular 2 y sucesivas, ya que AngularJS fue una primera versión que sufrió un cambio muy complejo y que nada tiene que ver con las versiones actuales de Angular. Aquí deberéis tener especial cuidado con buscar información de framework y revisar cuando se esté hablando de Angular 1 (AngularJS) o Angular 2 y sucesivos, que las cosas cambian mucho.

A partir de este momento siempre que hablemos de Angular estaremos hablando de la versión actual, que es la 13.

El objetivo de este módulo es adquirir las herramientas necesarias para trabajar con Angular en todas sus versiones desde la 2 en adelante. Para ello vamos a tener que aprender no solo a manejar el framework, sino herramientas como Typescript y patrones de desarrollo como MVC (modelo vista controlador).

18.2. ¿Cómo funciona Angular?

¿Qué es Angular?

Angular es un framework front-end basado en JavaScript y Typescript. Se usa para el desarrollo de aplicaciones web multiplataforma y está pensado para desarrollar aplicaciones del tipo SPA (Single Page Application).

¿Y qué es un SPA? Traducido del inglés sería, una aplicación que ejecuta su contenido en una sola página, pero realmente no es el termino exacto, sería una aplicación web o página web que interactúa con el usuario reescribiendo dinámicamente la página con los nuevos datos que le lleguen del servidor web, esto permite que la carga del contenido sea más rápida y la experiencia de usuario más satisfactoria en cuanto a rendimiento.

Esto viene a sustituir a la forma tradicional de la carga de contenidos donde al cambiar de página se recarga todo el contenido de esta con la perdida de rendimiento que ello produce.

Una SPA funciona cargando el contenido HTML, CSS y Typescript (JavaScript compilado) por completo al abrir la web, al ir pasando de una sección a otra, solo necesita cargar el contenido que cambie nuevo, y no el resto de la página que ya está cargada.

Que sea una única página no implica que tenga todo el contenido cargado en una sección y que no exista distintas vistas. Lo que realmente hace Angular es cargar un módulo que gestiona toda la web y un contenedor principal que se encarga de cargar el contenido dinámico dentro del mismo, haciendo que cambie el aspecto y contenido de la página.

Es un framework modular que se basa en la creación de webs a través de componentes. Este concepto es muy práctico ya que permite crear distintos componentes dentro de una web, que realicen las funciones que necesitemos y la unión de dichos componentes me permiten crear aplicaciones web complejas.

El concepto de desarrollo por componentes es la base para crear aplicaciones web robustas y escalables, así como nos permite un desarrollo más ágil ya que podemos reutilizar los componentes que desarrollamos en varios proyectos.

18.3. Características principales de Angular

Las principales características en las cuales se basan todos los desarrollos que realicemos con Angular son las siguientes:

- ▶ Es un framework que me permite desarrollar **aplicaciones escalables y robustas**, ya que al estar bien compartimentado sabes donde esta cada cosa y separa muy bien la lógica, de los estilos y del interfaz.
- ▶ El tener todo compartimentado y el desarrollo por componentes me permite desarrollar aplicaciones en grupos de trabajo lo que me permite un desarrollo más ágil y una disminución de costes de producción.
- ▶ Es bastante **fácil de mantener y actualizar**.
- ▶ Al basarse en desarrollo por componentes el ciclo de vida de las aplicaciones es más largo ya que podemos reutilizar nuestros desarrollos en otros proyectos con muy pocos cambios o casi ninguno.
- ▶ El rendimiento y la **carga de la aplicación es muy rápida**.

- ▶ Angular sirve para proyectos sencillos de una web con un único desarrollador hasta una aplicación web de ámbito empresarial donde varios programadores tengan asignados diferentes roles y tareas.

18.4. ¿Cómo trabajamos con Angular?

En el siguiente enlace se puede encontrar esta y otra información relevante sobre Angular, todo lo explicado en estas líneas está fundamentado en la documentación oficial del framework.

<https://angular.io/docs>

Vamos a argumentar en unos pocos puntos cada una de las ventajas que me aportará trabajar con un framework como Angular, que van muy ligadas de la mano con las características que hemos comentado en el punto anterior.

Generación de código

Angular transforma las plantillas en código altamente optimizado para el trabajo en una máquina virtual JavaScript. Nos permite tener todos los beneficios de la creación manual de código, pero con la potencia de cualquiera de los mejores frameworks.

Universal

Es multiplataforma y dispositivo, el código generado es puramente HTML y CSS, por lo que podemos renderizarlo en todos los navegadores.

División de código

El usuario únicamente carga la parte de la página que necesita usar en cada momento, con lo que ahorramos muchos tiempos de carga. La creación de componentes reutilizables favorece la velocidad de carga en el navegador.

Sistema de plantillas

Angular posee un sistema de plantillas que me permite separar totalmente el interfaz visual (HTML y CSS), de la lógica (Typescript) de una forma muy simple y sin tener que capturar los elementos del DOM como en JavaScript.

Angular CLI

Es el cliente de angular que se ejecuta por terminal de línea de comandos y que nos aligera tanto la creación de proyectos de Angular como la de los componentes, así como levantar el servidor de trabajo o construir los archivos que posteriormente subiremos a al servidor para publicar nuestra aplicación en internet.

Testing

Angular se encuentra adaptado para aplicar el desarrollo basado en Test Unitarios y trabaja con las librerías de prueba más importantes.

18.5. Herramientas para trabajar con Angular

Antes de instalar Angular y empezar a trabajar con él, necesitamos tener una serie de herramientas que son absolutamente necesarias para trabajar con Angular y que sin ellas no podríamos desarrollar nuestro trabajo.

Muchas de ellas ya la venís usando en los temas anteriores, pero no viene mal recordar algunas de ellas y otras nuevas con las que necesitaremos trabajar.

Lo primero que necesitamos es un editor de código o IDE para ello vamos a usar Visual Studio Code, aquí os dejo el enlace de descarga. Es un editor de código gratuito que tiene un sistema de plugins muy completo que aumenta sus posibilidades del editor hasta el infinito.

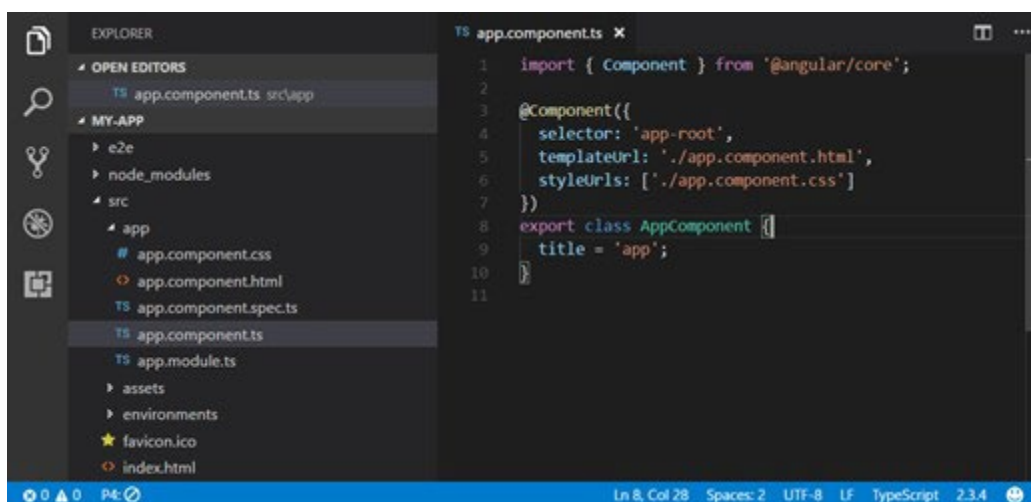


Figura 1. Árbol de ficheros de Angular. Fuente: elaboración propia.

Dentro de las extensiones de Visual Studio Code os recomiendo que instaléis Angular Language Service ya que es un plugin que me ayuda mucho en la escritura de código con Angular.

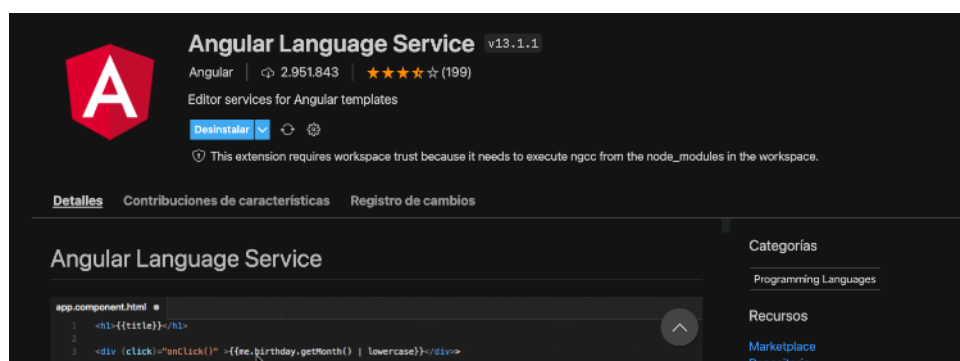


Figura 2. Plugin de Visual Studio Code. Fuente: elaboración propia.

Otra herramienta indispensable para nosotros es Node que, aunque tendrá un tema a parte dentro de este Máster lo necesitamos porque con Node se instala un gestor de paquete llamado NPM que es vital para poder instalar el cliente de Angular que me permitirá crear aplicaciones con este framework.

Para instalar Node nos iremos a su página oficial y nos descargaremos el instalable para nuestro sistema operativo a través del siguiente enlace:

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

The screenshot shows the Node.js download page. It has two main tabs: 'LTS' (Recommended For Most Users) and 'Current' (Latest Features). Under 'LTS', there are three download options: 'Windows Installer' (node-v16.13.2-x86.msi), 'macOS Installer' (node-v16.13.2.pkg), and 'Source Code' (node-v16.13.2.tar.gz). Below these, there is a list of download links: 'Windows Installer (.msi)', 'Windows Binary (.zip)', 'macOS Installer (.pkg)', 'macOS Binary (.tar.gz)', 'Linux Binaries (x64)', 'Linux Binaries (ARM)', and 'Source Code'. To the right of this list is a table with two columns: '32-bit' and '64-bit'. The table contains links for '64-bit / ARM64', '64-bit', 'ARM64', '64-bit', 'ARMv7', 'ARMv8', and 'node-v16.13.2.tar.gz'. Below the table, there is a section titled 'Additional Platforms' with a list of download links: 'Docker Image', 'Linux on Power LE Systems', 'Linux on System z', and 'AIX on Power Systems'. To the right of this list is a table with two columns: 'Official Node.js Docker Image' and '64-bit'. The table contains links for '64-bit', '64-bit', and '64-bit'.

| Platform | 32-bit | 64-bit |
|--------------------------|-----------------------------------|----------------------------------|
| Windows Installer (.msi) | node-v16.13.2-x86.msi | node-v16.13.2-x64.msi |
| Windows Binary (.zip) | node-v16.13.2-x86.zip | node-v16.13.2-x64.zip |
| macOS Installer (.pkg) | node-v16.13.2.pkg | node-v16.13.2.pkg |
| macOS Binary (.tar.gz) | node-v16.13.2.pkg | node-v16.13.2.pkg |
| Linux Binaries (x64) | node-v16.13.2-linux-x64.tar.gz | node-v16.13.2-linux-x64.tar.gz |
| Linux Binaries (ARM) | node-v16.13.2-linux-armv7l.tar.gz | node-v16.13.2-linux-arm64.tar.gz |
| Source Code | node-v16.13.2.tar.gz | node-v16.13.2.tar.gz |

Additional Platforms

| Platform | Official Node.js Docker Image |
|---------------------------|-----------------------------------|
| Docker Image | node:16 |
| Linux on Power LE Systems | node-v16.13.2-linuxppc64le.tar.gz |
| Linux on System z | node-v16.13.2-linuxs390x.tar.gz |
| AIX on Power Systems | node-v16.13.2-aixppc64.tar.gz |

Figura 3. Captura página web de Node JS. Fuente: elaboración propia.

De todas las opciones para nuestro sistema operativo y la versión LTS seleccionamos la que mejor nos convenga. Se descargará y ejecutará un instalador estándar el cual nos pedirá una serie de confirmaciones para proceder a la instalación.

Para comprobar que todo ha terminado de manera correcta podemos abrir una instancia del terminal dentro de nuestro ordenador y lanzar el comando.

```
node --version  
v14.17.1
```

Si el resultado de esta ejecución nos devuelve información sobre la versión que hemos instalado previamente quiere decir que hemos completado la instalación de NodeJS correctamente en nuestro ordenador.

A partir de este momento y podemos usar el gestor de dependencias NPM para instalar cualquier librería de JavaScript necesaria para trabajar, incluido Angular.

18.6. ¿Cómo instalar Angular?

Para trabajar con Angular vamos a tener que instalar cliente de Angular o lo que se conoce como Angular CLI, es una herramienta de línea de comandos que me simplifica mucho las acciones a la hora de generar un nuevo proyecto o crear nuevos componentes, será una herramienta que usaremos continuamente así que debemos perder el miedo a usarla desde el principio.

Angular CLI realiza una serie de tareas para que no tengas problemas. Aquí hay unos ejemplos:

| | |
|--------------------|---|
| ng build | Compila una aplicación Angular en un directorio de salida llamado dist/ en la ruta de salida dada. Debe ejecutarse desde dentro de un directorio de espacio de trabajo. |
| ng serve | Me permite levantar el servidor de Desarrollo para arrancar la aplicación y visualizar los cambios que voy desarrollando en tiempo real. |
| ng generate | Me permite generar componentes, servicios, pipes y cualquier estructura necesaria para trabajar en angular. Gracias a este script podemos crear y vincular los componentes con sus archivos de vistas y html con mucha facilidad. |
| ng test | Nos permite generar pruebas unitarias dentro de un proyecto determinado. |

Tabla 1. Tareas de Angular CLI. Fuente: elaboración propia.

Para instalar Angular Cli solo tienes que usar el gestor de paquetes NPM que hemos instalado anterior mente desde la terminal o línea de comandos. Abre el CMD o terminal y ejecuta la siguiente instrucción.

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

Este comando lo que me permite instalar el Angular CLI de forma global, de ahí el -g, dentro de nuestro equipo.

Para comprobar que angular este correctamente instalado en nuestro equipo escribimos el siguiente comando.

```
ng -versión
```

Que deberá lanzarnos como respuesta algo parecido a lo que vais a ver en la imagen siguiente.

```
[> ng --version

Angular CLI
Angular CLI: 13.0.1
Node: 14.17.1
Package Manager: npm 7.19.0
OS: darwin x64

Angular: undefined
...

Package      Version
-----
@angular-devkit/architect 0.1300.1 (cli-only)
@angular-devkit/core      13.0.1 (cli-only)
@angular-devkit/schematics 13.0.1 (cli-only)
@schematics/angular       13.0.1 (cli-only)
```

Figura 4. Captura terminal ng -version. Fuente: elaboración propia.

Una vez instalada esta herramienta y las anteriores ya podemos empezar a crear nuestras aplicaciones en Angular.

Crea una carpeta dentro de un directorio de trabajo a tu elección, y dentro de esa carpeta, desde la terminal o línea de comandos, ejecuta al comando que te permitirá crear una aplicación de Angular.

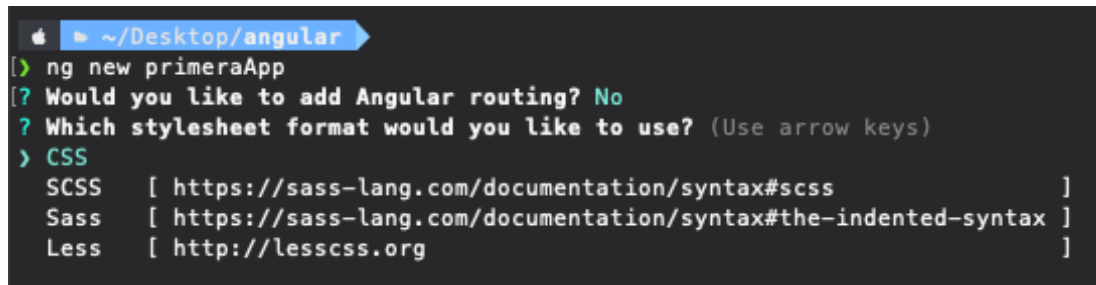
Dicho comando es:

```
ng new primeraApp
```

Donde **primeraApp** corresponde al nombre que le pongamos a nuestra aplicación de angular.

Este comando ejecutará un conjunto de ordenes que te realizará una serie de preguntas. La primera pregunta será si quiero **Angular Routing** a la que de momento contestaremos que no (más adelante cuando expliquemos rutas esta opción

cambiará) y la siguiente será que motor de CSS usaremos a la que moviéndonos con las flechas seleccionaremos en primera instancia CSS (igualmente más adelante realizaremos desarrollos con SCSS que habéis visto en temas anteriores).



```
~/Desktop/angular
[>] ng new primeraApp
[?] Would you like to add Angular routing? No
[?] 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 ]
```

Figura 5. Captura terminal creación de proyecto de Angular. Fuente: elaboración propia.

Una vez acabe el proceso de carga de la aplicación en Angular (puede tardar tiempo dependiendo del equipo con el que trabajes y tu conexión a internet).

Lo que está ocurriendo en este proceso es que se está creando una carpeta llamada primeraApp, una vez termine este proceso nos meteremos en la carpeta que Angular ha creado y allí ejecutaremos el comando:

```
cd primeraApp
ng serve
```

Este comando lo que hará será compilar el código Typescript y levantar un servidor en local que me permitirá ver la aplicación funcionando.

La aplicación por defecto se levanta en un puerto del ordenador que es un puerto local 4200. Así que si vas a un navegador y pones:

<http://localhost:4200>

Veras que una aplicación se levanta que tiene más o menos este aspecto.

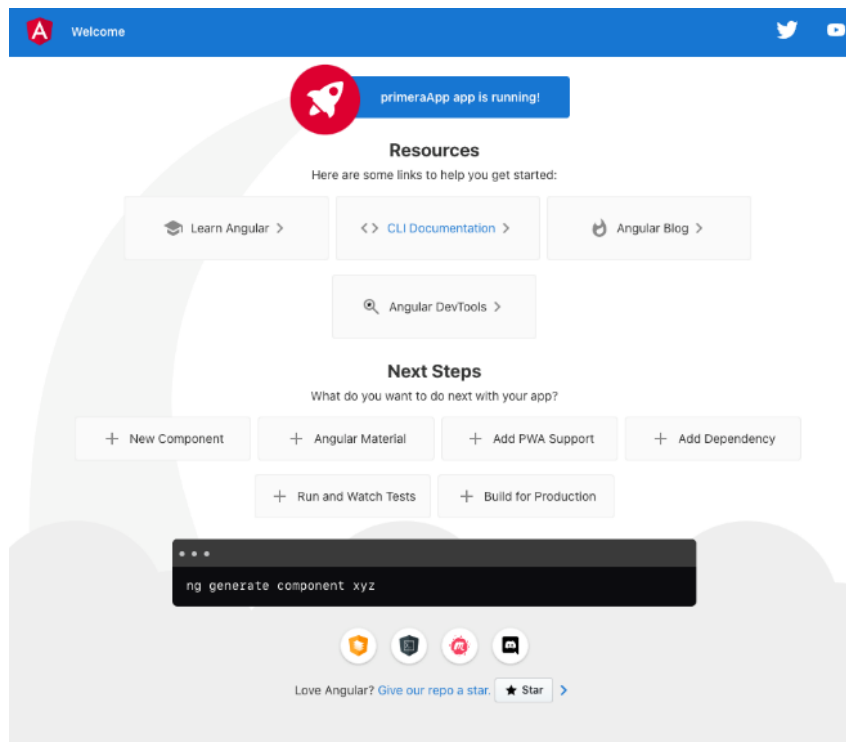


Figura 6. Pantalla inicial de un proyecto de Angular. Fuente: elaboración propia.

Esta es la aplicación por defecto que arranca angular la primera vez que levantamos el servidor.

Dentro de la carpeta de trabajo vamos a ver como Angular crea un sistema de carpetas similar al de la imagen, en este sistema puedes encontrar todos los archivos necesarios para que la aplicación de angular funcione.

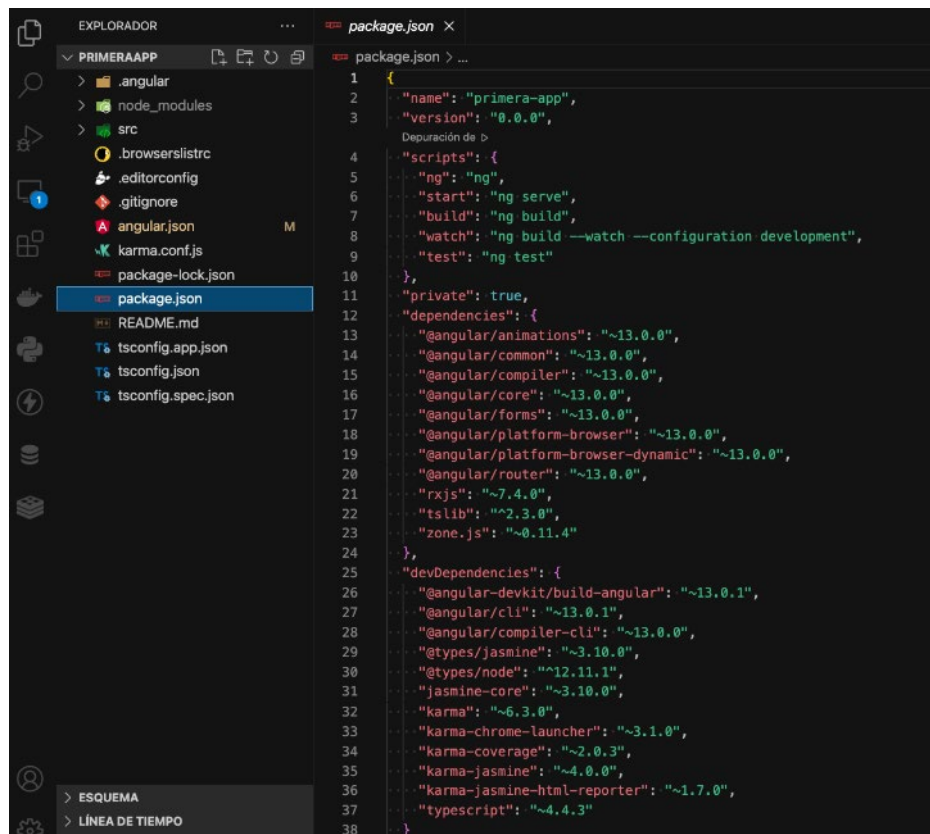


Figura 7. Árbol de ficheros de Angular, fichero package.json. Fuente: elaboración propia.

Vamos a hacer una introducción rápida de algunos ficheros importantes para saber para qué sirven, aunque según avance el curso.

Directorio node_modules

Contiene todas las dependencias del proyecto. Este directorio no tenemos que editarlo nosotros ya que NODE lo hará por nosotros.

En caso de que necesitemos agregar una dependencia o librería a nuestro proyecto de angular lo haremos por medio del comando.

```
npm install <paquete> --save
```

La bandera --save es opcional, pero se usa para marcar si la dependencia que deseamos instalar es necesaria solo para el desarrollo de la aplicación, es decir si

ponemos el `--save` la librería instalada es necesaria solo para desarrollo con lo que cuando publiquemos en producción no será necesaria tenerla. La información de todos los paquetes del proyecto se pueden ver en el archivo `package.json`.

Directorio `src`

Contiene el código de nuestra aplicación, en este directorio es donde nosotros escribimos nuestro código. Los archivos más importantes son el archivo `style.css`, y el `index.html` que es el archivo donde corre toda nuestra aplicación.

Directorio `src/app`

Contiene los archivos que son los responsables del funcionamiento de nuestra aplicación. Dentro de esta carpeta cabe resaltar el archivo `app.module.ts` que es el módulo principal de nuestra aplicación y el que se encarga de decirle al componente principal que otras librerías necesitas para trabajar y donde están situadas así como donde se encuentran los componentes que creemos y como se enlazan entre ellos. Una aplicación tiene como mínimo un modulo

Directorio `src/assets`

Contiene los archivos estáticos de la aplicación como imágenes, archivos audio etc.

Directorio `src/environments`

Contiene archivos con variables de configuración de entorno escritos en TypeScript para los entornos de desarrollo y/o producción. Aquí es donde se configuran claves y apis de acceso a servicios de terceros que necesita mi aplicación para funcionar.

Otro archivo es `angular.json`, este es esencial, tiene muchas propiedades. Va a permitir gestionar el espacio de trabajo, así como las configuraciones de los diferentes

proyectos de la aplicación Angular. Es importante conocer las configuraciones principales con el objetivo de asegurarse del correcto dominio de un proyecto realizado con ayuda de Angular CLI.

Por ejemplo, en este archivo podemos cambiar donde se cargan los archivos css y donde están alojadas las librerías externas de terceros en javascript.

La modificación de este fichero requiere que volvamos a ejecutar el servidor a través de:

```
ng serve
```

Recuerda que si el servidor esta levantado, te dejará la terminal bloqueada, no te preocupes por eso es correcto, cada vez que realices algún desarrollo dentro de la app la terminal volverá a recompilar los archivos de trabajo y recargará la aplicación en el navegador de forma automática.

Para parar el servidor y volver a recuperar el uso de la terminal deberás pulsar estas teclas:



Figura 8. Combinación de teclas para parar un proyecto de Angular. Fuente: <https://necesitoayuda.eu/>

Y hasta aquí el primer acercamiento a una aplicación en Angular durante el desarrollo de este módulo iremos desengranando poco a poco estos y más puntos que nos permitirán desarrollar aplicaciones web cada vez más complejas y funcionales.

En el siguiente vídeo, *Instalación de Angular y creación de un nuevo proyecto*, se explica cómo instalar Angular y cómo se crea un proyecto desde terminal.



Accede al vídeo

Tema 19. TypeScript

19.1. Introducción y objetivos

JavaScript, como lenguaje, se enfrenta a algunas limitaciones importantes que se han ido solucionando con las nuevas versiones del estándar web.

Algunas de ellas como el tipado débil de las variables, el uso de estructuras de datos más complejas o el aviso de errores en tiempo de desarrollo no de ejecución son algunos puntos que JavaScript hoy en día no ha conseguido solventar y que TypeScript nos ayuda solventarlo

El hecho de que estudiemos TypeScript en este curso es porque angular lo usa como piedra fundamental del desarrollo de sus componentes, así que este tema intenta servir como punto de partida y toma de contacto para aprender a trabajar con TypeScript.

Con TypeScript nos enfrentamos a un conjunto de herramientas creadas por encima de JavaScript.

Un resumen muy escueto de cómo funciona TypeScript, es un lenguaje compilado, este tipo de lenguajes necesitan de una herramienta para transformar código TypeScript en JavaScript, y en angular esta herramienta ya viene implementada y si lanza desde

ng serve

El objetivo fundamental de este tema es aprender los conceptos más importantes de TypeScript y como se aplica para usarlo como herramienta de desarrollo en nuestras aplicaciones de angular.

19.2. Instalación y uso TypeScript

Aunque TypeScript en Angular no necesita de ninguna instalación es cierto que TypeScript se puede usar no solo en Angular si no también en solitario para el desarrollo con JavaScript o en otros frameworks como React, Nest JS, etc.

Aprender TypeScript no es muy difícil si ya sabes JavaScript y te aporta herramientas muy interesantes.

La manera más fácil de usar TypeScript de forma nativa es instalarlo por defecto dentro de nuestra máquina de trabajo

```
npm install -g typescript
```

Recuerda que la bandera -g significa que no instalamos TypeScript dentro de una carpeta concreta si no de forma global en nuestro equipo.

Para estar seguros de que TypeScript se ha instalado correctamente tenemos que aplicar el siguiente comando en nuestra terminal. La respuesta esperada es la versión de TypeScript actual que acabamos de instalar.

```
tsc --version
```

Puntos a tener en cuenta cuando se usa TypeScript

Los ficheros con los que vamos a trabajar tendrán extensión .ts no extensión .js como estábamos usando hasta ahora.

Para poder trabajar con ellos debemos compilarlos y transformarlos a JavaScript con el siguiente comando.

```
tsc nombre_fichero.ts
```

19.3. Tipado

Una de las mejoras que trae TypeScript respecto a JavaScript es el tipado fuerte de variables. Este es opcional pero muy recomendable por parte de los desarrolladores ya que nos ayuda a cometer menos errores a la hora de programar.

Por qué es tan importante el tipado para los desarrolladores:

- ▶ Mediante el uso de los diferentes tipos a la hora de definir nuestras variables, limitamos su uso.
- ▶ Las variables que creamos solo vamos a poder utilizarlas para almacenar valores de esos tipos en concreto.
- ▶ Para especificar el tipo de unas variables podemos hacerlo con la siguiente estructura.

```
let nombreVariable: tipo = valor  
let nombre: string = "Juan"
```

Diferentes tipos de datos en TypeScript y cómo se usan

Boolean

Se trata del tipo más básico, en el que especificamos si el valor es verdadero (true) o falso (false).

```
let termina: boolean = false
```


Number

Es la definición genérica que usamos para hacer referencia a cualquier tipo de número en TypeScript.

Aparte de eso, el tipo `number` también soporta el uso de binarios, octales y hexadecimales.

```
let decimal: number = 23
let decimal2: number = 12.3
let hex: number = 0xf00d
let binario: number = 0b1001
let octal: number = 0o744
```

String

Utilizamos el tipo `string` para hacer referencia a todas las cadenas de tipo literal. Podemos limitar estas cadenas con comillas dobles (`"`), simples (`'`) o podemos utilizar los template strings típicos de JavaScript (```).

```
let nombre: string = "Roberto"
let apellidos: string = 'García'
let saludo: string = `Hola, me llamo ${nombre} ${apellidos}`
```

Array

Del mismo modo que pasa en JavaScript, los arrays nos permiten almacenar varios valores dentro de una estructura.

La diferencia aquí es que vamos a especificar el tipo de los valores almacenados en nuestro array.

Se puede especificar de dos maneras:

```
let list: number[] = [1,2,3,4,5]
let list2: Array<number> = [6,7,8,9]
```

Tupla

Este tipo de dato nos permite representar un array que, normalmente dispone un tamaño fijo y en el que limitamos a dos únicos tipos los datos que vamos a poder incluir dentro del mismo. Lo vemos con un ejemplo:

```
let t: [string, number]
t = ['hola', 32]
// Únicamente recibe string y number
```

Se podrían incluir más elementos de los tipos especificados, pero se suele limitar su uso a pares de valores.

Enum

Utilizamos este tipo de dato específico para aplicar un nombre más amigable a un conjunto de datos numéricos.

```
enum Size {Small, Medium, Big}
let s: Size = Size.Medium
```

Por defecto, los elementos de tipo enum comienzan a definir sus valores a partir de 0, pero podemos cambiar ese comportamiento, modificando alguno de los valores. Vemos cómo modificar el valor de inicio por defecto.

```
enum Size {Small=1, Medium, Big}
let s: Size = Size.Medium
console.log(s) // 2
```

Incluso podríamos definir todos los valores.

```
enum Size {Small=3, Medium=6, Big=9}
let s: Size = Size.Medium
console.log(s) // 6
```

Podemos recuperar también el nombre relacionado con alguno de los valores:

```
enum Size {Small=3, Medium=6, Big=9}
let nbColor: string = Size[6]
console.log(nbColor) // Medium
```

Any

Utilizamos este tipo para definir aquellas variables de las cuales desconocemos su tipo.

Este tipo de variables suelen venir del análisis de contenido dinámico o de librerías de terceros que desconocemos.

```
let duda: any = 4
duda = "almacenamos un string"
duda = true
```

```
console.log(duda) // true
```

Funciones

En typescript las funciones que reciben o que devuelven parámetros también pueden tipar los datos tanto en un sitio como en otro.

```
function sumar(pNumeroA: number, pNumeroB: number): number {
    let resultado: number = pNumeroA + pNumeroB;
    return resultado;
}
```

Cuando una función no devuelve nada usamos el tipo de dato contrario a **ANY** que es **VOID**.

```
function saludo(): void{
    console.log("Hola Mundo!!")
}
```

Muchas veces necesitamos trabajar de manera concreta con variables de las cuales desconocemos su tipo.

Para poder decirle al compilador que «confíe en nosotros» y trate a las variables con el tipo que necesitamos, podemos hacerlo de dos maneras diferentes.

```
let valor: any = "Es una cadena"
```

```
let sizeValor: number = (<string>valor).length
let sizeValor2: number = (valor as string).length
```

19.4. Interfaces

Técnicamente un interfaz es un mecanismo de la programación orientada a objetos que trata de suplir la carencia de la herencia múltiple. Casi todos los lenguajes de programación que implementan la programación orientada a objetos no ofrecen la posibilidad de definir una clase que extienda de varias clases a la vez y muchas veces sería lo deseable. Y es aquí donde los interfaces cobran su uso.

Mediante las interfaces de TypeScript podemos definir contratos que, las clases que implementen dichos interfaces tendrán que cumplir.

De esta manera nos aseguramos de que las clases que implementan ciertos interfaces van a tener los mismos métodos y propiedades, aparte de las suyas propias.

```
interface IPersona{
    name: string
    getName(): string
}

class Persona implements IPersona{
    name: string
    getName(): string{
        return this.name
    }
}
```

Si en el ejemplo anterior, la clase Persona no cumple con alguno de los nombres o tipos de las propiedades o métodos definidos en el interfaz, obtendremos un error de Typescript indicándonos que algo va mal en nuestro desarrollo.

Como has podido ver los interfaces no son difíciles de definir y usar. En el mundo de la programación orientada a objetos se usan bastante y para nosotros suponen una ayuda a la hora de definir que métodos o que propiedades va a tener nuestra clase.

El uso de interfaces nos obliga a mantener buenas prácticas de programación y a cambio nos provee de una serie de mecanismos que nos permite detectar errores de una forma más fácil.

19.5. Decoradores

Un decorador es una función que, dependiendo de que cosa queramos decorar, sus argumentos serán diferentes.

Usan la forma `@expression` donde «expression» evaluará la función que será llamada. A continuación, explicaré los decoradores más frecuentes:

Decoradores de clase

Es aplicado al constructor de la clase y puede ser usado para observar, modificar o reemplazar la definición inicial de la clase. Su único argumento es target que vendría siendo la clase decorada, tipado como Function o any:

```
function Bienvenida(target: Function): void{
    target.prototype.saludo = function(): void{
        console.log("Hola Mundo")
    }
}
```

```
@Bienvenida
class Saludar{
    constructor(){}
}
```

```
let s = new Saludar()
s.saludo()
```

Si quisiéramos modificarlo, agregando un parámetro en la llamada al decorador, podríamos hacerlo de la siguiente manera

```
function Bienvenida(saludo: string){
    return function(target: Function){
        target.prototype.saludo = function(): void{
            console.log(saludo)
        }
    }
}
```

```
@Bienvenida("Saludando")
class Saludar{
    constructor(){}
}
```

```
let s = new Saludar()
s.saludo()
```

Podemos decorar de la misma manera, propiedades definidas en nuestras clases. Primero definimos la función decoradora.

```
function LogCambios (target: Object, key: string){
  var valueProp: string = this[key]
  if(delete this[key]){
    Object.defineProperty(target, key, {
      get: function(){
        return valueProp
      },
      set: function(newValue){
        valueProp = newValue
        console.log(`${key} es ahora ${valueProp}`)
      }
    })
  }
}
```

Posteriormente hacemos uso de dicho decorador

```
class Animal{
  @LogCambios
  name: string
}

var a = new Animal()
a.name = 'Pipo'
```

Tema 20. Programas y módulos

20.1. Introducción y objetivos

El objetivo de este tema es presentarte las partes y archivos en angular y que entiendas como se estructura una aplicación por módulos. Vamos a ver como se gestiona el módulo principal de nuestra aplicación y como nos ayuda a que todos los elementos de nuestros sistemas puedan saber y conocer que componentes y librerías pueden usar y donde tiene que ir a la hora de cargar cierto contenido.

La estructura general parte de un conjunto de librerías, las obligatorias, que componen el núcleo central de Angular y multitud de opcionales.

Podemos crear nuestras aplicaciones Angular mediante la creación de plantillas dentro de componentes propios, agregando servicios que manejen la lógica de nuestra aplicación y empaquetando nuestras creaciones dentro de módulos.

A partir de ahí, simplemente tendríamos que determinar cuál es el módulo raíz a partir del cual arrancaría nuestra aplicación.

20.2. Partes de una aplicación de Angular

Las aplicaciones de Angular son modulares, a partir de su propio sistema, denominado NgModules.

Por defecto, toda aplicación Angular tendrá como mínimo un módulo, el módulo raíz (**root**), conocido normalmente como AppModule.

Una aplicación pequeña lo normal es que esté constituida por un único módulo, pero lo habitual es que nuestras aplicaciones se dividan en diferentes módulos, haciendo referencia a cada una de las características que vamos a tratar.

Por lo general, vamos a ver, que la mayoría de los ejemplos realizados en clase tiene dos módulos, uno para la gestión de rutas y otro para la gestión de librería y componentes, aunque como hemos dicho puede haber más módulos.

Los **Módulos** de Angular se representan a partir del decorador `@NgModule`. Este decorador recibe la información de los metadatos que identifican el módulo con el que vamos a trabajar.

Las propiedades más importantes son las siguientes:

- ▶ **Declarations:** especifica las vistas contenidas dentro del módulo en cuestión. Tenemos 3 tipos de vistas diferentes: **componentes**, **pipes** y **directivas**.
- ▶ **Exports:** definimos aquellos componentes que serán visibles desde otros módulos externos.
- ▶ **Imports:** nos permite especificar qué módulos externos vamos a necesitar dentro del módulo que estamos desarrollando.
- ▶ **Providers:** lista de los servicios generados dentro de este módulo y que serán accesibles desde cualquier parte de la aplicación.
- ▶ **Bootstrap:** define la vista principal de la aplicación. Sería lo que podemos denominar el componente principal.

Un módulo puede agrupar diversos componentes relacionados entre sí.

Los **componentes**, son lo que en programación denominamos las vistas. Como casi todo en angular son clases que definen métodos y propiedades que usa la propia vista para mostrar un resultado por pantalla.

Pueden tener atributos tanto de entrada (decorador @Input) como de salida (decorador @Output). También podemos definir al componente como:

Componente = template + metadatos + clase

Los **templates** definen la vista de un componente, el html y el css que va a usar el componente para mostrar el resultado.

Los **decoradores** son funciones que se colocan delante de la definición de una propiedad, clase a función y que ofrecen la posibilidad de configurar el comportamiento de esa clase, propiedad o función a la que decoran, son los **metadatos** de los componentes.

El **Data Binding** nos permite pintar datos o intercambiar datos con el template y la clase del componente, es la forma en la que comunicamos la parte de la lógica con la parte de la vista y podemos mostrar el resultado.

Existe cuatro tipos de **Data Binding** que durante el curso iremos explicando con mayor profundidad.

Veamos un poco por encima en qué consisten:

- ▶ **Interpolación:** nos permite pintar dentro del html cualquier valor que este definido dentro de una propiedad de clase de nuestro componente. Debemos definir esta propiedad e inicializarla dentro de nuestra clase de componente, de esta forma podremos pintar dicho valor a través de las dobles llaves de esta manera {{propiedad}}.
- ▶ **Event binding:** Nos permite ejecutar un evento desde el HTML de nuestro componente, de esta forma (click) = "hacerClick()". Dicha función deberá estar dentro del .ts del propio componente y se ejecutará siempre y cuando el

usuario interaccione con el interfaz, en este caso haciendo click. Nos permite comunicar información desde el HTML hacia el TS.

- **Property binding:** Es la forma que tenemos de asignar a una propiedad de HTML el valor de una propiedad creada dentro de nuestro componente.

[src] = “ propiedad ”

Es como realizar una interpolación dentro de un atributo de HTML, permitiéndome insertar cualquier sentencia JavaScript valida dentro de los atributos del HTML.

- **Two-way binding:** Me permite comunicar HTML y TS del componente de forma bidireccional, es decir, si el valor cambia en el html se guarda en el TS y biceversa.

Las **directivas** permiten añadir comportamiento dinamico al html mediante un atributo o selector. Existe dos directivas estructurales `*ngIf` y `*ngFor`, y otras de atributo que modifican el aspecto visual de un elemento del DOM, estas son `[ngClass]` y `[ngStyle]`. También podemos tener directivas mixtas que tienen parte estructural y parte de atributo como puede ser `ngSwitch`.

Los **servicios** son clases que permiten comunicar componentes entre sí y además obtener datos externos a través de peticiones por HTTP un servidor. Se encarga de distribuir la información y segmentarla a través de funciones para que puedas pedir cualquier dato que necesite nuestros componentes.

Los **injectables** o lo que se conoce como inyección de dependencias, permite suministrar funcionalidades a un componente o a un servicio sin que este tenga que crearlos. Usan el decorador `@Injectable`. Están pensados para proporcionar funcionalidades y modularizar el código.

20.3. Módulo principal

Al crear una aplicación, por defecto se crea el módulo `app.module.ts`. Veamos como es el módulo principal y luego estudiemos cada una de sus partes.

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

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

En este módulo se puede apreciar varias partes.

Los **imports**:

- ▶ **BrowserModule**: nos permite mostrar la aplicación dentro de los navegadores y nos asegura retrocompatibilidad.
- ▶ **NgModule**: Es un decorador que nos marca que la clase que estamos visualizando es un módulo.
- ▶ **AppRoutingModule**: Es el módulo que nos permite la gestión de rutas en Angular.
- ▶ **AppComponent**: componente principal del módulo.

El decorador `@NgModule` tiene tres array de elementos que nos permite dar de alta los diferentes componentes y librerías que necesitará mi aplicación:

- ▶ **Declarations:** En este array se cargan los diferentes componentes que creemos dentro de nuestra aplicación, por defecto se carga siempre el componente principal.
- ▶ **Imports:** En este array se cargan todas las librerías que Angular necesite para trabajar y que no estén cargadas de forma inicial en la aplicación, por ejemplo, `BrowserModule`.
- ▶ **Providers:** Aquí podemos cargar elementos como los servicios, pero inicialmente no carga nada.
- ▶ **Bootstrap:** Nos marca que componente es **el principal en este caso** `AppComponent`.

Y el último la clase `AppModule` que permite cargar nuestra aplicación.

20.4. Arquitectura por módulos

Angular es un framework para la creación de aplicaciones de cliente basadas en HTML y JavaScript (o algún lenguaje que pueda compilarse en JavaScript, como es TypeScript).

La estructura general parte de un conjunto de librerías, las obligatorias, que componen el núcleo central de Angular y multitud de opcionales.

Podemos crear nuestras aplicaciones Angular mediante la creación de plantillas dentro de componentes propios, agregando servicios que manejen la lógica de nuestra aplicación y empaquetando nuestras creaciones dentro de módulos.

A partir de ahí, simplemente tendríamos que determinar cuál es el módulo raíz a partir del cual arrancaría nuestra aplicación.

A la hora de organizar nuestro código deseamos que nuestra aplicación sea escalable y limpia, y en esa escalabilidad nos ayuda a encontrar una estructura de carpetas adecuada para nuestro proyecto.

El tener una estructura bien definida nos ayuda a ubicar los diferentes elementos que vamos a ir creando, y nos da una mejor visibilidad.

Cada estructura varía, como cada proyecto o aplicación, antes de crear nuestros componentes, módulos, servicios, etc., debemos tener claro cuál será la responsabilidad que este va a tener.

De acuerdo con nuestro proyecto podemos tener diferentes formas de organizar nuestros archivos, sea crear módulos por cada nueva funcionalidad que vaya a tener nuestra aplicación, o incluso crear módulos, donde agrupamos ciertos elementos de acuerdo con su uso, si es común, o compartido.

Podemos crear un módulo principal de la aplicación del cual dependan otros módulos como por ejemplo el de rutas y a partir de ahí hacer un desarrollo centrado en los componentes.

20.5. Creación de módulos

Para facilitar las tareas a la hora de generar las partes de un proyecto de Angular ya hablamos de la herramienta Angular CLI y que ya hemos usado para generar un proyecto. El comando que vamos a usar ahora para generar un módulo nuevo es «generate» y a continuación tenemos que indicar que es lo que se quiere generar ya que dicho comando lo usaremos a lo largo del curso para generar componentes, servicios, etc.

El comando que me permite generar un módulo es:

```
ng generate module nombre
```

module nombre equivaldría a la ruta donde queremos guardar el módulo que estamos creando, que normalmente es `src`. Dentro de esta carpeta podemos crear otra carpeta. Este comando se debe aplicar dentro de la terminal del proyecto.

Tema 21. Componentes

21.1. Introducción y objetivos

Una de las máximas en Angular es que **todo es un componente**. Representan la manera que tenemos en Angular para definir las unidades que vamos a visualizar en nuestras páginas y que contendrán la lógica de estas.

Por norma general, nuestras aplicaciones Angular estarán conformadas por un **árbol de componentes** relacionados entre ellos. Según el equipo de desarrollo de Angular:

«Un componente controla una parte de la pantalla llamada vista y se encarga de definir los diferentes bloques reutilizables por la interfaz de usuario de nuestras aplicaciones».

El objetivo principal que queremos cubrir en este tema es que entendamos cual es la importancia del desarrollo de componentes en Angular y como plantearnos la creación de un sistema basado en componentes.

21.2. ¿Qué es un componente?

Un componente es una clase a la que le precede un decorador `@Component` y que determina que ficheros conforman el componente. Dicho decorador suele determinar que un componente tiene una etiqueta que nos sirve para activar el componente dentro de otro o del componente principal.

Básicamente los componentes en Angular se organizan en una estructura de árbol teniendo un componente principal del cual depende todos los demás.

El trabajo con componentes en Angular dispone de dos partes fundamentales:

- ▶ **La definición del componente como elemento html.** En este momento queda definido cómo vamos a acceder a dicho componente, la plantilla que lo representa y demás elementos relacionados con su aspecto.
- ▶ **El desarrollo de la lógica del componente.** Todos los componentes llevan asociada una clase donde vamos a encontrar las acciones que van a poder llevar a cabo cada uno de los elementos declarados dentro de nuestro Componente.

22.2. Partes de un componente

Antes de ponernos a definir las partes de un componente veamos un componente tipo para poder hacernos una idea de cada una de sus partes.

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

@Component({
  selector: 'app-primer-componente',
  templateUrl: './componente1.component.html',
  styleUrls: ['./componente1.component.css']
})
export class Componente1Component implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

Las primeras sentencias de nuestra clase definen todos aquellos import que nos permiten acceder a aquellas librerías que vamos a utilizar dentro de la clase.

Según vayamos conociendo o creando nuevas clases relacionadas con Angular y necesitemos usarlas dentro de nuestras clases, debemos agregar los `import` correspondientes.

En este caso únicamente estamos importando la librería `@angular/core`, donde encontramos la base para la creación de componentes en Angular

Por ejemplo, es donde se encuentra definido el decorador `@Component`. Mediante el decorador `@Component` especificamos los metadatos que van a caracterizar este componente.

El decorador se debe aplicar sobre la clase donde posteriormente implementaremos las acciones de nuestro componente.

Un componente se puede componer de tres partes:

¿Qué son las anotaciones? son los metadatos que agregamos a nuestro código y que permite definir ciertas propiedades, las más importantes son:

- ▶ **Selector:** Se trata de la propiedad que define cómo vamos a hacer referencia a este componente dentro del DOM de nuestra página. El nombre debe ir en minúsculas y separado por guiones en caso de que contenga más de una palabra. Cuando queramos utilizar nuestro componente, lo especificaremos de la siguiente manera:

```
<app-primer-componente></app-primer-componente>
```

- ▶ `template / templateUrl`: Con estas propiedades especificamos el contenido que va a renderizar nuestro componente cuando en el DOM situemos el selector que lo define.

El primero, define el contenido de manera INLINE, mientras que el segundo nos permite concretar un fichero .html donde encontraremos la plantilla que vamos a renderizar.

- `styles / styleUrls`: Podemos definir los estilos asociados a nuestro componente a través de estas dos propiedades. Mediante la propiedad `style` podemos definir los estilos de nuestro componente en línea.

Debemos trabajar como si estuviésemos generando los estilos para cualquier página, es decir, establecer clases, modificar elementos a través de su identificador o a través de su tag.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-primer-componente',
  template: `<h1>Hola Angular 2</h1>`,
  styles: [
    h1 { color: #fff; }
    .description {
      font-style: italic;
      color: green;
    }
  ]
})
export class AppComponent { }
```

Podemos incluso dividir los diferentes estilos en cadenas de caracteres contenidas dentro del array de `styles`.

La parte negativa de usar esta nomenclatura es que la definición de los parámetros de nuestro componente se vuelve demasiado compleja e ilegible, lo que entorpece las posibles tareas de mejora y escalabilidad.

Mediante la propiedad `styleUrls` podemos permitirnos definir los estilos asociados a un componente en un fichero `.css` externo.

Al ser un array de literales, nos permite definir tantas hojas de estilo como sean necesarias para nuestros componentes.

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

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

Aparte de las dos propiedades de estilos que tenemos disponibles en la definición de los metadatos de `@Component`, podemos definir los estilos de nuestros componentes de dos maneras diferentes.

La primera sería agregando la etiqueta `<style>` delante del html que vamos a renderizar con el componente.

De igual manera, podemos asignar el atributo `style` a cualquiera de los elementos que coloquemos dentro de nuestra plantilla.

21.3. Cómo crear un componente

Generalmente, un componente está formado por cuatro archivos, aunque uno de ellos no es obligatorio y se puede omitir.

- ▶ `app.component.html`: Es la vista que contiene el HTML del componente y donde se van a realizar otras las funcionalidades de renderizado.
- ▶ `app.component.css`: Define la hoja de estilos de un componente. Nos permite repetir otras etiquetas que hayamos usado dentro de otros componentes ya que angular les añade una capa numérica a las hojas de estilos y así no se repiten.
- ▶ `app.component.ts`: Esta escrito en TypeScript y es el **controlador** de nuestra aplicación. Es donde está definido toda la lógica de negocio de nuestro componente y donde implementaremos todos los métodos necesarios para que nuestro componente funcione.
- ▶ `app.component.spec.ts`: los ficheros `spec.ts` nos sirven para hacer test unitarios y que nuestro resultado sea más fiable.

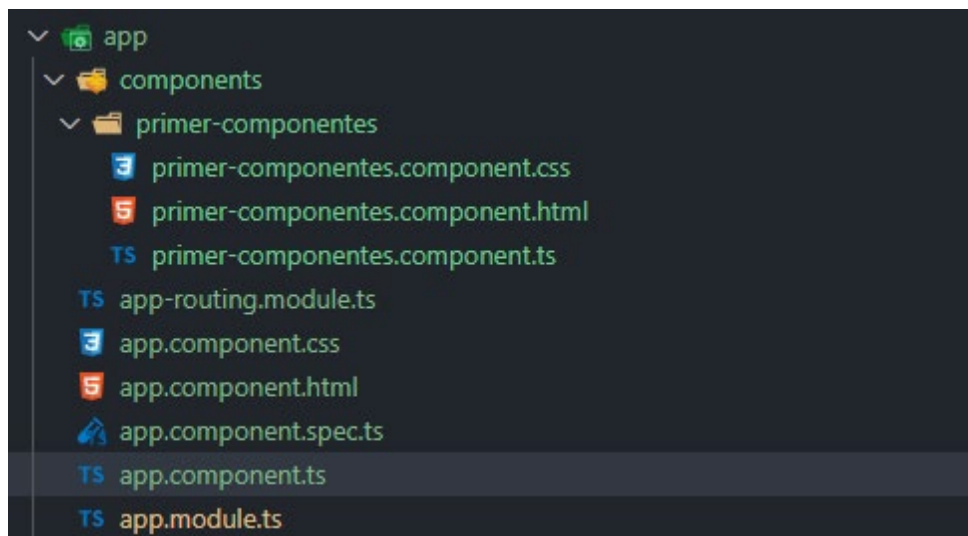


Figura 9. Árbol de carpetas de un componente. Fuente: elaboración propia.

Para crear un componente nos vamos a servir de la herramienta angular CLI que instalamos al principio de este tema y que entre otras muchas cosas nos va a servir para crear componentes y otros elementos de una aplicación en angular.

Para crear un componente escribimos en la terminal elegida el comando:

```
ng generate component components/nombreComponente --skip-tests
```

También tienes un formato corto para este script en terminal:

```
ng g c components/nombreComponente
```

El skip-tests sirve para evitar la creación del fichero de test en cada component usará siempre que quieres que dicho fichero no aparezca en la carpeta del componente.

21.4. Componente principal y componentes hijos.

¿Cómo se enlazan?

Una de las herramientas más potentes dentro de Angular es la posibilidad de **anidar componentes**.

Para crear una aplicación compleja, podemos implementar los componentes de la manera más sencilla posible para posteriormente, combinarlos entre sí.

Si queremos utilizar un componente dentro de la estructura html de otro, lo podemos hacer incluyendo el selector del componente hijo dentro del espacio donde queramos que se renderice del componente padre

Para que esto sea posible, no debemos olvidar **declarar nuestros componentes dentro del módulo donde estemos trabajando**.

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

import { AppRoutingModule } from './app-routing.module';
```

```

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

//aquí importamos el componente
import { PrimerComponentesComponent } from './components/primer-
componentes/primer-componentes.component';

@NgModule({
  declarations: [
    AppComponent,
    // importantísimo añadirlo al array de declarations, esto lo hace por
    // nosotros angular cli pero debemos comprobarlo.
    PrimerComponentesComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Por último, utilizamos el componente hijo dentro de la plantilla del padre. Recordamos que ahora mismo solo tenemos un componente padre que es `app.component.ts`. Pero esto es extrapolable a otros componentes no solo al componente principal.

Enlazar ambos componentes tenemos que poner la etiqueta del componente hijo dentro del fichero `.html` del padre. De esta forma:



Figura 10. Cómo se enlazan dos componentes. Fuente: elaboración propia.

Esta forma de enlazar componentes dentro de otros componentes. Si ahora intentamos levantar nuestra aplicación para ver el resultado, solo tenemos que hacer el siguiente comando en la terminal.

```
ng serve -o
```

Esto arrancará nuestra terminal y el resultado que veremos será el siguiente, ya que dentro de nuestro componente principal habremos cargado el componente primer-componente visualizándose de la siguiente manera.

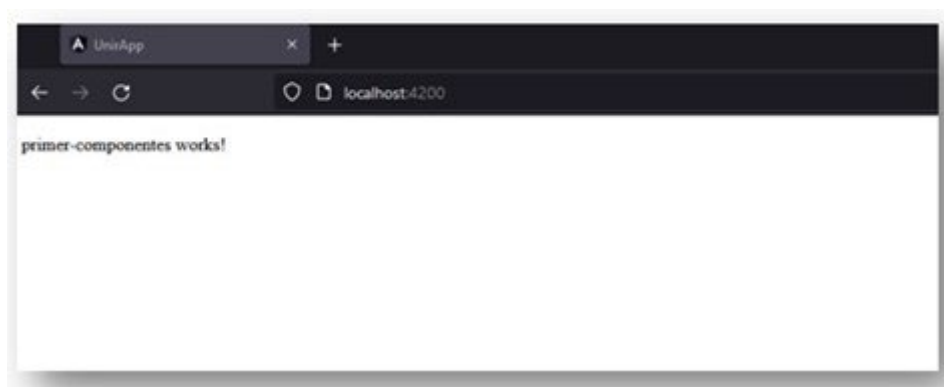


Figura 11. Resultado de la carga del primer componente. Fuente: elaboración propia.

Si modificamos algo del componente, por ejemplo, poner un título en el html del propio componente, de esta forma:

```
<h1>Titulo del primer componente</h1>
<p>Primer parrafo</p>
<p>Segundo parrafo</p>
```


El resultado que saldrá por pantalla tendrá la siguiente pinta:

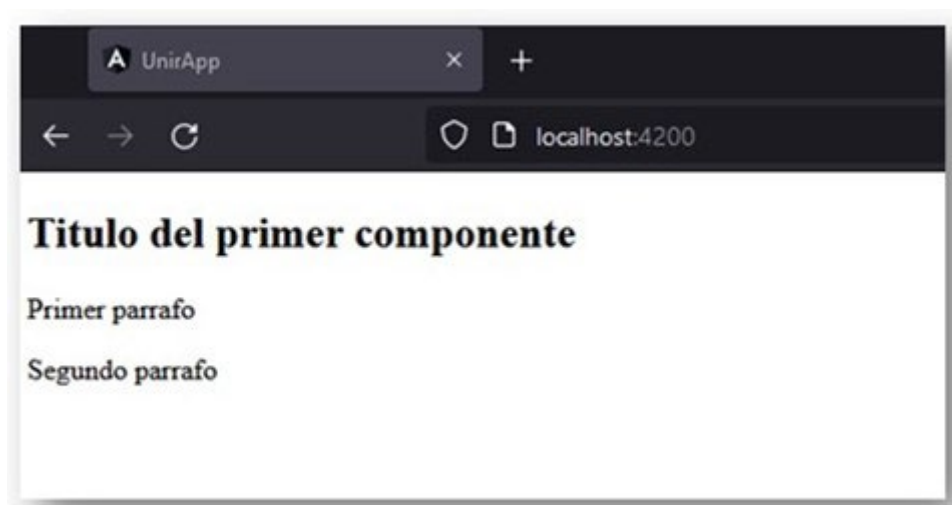


Figura 12. Resultado del navegador. Fuente: elaboración propia.

Simplemente hemos incluido algo de maquetación en html dentro del componente sin ninguna funcionalidad, pero el resultado igualmente se visualiza en pantalla.

En el siguiente tema veremos como enlazar el html y css de cada componente a través de su sistema de templating.

En este vídeo, *Creación de componentes*, vamos a ver todos los pasos para la creación de componentes, vamos a crear varios componentes y los vamos a enlazar entre ellos para que se comprenda bien cuál es el proceso y todas sus partes.



Accede al vídeo

21.5. Funciones de ciclo de vida de un componente

Todos los componentes de Angular disponen de una serie de métodos que podemos implementar y que se ejecutan automáticamente en determinados momentos del ciclo de vida del componente, dándonos la posibilidad de interactuar con cada uno de estos momentos.

De esta manera seremos capaces de realizar diferentes acciones desde la creación del componente hasta su destrucción.

El siguiente gráfico muestra el orden de ejecución de los diferentes métodos del ciclo de vida de un componente.



Figura 13. Orden de ejecución del ciclo de vida de un componente. Fuente: elaboración propia.

Todas ellas tienen un tiempo de ejecución al que podemos engancharnos para ejecutar algunas de nuestras acciones dentro del componente, algunas incluso se solapan y pueden llegar a confundirse por que se ejecutan al mismo tiempo.

Vamos a hacer un repaso por las más importantes y que vamos a usar la mayoría de las veces en nuestros desarrollos.

Constructor

Es el método que carga la lógica y he inicializa las propiedades del componente.

ngOnChanges

Este método es llamado cuando se aprecia algún cambio en cualquiera de las propiedades que recibe nuestro componente. Vemos un pequeño ejemplo de implementación.

```
ngOnChanges(changes: SimpleChange) {  
  for (let propName in changes) {  
    let chng: any = changes[propName];  
    let cur = JSON.stringify(chng.currentValue);  
    let prev = JSON.stringify(chng.previousValue);  
    console.log(`${propName}: ValorActual = ${cur}, ValorPrevio =  
    ${prev}`);  
  }  
}
```

Estos cambios se aprecian siempre y cuando modifiquemos el valor que le estamos pasando al componente desde el elemento donde lo tenemos definido.

Estos cambios únicamente se observan en un primer nivel.

Es decir, si estamos recibiendo un objeto como valor de un input, las modificaciones que se realicen dentro de los atributos de ese objeto no serán detectadas por el método.

ngOnInit

Ocurre tras la primera ejecución de ngOnChanges.

Es un método que solo ocurre una vez cuando el componente se carga por primera vez. Es un método que está implementado en el interfaz `OnInit` y que inicializa no solo la lógica del componente si no también la parte del HTML.

Implementamos este método por dos razones:

- ▶ Para realizar acciones complejas justo después del constructor.
- ▶ Para configurar el componente después de haber definido los inputs de este.

En el constructor no se deben hacer tareas costosas como la obtención de datos para el componente o la llamada a servicios. Este método sería el lugar idóneo para llevar a cabo estas tareas.

Este método se ejecuta prácticamente a continuación de la creación del componente.

`ngDoCheck`

Hemos visto cómo podemos utilizar el método `ngOnChanges` para recibir los cambios en las diferentes propiedades de nuestros componentes. Estos cambios no se reflejan si las propiedades con las que trabajamos son objetos complejos o arrays. Para ello podemos implementar el método `ngDoCheck`, el cual nos permite trabajar con una comprobación personalizada de los datos.

Este método se lanza después de detectar cualquier tipo de cambio en los datos de nuestro componente. No se debe abusar del uso de este método ya que puede afectar al rendimiento de nuestra aplicación

`ngAfterContentInit`

Se lanza **después de inicializar el contenido del componente**.

`ngAfterContentChecked`

Se ejecuta **tras cada comprobación del contenido del componente**. Comprueba el contenido visualizado en el componente.

`ngAfterViewInit`

Se ejecuta **después de que las vistas del componente se inicialicen** y después de `ngAfterContentChecked`.

`ngAfterViewChecked`

Se ejecuta **después de cada comprobación de la vista de un componente**. Se llama después de `ngAfterViewInit` y de cada `ngAfterContentChecked`.

`ngOnDestroy`

Se trata del método donde colocaremos las **acciones de limpieza de nuestro componente**.

En este momento deberíamos avisar al resto de componentes, si fuera preciso, que el componente actual va a desaparecer. También es el lugar adecuado para liberar recursos utilizados durante el ciclo de vida del componente.

Hay que tener en cuenta las posibles fugas de memoria que pueda provocar nuestro componente.

Tema 22. Templating

22.1. Introducción y objetivos

Nos encontramos ante un tema corto, pero bastante visual, ya que en el vamos a intentar cubrir como pasamos de un HTML y un CSS a un componente de angular.

En los primeros temas de HTML y CSS aprendimos todo lo necesario para maquetar de forma estática una web, y con JavaScript le dimos funcionalidad.

En este tema vamos a ver cómo podemos juntar esto en Angular y así poder dar vistosidad a nuestros desarrollos, uniendo nuestra lógica de programación con nuestro HTML y dándole estilos a través de CSS.

Este es el principio de una serie de temas donde vamos a hacer interactuar a nuestra lógica con nuestras vistas, siguiendo el paradigma de programación de MVC, que comentamos al principio en el tema de introducción.

22.2. Sintaxis de plantillas

Hasta el momento, las plantillas que hemos creado para nuestros componentes eran totalmente estáticas.

Angular nos ofrece la posibilidad de ampliar nuestras plantillas e incorporar elementos dinámicos.

Uno de los conceptos más importantes de este **framework** es cómo podemos hacer para enlazar los elementos creados en la clase de nuestro componente con la plantilla que finalmente renderizamos.

Vamos a ver las como comunicamos el archivo `.ts` con el `.html`.

Ya hemos visto que nuestro componente tiene un decorador que asigna, el selector con el que se carga el componente, el CSS que da los estilos al componente y el HTML que renderiza.

En este tema vamos a analizar uno de los primeros elementos del Data Binding, este es uno de los conceptos más importantes en Angular para definir la comunicación entre el componente y el DOM (archivo HTML). Este concepto hace muchos más fácil el pintar los datos variables dentro del HTML simplificando muchísimo el uso del DOM con respecto al uso de JavaScript nativo. El Data Binding son un conjunto de herramientas que Angular nos provee para estos menesteres.

De tal forma que no solo vamos a poder pintar datos en el HTML si no variar estilos y propiedades del DOM de una forma más sencilla.

La comunicación dentro del componente se puede realizar en varios sentidos y dependiendo de este tenemos diferentes tipos de elementos.

Dentro del Data Binding hay tres tipos diferentes de comunicación entre plantilla, lógica y estilos. Veamos los tres modos y en los próximos temas analizaremos en profundidad cada uno de ellos.

- ▶ One Way Data Binding (del componente al DOM): **Interpolación, Property Binding y Class Binding.**
- ▶ One Way Data Binding (del DOM al componente): **Event Binding y \$event.**
- ▶ Two Way Data Biding (del componente al DOM y viceversa): **FormsModule y [(ngModel)].**

En este primer tema vamos a hablar de la interpolación como la forma más sencilla de visualizar propiedades de un componente en su template. Es la capacidad que tenemos de incluir datos dentro de nuestras plantillas.

Para realizar una interpolación solo necesitamos crear una propiedad dentro de la clase del componente. Este enlace se realiza incluyendo el código dinámico dentro de los símbolos `{{ }}`.

Es la estructura determinada por las llaves podemos incluir cualquier sentencia JavaScript válida excepto:

- ▶ Asignaciones.
- ▶ Creación de nuevas variables.
- ▶ Expresiones encadenadas.
- ▶ Incrementos/decrementos.

Cualquier sentencia que incluyamos dentro de las llaves será ejecutada como si de JavaScript se tratase.

Pero sin duda, el uso más habitual para esta técnica es la de mostrar el valor de las diferentes propiedades contenidas dentro de la clase del componente

El enlace entre la clase y la plantilla es directo y, por lo tanto, a través del nombre de la propiedad, podemos renderizar su valor en la plantilla.

Veamos un ejemplo:

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

@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})
```



```

}))
export class PrimerComponentesComponent implements OnInit {

    //creamos una propiedad de clase, es como la creación de una variable pero
    sin la palabra let.

    titulo: string;
    //inicializamos en el mismo punto de la creación
    numero: number = 2;
    constructor() {

        //las propiedades hay que inicializarlas y podemos hacerlo en el
        constructor o en la propia declaración de la propiedad. Para referirnos a
        la propiedad dentro de la clase usamos la palabra this.
        this.titulo = 'Titulo del Componente'

    }

    ngOnInit(): void {
    }

}

```

En el ejemplo de arriba vemos que hemos creado dos propiedades **título** y **numero**. Las propiedades se declaran por encima del constructor como si de dos variables globales se tratasen, pero sin usar la palabra `let`, `var`, `const` que usábamos en JavaScript.

Dichas propiedades aparte de declararla y asignarles un tipo, tenemos la obligación de inicializarlas, y esta tarea la podemos realizar en dos zonas, o en la misma declaración de la propiedad, asignándole un valor por defecto o en el constructor de la clase donde tendremos que usar la palabra reservada **this** para referirnos a la propia propiedad y asignarle un valor.

A través de esta propiedad y la interpolación podemos visualizar el valor de estos datos dentro del HTML de la siguiente forma.

```
<h1>{{ titulo }}</h1>
<p>{{ numero }}</p>
<p>Segundo</p>
```

De esta forma el resultado que aparecerá en nuestra pantalla será el siguiente.

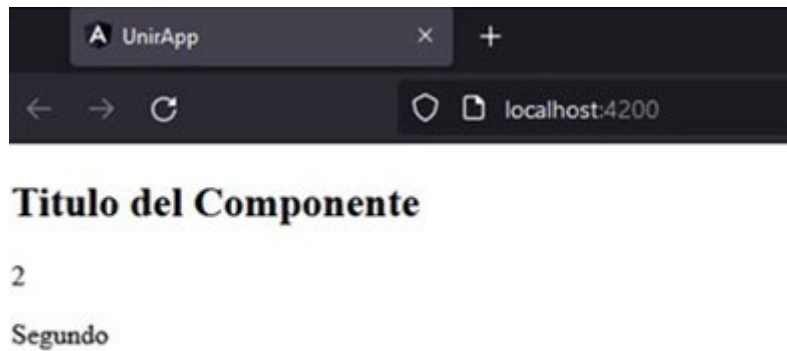


Figura 14. Renderizado de la interpolación de título y número. Fuente: elaboración propia.

Aunque es la forma más sencilla de comunicar valores variables entre la lógica y el html, también podemos hacerlo a través de métodos dentro del componente.

Podemos declarar una función dentro del componente que devuelva un texto y dicha función pintarla como resultado dentro de nuestro HTML. Veamos su ejecución.

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

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

  //creamos una propiedad de clase, es como la creación de una variable pero
  //sin la palabra let.

  titulo: string;
  //inicializamos en el mismo punto de la creación
```

```

numero: number = 2;
constructor() {
    //Las propiedades hay que inicializarlas y podemos hacerlo en el
    constructor o en la propia declaración de la propiedad. Para referirnos a
    la propiedad dentro de la clase usamos la palabra this.
    this.titulo = 'Titulo del Componente'

}

ngOnInit(): void {
}

getParrafo(): string {
    return 'Este parrafo me esta llegando del return de una función dentro
de mi componente.'
}
}

```

Luego la pintamos en el HTML del componente de la siguiente forma:

```

<h1>{{ titulo }}</h1>
<p>{{ numero }}</p>
<p>{{ getParrafo() }}</p>

```

La función `getParrafo()` devuelve un texto que se renderiza en la interpolación del HTML dando como resultado lo siguiente.



Figura 15. Renderizado de un return de la función. Fuente: elaboración propia.

Cuando el resultado que queremos pintar tiene etiqueta HTML la cosa cambia un poco por qué no podemos usar la interpolación como tal si no que debemos usar la propiedad `innerHTML` del DOM, aunque en este caso lo haremos de forma un poco distinta a como se hace en JavaScript.

Pongamos un ejemplo, vamos a crear una propiedad y un método que contengan un texto con etiquetas HTML en su interior de la siguiente forma.

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

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

  parrafo: string = 'En un lugar de la <strong>mancha</strong>'

  constructor() {}

  ngOnInit(): void {}

  getContenido(): string {
    return `Este contenido es un template literal y tiene un listado
      <ul>
        <li>opción 1</li>
        <li>opción 2</li>
      </ul>
    `;
  }
}
```

La propiedad solo tiene una etiqueta `strong` y el método tiene un template literal que renderiza un listado no numerado.

Si usásemos una interpolación para pintar estos elementos de esta forma.

```
<p>{{ parrafo }}</p>
<p>{{ getContent() }}</p>
```

El resultado obtenido no sería el esperado ya que las etiquetas se verían como si de texto se tratase, obteniendo este resultado.



Figura 16. Renderizado erróneo de contenido HTML dentro de la plantilla. Fuente: elaboración propia.

Como podéis ver las etiquetas forman parte del contenido y no son interpretadas por el navegador de forma correcta.

Pero ¿cómo podríamos visualizar el contenido de esos elementos dentro del HTML? Pues usando la propiedad de Angular `[innerHTML]` dentro de la etiqueta del HTML de la siguiente forma.

```
<p [innerHTML]="parrafo"></p>
<p [innerHTML]="getContent()"></p>
```

Asignamos como atributo del párrafo dicha propiedad o igualamos al valor que retorna la función para así obtener ahora este resultado. Ahora si visualizándose el contenido HTML de forma correcta.



Figura 17. Renderizado correcto de contenido HTML dentro de la plantilla. Fuente: elaboración propia.

En el siguiente vídeo, *Estilos de plantilla en Angular*, se explica la creación de un componente donde aplicamos estilos de plantilla para ver cómo generar un interfaz dentro con varios componentes. Solo visual no funcional.



Accede al vídeo

22.3. Hojas de estilos de componente

A la hora de trabajar con estilos de CSS Angular también nos aporta soluciones bastante interesantes.

La primera es que al tener una hoja de estilos por componente los estilos de este son independientes de otros componentes, lo cual hace que no nos tengamos que preocupar del nombre de las etiquetas de otro componente porque Angular las hace independientes.

Si queremos que algún estilo afecte a todos los componentes tenemos la hoja de estilos principal `style.css`.

En esta hoja de estilos podemos cargar los estilos que van a ser comunes y que queremos que afecten a todos los componentes.

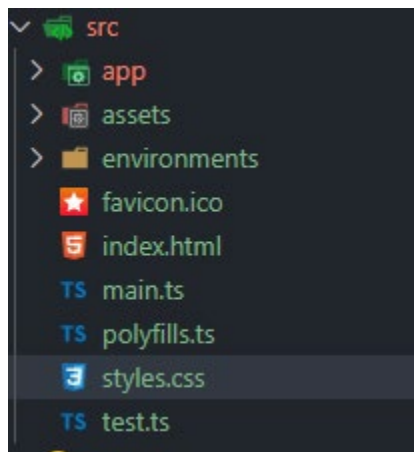


Figura 18. Style.css. Fuente: elaboración propia.

Por el contrario, si usamos las hojas de estilos propias de cada componente, da igual que estos sean hijos de otros, dichos estilos solo afectaran al componente al que pertenezca esa hoja de estilos, y exclusivamente a él.

Esto hace que no tengamos que estar pensando en si ese nombre de clase que hemos puesto ya lo hemos usado en otro componente y sobre todo permite que si dos desarrolladores trabajan en distintos componentes y usan clases iguales su resultado no se vea afectado por el desarrollo del otro.

Uso de Sass

Dentro de Angular también podemos usar Sass como preprocesador de CSS. Solo hay que tener en cuenta ciertos parámetros.

El primero es que cuando creamos el proyecto debemos marcar que este va a ser con SASS o con SCSS según preferencia. Al elegir una de estas opciones nuestras hojas de estilo CSS desaparecerán para dejar paso a SCSS tanto como hojas generales como dentro de los componentes.

```
C:\Users\janto\proyectos\cursos
λ ng new sass
? Would you like to add Angular routing? Yes
? 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 ]
```

Figura 19. Menú de elección de SCSS. Fuente: elaboración propia.

El propio compilador de Angular se encargará de transformarlas a CSS y que podemos ver el resultado en tiempo real dentro de nuestro localhost.

Otro punto importante a la hora de trabajar con SCSS en Angular es que tenemos que configurar el angular.json para decirle donde se van a alojar los ficheros partials de SCSS dentro de nuestra aplicación. Lo normal para que estén accesibles para todos los componentes que los importen es en la carpeta assets, de la siguiente forma. Damos de alta justo debajo de script una propiedad stylePreprocessorOptions que contiene otro propiedad donde se incluyen la ruta donde van a ser alojados los archivos partials de CSS, esos que empiezan por guión bajo _variables.scss, etc.

```
"styles": [
  "src/styles.css"
],
"scripts": [],
"stylePreprocessorOptions": {
  "includePaths": ["assets/partials"]
}
```

A partir de colocar dentro de esa carpeta todos los archivos parciales de angular, cada componente los podrá importar de forma personal el que necesite en cada momento y de forma independiente.

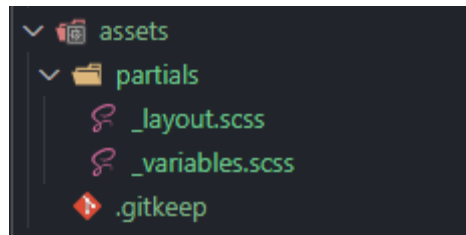


Figura 20. Sistema de carpetaas SCSS dentro de un proyecto. Fuente: elaboración propia.

Conclusión

De esta forma podemos dar estilos a nuestros componentes ya sea a través de CSS puro y duro como a través de SASS, trabajando de forma individual con cada componente cuando sea necesario y de forma genérica a través de la hoja principal ya sea `style.css` o `style.scss`.

El uso de cualquiera de estas tecnologías de estilo dependerá del equipo de trabajo ya que son tecnologías muy parejas pero que ofrecen ventajas e inconvenientes a la hora de maquetar. Conocerlas y usarlas bien es fundamental para obtener un resultado óptimo visualmente en nuestros desarrollos.

Tema 23. Data Binding: One-way and Two-ways

23.1. Introducción y objetivos

Este es el segundo tema donde hablamos del Data Binding, como parte importante de la comunicación entre la lógica y la vista del componente.

En el tema anterior hablamos del One Way Data Binding a través de la interpolación de contenidos. En este vamos a continuar con el **One Way Data Binding**, pero esta vez modificando propiedades de las etiquetas HTML a través de propiedades y funciones. Es lo que se llama el Property Binding y me permite modificar aspectos visuales de mis plantillas

Otra parte importante de este tema también vamos a tratar con la comunicación bidireccional entre lógica y HTML a través del **Two Way Data Biding**.

En definitiva, con este tema cerramos el contenido que explica como un componente de angular conecta la lógica de los archivos TypeScript con el HTML y CSS de nuestro componente, consiguiendo así un manejo del DOM que nos facilita muchísimo el trabajo dentro de nuestra aplicación de Angular.

23.2. Property Binding

Cuando necesitemos enlazar propiedades de componente a atributos del HTML debemos usar Property Binding de la misma forma que usamos la interpolación para

meter valores dentro del contenido del HTML (que también podría usarse en estos casos).

La forma de implementarlo es la siguiente:

Le asignas los corchetes al atributo del DOM y al otro lado de la igualdad el valor de la propiedad del TS de la siguiente forma.

```
[atributoDOM] ="propiedadTs"
```

La asignación seguiría siendo igual que si utilizamos un literal, pero en este caso, el valor lo obtendremos de evaluar el contenido situado entre comillas.

Este contenido podría ser una de las propiedades de nuestra clase, la ejecución de un método o incluso una sentencia TypeScript.

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

@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})
export class PrimerComponentesComponent implements OnInit {
  ruta: string = 'http://www.google.es'
  constructor() { }
  ngOnInit(): void { }
}
```

En el ejemplo que tenemos arriba asignamos a una propiedad de clase ruta el valor de una url con destino a Google. Si queremos asignar dicho valor a una propiedad dentro del HTML, lo haremos de esta forma.

```
<a [href]="ruta">IR A GOOGLE</a>
```

De tal forma que cuando renderizamos el resultado ese enlace nos lleva de a Google de manera directa. Esto podría producirse con cualquier propiedad de HTML ya sea `src` de imagen, texto alternativo `alt`, `id`, `class`, `title`, en definitiva, cualquier propiedad del HTML puede tener asignada un valor variable en el componente.

Si eliminamos los corchetes, toma el contenido entre comillas como un literal. Con lo que el resultado esperado en el caso del ejemplo anterior es que pusiese dentro del `href="ruta"` en lugar de `href=http://google.es`

Podemos conseguir el mismo efecto con la siguiente sintaxis `href = {{ titulo }}`, aunque desde la documentación oficial de Angular, nos explican que debemos evitar esta forma de aplicarlo e intentar acostumbrarnos a la forma de corchetes ya que el resultado funciona mejor dentro del componente.

Class Binding

Las clases son propiedades también y pueden funcionar de manera muy similar a la de los ejemplos anteriores.

Es decir, podemos tener dentro del typescript una propiedad `color` que se la asigne un valor rojo que corresponda al valor de una clase de `css` de la siguiente manera.

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

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

  ruta: string = 'http://www.google.es'
  color: string = 'rojo'
  constructor() { }
```

```
ngOnInit(): void { }

}
```

Al llevar esta propiedad a nuestro HTML usaremos los [] como forma de asignar el valor de la propiedad a la clase:

```
<a [href]="ruta" [class]="color">IR A GOOGLE</a>
```

Consiguiendo como resultado los siguiente:

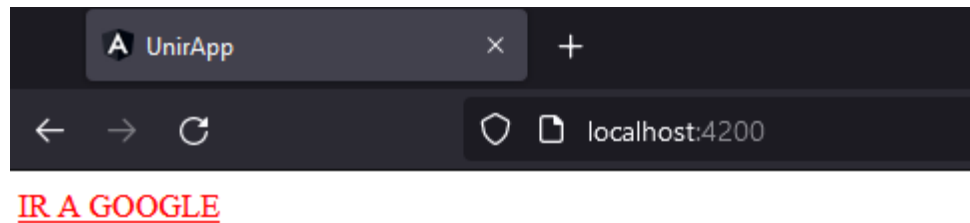


Figura 21. Resultado del Property Binding. Fuente: elaboración propia.

De igual forma también nos permite especificar una clase de CSS para añadir a nuestro HTML, de tal forma que si le añadimos una propiedad del TS que tenga un valor booleano puedes activar el valor de una clase dando el valor true.

Veamos un ejemplo:

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

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

```

    styleUrls: ['./primer-componentes.component.css']
  })
  export class PrimerComponentesComponent implements OnInit {

    ruta: string = 'http://www.google.es'
    color: string = 'rojo'
    stock: boolean = true;
    constructor() { }

    ngOnInit(): void { }

  }

```

La propiedad stock es booleana y la vamos a usar para definir que nos pinte una clase rojo si su valor es true de la siguiente forma.

```

<a [href]="ruta" [class.rojo]="stock">IR A GOOGLE</a>

.rojo{
  color:red
}

```

23.2. Two way data binding: FormsModule y [(ngModel)]. Doble comunicación

Se trata de una de las técnicas más usadas para la recopilación de datos por parte del usuario.

Esta herramienta nos da la posibilidad de enlazar un campo de texto definido dentro de nuestra plantilla con cualquiera de las propiedades del componente, de una manera bidireccional.

Para poder hacer uso de esta característica utilizamos la directiva ngModel.

Para poder hacer uso de esta directiva, dentro de la definición de nuestro módulo, debemos importar la librería FormsModule.

Nuestro app.module.ts quería de la siguiente forma.

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

Posteriormente podemos usar la directiva dentro de nuestras plantillas de la siguiente manera:

```
<input [(ngModel)]="textoInput" />
<h2>{{ textoInput }}</h2>
```

Lo que hacemos con la directiva [(ngModel)] es asociar una propiedad de Typescript con la directiva ngModel haciendo que así cualquier modificación en el input se guarde en la propiedad y viceversa, obteniendo así una comunicación bidireccional.

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

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

```
    textoInput: string = "Hola con la variable iniciada"
    constructor() { }

    ngOnInit(): void { }
}
```

En este caso concreto, el valor del campo de texto queda enlazado en ambos sentidos con la propiedad `textoInput` definida dentro de nuestra clase.

Si modificamos la variable, se modifica el valor del campo de texto y viceversa.

Esto es una forma alternativa a los formularios de angular que nos permite recoger datos por parte del usuario de nuestra aplicación asociando a una propiedad del TypeScript.

En el tema siguiente veremos como a través de eventos podemos usar el Two Way Data Biding para asociar por ejemplo a un objeto el resultado de dos inputs de tal forma que se recoja en cada una de las propiedades del propio objeto los valores del usuario introduce dentro de cada input.

En este vídeo, *Comunicación entre el TS del componente y el HTML*, se trata la creación de propiedades que me permiten pintar valores dentro del HTML y modificar también atributos de HTML.



Accede al vídeo

Tema 24. Event Binding

24.1. Introducción y objetivos

El objetivo que buscamos en este tema es interactuar con el usuario de nuestra web. Una página web no es un ente estático si no es un lugar donde el usuario puede interactuar y modificar elementos de la web, ya sea a través de formularios, botones o filtros, el usuario es una parte muy importante del desarrollo de nuestras aplicaciones ya que a través de su interacción recibimos muchísima información que nos ayuda a tomar decisiones.

Esto da lugar a los eventos, y estos están presentes en muchas aplicaciones y de diferentes formas. En JavaScript los manejábamos a través de listeners y Angular nos aporta una forma muy sencilla de manejarlos y recoger los valores que emite nuestra interfaz.

En este tema vamos a ver las diferentes formas que tenemos de capturar los eventos y de mandar información, esta vez, desde el HTML hacia nuestro fichero TypeScript.

24.2. Creación de eventos en Angular

Event Binding nos permite implementar dentro de Angular un sistema de eventos o funciones que nos posibilita conectar el HTML y la interacción del usuario y cambiar el resultado.

El método que llamamos tras la ejecución del evento por parte del usuario nos permite recoger datos del propio DOM o cualquier dato que le pasemos por parámetros a dicha función. Para llamar a un evento invocamos el tipo de evento sin

la palabra «on» entre paréntesis seguido de un igual y la llamada a la función definida dentro de Typescript.

Podemos utilizar los eventos definidos de manera nativa dentro del navegador o crear los nuestros propios.

Para poder definir qué evento estamos capturando utilizamos los paréntesis para limitarlos.

Vemos un ejemplo para capturar la acción de **click** sobre un botón.

```
<button (click)="hacerClick()">Pulsame</button>
```

Esta captura de código que visualizas es un elemento que se sitúa en el HTML del componente. Entre paréntesis puedes ver el tipo de evento sin la palabra «on» y entre las “” puedes ver el método de clase que se ejecuta cuando se produce dicho evento.

Este método tiene que estar definido dentro de la clase del componente de la siguiente forma.

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

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

  constructor() { }

  ngOnInit(): void { }

  hacerClick() {
```

```

    alert('has hecho click en el boton')
  }
}

```

Si ejecutas la interfaz lo que vas a poder observar si haces click en el botón es que esta interactúa y te devuelve una respuesta en forma de alerta.

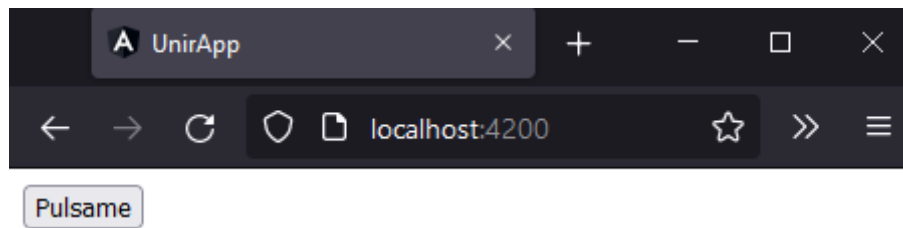


Figura 22. Resultado de la vista de componente antes de hacer click. Fuente: elaboración propia.

Al pulsar en el botón la interfaz captura el evento llama al método de la clase y se produce el resultado, en este caso una alerta de JavaScript que nos lanza un mensaje de la siguiente forma.

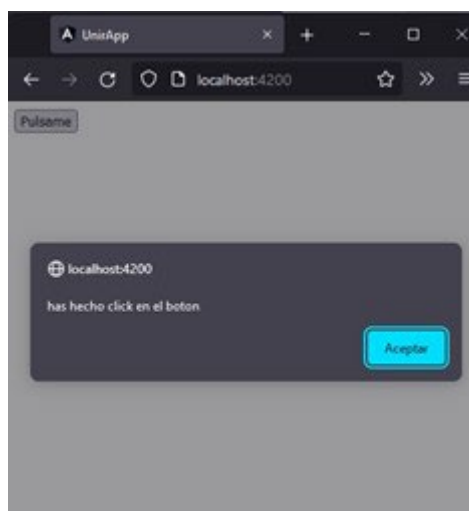


Figura 23. Salida por pantalla de la resolución del evento. Fuente: elaboración propia.

Si unimos la consecución de un evento con lo explicado en el tema anterior podemos asociar el resultado de varios inputs a través del Two Way Data Biding. Os recuerdo los pasos para poder realizar esta técnica:

- ▶ Importar **FormsModule** en App-module.
- ▶ Crear una propiedad dentro de la clase, esta vez será un objeto JSON.
- ▶ Asociar a cada uno de los inputs una propiedad del objeto.
- ▶ Cuando hagamos **click** en el botón sacaremos por consola el resultado del objeto ya relleno.

Como ya vimos cómo se hacían las importaciones en el tema anterior me voy a centrar solo en la creación y asociación de las propiedades del objeto y en la ejecución del evento.

```
export class PrimerComponentesComponent implements OnInit {  
  
    //creamos la propiedad  
    alumno: any;  
  
    constructor() {  
        //inicializamos los valores del objeto  
        this.alumno = {  
            nombre: "",  
            edad: 0  
        }  
    }  
}
```

En el fragmento de código anterior vemos como se declara un objeto alumno y lo inicializamos dentro del constructor con las propiedades nombre y edad, con los valores por defecto

En el siguiente fragmento de código vamos a ver como montamos el evento dentro de un botón y como asignamos a cada input los valores de los diferentes elementos.

```
<input type="text" [(ngModel)]="alumno.nombre" placeholder="introduce un nombre">
<input type="number" [(ngModel)]="alumno.edad" placeholder="introduce una edad">
<button (click)="capturarDatos()">Pulsame</button>
```

En el Typescript habrá que dar de alta el método de `capturarDatos()` donde simplemente mostraremos los valores del objeto `alumno` por consola.

```
export class PrimerComponentesComponent implements OnInit {
  //creamos la propiedad
  alumno: any;
  constructor() {
    //inicializamos los valores del objeto
    this.alumno = {
      nombre: "",
      edad: 0
    }
  }
  ngOnInit(): void { }

  capturarDatos() {
    console.log(this.alumno);
  }
}
```

El resultado que se produce en la consola del navegador es el siguiente.

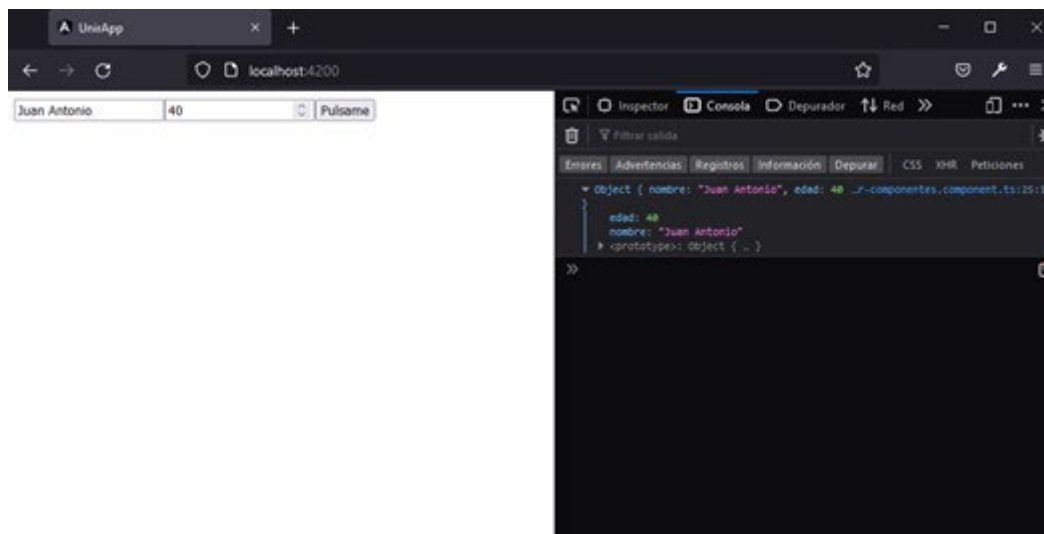


Figura 24. Captura de resultado del evento en consola al hacer click. Fuente: elaboración propia.

Como puedes ver el tomar datos de esta forma combinando las dos técnicas me permite una comunicación bidireccional entre HTML y TypeScript. De esta forma podemos empezar a interactuar entre el usuario y nuestra aplicación.

Por descontado que no hace falta el crear objetos para asignar campos, pero es una buena práctica ya que en un único objeto tengo capturados todos los valores de los campos.

Los eventos que se pueden producir en una aplicación no son solo de click, por supuesto, sino que hay un abanico amplio de evento que pueden resultar interesantes y que se aplican de la misma forma

```
<div (mouseover)=handleOver()">  
<input type="text" (keydown)=handleKey()">  
<form (submit)=handleSubmit()">
```

24.3. \$event

Dentro de los paréntesis del método que lanza el evento podemos insertar cualquier tipo de parámetro, ya sea un string, un número o un objeto completo. Esto podrá cambiar dependiendo de las necesidades que tengamos dentro de nuestro proyecto.

Uno de los argumentos más importantes que podemos pasarle a un evento es `$event`, ya que me permite enviar información del objeto que está lanzando en evento y poder capturar de él, cierta información necesaria para la ejecución de mis acciones. Depende del evento que estemos capturando podremos recuperar unas propiedades u otras de `$event`.

Volvamos al ejemplo del botón simple y veamos qué posibilidades nos ofrece en este caso pasarle como parámetro al método el `$event`.

```
<button (click)="capturarDatos($event)">Pulsame</button>
```

Como podéis observar en este caso, dentro de los paréntesis de la función tenemos como argumento de nuestra función el `$event` cuando se ejecute la acción descrita, en el método `capturarDatos()` tendremos toda la información que se produce del evento, como por ejemplo la posición en el escenario, el tipo de evento de click, y sobre todo y lo más importante el objeto que lanza el evento, en este caso un button de html.

```
capturarDatos($event:any) {  
  console.log($event);  
}
```

El resultado que nos arroja la consola de nuestro navegador es el siguiente. Como veis la información enviada es bastante extensa, y entre toda esa información el elemento más importante es el target ya que representa el elemento del DOM que tocamos.

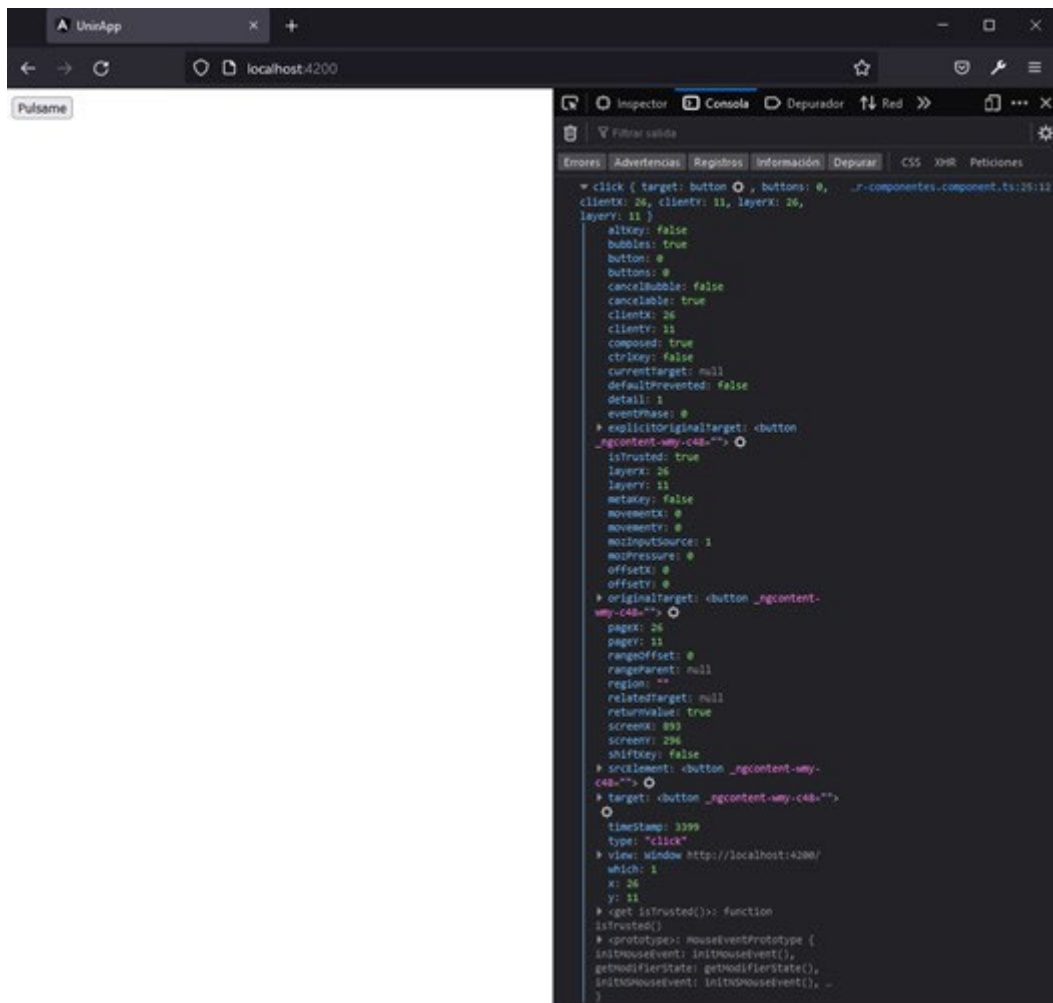


Figura 25. Captura de la salida de consola del navegador. Fuente: elaboración propia.

El target al igual que en JavaScript nativo representa el elemento del DOM que ejecuta el evento, y a partir de este elemento podemos capturar una propiedad del objeto, como por ejemplo el id, o el texto del botón con `innerText`, al igual que hacíamos en JavaScript. Por ejemplo `$event` en caso de un evento de teclado me ofrece la posibilidad de sacar el código de tecla para diferenciar entre que tecla del teclado estoy haciendo `keydown`, a través del `$event.keyCode`.

Se debe recordar que a través de un evento podemos cambiar el resultado de cualquier propiedad de nuestro TypeScript lo que me permitiría cambiar una clase o un estilo a través de `Property Binding` o un texto a través de la interpolación.

Combinando todo lo aprendido hasta ahora, podemos empezar a desarrollar ya aplicaciones un poco más complejas con interacción del usuario.

En este vídeo, *Gestión de Eventos en Angular*, se trata la creación de eventos dentro de Angular para poder interactuar con el usuario.



Accede al vídeo

En el siguiente vídeo, *Two Way Data Biding con Eventos*, se habla de la comunicación bidireccional entre HTML y TS y la interacción con el usuario.



Accede al vídeo

Tema 25. Input y Output

25.1. Introducción y objetivos

El objetivo que buscamos en este tema es la comunicación entre componentes. Puede realizarse de diversas maneras (servicios, observables, etc.), pero en este tema, utilizaremos los decoradores `@Input` y `@Output`. El `@Input` es un decorador que van asociado a una propiedad de la clase de nuestro componente, gracias a ellos podemos pasar datos desde un componente padre hacia un componente hijo. Con `@Output`, realizaremos lo contrario. Pasaremos datos desde un hijo hacia su padre. Vamos a ver detenidamente todos los pasos a seguir para usar cada uno de ellos y que particularidades tienen.

El objetivo de este tema es manejarlos con soltura para tener la capacidad de decidir cuando es más recomendable usarlo y cuando, no. Como hemos dicho hay diferentes formas de comunicarse entre componentes y aprender `@Inputs` y `@Outputs` no implica que tengamos que usarlos siempre.

En este tema nos centraremos en manejarlos bien, incluso muchas veces en momentos en los que no sería necesario, pero de esta forma nos ayude a comprender como y cuando es la mejor manera de usarlo.

25.2. Comunicación entre componentes

Dentro de una página web nos podemos encontrar diferentes tipos de componentes que realizan acciones distintas pero que tienen relación con cada uno de los otros componentes que les rodean, en mucho caso estos componentes deben comunicarse algunas cosas y para ello podemos usar diferentes formas.

Como hemos explicado en los objetivos del tema existen muchas formas de comunicar componentes, dependiendo si estos están relacionados entre sí o no.

Durante este curso iremos viendo formas de pasar información entre dos componentes, entre un componente padre y un componente hijo, entre el componente y la BBDD a través del servicio. Es decir que gran parte del acceso a los datos que tienen los componentes la obtienen de la comunicación que puede existir entre los diferentes métodos de comunicarse.

Por matizar un poco más esto nos podemos encontrar con los siguientes casos.

Un componente quiere pasarle información a otro componente que este contenido en él. Es decir, un componente padre quiere comunicar algo a su hijo. Es caso es bastante común dentro de la comunicación entre componentes. En este caso realizaremos la comunicación mediante un `@Input`.



Figura 26. Explicación gráfica de un input. Fuente: elaboración propia.

Un componente que este contenido dentro de otro quiere comunicarle algo a su contenedor. Es decir, un componente **hijo** quiere comunicar algo a un componente **padre**. En este caso la comunicación se debe realizar a través de un evento personalizado o `@Output`.



Figura 27. Explicación gráfica del output. Fuente: elaboración propia.

Dos componentes que no están relacionados entre sí quieren transmitirse información. En este caso al no haber ningún tipo de relación, directa, entre ellos hay que encontrar un tercer elemento que me permita comunicarme parcialmente con cada uno de los componentes, para ello vamos a usar diferentes técnicas dependiendo de donde se encuentre los datos. Aunque la más común y que veremos más adelante en el curso es la comunicación usando un servicio.

Más adelante el servicio también nos aportará como funcionalidad la posibilidad de hacer consultas a la BBDD externa a la aplicación y poder traer datos para luego enviárselos a los otros componentes de la aplicación. Sería como el director de juego o base en baloncesto, que reparte el juego entre el resto de sus compañeros.

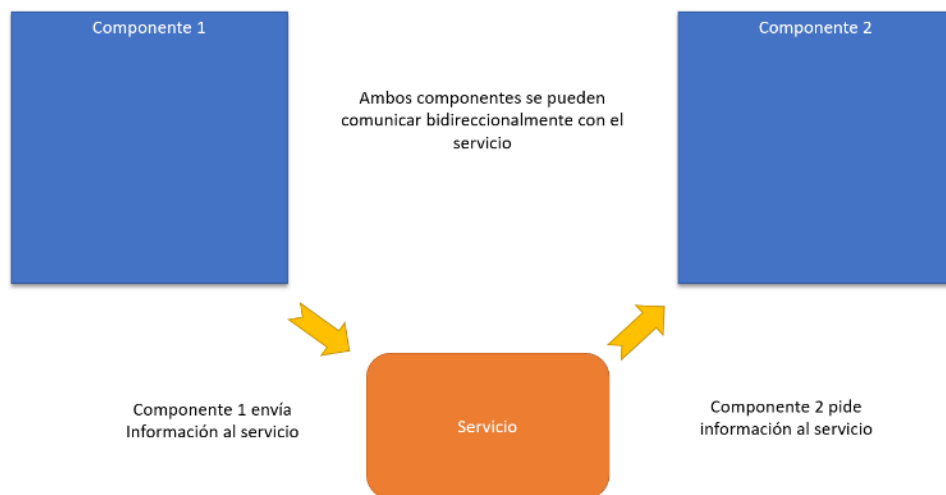


Figura 28. Comunicación a través del servicio. Fuente: elaboración propia.

En este tema nos vamos a centrar exclusivamente en los dos primeros, y más adelante le daremos al servicio un tema propio donde hablaremos de todas sus particularidades.

25.3. Decorador @Input(). Del padre al hijo

Un decorador @input, como ya hemos explicado antes, nos permite comunicar un padre con su hijo y que este le pueda enviar diferentes valores al hijo para cambiar su estado, o simplemente pintar un resultado diferente.

Lo primero que tenemos que hacer para usar el Input es importarlo en el componente que vayamos a usarlo de la siguiente forma.

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

El siguiente paso es decorar la propiedad con la que vamos a trabajar, para ello como con cualquier decorador le ponemos la función decoradora por delante de la propiedad la cual queremos convertir en Input. Es importante y TypeScript nos avisará de ello, que le pongamos el tipo del dato que va a almacenar ya sea un

número o un string, etc. También si TypeScript está en modo estricto nos obligará a inicializar el valor. Esto como cualquier propiedad podréis hacerlo en la misma línea de la declaración o dentro del método constructor de la clase.

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

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

  //Creación de un input e inicializamos el valor.
  @Input() texto: string = "";

  constructor() {
  }

  ngOnInit(): void { }

}
```

Ahora tenemos que llenar esa propiedad con un dato, y este se lo vamos a pasar a través del selector que carga el componente dentro de otro, ya sea el principal como otro cualquiera.

```
<section>
  <app-primer-componente [texto]='Una cadena de texto que viene del
padre'></app-primer-componente>
</section>
```

Como podéis ver al crear una propiedad texto dentro de la etiqueta del componente, está la ponemos entre corchetes, para que admita el uso de sentencias JavaScript validas como valor del input, en este caso le hemos pasado una cadena de texto, de ahí que el texto este entre comillas simples.

Tal y como veíamos anteriormente, este tipo de atributos también pueden recoger su valor evaluando el contenido entre comillas, mediante la nomenclatura de corchetes.

Podemos definir un alias dentro de la llamada al decorador para especificar un nombre diferente para el atributo que usaremos en la declaración del tag

```
//Creacion de un input
@Input('miTexto') texto: string = "";
```

Si usamos ese alias tenemos que cambiar el valor del input en la etiqueta de creación del componente.

```
<section>
  <app-primer-componente [miTexto]='Una cadena de texto que viene del
padre'"></app-primer-componente>
</section>
```

Una vez este cargado todo el componente podremos disponer de este input que al ser una propiedad de la clase puede renderizarse perfectamente dentro del HTML del propio componente.

Si os acordáis de las funciones de ciclo de vida de los componentes lo input y los outputs se cargan con el `ngOnChanges` o con el `ngOnInit`.

```
//OJO DENTRO DEL HTML DEL COMPONENTE NO USAMOS EL
//ALIAS SINO EL NOMBRE DE LA PROPIEDAD
<p>{{texto}}</p>
```

Visualizar el resultado por el navegador, podréis comprobar que os sale el texto que habríais introducido dentro del del input.

Recordar: Ese texto puede ser una variable o constante, así como una cualquiera propiedad que tenga el componente padre.

En este vídeo, *Inputs en Angular*, se trata la comunicación del padre al hijo a través de inputs.



Accede al vídeo

25.4. Decorador @Output(). Del hijo al padre

Este tipo de decorador lo usamos para pasar información del hijo al padre, el padre esperará un aviso o evento por parte del hijo con el que gestionará la información que este le envía.

Para usar un decorador output al igual que el input hay que importarla en cada componente donde se vaya a usar de la siguiente forma.

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

también debemos importar la librería EventEmitter que es la que me ofrece la posibilidad de emitir un evento personalizado al padre.

Creamos la propiedad que hacer de output y la inicializamos.

```
import { Component, EventEmitter, OnInit, Output } from '@angular/core';
@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})

export class PrimerComponentesComponent implements OnInit {

  //Creacion de un output
```



```

@Output() delete = new EventEmitter()

constructor() {
}

ngOnInit(): void { }
}

```

Ahora nos toca crear una función que nos permite llenar el output y emitirle el resultado al padre para que este actúe en consecuencia

Por ejemplo, podemos crear en el HTML un evento que nos envíe un número y este se lo comuniquemos al padre, en el evento onDelete vamos a enviar como parámetro un número y este tenemos que hacérselo llegar al padre para que lo saque por consola.

Primer paso crear el evento y recoger el valor que nos envía.

```

<button (click)="onDelete(5)">Enviar numero</button>

```

Este evento se está produciendo en el HTML del componente con lo que tendremos que crear una función onDelete() en el TypeScript de dicho componente de la siguiente manera:

```

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

@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})

export class PrimerComponentesComponent implements OnInit {

```

```

//Creacion de un output
@Output() delete = new EventEmitter()

constructor() {
}

ngOnInit(): void { }

onDelete(numero: number) {
  this.delete.emit(numero)
}
}

```

Al recoger el parámetro número del propio evento este se emite hacia al padre creando un evento personalizado dentro de la etiqueta que carga el propio componente.

En el HTML del componente padre la etiqueta del componente antes mencionado tendrá un evento personalizado que se llamará exactamente igual que el nombre del output declarado. Este evento llamará a una función declarada dentro del padre.

```

<section>
  <app-primer-componente      (delete)="handleDeleteFather($event)"></app-
primer-componente>
</section>

```

Este método o evento personalizado recibe un \$event que en este caso representa al valor del número que le estamos pasando por parámetro. Este valor podremos recogerlo a través de la función handleDeleteFather(\$event) que tendremos que crear dentro del componente padre.

```

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

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

```

```
    templateUrl: './app.component.html',  
    styleUrls: ['./app.component.css']  
  })  
  export class AppComponent {  
  
    handleDeleteFather($event: any) {  
      console.log($event)  
    }  
  }  
}
```

Cerrando de esta manera el ciclo en el cual el componente hijo le manda un valor al padre y este es capaz de recogerlo a través de un componente personalizado.

En este vídeo, *Outputs en Angular*, se trata la comunicación del hijo al padre a través de outputs.



Accede al vídeo

Tema 26. Directivas

26.1. Introducción y objetivos

En este tema vamos a tratar el tema de las directivas. Podemos considerar como directivas, cualquiera de los elementos que representamos dentro de las plantillas de nuestras páginas web, ya sean etiquetas o atributos.

El objetivo de este tema es conocer los diferentes tipos de directivas y aprender a aplicarlos, para si obtener diferentes resultados dentro de nuestro HTML del componente.

Las Directivas extienden la funcionalidad del HTML usando para ello una nueva sintaxis. Con ella podemos usar lógica que será ejecutada en el DOM (Document Object Model) y así «olvidarnos» del manejo del DOM que hacía JavaScript de forma nativa, ayudándonos a gestionar la aparición, replica y estilos de nuestros elementos en Angular.

26.2. Directivas: definición y tipos

Podemos considerar como directivas, cualquiera de los elementos que representamos dentro de las plantillas de nuestras páginas web, ya sean etiquetas o atributos.

Por lo tanto, los componentes que definamos en nuestras aplicaciones también pueden ser considerados como directivas.

En este caso, lo destacable es que estas directivas especiales llevan asociadas una plantilla HTML.

Aparte, disponemos de dos tipos diferentes de directivas: de atributo y estructurales. Las directivas de atributo se encargan de modificar la apariencia o el comportamiento de cualquiera de los elementos que podamos encontrar dentro del DOM.

Existen tres, una de ellas ya la conocemos que es la que nos permite hacer el two Data Binding.

- ▶ `ngClass`: nos permite añadir dinámicamente clases de `css` de forma fácil. Tiene varias formas de implementarse y me da mucha más versatilidad que el `property binding`.
- ▶ `ngModel`: Necesario para aplicar el Two Data Binding, asocia el valor de una propiedad del TS a un elemento en el que el usuario puede introducir valores, como por ejemplo un `input`.
- ▶ `ngStyle`: permite asignar varios estilos en línea a un elemento.

Por otro lado, las directivas estructurales modifican el contenido de nuestras plantillas mostrando u ocultando diferentes elementos en función de la serie de condiciones que necesitemos. También son tres.

- ▶ `ngFor`: permite repetir estructuras de HTML iterables por medio de recorrer una propiedad del TS de tipo `array`. El ámbito del HTML sobre el que se aplica lo llamamos superficie iterable.
- ▶ `ngIf`: en base a un valor booleano, crea o elimina elementos en del HTML.
- ▶ `ngSwitch`: Otra sentencia condicional que tiene parte estructural y parte de atributo, permite asignarle a un elemento HTML la posibilidad de aparecer y desaparecer por medio del valor de una propiedad del TS.

Vamos a analizar las más características para terminar viendo cómo podemos crear las nuestras propias.

26.3. ngStyle

La primera de las directivas de atributo es ngStyle.

Mediante esta directiva podemos aplicar a nuestras plantillas, un objeto con los diferentes estilos que necesitemos en cada caso.

Los valores asignados a cada uno de los estilos que apliquemos a través de esta directiva pueden ser fijos o, por el contrario, podemos utilizar alguna de las variables de nuestro componente para darle un carácter más dinámico.

Lo vemos con una serie de ejemplos. Todo el código que vais a visualizar está en su mayoría creado dentro del HTML del componente en el que trabajéis.

```
<p [ngStyle]="{'color': 'blue', 'font-size': '12px', 'font-weight':  
'bold'}">Párrafo de texto</p>
```

También podemos asignar los valores a través de las propiedades o métodos de nuestra clase

```
import { Component, OnInit, SimpleChange } from '@angular/core';  
@Component({  
  selector: 'hola',  
  template: `<p [ngStyle]="paragStyles">Párrafo de texto</p>`  
})  
export class HolaComponent {  
  weight = 'lighter'  
  paragStyles = {  
    'color': this.getRandomColor(),
```

```

        'font-size': '14px',
        'weight': this.weight
    }
    getRandomColor() {
        return '#' + Math.floor(Math.random() * 16777215).toString(16)
    }
}

```

Y el resultado en esta ocasión en el HTML se visualizaría de esta forma

```
<p [ngStyle]='{color: 'blue', 'font-size': '12px', 'font-weight': weight}'>Párrafo de texto</p>
```

```
<p [ngStyle]="paragStyles">Párrafo de texto</p>
```

En este vídeo, *Directiva ngStyle*, se explica cómo se usa la directiva ngStyle.



Accede al vídeo

26.4. ngClass

Mediante la directiva ngClass podemos asignar clases CSS a nuestros componentes. Podemos hacerlo de tres maneras diferentes.

Mediante un array de clases:

Asignamos a la directiva un array con todas las clases de CSS que queremos aplicarle, por supuesto para que este funcione dichas clases tiene que estar creadas dentro del CSS del componente.

```
<p class="fondoAzul textoRojo" [ngClass]="['textoRojo',  
'fondoAzul']">Lorem aliquid!</p>
```

Mediante un string:

Esta es la forma menos recomendable porque es muy parecida al usar una clase estática de las de CSS de toda la vida, pero es una alternativa que puede ser interesante en algún caso, como por ejemplo recibir todo un string como propiedad de una clase.

```
<p [ngClass]="['texto-negrita verde']">Párrafo de texto</p>
```

Mediante un objeto:

Mediante un objeto, en el cual se especifica mediante un boolean, qué clases se aplican y cuáles no.

Esta quizás es la más interesante ya que te permite activar o desactivar valores dependientes de valores de propiedades que tenga tu TypeScript.

```
<p [ngClass]="{'fondoRojo': activo, 'fondoAzul': !activo}">Lorem eaque!</p>  
<button (click)="cambiarActivo()">Cambiar activo</button>
```

Si creásemos una propiedad dentro de nuestra clase de componente que se llamase activo la clase de CSS fondoRojo y fondoAzul se activarían siempre y cuando su valor fuese true o false.

El código en nuestro componente quedaría de la siguiente manera.

```
import { Component, OnInit } from '@angular/core';  
  
@Component({  
  selector: 'app-ng-class',
```



```

    templateUrl: './ng-class.component.html',
    styleUrls: ['./ng-class.component.css']
  })
  export class NgClassComponent implements OnInit {

    texto: string = "textoRojo"
    activo: boolean = false
    constructor() { }

    ngOnInit(): void {
    }

    cambiarEstilo() {
      this.texto = (this.texto == 'textoRojo') ? 'textoVerde' : 'textoRojo'
    }

    cambiarActivo() {
      this.activo = !this.activo;
    }
  }

```

En este vídeo, *Directiva ngClass*, se explica cómo se usa una directiva ngClass.



Accede al vídeo

26.5. ngIf

Es una directiva estructural, **ngIf* que nos permite cargar o eliminar cierta parte de la estructura de un componente a través de una expresión booleana. Si el valor que hay dentro del **ngIf* es **true** la estructura de HTML de dicho componente se visualiza y si es **false** no se carga. Al contrario que las propiedades de css display o visibility que tan solo lo ocultan de la vista, pero lo cargan dentro del componente.

Distinguimos que es una directiva estructural por que se nombre con un asterisco delante, todas las directivas de atributo que hemos visto están entre [].

¿Por qué eliminamos el contenido en vez de ocultarlo?

Podríamos realizar el mismo trabajo que hace esta directiva con la aplicación de ciertos estilos CSS que nos permitan mostrar/ocultar el HTML.

Con `*ngIf` lo eliminamos directamente.

El comportamiento con CSS es el siguiente:

- ▶ Cuando ocultamos un componente o parte de nuestra página, su comportamiento sigue activo.
- ▶ Sigue escuchando los posibles eventos que reciba la aplicación.
- ▶ Angular seguirá escuchando los posibles cambios que le afecten y le seguirá enviando los datos que se modifiquen.
- ▶ Sigue usando recursos de nuestra aplicación.
- ▶ Por otro lado, el cambio de oculto a no es muy rápido.

El trabajo con `*ngIf` es diferente:

- ▶ Si ocultamos bloques de código, afecta a su rendimiento, ya que se elimina del DOM.
- ▶ Deja de recibir evento.
- ▶ El componente es destruido y se ejecutan los métodos finales del ciclo de vida del componente.
- ▶ En este caso, se emplean muchos recursos para regenerar el componente.

A continuación, se muestra un ejemplo:

```
<p *ngIf=" visible === 'none' ">Este se oculta con ngIf</p>
```

Si lo que está entre las comillas es igual a true el elemento que tiene el ámbito del *ngIf se visualiza y si es false no se carga.

Hemos visto cómo podemos mostrar u ocultar información dentro de nuestros componentes mediante el uso de *ngIf, pero también podemos trabajar con el valor contrario a la condición que estemos usando.

Para ello, contamos con la expresión else definida dentro de la expresión de *ngIf

```
<p *ngIf="mostrar; else elseBlock">Contenido</p>
<ng-template #elseBlock>
  <p>Contenido ELSE</p>
</ng-template>
<button (click)="toggleMostrar()">Aparece/Desaparece</button>
```

En el ejemplo que vemos arriba creamos una función dentro del TypeScript del componente que nos permite cambiar el valor de la propiedad mostrar de true a false de tal forma que el ng-template y el p se intercambian, apareciendo y desapareciendo de nuestra vista.

Otra posibilidad para interactuar con condicionales de este tipo es hacerlo con la estructura **then... else...**

```
<p *ngIf="mostrar; then content else elseBlock">Esto se
  ignora</p>
<ng-template #content>
  <p>CONTENIDO TRUE</p>
</ng-template>
<ng-template #elseBlock>
  <p>Contenido ELSE</p>
</ng-template>
<button (click)="toggleMostrar()">Aparece/Desaparece</button>
```

En este caso, podríamos utilizar únicamente **then** dentro de la estructura condicional.

En este vídeo, *Directiva ngIf*, se explica cómo se usa una directiva ngIf.



Accede al vídeo

26.6. ngFor

La directiva `*ngFor` permite insertar bloques de código HTML de forma dinámica, basándose en una lista de elementos (en general, arrays). Esta directiva nos permite repetir una serie de contenido en función de los elementos de una colección definida dentro de nuestro componente.

Para poder utilizarla, necesitamos partir del uso de un array dentro de la clase que representa nuestro componente. Algo así.

```
episodios: any[] = [];  
constructor() {  
  this.todosEpisodios = new Array(  
    { title: 'Winter <span class="rojo">Is Coming</span>', director: 'Tim  
Van Patten' },  
    { title: 'The Kingsroad', director: 'Tim Van Patten' },  
    { title: 'Lord Snow', director: 'Brian Kirk' },  
    { title: 'Cripples,<span class="rojo"> Bastards, and Broken  
Things</span>', director: 'Brian Kirk' },  
    { title: 'The Wolf and the Lion', director: 'Brian Kirk' },  
    { title: 'A Golden Crown', director: 'Daniel Minahan' },  
    { title: 'You Win or You Die', director: 'Daniel Minahan' },  
    { title: 'The Pointy End', director: 'Daniel Minahan' },  
  )  
}
```

El siguiente paso será recorrer cada uno de los elementos del array dentro de nuestra plantilla.

```
<ul>
  <li *ngFor="let episodio of
episodios">{{episodio.title}}</li>
</ul>
```

El `*ngFor` se aplica sobre el elemento HTML que se va a repetir en función de la cantidad de elementos de mi lista. Es lo que se llama superficie iterable, es el componente HTML debemos localizar para que la repetición de nuestra plantilla tenga sentido.

Aparte de la recuperación de cada uno de los elementos del array con el que estamos trabajando, podemos recuperar más información ofrecida por el bucle.

Las variables que podemos recuperar son las siguientes:

- ▶ **index:** devuelve la posición que nos encontramos en el bucle.
- ▶ **first:** es true si nos encontramos en el primer elemento del bucle.
- ▶ **last:** es true si nos encontramos en el último elemento de la colección.
- ▶ **even:** valor true si es elemento es par.
- ▶ **odd:** valor true si el elemento es impar.

La manera de acceder a estos valores es la siguiente:

```
<ul>
  <li *ngFor="let episodio of episodios; let i = index; let first = first;">
    {{i}} - {{episodio.title}} {{first}}
  </li>
</ul>
```

Todas estas variables, al usar la definición **let** acotan su ámbito al interior del bucle.

La palabra clave en la iteración es **OF**. No debemos confundirla con **IN**. Un ejemplo de plantilla combinando `*ngFor` y `[ngClass]` podría ser el siguiente:

```
<ul>
  <li *ngFor="let episodio of episodios; let par = even; let impar = odd;"
      [ngClass]="{ parStyle: par, imparStyle: impar }">
    {{i}} - {{episodio.title}} {{first}}
  </li>
</ul>
```

Si modificamos los elementos del array, el bucle tiende a adaptarse a los nuevos elementos y reorganiza la plantilla en función del menor gasto de procesador.

Si queremos optimizar el proceso del bucle `*ngFor`, podemos utilizar **trackBy** para indicar a nuestra aplicación qué elementos se están modificando.

De esta manera, únicamente se modifican en la plantilla aquellos elementos que cambien, reduciendo el gasto de procesador de nuestra aplicación.

Vemos un ejemplo a partir de esta lista:

```
<ul>
  <li *ngFor="let episodio of episodios; trackBy: episodio?.id;">
    {{i}} - {{episodio.title}} {{first}}
  </li>
</ul>
```

En este vídeo, *Directiva ngFor*, explica cómo se usa una directiva `ngFor`.



Accede al vídeo

26.7. ngSwitch

Su funcionamiento es similar a la estructura switch que podemos encontrar en cualquier lenguaje de programación. Se encarga de evaluar un valor y mostrar el contenido que coincida con dicho valor. Se forma a partir de dos directivas, una de **atributo** y otra **estructural**.

Con ngSwitch evaluamos el elemento que necesitamos comparar.

Con ngSwitchCase especificamos cada uno de los valores que va a poder tomar.

Si ninguno de los valores anteriores coincide, se muestra la plantilla delimitada con la directiva ngSwitchDefault.

Este sería un ejemplo de plantilla usando ngSwitch:

```
<div [ngSwitch]="opcion">
  <p *ngSwitchCase="1">Opción 1</p>
  <p *ngSwitchCase="2">Opción 2</p>
  <p *ngSwitchCase="3">Opción 3</p>
  <p *ngSwitchCase="4">Opción 4</p>
  <p *ngSwitchDefault>Ninguna opción</p>
</div>
```

La diferencia de uso entre *ngIf y [ngSwitch] es que, con este último, la condición solo se evalúa una vez y a raíz de esa evaluación, se muestran unas partes u otras de nuestro componente.

Todas las directivas que hemos visto hasta ahora se pueden anidar unas con otras para conseguir plantillas más complejas.

26.8. Creación de tu propia directiva

Una vez conocemos las diferentes directivas que podemos utilizar dentro de nuestras plantillas, vamos a analizar cómo podemos crear las nuestras propias.

Para generar nuestras propias directivas tenemos a nuestra disposición el decorador `@Directive`.

Para crear una directiva usamos este código desde terminal al igual que hacíamos para crear componentes.

```
ng g directive nomDirectiva
```

Este decorador se aplica sobre una clase como las vistas anteriormente.

```
import { Directive } from '@angular/core';
@Directive({
  selector: "[neoColor]"
})
export class ColorDirective {
}
```

El primer elemento a destacar es el atributo selector, mediante el cual indicamos cómo vamos a representar nuestra directiva en el momento de utilizarla.

Para poder definir un selector en nuestras directivas necesitamos usar nomenclatura CSS.

Si mantenemos la sintaxis vista en el ejemplo anterior, el modo para ejecutar dicha directiva sería el siguiente:

```
<p neoColor>Contenido</p>
```


Aunque si lo preferimos, podemos definir nuestra directiva como una clase más de CSS aplicada sobre cualquiera de los elementos de nuestro DOM.

Para alcanzar este objetivo tendremos que modificar el selector y la forma de aplicarlo:

```
@Directive({  
  selector: ".neoColor"  
})
```

```
<p class="neoColor">Contenido</p>
```

Como cualquiera de las clases que desarrollamos dentro de nuestra estructura de aplicación Angular, la clase que representa nuestra directiva dispone de su método constructor.

Dentro del método constructor podemos inyectar un objeto de tipo `ElementRef`, el cual nos da acceso al elemento del DOM sobre el cual estamos aplicando dicha directiva.

Para poder interactuar con dicho elemento tenemos que hacerlo a través de la propiedad `nativeElement` del objeto inyectado.

Vemos en un ejemplo cómo podríamos modificar algún estilo del elemento sobre el cual aplicamos la directiva.

```
import { Directive, ElementRef } from '@angular/core';  
@Directive({  
  selector: "[neoColor]"  
})  
export class ColorDirective {  
  constructor(el: ElementRef) {  
    el.nativeElement.style.color = 'red'  
  }  
}
```

```
}  
  
}
```

En el ejemplo anterior estamos asumiendo que nuestra aplicación se está ejecutando sobre un navegador, pero esto no siempre será así.

Angular se ha diseñado para que pueda ser utilizado en diferentes entornos, incluso podríamos trabajar del lado servidor gracias a Node o en cualquier dispositivo móvil.

Por lo que disponemos de una herramienta mediante la cual podemos modificar propiedades de nuestros elementos, independientemente de la plataforma en la que nos encontremos, se trata del objeto **Renderer**.

Modificamos el ejemplo anterior:

```
import { Directive, ElementRef, Renderer } from '@angular/core';  
@Directive({  
  selector: "[neoColor]"  
})  
export class ColorDirective {  
  constructor(el: ElementRef, renderer: Renderer) {  
    renderer.setStyle(el.nativeElement, 'color', 'red')  
  }  
}
```

En este caso, si visualizamos nuestra aplicación en un entorno diferente al web, el objeto **Renderer** será el encargado de llamar a las funciones apropiadas en cada caso para conseguir la modificación que estamos aplicando.

De esta manera eliminamos la limitación del navegador.

En este vídeo, *Creación de una directiva propia*, se explica cómo se crea una directiva propia.



Accede al vídeo

Tema 27. Inyección de dependencias

27.1. Introducción y objetivos

Este es un tema que no tiene la extensión de otros temas más importantes dentro de Angular, pero que explica muy bien un patrón de diseño que vamos a usar mucho en temas posteriores, que es la inyección de dependencias.

El objetivo que intentamos cumplir dentro de este tema no es otro que entender como es este patrón de diseño para que no nos resulte extraño cuando nos lo encontremos en temas posteriores.

Intentaremos explicar en un breve tema que es, como se usa una dependencia y cómo podemos aislarlas de nuestras clases para que estas sean más fuertes, flexibles y escalables.

27.2. ¿Qué es y para qué se usa?

La inyección de dependencias es un patrón de diseño muy usado en Angular. Se basa en que las clases no crean sus dependencias por sí mismas, sino que las reciben de fuentes externas.

Por ejemplo, si tienes un componente que depende de un servicio, con el que se comunica con la BBDD o con otros componentes, tal y como explicamos en el tema de inputs y outputs, no creas ese servicio en cada componente donde lo vas a usar,

si no que en el constructor del componente solicitas e inyectas el servicio que vas a usar y Angular te lo trae.

Este concepto no solo lo vamos a usar en los servicios si no en otros aspectos de angular como las rutas, por ejemplo.

De esta forma **nuestro código esta desacoplado**, ya que **cada cosa se encarga de lo que debe**, y el componente solo lo consume.

El ejemplo más común de una inyección de dependencias es el servicio que analizaremos más adelante junto con las peticiones externa por HTTP, pero del que ahora haremos una pequeña introducción.

El ejemplo que vamos a ver a continuación con la creación de un servicio viene a explicar lo que es una inyección de dependencia de forma clara. Un componente necesita de un elemento, en este caso un servicio, para funcionar correctamente, así que en lugar de realizar esa funcionalidad dentro del componente la sacamos fuera y así otros componentes la pueden usar a futuro.

Para usar esa dependencia tenemos por supuesto que importarla, pero para poder trabajar con ella debemos pasarla o inyectarla como parámetro dentro de la función constructor de nuestra clase.

Veamos un ejemplo práctico de este interesante concepto teórico para que quede más claro su uso.

Antes de nada, también puntualizar una cosa, una dependencia no tiene que ser solamente un servicio podría ser una función, por ejemplo, o un valor.

Cuando Angular crea una nueva instancia de una clase de componente, determina qué servicios u otras dependencias necesita ese componente al observar los tipos de parámetros del constructor.

Veamos un ejemplo visual:

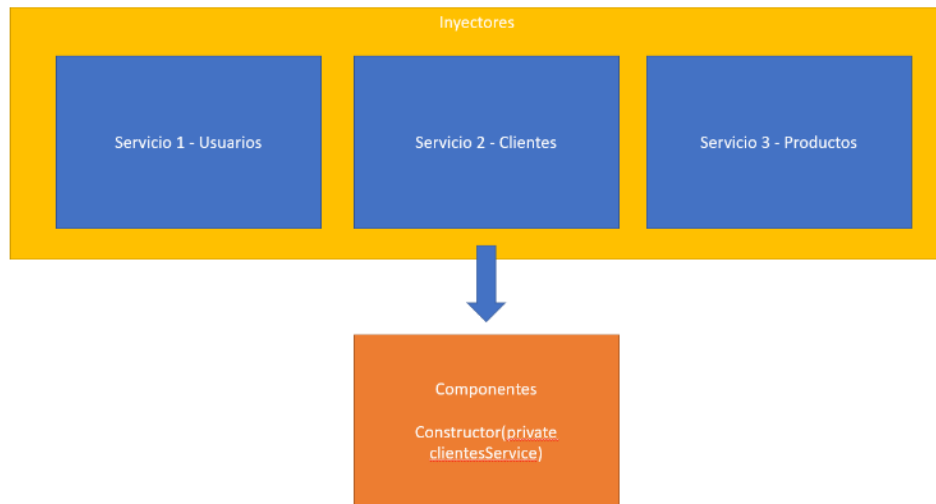


Figura 29. Explicación gráfica de inyección de dependencias. Fuente: elaboración propia.

27.3. Ejemplo de inyección de dependencia

Lo primero que necesitamos es crear una dependencia que podamos inyectar dentro de un componente.

En este caso y aunque tenemos un tema pensado exclusivamente para la generación de servicios, vamos a usar estos como ejemplo de inyección de dependencias puesto que es el ejemplo más claro dentro de los desarrollos propios que vamos a poder realizar dentro de un proyecto en Angular.

Para crear un servicio al igual que generamos una directiva y un componente tendremos que abrir nuestra terminal y situarnos dentro de la carpeta de proyecto. Una vez allí pondremos el siguiente código dentro en nuestro terminal.

```
ng generate service services/clientes --skip-tests
```

Esto generará dentro de la carpeta de trabajo src/app una carpeta services donde se alojarán los diferentes servicios de nuestra aplicación.

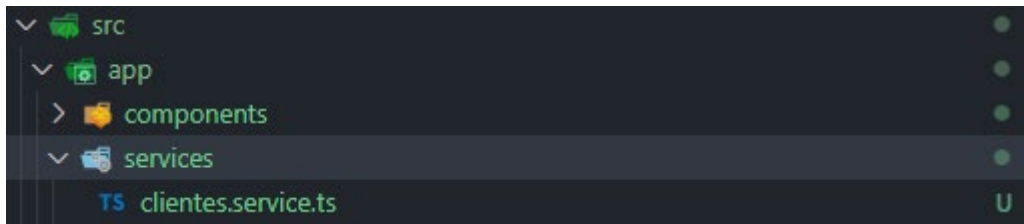


Figura 30. Árbol de carpetas tras la ejecución de la creación del servicio. Fuente: elaboración propia.

El servicio generado, para el ejemplo, va a contener un array de colores que le tenemos que hacer llegar a los diferentes componentes que consulten al servicio, para ello vamos a generar una propiedad privada dentro de la clase, llamada colores de la siguiente forma.

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

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

  private colores: any[] = ['blue', 'red', 'violet', 'yellow', 'green']
  constructor() { }
}
```

Hay dos puntos importantes en este código que se ha generado.

- El primer dato importante es que la clase está decorada por la función `@Injectable()`, un decorador que está especificando que este servicio está disponible para el root de la aplicación, es decir para todos los componente que dependan del módulo principal.

- El otro punto importante es que hemos creado un array de colores como **propiedad privada** dentro de la clase con lo que solo es accesible dentro de las funciones de la propia clase.

La única forma que tenemos de dar acceso a esa propiedad a otros componentes es creando un método dentro de esta clase que me retorne el valor de dicha propiedad, lo que **POO** se llama **GETTER**. De la siguiente forma.

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

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

  private colores: any[] = ['blue', 'red', 'violet', 'yellow', 'green']
  constructor() { }

  getColores(): any[]{
    return this.colores;
  }
}
```

Esto que acabamos de crear es un injectable de clientes que me devuelve un array de colores, que bien podría significar un array de categorías de clientes o una lista de clientes.

Ahora bien **¿cómo podemos usar esto dentro de nuestro componente?** Pues usando la DI (Dependency Injection) o inyección de dependencias.

Todo componente que recoja como parámetro de su constructor este servicio podrá usar sus método y propiedades como si estuviesen creadas dentro del propio componente.

Veamos cómo se lleva a cabo esta acción.

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

@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})

export class PrimerComponentesComponent implements OnInit {

  arrayColores: any[] = []

  constructor(private clientesServices: ClientesService) {
  }

  ngOnInit(): void {
    this.arrayColores = this.clientesServices.getColores()
  }

}
```

Dentro del componente hemos creado una propiedad para almacenar el array de colores y por ejemplo poderlo pintar dentro del HTML con un `*ngFor`.

Dentro del `constructor()` como argumento privado hemos inyectado el servicio creado previamente y lo hemos importado en la zona de importación arriba de nuestro fichero.

Ahora podemos, por ejemplo llenar el `arrayColores` cuando el componente está cargado, llamando al servicio inyectado previamente y ejecutando el método `getColores()` que habíamos creado en el servicio.

Con esta técnica el servicio puede suministrar esta información a cualquier componente que se la solicite de forma fácil, **simplemente aplicando una inyección de dependencias**.

Los servicios, ya lo veremos más adelante, puede contener multitud de funciones que sirvan datos a diferentes componentes, de esta forma podemos encapsular código reutilizable en cualquier situación y en cualquiera de nuestros componentes.

Tema 28. Formularios y validaciones

28.1. Introducción y objetivos

Las dos acciones fundamentales que cumple cualquier aplicación web son:

- ▶ Mostrar datos al cliente.
- ▶ Recoger datos por parte del cliente.

El objetivo de este tema es entender cómo funcionan los formularios en Angular como herramienta para recoger información que el usuario puede introducir en nuestras aplicaciones, ya sea para registrar datos o hacer un proceso de login, etc.

El elemento básico para la recuperación de estos datos dentro de una webapp son los formularios y conocerlos bien y ver qué posibilidades tienen es fundamental para realizar buenas aplicaciones que permitan al usuario interactuar con la propia aplicación.

Hasta ahora solo conocíamos una forma de recoger datos de un usuario que era con el Two Way Data Biding, a la que a cualquier input le asociábamos una propiedad de nuestra clase del componente y automáticamente se llenaba con los datos que el usuario introducía.

Esto está muy bien, pero es limitado en cuando a funcionalidad se refiere. Por esta razón los formularios son parte fundamental del desarrollo de una aplicación ya que cumplen ciertas funcionalidades.

- ▶ Recogen los datos por parte del usuario
- ▶ Controlan los cambios que se produzcan en los propios campos del formulario
- ▶ Deben validar dichos campos.

- ▶ Es importante que muestren los posibles errores derivados de los posibles datos introducidos

Angular posee un módulo especial para trabajar con todo lo relacionado a los formularios. Todas las funcionalidades vistas anteriormente ya se encuentran implementadas dentro de este módulo. Posee una gran variedad de validadores, aparte de la posibilidad de crear los nuestros propios.

Existe la posibilidad también de trabajar con validadores asíncronos que revisen los datos del usuario mientras los está introduciendo. Dicho módulo nos va a permitir trabajar con los campos de nuestros formularios como si fuesen propiedades dentro de un objeto

Existen dos maneras para definir formularios dentro de Angular:

- ▶ **Template driven:** son todos aquellos formularios que solo dependen de la plantilla del componente donde lo estemos definiendo.
- ▶ **Model driven:** en este caso, la configuración de los diferentes campos del formulario se realiza dentro de la clase del componente asociado. Son la opción más recomendable.

28.2. Formulario de tipo Template. FormsModule

Como ya hemos dicho en la introducción se tratan de módulos que se importan dentro de nuestro modulo principal para que todos nuestros componentes puedan usarlo.

Con lo que el primer paso en la creación de formularios, y poder usar esta potente característica dentro de nuestros componentes es importar el módulo `FormsModule` dentro del `app.module.ts`.

Este módulo es el que me permitía también trabajar con el Two Data Binding y es necesario importarlo para poder usar los formularios de tipo template.

Veamos cómo queda nuestro app-module.ts y como se importa dicho módulo.

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { PrimerComponentesComponent } from './components/primer-
componentes/primer-componentes.component';

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

En las plantillas donde estemos trabajando, debemos indicar qué formulario vamos a tratar a través de la directiva ngForm.

Debemos concretar qué campos son los que vamos a tratar a través de Angular.

En cada campo que vamos a tratar debemos incluir la directiva ngModel.

Aparte, mediante el evento `ngSubmit` podemos especificar qué método se va a encargar de recibir los datos del formulario cuando el usuario lo envíe.

Veamos un ejemplo de formulario en la parte del HTML del componente:

```
<h1>Formulario</h1>
<form          novalidate          #nuevaPersonaForm="ngForm"
  (ngSubmit)="onSubmit(nuevaPersonaForm.value)">
  <input type="text" name="nombre" ngModel />
  <button type="submit">Enviar</button>
</form>
```

Cada campo que necesitemos tratar va identificado con la directiva `ngModel` y un atributo `name` con el nombre del campo.

Generamos una nueva variable de plantilla (`#nuevaPersonaForm`) a la cual le asignamos el valor del formulario.

Esta variable nos permite recuperar los datos del formulario dentro del método que maneja dicho formulario.

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

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

  onSubmit(values: any) {
    console.log(values)
  }
}
```

`values` representa un objeto JSON que me devuelve un objeto cuyas claves son los `name` que le hemos puesto a cada uno de los campos de nuestro formulario.

Dentro del método `onSubmit()` puede comprobar que te llegan todos los datos del formulario que acabas de crear como un objeto que posteriormente puede usar para mandar a un servicio, a la base de datos a un array para almacenarlo.

Este tipo de formulario son muy sencillos y me ofrecen de forma fácil y rápida una forma de recoger valores del usuario de la página web.

En este vídeo, *Formulario tipo template*, se trata la creación de un formulario de tipo template. Uso y recogida de datos.



Accede al vídeo

28.3. Formulario de tipo Model. Reactive Forms

En este caso, es la clase del componente la encargada de definir el formulario.

Nos ofrece la posibilidad de:

- ▶ Creación de los campos.
- ▶ Generación de reglas de validación.
- ▶ Posibilidad de tratar con los cambios que se realicen en los campos del formulario.
- ▶ Se pueden generar de una manera más cómoda pruebas unitarias sobre los datos de un formulario.

La diferencia con la anterior fórmula reside en la potencia de nuestros desarrollos, es un tipo de formulario que me ofrece muchas más cosas que el anterior modelo y mucha más versatilidad. Template Driven sin embargo, es mucho más rápida en su implementación.

Para trabajar con este tipo de formularios, debemos importar dentro de la definición de nuestro módulo `ReactiveFormsModule`.

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { PrimerComponentesComponent } from './components/primer-
componentes/primer-componentes.component';

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

Es posible que en algunos proyectos necesitemos tener activos los dos módulos.

Trabajaremos con dos objetos básicos, **FormGroup** y **FormControl**, para generar grupos de controles y para identificar los controles en sí, respectivamente.

Vemos un ejemplo de clase en la que incorporamos la definición de un formulario.

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

@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})

export class PrimerComponentesComponent implements OnInit {

  form: FormGroup;

  constructor() {
    this.form = new FormGroup({
      nombre: new FormControl(''),
      apellidos: new FormControl(''),
      edad: new FormControl(''),
      direccion: new FormControl('')
    })
  }

  ngOnInit() {
  }

}
```

El siguiente paso sería modificar nuestra plantilla HTML para poder trabajar con los campos que hemos definido dentro del objeto formulario.

Ya no necesitamos especificar la directiva `ngModel` para cada uno de los campos, pero sí debemos concretar con qué propiedad se enlazan de las definidas dentro del objeto.

Esto lo conseguimos gracias a la directiva `formControlName`.

Vemos un ejemplo de plantilla a partir de los campos anteriores.

```
<form novalidate [formGroup]="form" (ngSubmit)="onSubmit()">
  <input type="text" name="nombre" formControlName="nombre">
  <input type="text" name="apellidos" formControlName="apellidos">
  <input type="text" name="edad" formControlName="edad">
  <input type="text" name="direccion" formControlName="direccion">
  <button type="submit">Enviar</button>
</form>
```

El evento `onSubmit()` recogerá todos los datos del formulario y los sacará por consola, de la misma forma que el formulario anterior me devolverá un objeto cuyas claves serán cada uno de los `formControlName` que tiene cada elemento del formulario.

28.4. Validaciones de Angular y validaciones personalizadas

Los formularios en Angular soportan dos tipos de validaciones:

- ▶ Las built-in validations.
- ▶ Las validaciones personalizadas.

Las validaciones del primer tipo las encontramos definidas dentro de la clase **Validators** en el paquete `forms` de Angular.

En la definición de nuestros controles, podemos agregar los validadores que necesitemos para cada campo concreto, como segundo parámetro recibimos un array de validadores donde podemos tener la cantidad de validadores por campo que deseemos.

Vemos un ejemplo con el formulario anterior a partir de una expresión regular para el nombre y de un validador required para la edad.

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

@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})

export class PrimerComponentesComponent implements OnInit {

  form: FormGroup;
  constructor() {
    this.form = new FormGroup({
      nombre: new FormControl('', [
        Validators.pattern('[\\w\\-\\s\\/]+'),
      ]),
      apellidos: new FormControl(''),
      edad: new FormControl('', [
        Validators.required
      ]),
      direccion: new FormControl('')
    })
  }

  ngOnInit() {

  }

  onSubmit(){
    console.log(this.form.value)
  }
}
```

Con la inclusión del segundo parámetro en la creación del objeto `FormControl` hemos conseguido que Angular detecte cuando el campo está incorrecto y en ese caso nos agrega al campo una nueva clase CSS llamada `ng-invalid`.

Si el campo es válido, elimina esa clase y coloca `ng-valid`.

Es tarea del desarrollador asegurarse que todos los campos están correctos antes de enviar el formulario. Angular **no se encarga de bloquear el envío**.

A partir de la instancia generada para manejar el formulario, podemos trabajar con las propiedades `valid` / `invalid`.

Si todos los campos cumplen con sus validaciones la propiedad `valid` obtiene el valor **true**.

Por lo tanto, podríamos controlar el envío del formulario con validaciones de este estilo:

```
<button type="submit" [disabled]="!form.valid">Enviar</button>
```

Dependiendo de la interacción que hayamos tenido con nuestros controles, vamos a ir viendo cómo se modifica su estado.

Aparte de los valores `valid/invalid` disponemos de otra serie de propiedades que nos permiten indicar qué estado tienen nuestros controles en cada momento.

Con `dirty` sabemos si el usuario ha modificado el valor del control o no. El caso contrario nos lo devuelve la propiedad `pristine`.

```
<pre>
Dirty -> {{ form.controls.nombre.dirty }}
Pristine -> {{ form.controls.nombre.pristine }}
</pre>
```

Obtendremos el valor true dentro de la propiedad touched si el usuario ha puesto el foco en un control y posteriormente pasa a otro.

Por ejemplo, cuando trabajamos sobre un campo y posteriormente, con el tabulador o haciendo **click** en otro control, cambiamos el foco.

La diferencia entre touched y dirty es que se puede activar la primera sin necesidad de modificar su valor.

El contrario es untouched.

```
<pre>
Touched -> {{ form.controls.nombre.touched }}
Untouched -> {{ form.controls.nombre.untouched }}
</pre>
```

Nota la etiqueta <pre> solo es para que se visualice en formato código por si queréis ponerlo a modo de comentario dentro de vuestros formularios, no significa absolutamente nada dentro del Angular

Si los validadores de un control concreto devuelve true, tendremos activa la propiedad valid.

El opuesto a esta propiedad es invalid.

```
pre>
Valid -> {{ form.controls.nombre.valid }}
Invalid -> {{ form.controls.nombre.invalid }}
</pre>
```

Validadores Custom

Para crear nuestros propios validadores necesitamos generar un método que recibe como parámetro el control a validar y devuelve null si está correcto o un objeto especificando los errores si la validación es incorrecta.

Veamos un ejemplo de cómo podríamos validar si la edad en nuestro formulario se encuentra entre los 18 y los 65.

```
edadValidator(control) {  
  if (control.value.trim().length === 0) return null  
  let edad = parseInt(control.value)  
  let minEdad = 18  
  let maxEdad = 65  
  if (edad >= minEdad && edad <= maxEdad) {  
    return null  
  } else {  
    return { 'message': 'La edad debe estar entre 18 y 65' }  
  }  
}
```

Este validador debemos ponerlo dentro del formControlName en el array de validadores del campo donde se aplique.

Ejemplo.

```
this.form = new FormGroup({  
  nombre: new FormControl('', [  
    Validators.pattern('[\\w\\-\\s\\/]+'),  
  ]),  
  apellidos: new FormControl(''),  
  edad: new FormControl('', [  
    Validators.required,  
    this.edadValidator  
  ]),  
  direccion: new FormControl('')
```

```
})
```

Manejo de errores

Aparte de las propiedades anteriores, para cada control disponemos de la propiedad `errors`, dentro de la cual queda definido si dicho control pasa o no los validadores.

Vemos un ejemplo con los campos anteriores.

```
<input type="text" name="edad" formControlName="edad">
<div *ngIf="form.controls.nombre.errors">
  La edad es incorrecta
</div>
```

Se puede ser incluso más específico haciendo referencia al validador en concreto sobre el cual queramos llamar la atención.

```
<input type="text" name="nombre" formControlName="nombre">
<div *ngIf="form.controls.nombre.errors?.required">
  El nombre es un campo requerido
</div>
```

El símbolo «?» indica que se va a validar la propiedad `required` siempre y cuando `errors` sea diferente de `null`.

Para nuestros validadores personalizados, podemos acceder al objeto que devolvemos en caso de que no se valide el valor.

```
<input type="text" name="edad" formControlName="edad">
<div *ngIf="form.controls.edad.errors">
  {{ form.controls.edad.errors?.message }}
</div>
```

Y así es como podemos gestionar las validaciones gráficamente dentro de nuestros formularios de tipo Model para lanzar los errores del usuario a nuestro interfaz.

En este vídeo, *Formulario de tipo Model*, explica el uso y recogida de datos con un formulario Model con Validaciones.



Accede al vídeo

Tema 29. Rutas y navegación

29.1. Introducción y objetivos

El objetivo principal de este tema es ver cómo podemos organizar nuestras aplicaciones en Angular para poder disponer de las características habituales de navegación web.

Siempre debemos tener en cuenta que estamos trabajando con aplicaciones de tipo SPA (Single Page Application), por lo que no podemos aplicar los mismos conceptos de enlaces que usamos cuando estamos maquetando con HTML.

Para poder trabajar con una página de tipo SPA podríamos activar o desactivar componentes dependiendo del estado de la aplicación que quisiéramos mostrar en cada caso.

En ese caso la **url** de nuestro navegador no se modificaría y podría llevarnos a una serie de problemas:

- ▶ Si refrescamos la aplicación volveríamos al principio.
- ▶ No se podrían guardar marcadores en el navegador para volver más adelante.
- ▶ Si compartimos el enlace con otra persona, no va a recoger el mismo estado en el que nos encontramos.

En las aplicaciones web tradicionales, cada vez que el usuario modifica la url se realiza una petición al servidor, que es el encargado de devolvernos el estado actual de la aplicación.

En nuestro caso, vamos a intentar evitar esto, mediante lo que podríamos llamar **Client Side Routing**.

El navegador donde estamos visualizando nuestra aplicación Angular será el encargado de construir el estado de la aplicación sin necesidad de esperar la respuesta del servidor.

El servidor únicamente nos devuelve la primera página a partir de la petición inicial.

Las ventajas en el uso de SPA son las siguientes:

- ▶ **Pueden ser más rápidas.** El cliente es capaz de actualizar la página sin necesidad de perder tiempo realizando peticiones al servidor.
- ▶ **Se requiere menos ancho de banda.** En vez de recuperar grandes páginas HTML del servidor solo pedimos los datos necesarios para mostrar en el cliente.
- ▶ **Se abarata el trabajo.** Un único desarrollador es capaz de enfrentarse a la mayoría de las funcionalidades necesarias para el desarrollo de una aplicación web.

29.2. Configuración básica. Módulo de rutas

Para poder manejar las rutas de nuestra aplicación Angular, disponemos del módulo **Router**.

La configuración de este módulo se suele encontrar dentro de un fichero llamado **app.routing.ts**.

Dentro de dicho fichero debemos implementar un array de objetos de tipo **Route** que van a identificar cada una de las rutas de nuestra aplicación.

Dentro de dicho fichero debemos implementar un array de objetos de tipo **Route** que van a identificar cada una de las rutas de nuestra aplicación.

Vemos una implementación sencilla de dicho fichero de configuración.

app.routing.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HolaComponent } from '../hola/hola.component';
import { HomeComponent } from '../home/home.component';

const routes: Routes = [
  { path: '', pathMatch: 'full', component: HomeComponent },
  { path: 'hola', component: HolaComponent }
];

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

Los objetos de tipo ruta básicos disponen de dos propiedades:

- ▶ **path:** define la url que vamos a manejar.
- ▶ **component:** se trata del nombre del componente que vamos a visualizar cuando se acceda a dicha ruta.

Hay que tener en cuenta que un objeto de tipo **Routes** no es más que una representación para poder utilizar un array de objetos de tipo **Route**.

El siguiente paso es comunicarle a nuestra aplicación cuáles serán las rutas que va a poder utilizar.

Para ello, podemos hacerlo dentro de los **imports** de **app.module.ts**.

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

import { AppRoutingModuleModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { PrimerComponentesComponent } from './components/primer-
componentes/primer-componentes.component';

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

Con estos simples pasos quedaría definida la configuración para algunas rutas simples en nuestra aplicación.

La pregunta es, **¿Dónde vamos a mostrar los componentes a los que hace referencia cada una de las rutas?**

Para representar el contenido dinámico de nuestra aplicación necesitamos llamar a la directiva **router-outlet** en alguna de nuestras plantillas HTML.

Esta directiva se encarga de situar dentro de la plantilla general de nuestra aplicación qué espacio vamos a ocupar con los componentes definidos dentro de las rutas

```
<header>
  Cabecera
</header>
<main>
  <router-outlet></router-outlet>
</main>
<footer>
  Footer
</footer>
```

En este ejemplo, la cabecera y el pie de página se mantienen siempre igual, mientras que la parte principal de la aplicación se modifica en función de la ruta que tengamos seleccionada.

Podemos configurar nuestras rutas de múltiples formas.

Una de las propiedades útiles puede ser `redirectTo`, la cual nos permite redireccionar el resultado de una ruta a otra de las que tengamos configuradas.

```
const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'home', component: HomeComponent},
  {path: 'hola', component: HolaComponent},
  {path: 'saludo', redirectTo: 'hola'}
];
```

En el ejemplo anterior, únicamente dos de las rutas renderizan un componente, el resto simplemente redireccionan a las rutas principales.

Por lo tanto, si en el navegador pasamos la ruta `‘/saludo’`, se modificará dicha ruta y ejecutará el componente asociado a `‘/hola’`, sobre la cual hemos redireccionado.

La propiedad `pathMatch='full'` le indica a Angular que, solo debe registrar dicha ruta si encuentra exactamente el string definido en la propiedad `path`.

Una buena práctica a la hora de definir nuestras rutas es la de especificar qué pasa si ninguno de los objetos definidos es capaz de responder a la ruta del navegador.

Podemos solucionarlo del siguiente modo:

```
const routes: Routes = [  
  {path: '', redirectTo: 'home', pathMatch: 'full'},  
  {path: 'home', component: HomeComponent},  
  {path: 'hola', component: HolaComponent},  
  {path: 'saludo', redirectTo: 'hola'},  
  {path: '**', component: HomeComponent}  
];
```

Para poder navegar entre las diferentes rutas que conforman nuestra aplicación no podemos usar el método tradicional (`a href`).

Si usamos enlaces normales, cada vez que hagamos **click** en alguno de ellos, el servidor lo interpretará como una nueva llamada y devolverá la aplicación completa de nuevo.

Podemos solventar, de manera provisional, este funcionamiento, mediante el uso de `HashLocationStrategy`.

```
@NgModule({  
  imports: [RouterModule.forRoot(routes, {useHash: true})],  
  exports: [RouterModule]  
})
```

Con esta nueva configuración, la aplicación se encarga de agregar el símbolo `#` entre el dominio y la ruta que estemos analizando.

Mediante el uso de este símbolo, cuando modificamos la ruta, el servidor entiende que no es una ruta nueva y por tanto no se realiza la petición.

Con este pequeño cambio podríamos utilizar los enlaces normales, pero no es una solución completa.

En Angular disponemos de diferentes formas para poder especificar la navegación dentro de nuestras aplicaciones.

La primera de ellas sería **inyectando el servicio Router** dentro de nuestros componentes.

Vemos un ejemplo sobre un nuevo componente.

```
<ul>
  <li><a (click)="goToPage('home')">Home</a></li>
  <li><a (click)="goToPage('hola')">Saludo</a></li>
</ul>
```

La clase del componente que emite esos eventos quedaría de la siguiente manera.

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})

export class PrimerComponentesComponent implements OnInit {
  constructor(private router: Router) {
  }
  ngOnInit() { }

  goToPage(path: any) {
```

```
this.router.navigate([path])
}
}
```

El parámetro que se pasa a la función **navigate** es un poco extraño.

Se trata de un array, el cual se corresponde con cada una de las partes de la **url** a la que queremos navegar.

Por ejemplo, si tenemos definida la siguiente ruta:

```
{path: 'hola/mundo/angular', component: HolaComponent}
```

Podemos hacer la llamada al método **navigate** de la siguiente manera:

```
this.router.navigate(['hola', 'mundo', 'angular'])
```

A continuación, puedes ver el siguiente vídeo, *Creación de rutas y enlazarlas*.



Accede al vídeo

29.3. RouterLink y RouterLinkActivate

Aparte del uso de **Router** para navegar a través de la aplicación tenemos la posibilidad de utilizar la directiva **routerLink** para hacerlo directamente desde la plantilla.

Modificamos la plantilla de la cabecera anterior.

```
<ul>
  <li><a [routerLink]="['home']">Home</a></li>
  <li><a [routerLink]="['hola']">Saludo</a></li>
</ul>
```

La directiva **routerLink** recibe los mismos parámetros que el método **navigate**.

Como ayuda en el trabajo con nuestros enlaces, disponemos de la directiva **routerLinkActive**.

Nos permite pasar un array con las clases que se le aplicarán a la ruta activa en cada momento.

Esto nos permite especificar una serie de estilos CSS concretos a la ruta activa.

```
<ul>
  <li><a [routerLink]="['home']"
    [routerLinkActive]="['active']">Home</a></li>
  <li><a [routerLink]="['hola']"
    [routerLinkActive]="['active']">Saludo</a></li>
</ul>
```

En ocasiones necesitaremos que nuestras rutas contengan ciertos elementos variables.

Podemos definir variables dentro de nuestras rutas de la siguiente manera:

```
{ path: 'hola/:saludo', component: HolaComponent }
```

Las variables que definamos deben ir precedidas del símbolo. Podemos definir tantas variables como necesitemos para enviar información a los componentes que resuelvan cada una de nuestras rutas.

El siguiente objetivo consiste en recibir esas variables dentro del componente.

En este vídeo, *Pasar parámetros de ruta*, vemos cómo se pasan los parámetros por la ruta.



Accede al vídeo

29.4. Parámetros de ruta: Activated Route

Para poder recuperar este tipo de variables dentro del componente que corresponda disponemos del objeto `ActivatedRoute`.

Debemos inyectar una instancia en nuestros componentes y posteriormente acceder a la propiedad `params` para recuperar los valores recibidos.

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

@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})

export class PrimerComponentesComponent implements OnInit {

  //inyección de dependencia de activatedRoute
  constructor(private activatedRoute: ActivatedRoute) {}

  ngOnInit() {
```

```
// Observable que se suscribe a los cambios en la ruta activa
this.activatedRoute.params.subscribe(params => console.log(params))
}

}
```

Existen ocasiones en las que necesitamos anidar diferentes rutas para que, dentro de un mismo componente, podamos diferenciar la información.

Estas rutas deben compartir, aparte de la estructura, las variables que definamos.

Para ello, podemos utilizar la propiedad `children` en la definición de la ruta.

Lo vemos con un ejemplo sobre las siguientes rutas:

- ▶ `/autor/1234`: devuelve los datos del autor cuyo identificador es 1234.
- ▶ `/autor/1234/libros`: devuelve todos los libros del autor 1234.
- ▶ `/autor/1234/reviews`: devuelve todas las reseñas del autor 1234.

Todas las rutas deberían estar relacionadas para poder agilizar el trabajo y posibilitar la sencilla ampliación del código.

29.5. Rutas Hijas

Para poder trabajar con estas rutas anidadas, necesitamos especificar, dentro de la plantilla padre, dónde se van a renderizar.

Para ello, solo tenemos que incluir la directiva **router-outlet** donde necesitemos. La generación de las rutas sería la siguiente.

```

{
  path: 'autor/:id', component: AutorComponent,
  children: [
    { path: 'libros', component: LibrosComponent },
    { path: 'reviews', component: ReviewsComponent },
  ]
}

```

Dentro de la plantilla HTML de AutorComponent incluimos la directiva **router-outlet** indicando dónde aparecerán los componentes hijo.

Para acceder a los parámetros del padre desde cualquiera de los componentes hijo, podemos hacerlo de la siguiente manera.

```

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})
export class PrimerComponentesComponent implements OnInit {
  constructor(private activatedRoute: ActivatedRoute) {}
  ngOnInit() {
    this.activatedRoute.parent.params.subscribe(params =>
console.log(params))
  }
}

```

A continuación, puedes ver el siguiente vídeo, *Creación de rutas hijas*.



Accede al vídeo

29.6. LocalStorage y Guards

En las aplicaciones tradicionales, cuando se accede a una ruta no permitida, el servidor devuelve el estado 403 indicando al cliente el error.

Otra opción es que se redirija al usuario hacia otra página que informe del posible error de acceso.

En nuestras aplicaciones de tipo SPA podemos lograr esta funcionalidad gracias a los **Router Guards**.

Utilizando esta metodología podemos prevenir el acceso del usuario a zonas específicas de nuestra aplicación.

Existen varios tipos de **Guards**:

- ▶ **CanActivate**. Comprueba si el usuario puede acceder a una ruta en concreto.
- ▶ **CanActivateChild**. Comprueba si el usuario puede acceder a una ruta concreta dentro de tipo **children**.
- ▶ **CanDeactivate**. Comprueba si el usuario puede abandonar una ruta.
- ▶ **Resolve**. Obtiene los datos de la ruta antes de que se active.
- ▶ **CanLoad**. Comprueba si un usuario puede desplazarse hasta una ruta de otro módulo.

Para una ruta podemos implementar el número de **Guards** que necesitamos.

Los **Guards** se implementan como servicios, por lo que necesitamos generar una clase con el decorador `@Injectable`.

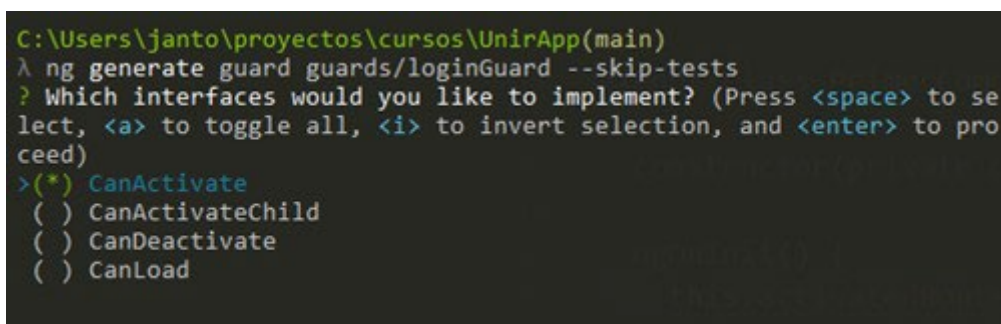
Estas clases tendrán que devolver **true** o **false** dependiendo si se cumplen o no los condicionales que especifiquemos

También podrían devolver una **Promesa** o un objeto **Observable** si la serie de comprobaciones que necesitemos hacer tienen una carga superior al habitual.

Para crear un guard tenemos que volver al terminal para teclear el siguiente código

```
ng generate guard guards/loginGuard --skip-tests
```

Cuando ejecutemos este comando en la terminal nos preguntará que tipo de guard queremos aplicar.



```
C:\Users\janto\proyectos\cursos\UnirApp(main)
λ ng generate guard guards/loginGuard --skip-tests
? Which interfaces would you like to implement? (Press <space> to se
lect, <a> to toggle all, <i> to invert selection, and <enter> to pro
ceed)
>(*) CanActivate
  ( ) CanActivateChild
  ( ) CanDeactivate
  ( ) CanLoad
```

Figura 31. Menú de elección de tipo de guard en terminal. Fuente: elaboración propia.

```
import { CanActivate } from '@angular/router'
export class SiemprePasaGuard implements CanActivate {
  canActivate() {
    console.log('Entrando Siempre Pasa Guard')
    return true
  }
}
```

El funcionamiento de este **Guard** es totalmente trivial ya que siempre devuelve true. Podríamos definir todas las comprobaciones necesarias dentro del método que implementa para permitir o no el acceso al cliente. Debemos declarar la clase que hemos desarrollado dentro de los **providers** globales de nuestro módulo. Por último, especificar, en la ruta donde queramos filtrar el acceso, qué **guards** son los que va a utilizar.

```
{ path: 'home', component: HomeComponent, canActivate: [loginGuard]
```

Como la propiedad `canActivate` de la ruta es un array, podríamos especificar todos los **guards** que necesitemos.

Vemos otro ejemplo a partir del guard `CanDeactivate`.

Habitualmente, usamos este método para comprobar si el usuario puede navegar hacia otra ruta, dejando algún elemento sin salvar en el componente anterior.

Una de las ventajas que nos ofrece esta manera de trabajar es que tenemos acceso directo al componente que estamos abandonando y deberíamos ser capaces de realizar comprobaciones sobre sus propiedades.

El primer paso es definir la clase.

```
import { HomeComponent } from './home/home.component';
import { Injectable } from '@angular/core';
import { CanDeactivate } from '@angular/router';
@Injectable()
export class PuedoSalirGuard implements CanDeactivate<HomeComponent>{
  canDeactivate(component: HomeComponent): boolean {
    return component.canDeactivate()
  }
}
```

En el ejemplo anterior hemos podido acceder desde el **Guard** al Componente sobre el que estamos trabajando, por lo que podemos hacer cualquier tipo de comprobación sobre su estado.

Aparte del componente, esta serie de métodos pueden recibir como parámetro un objeto de tipo `ActivatedRouteSnapshot` y/o uno de tipo `RouterStateSnapshot`.

El primero de ellos corresponde a la siguiente ruta que se va a activar si pasa el guard. El segundo corresponde al **Router** que obtendremos si pasamos el guard. Con estos dos objetos podemos recuperar datos sobre los parámetros usados o la url.

Tema 30. Servicios y Http

30.1. Introducción y objetivos

En temas anteriores vimos la importancia que tenía para Angular que ciertos elementos fueran inyectados dentro de los componentes que se iban a usar.

Introducimos brevemente el concepto de servicio y como se implementaba. Ahora el objetivo de este tema es ahondar un poco más en el concepto de servicios, pero esta vez no solo para enviar información entre componentes si no para hacer peticiones HTTP externas a servicios API-REST con la intención de pedir datos para posteriormente pasárselos a los diferentes componentes que maneje nuestra aplicación.

Dentro de Angular los servicios son de gran utilidad para poder conectar nuestra aplicación con elementos externos a ella, y para ello Angular nos provee de un módulo que me permite gestionar esas peticiones, que se llama `HttpModule`.

En este tema vamos a ver que mezclando estos dos conceptos podemos suministrar a nuestras aplicaciones de Angular un sinfín de posibilidades.

30.2. ¿Qué es un servicio?

Aunque en un tema anterior vimos lo que era un servicio vamos a repasar el concepto ya que es un punto muy importante de las aplicaciones de Angular.

Resumiendo, brevemente, un servicio es una clase de Angular que posee un conjunto de funciones de JavaScript que se encargan de realizar una tarea específica dentro de la aplicación.

El concepto de servicio dentro de Angular se alcanza gracias al uso de la **inyección de dependencias**.

Son extremadamente útiles si tenemos que compartir información entre los componentes que forma la aplicación.

Como ya vimos un servicio se inyecta en un componente siguiendo estos dos pasos:

- ▶ El primero es el registro de aquellos servicios que queremos poner en conocimiento de Angular para su inyección. Este registro lo alcanzamos gracias a los decoradores disponibles (`@Injectable`).
- ▶ El segundo es la propia inyección de dichos servicios dentro de los constructores de las clases donde vayan a ser usados. Puede hacerse también a través de los decoradores de Inyección (`@Inject`).

Cuando en cualquier clase de nuestra aplicación requiramos uno de estos servicios, Angular comprueba si ya existe una instancia de esa clase creada con anterioridad.

Si ya existe, recoge la instancia y la inyecta. Si no existe, instancia un nuevo objeto de la clase que nos interese, lo inyecta y lo almacena para inyecciones posteriores.

Esta forma de manejar clases en programación sigue un patrón de desarrollo muy concreto que, aunque no hace falta entrar en detalle en este tema se denomina **singleton**.

Todos estos servicios se mantienen en memoria mientras el usuario trabaja. Como la página no se refresca, esta serie de elementos se mantienen activos.

Los servicios registrados existen en el componente que lo registra y sus hijos. Si los registramos al inicio de nuestra aplicación, estará disponible para todos los componentes que vayamos generando.

Para **crear nuestras propias clases inyectables** podemos usar el decorador `@Injectable`.

El concepto de servicio no es algo específico de Angular, ya que se trata de un patrón muy usado en otro tipo de frameworks o desarrollos.

Se trata de **una clase** que contiene funcionalidad que nosotros podemos exportar dentro de nuestra aplicación.

No necesitamos ningún código específico de Angular para trabajar con nuestros servicios, son clases normales

Ya vimos en temas anteriores que para crear un servicio aplicamos el siguiente comando en la terminal dentro del proyecto donde estemos trabajando.

```
ng generate service services/clientes --skip-tests
```

A continuación, puedes ver el siguiente vídeo, *Creación de un servicio*.



Accede al vídeo

30.3. Comunicación del servicio con los componentes

¿Por qué debemos usar los servicios?

Para definir clases que nos permitan trabajar con una serie de datos y organizar cómo vamos a recibirlos dentro de nuestra aplicación.

También para compartir métodos útiles que serán usados por la mayoría de componentes de nuestra aplicación.

Para compartir métodos útiles que serán usados por la mayoría de las componentes de nuestra aplicación.

Angular tiene sus propios servicios con funcionalidades específicas (Http, FormBuilder, Router...). Con el uso de este tipo de servicios vamos a ser capaces de aislar nuestros componentes de la lógica de negocio de nuestra aplicación.

Por ejemplo, si estamos desarrollando las pruebas unitarias de nuestra aplicación, es muy sencillo, a través de un servicio, simular los datos que entran en nuestros componentes y así poder implementar cualquier caso de uso.

En este caso nos centramos más en el buen funcionamiento del componente que en organizar los datos que necesitamos para su funcionamiento.

En el momento de decidir si vamos a crear un servicio o no, tenemos que hacernos el siguiente planteamiento:

Generamos un servicio si lo que va a representar es información independiente de la plantilla que posteriormente utilizaremos para mostrarlo.

Generamos un servicio si dicha información vamos a necesitar que se represente dentro de nuestra aplicación de varias maneras diferentes.

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class ClientesService {
  constructor() { }
}
```

La inclusión del decorador Injectable permite a Angular situar esta clase como una de las disponibles para poder ser inyectada en cualquier otro elemento de la aplicación.

Uno de los conceptos importantes a la hora de desarrollar nuestros propios servicios es hacerlos totalmente opacos a los posibles consumidores de datos.

Es decir, debemos **seguir las buenas prácticas de encapsulación**: no dar acceso público a las propiedades de nuestra clase y disponer de una serie de métodos que serán los encargados de recoger los datos correspondientes.

Desde fuera de la clase se podrá llamar a esa serie de métodos, pero se desconoce cómo se obtienen los datos que devuelven.

Simplemente se debe dar a conocer de los métodos cuál es su nomenclatura, los parámetros que reciben y el resultado que devuelven.

El desarrollo de servicios nos da mucha flexibilidad a la hora de recoger los datos que vamos a mostrar dentro de nuestra aplicación.

Si vamos a trabajar con datos de producción, nuestros servicios apuntarán a la base de datos de producción.

Si estamos en un entorno de pruebas podemos apuntar nuestros servicios a cualquier fichero estático que nos permita recuperar los datos idóneos para las pruebas (Mock).

Veamos un ejemplo sencillo a partir del ejemplo anterior.

Lo primero que vamos a hacer es generar un interfaz en TypeScript que de forma a un modelo de tipo Persona. En el cómo ya vimos en el tema de TypeScript generamos un contrato que al implementarlo estamos obligando a nuestro array de datos a seguir ese modelo con esos campos.

Para generar un interfaz en angular aplicamos el siguiente código en nuestra terminal dentro de la carpeta de proyecto.

```
ng generate interface interfaces/cliente
```

El interfaz define un cliente con lo que el nombre del archivo será en singular.

```
export interface Cliente {  
  nombre: string;  
  apellidos: string;  
  direccion: string;  
}
```

Una vez generado el interfaz y por lo tanto el contrato que tiene que seguir nuestro array de datos creamos dentro de una carpeta **db** un archivo **mock-clientes.ts**

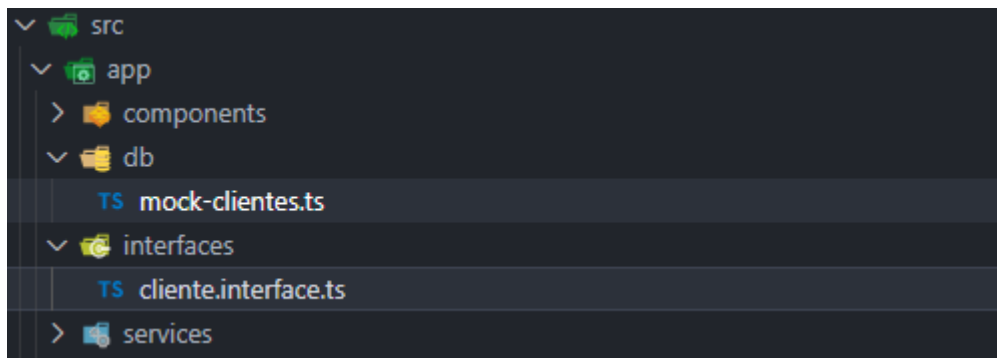


Figura 32. Árbol de carpetas. Fuente: elaboración propia.

El fichero contendrá un array de clientes que seguirá el modelo del interfaz de cliente, de la siguiente forma.

```
import { Cliente } from "../interfaces/cliente.interface";

export const Clientes: Cliente[] = [
  { nombre: "Antonio", apellidos: "Garcia", direccion: "C Gran Vía" },
  { nombre: "Raul", apellidos: "Gonzalez", direccion: "C Callao" },
  { nombre: "Mario", apellidos: "Girón", direccion: "C Ballesta" },
  { nombre: "Rosa", apellidos: "Jimenez", direccion: "C La Paz" },
  { nombre: "Sandra", apellidos: "Marti", direccion: "C Infantas" }
]
```

Ahora los importamos dentro del servicio y creamos un método `getClientes()` que lo que haga es retornar el array de clientes. Dicho método permitirá a los componentes que inyecten este servicio obtener el listado de clientes de la lista mock.

```
import { Injectable } from '@angular/core';
import { Clientes } from '../db/mock-clientes';
import { Cliente } from '../interfaces/cliente.interface';

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

```

    constructor() { }

    getAll(): Cliente[] {
        return Clientes
    }
}

```

El siguiente paso para poder recuperar los datos del servicio que acabamos de crear sería importarlo dentro de la clase donde vayamos a usarlo.

Siguiendo la lógica de la Programación Orientada a Objetos, para poder usar esta clase, necesitaríamos instanciar un objeto.

Este concepto se soluciona mediante la utilización de la **inyección de dependencias** de dicha clase.

Para poder disponer de una instancia del servicio dentro del componente donde lo vamos a utilizar, simplemente debemos especificarlo en el constructor.

Vemos el ejemplo con el caso anterior.

Primero dentro de nuestro componente creamos un array de clientes que luego nos servirá para pintarlo dentro de nuestro HTML.

Dicho array lo llenamos a través del servicio inyectado dentro del componente, llamando al método que hemos implementado antes `getAll()`.

```

import { Component, OnInit } from '@angular/core';
import { Cliente } from 'src/app/interfaces/cliente.interface';
import { ClientesService } from 'src/app/services/clientes.service';

@Component({
    selector: 'app-primer-componente',

```



```

    templateUrl: './primer-componentes.component.html',
    styleUrls: ['./primer-componentes.component.css']
  })

  export class PrimerComponentesComponent implements OnInit {

    //inicializamos un array de tipo cliente vacio
    clientes: Cliente[] = [];
    constructor(private clientesServices: ClientesService) { }

    //dentro del metodo de carga del componente llamamos al servicios y
    llenamos el array de clientes
    ngOnInit() {
      this.clientes = this.clientesServices.getAll()
    }

  }

```

Una vez tengamos la propiedad array de clientes llena podremos ir a nuestra plantilla HTML y pintar nuestro resultado a través de una directiva `*ngFor`. (La maquetación realizarla como vosotros queráis).

```

<ul>
  <li *ngFor="let cliente of clientes">{{cliente.nombre}}
  {{cliente.apellidos}}</li>
</ul>

```

El constructor no hace falta que haga nada.

Esta sintaxis implica que se ha creado, dentro del componente, una nueva propiedad privada que apunta a una instancia generada a partir de la clase `ClientesServices`.

Como podemos observar, el componente es opaco a la obtención de los datos (mock, localStorage, server...).

30.4. HttpClientModule

Una de las capacidades básicas que deben tener nuestras aplicaciones web es la de poder conectarse con HTTP APIs externas.

A partir de estas conexiones podremos recuperar la mayoría de los datos a mostrar dentro de nuestras aplicaciones Angular.

Angular dispone de su propio módulo preparado para poder realizar dichas llamadas de manera muy sencilla.

Una de las premisas que debemos tener en cuenta si realizamos este tipo de trabajos es que cualquier conexión que conlleve una carga excesiva para nuestra aplicación, se debe realizar de manera asíncrona.

En ningún caso nuestra aplicación se debe quedar esperando a recibir cualquier tipo de respuesta.

El servicio `HttpClient` disponible dentro de la librería `@angular/common/http` nos ofrece una interfaz sencilla para poder realizar los diferentes tipos de peticiones **HTTP** necesarias para realizar comunicaciones a través de internet

El servicio trabaja por encima de `XMLHttpRequest`, facilitando el acceso a las diferentes herramientas que nos proporciona.

El uso de esta librería frente a la aplicación de las herramientas nativas de **JavaScript** nos proporciona múltiples ventajas cómo, por ejemplo, la cantidad de facilidades a la hora de realizar pruebas en la aplicación.

Para poder utilizar el servicio `HttpClient`, primero debemos incluir dentro de `AppModule` el módulo correspondiente (`HttpClientModule`).

```

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { PrimerComponentesComponent } from './components/primer-
componentes/primer-componentes.component';

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

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

```

Una vez incluido el módulo, podemos inyectar `HttpClient` dentro del servicio donde necesitemos usarlo. En nuestro caso en el servicio de clientes.

```

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

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

  constructor(private httpClient: HttpClient) { }

}

```

30.5. Peticiones GET, POST, PUT, DELETE

El protocolo Http dispone de una serie de verbos para cada una de las urls, los cuales nos permiten especificar qué acciones vamos a llevar a cabo.

Los verbos **Http** más utilizados son **get**, **put**, **post**, **delete**. Forma parte de lo que se conoce como CRUD de peticiones, con las que podemos guardar, leer, borrar y actualizar datos dentro de cualquier servicio de API.

- ▶ **Get:** Se usa para obtener datos del API.
- ▶ **Post:** Se usa para enviar y guardar nuevos datos en el API.
- ▶ **Put:** Se usa para actualizar datos dentro del API.
- ▶ **Delete:** Se usa para borrar datos dentro del API.

Dentro del servicio vamos a lanzar peticiones con los diferentes verbos para poder ver cómo podemos enfrentarnos a los diferentes casos.

30.6. Promesas y observables

Todas las funciones que aplicaremos a continuación devuelven como resultado un **observable**. Aunque luego veremos que podemos convertir esos observables en **promesas** de forma sencilla, y así gestionar con más flexibilidad las peticiones asíncronas.

En este punto vamos a ver y estudiar las dos formas de gestionar las peticiones HTTP tanto con **promesas** como con **observables**.

Para ilustrar este ejemplo vamos a usar una API-FAKE (un api gratuito que me sirve diferentes tipos de datos para hacer pruebas).

En este caso este api se llama JSONPlaceholder y esta es su url:
<https://jsonplaceholder.typicode.com/>

Dentro del apartado «Resources» encontrareis diferentes recursos que nos puede servir para hacer peticiones **get** y poder hacer prueba para traeros datos a vuestras aplicaciones. Tenéis tanto **post, photos y users**.

Resources

JSONPlaceholder comes with a set of 6 common resources:

| | |
|---------------------------|--------------|
| /posts | 100 posts |
| /comments | 500 comments |
| /albums | 100 albums |
| /photos | 5000 photos |
| /todos | 200 todos |
| /users | 10 users |

Figura 33. JSONPlaceholder Resources. Fuente: elaboración propia.

También puedes encontrar que con algunos recursos te dan Routes para probar los diferentes verbos anteriormente mencionados: **get, post, delete, put**.

Estos recursos son los que vamos a usar para ver cómo se hacen peticiones a api externas y como a través de observables y promesas podemos gestionarlas para después que nuestros componentes puedan usarlas.

Todos los ejemplos que vais a ver en esta parte del tema las rutas las hemos cogido de esta web como recurso para que podáis tranquilamente ir a buscarlas y consultarlas libremente, ya que son gratuitas.

Routes

All HTTP methods are supported. You can use http or https for your requests.

| | |
|--------|---|
| GET | /posts |
| GET | /posts/1 |
| GET | /posts/1/comments |
| GET | /comments?postId=1 |
| POST | /posts |
| PUT | /posts/1 |
| PATCH | /posts/1 |
| DELETE | /posts/1 |

Figura 34. JSON PLACEHOLDER. Captura de pantalla. Fuente: elaboración propia.

Observables

Los observables es la forma nativa en la que angular gestiona las peticiones a servidores externos. Para que os hagáis una idea sencilla sería como tener una persona permanente observando cualquier cambio que se produzca en un sitio y que nos avise cuando se produzca, pues bien, esto es un observable. Una función dentro del servicio de Angular que nos permite estar atentos a cualquier cambio en los datos y comunicárselo en tiempo real a nuestros componentes.

Para usar los **observables** no tenemos que hacer nada ya viene de forma nativa en angular, solo que cuando creamos la función que devuelva un observable se nos debe importar la librería nativa en el servicio de la siguiente manera.

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

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

```

constructor(private httpClient: HttpClient) { }

peticionGetObservable(): Observable<any[]> {
    return
    this.httpClient.get<any[]>('https://jsonplaceholder.typicode.com/users')
    }
}

```

Como veis arriba en las importaciones aparece los Observables de la librería rxjs de Angular.

Con ella podemos crear tal y como muestra el fragmento de código una función `peticionGetObservable()` que retorna una petición `get` a la url que esta entre paréntesis.

Para poder usar este método y traerme los datos necesito ir a un componente y hacer una llamada a la función que me devolverá o bien los datos o bien una respuesta en función del tipo de verbo que usemos para hacer la petición ya sea **get, post, put o delete**.

Veamos como llamaríamos desde un componente a este método y como manejaríamos el observable antes de meternos de lleno en las promesas.

```

import { Component, OnInit } from '@angular/core';
import { ClientesService } from 'src/app/services/clientes.service';

@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})

export class PrimerComponentesComponent implements OnInit {

```

```

//inicializamos un array de tipo cliente vacio
clientes: any[] = [];
constructor(private clientesServices: ClientesService) { }

//dentro del metodo de carga del componente llamamos al servicios y
llenamos el array de clientes
ngOnInit() {
  this.clientesServices.petitionGetObservable().subscribe(data => {
    this.clientes = data;
  })
}
}

```

Como podéis observar el componente cuando se carga en el ngOnInit hace una petición al servicio y como es un observable, tal y como hicimos anteriormente, con el activatedRoute, nos suscribimos a los cambios que se produzcan dentro de nuestro API almacenándolos en la propiedad this.clientes de tal forma que podemos pintarlo dentro de HTML de nuestro componente a través de una directiva ngFor por ejemplo.

El método suscribe es la función que usamos para gestionar los observables en angular nos suscribimos a los cambios y cuando nos llegan los almacenamos donde los necesitamos.

Promesas

Las promesas son probablemente uno de los avances más importantes para la gestión de peticiones asíncronas que se ha producido en JavaScript, y tanto si se usa con then/catch como con async/await, es una forma muy interesante y sobre todo continuista de como lo hacemos en JavaScript nativo.

Ahora vamos a hacer el mismo ejemplo anterior pero convertido en promesa y vamos a ver cómo gestionarlo dentro de nuestro componente con async/await.


```

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

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

  constructor(private httpClient: HttpClient) { }

  petitionGetPromesa(): Promise<any[]> {
    return
    lastValueFrom(this.httpClient.get<any[]>('https://jsonplaceholder.typicode
    .com/users'))
  }
}

```

Como vemos ahora un importamos la librería observable sino LastValueFrom, esta librería me aporta un funcionalidad que me permite convertir un Observable en una promesa, tal y como se muestra la función petitionGetPromesa(). Por lo demás la petición no cambia en absoluto a como lo hicimos antes.

Ahora bien, en nuestra componente lo gestionaremos de forma distinta ya que lo que recibimos ya no es un observable si no una promesa.

```

import { Component, OnInit } from '@angular/core';
import { ClientesService } from 'src/app/services/clientes.service';

@Component({
  selector: 'app-primer-componente',
  templateUrl: './primer-componentes.component.html',
  styleUrls: ['./primer-componentes.component.css']
})

```

```

export class PrimerComponentesComponent implements OnInit {

    //inicializamos un array de tipo cliente vacio
    clientes: any[] = [];
    constructor(private clientesServices: ClientesService) { }

    //dentro del metodo de carga del componente llamamos al servicios y
    //llenamos el array de clientes

    async ngOnInit() {
        this.clientes = await this.clientesServices.peticionGetPromesa()
    }

}

```

En este caso en el `ngOnInit` llenaremos la propiedad `this.clientes` con un `async/await` la petición es una promesa y una de las mejores formas de gestionarla es de esta forma.

Cuando el dato nos llegue lo almacenaremos de forma asíncrona dentro de nuestra propiedad `this.clientes` para posteriormente consumirla dentro de nuestro HTML a través de un `ngFor`, por ejemplo.

Bien, una vez visto el ciclo completo de una petición a un componente tanto con observables como con promesas vamos a ver diferentes ejemplos con distintos verbos de petición, esta vez simplemente la función que montaríamos dentro del servicio.

NOTA: Para gestionar un observable convertido en promesa podemos usar tanto los métodos `lastValueFrom` o `firstValueFrom` para transformar el resultado, indistintamente.

Veamos un ejemplo con cada uno de los verbos, esta vez solo desde el punto de vista de la función del servicio.

El método `get` recibe como parámetro la URL a la que vamos a enviar la petición y debemos concretar qué nos devuelve el método y qué resuelve la promesa.

```
peticionGet(): Promise<any[]> {  
    return  
    lastValueFrom(this.httpClient.get<any[]>('https://jsonplaceholder.typicode  
    .com/posts'));  
}
```

Usamos la petición `post` para poder enviar información al servidor con el que estamos interactuando. Debemos incluir el `body` como segundo parámetro.

```
peticionPost(): Promise<any> {  
    const body = {  
        param: 'value1',  
        param2: 'value2'  
    }  
    return  
    lastValueFrom(this.httpClient.post<any>('https://jsonplaceholder.typicode.  
    com/posts', body))  
}
```

De la misma manera podemos lanzar peticiones de tipo `put` a través del método `put`. El método se usa de la misma manera que `post`.

```
peticionPut(): Promise<any> {  
    const body = { param: 'value1' };  
    return  
    lastValueFrom(this.httpClient.put<any>('https://jsonplaceholder.typicode.c  
    om/post', body))  
}
```

Si necesitamos lanzar una petición `delete`, podemos hacerlo a través del método homónimo.

```
peticionDelete() {
```

```

return
lastValueFrom(this.httpClient.delete('https://jsonplaceholder.typicode.com
/post'));
}

```

Cualquiera de los métodos anteriores, cuando necesitemos ejecutarlos los tratamos con los métodos then y catch o con async-await.

En un ejemplo anterior lo gestionamos con async/await, aquí lo vemos con then/catch.

```

async ngOnInit() {
  this.clientesServices.peticionGet()
    .then(response => {
      // Recibe un objeto con la respuesta del servidor
      console.log(response);
    })
    .catch(err => {
      console.log(err);
    })
  const response = await this.personasService.peticionGet();
}

```

Cabeceras

Las cabeceras son todos aquellos metadatos que el navegador adjunto a la petición enviada al servidor.

Enviamos elementos como, la IP desde la cual se manda la petición o el navegador que utilizamos.

Determinadas APIs requieren que especifiquemos ciertas cabeceras personalizadas. Podemos implementar las cabeceras de la petición con la siguiente estructura.

Aquí ponemos un ejemplo de una petición desde el servicio que necesita un token de validación para que el servidor acepte mi petición de datos.

```
peticionGet(): Promise<any[]> {  
  const httpOptions = {  
    headers: new HttpHeaders({  
      'Authorization': 'TOKEN'  
    })  
  }  
  
  return  
  lastValueFrom(this.httpClient.get<any[]>('https://jsonplaceholder.typicode  
  .com/posts', httpOptions))  
  
}
```

Como veis creamos una const httpOptions que implemente un objeto HttpHeaders que previamente hay que haber importado en nuestro servicio

```
import { HttpClient, HttpHeaders } from '@angular/common/http';  
import { Injectable } from '@angular/core';  
import { lastValueFrom }
```

Esta es la forma de hacer peticiones a API's externas en Angular y los diferentes métodos y casuísticas que hay que tener en cuenta.

Como el uso de los diferentes métodos veréis que poco a poco el traer datos externos a nuestros componentes no es una tarea complicada, es más un conjunto amplio de pasos que dificultad.

A continuación, puedes ver el siguiente vídeo, *Peticiones HTTP*.



Accede al vídeo

Tema 31. Pipes

31.1. Introducción y objetivos

El objetivo de este tema es ver el uso y para qué sirven los Pipes en Angular.

Los pipes son estructuras muy interesantes que nos sirven para transformar visualmente los datos en nuestro HTML.

En algunas ocasiones nos veremos en la necesidad de transformar visualmente una fecha, un precio, un dato de la alguna forma y es donde los pipes cobran todo el sentido.

En este tema vamos a ver como se aplican los pipes y vamos a aprender cómo crear nuestro propio Pipe, cuando los que tenemos disponible en Angular no sean suficiente.

31.2. ¿Qué es un Pipes?

Los pipes son una herramienta de Angular que está pensada para transformar datos de forma visual sin modificar su valor dentro de la variable. Es decir, solo cambia visualmente cuando se muestra en el HTML.

Lo más interesante de los pipes es que la información no cambia solo lo hace el aspecto con el que se representa.

Un Pipe solo se usa dentro del HTML del componente y su sintaxis es de esta manera.

```
{{ dato | pipe }}
```

En este ejemplo tenemos un de los pipes más sencillos, el nombre está escrito en con letra capital, es decir *Alberto*, al ponerle el pipe uppercase el dato aparecerá como ALBERTO.

```
<p>{{ nombre | uppercase }}</p>
```

31.3. Pipes propios de Angular

Angular posee un listado de pipes que me permiten hacer muchas cosas. Vamos a ver ejemplo de algunos de ellos. En «A fondo» os dejaremos la documentación oficial de los pipes para que veáis el listado completo, aquí vamos a ver los más comunes, que ciertamente son prácticamente todos los que ofrece Angular.

Los pipes más comunes de Angular son:

- ▶ Decimal, number.
- ▶ Percent.
- ▶ Currency.
- ▶ Date.

Decimal/number

Es un pipe que me permite mostrar los datos con decimales y elegir con cuantos decimales. Pongamos que creamos una propiedad racional = 2. En este caso después de los dos puntos ponemos la cantidad de elementos tanto enteros como fraccionales.

```
<!-- minNumeroEnteros. minNumeroFrac - maximoNumFrac-->  
<p>{{ racional | number : '2.2-5' }}</p>
```

Percent

Lo usaremos para mostrar datos a nivel de porcentaje. Por ejemplo, si creamos una propiedad `iva = 0.21` y le aplicamos el pipe `percent` conseguiremos mostrar `21%`, siempre sin cambiar el dato de `0.21` de la propiedad.

```
<p>{{iva | percent}}</p>
```

Currency

Un pipe para representar una moneda, después de los dos puntos podemos poner el tipo de moneda que queremos representar. De tal forma de que si tenemos una propiedad `precio = 25` se representará como `$25` o `25€` dependiendo del tipo de moneda que elijamos.

```
<p>{{precio | currency: 'EUR'}}</p>
```

Date

Un pipe para representar una fecha ya sea de día o con hora, dependiendo de la configuración elegida en los dos puntos después del pipe.

```
<p>{{ fecha | date: 'dd-MM-yyyy h:mm:ss' }}</p>
```

En ejemplo una fecha sería representada como `15-04-2021 15:30:20`

Adjunto os mostramos algunos códigos de fecha útiles.

```
<!--
```

| Formato | Equivale | Salida |
|---------|---------------|-----------------|
| ----- | ----- | ----- |
| short | M/d/yy h:mm a | 6/15/15 9:03 AM |


```

medium      MMM d y h:mm:ss a      Jun 15 2015 9:03:01 AM
long        MMMM d y h:mm:ss a z  June 15 2015 at 9:03:01 AM GMT+1
shortDate    M/d/yy              6/15/15
mediumDate   MMM d y              Jun 15 2015
longDate     MMMM d y              June 15 2015
fullDate     EEEE MMMM d y        Monday June 15 2015
shortTime    h:mm a              9:03 AM
mediumTime   h:mm:ss a           9:03:01 AM
longTime     h:mm:ss a z         9:03:01 AM GMT+1
fullTime     h:mm:ss a zzzz      9:03:01 AM GMT+01:00
full         EEEE MMMM d y        Monday June 15 2015
            h:mm:ss a zzzz      at 9:03:01 AM GMT+01:00
-->

```

Estos son algunos ejemplos de los más comunes en Angular, ahora vamos a ver cómo crear nuestro propio Pipes para solucionar problemas de visualización que no se contemplan de forma nativa en Angular.

A continuación, puedes ver el siguiente vídeo, *Cómo usar un pipe*.



Accede al vídeo

31.4. Creación de pipe propio

Para crear nuestro propio pipe vamos a poner un ejemplo sencillo que no está contemplado por angular y vamos a ver cómo solucionarlo.

Imaginemos que tenemos una propiedad en minúsculo por ejemplo apellido= “perez” que está en minúscula y queremos convertirla en forma capitular, es decir, “Perez”. Angular no tiene ningún pipe que lo resuelva así que nos vamos a crear el nuestro propio.

Lo primero es generar el pipe, para ello como siempre usaremos la terminal y pondremos el siguiente código.

```
ng generate pipe pipes/capitalize --skip-tests
```

Esto generará un fichero que ts que estará encabezado por el decorador `@Pipe` al igual que los componentes están encabezados por el decorador `@Component`.

Un Pipe no es más que una clase que implementa un interfaz de `PipeTransform` que crea un contrato de clase que obliga a implementar el método `transform()`

El pipe se da de alta en el `app.module` al igual que los componentes creados por nosotros en el array de `declarations`.

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { PrimerComponentesComponent } from './components/primer-
componentes/primer-componentes.component';

import { CapitalizePipe } from './pipes/capitalize.pipe';

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

@NgModule({
  declarations: [
    AppComponent,
    PrimerComponentesComponent,
    CapitalizePipe
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
```

```

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

```

El método transform es el que implementa el campo dentro del pipe, lo que hace es recibir el valor como parámetro y dentro de la función transform se produce el cambio del contenido, y se retorna el valor transformado.

En nuestro ejemplo recibimos una cadena de caracteres, que convertimos en un array previamente, después cogemos el primer elemento lo pasamos a mayúscula para finalmente volver a unir el array con el primer carácter pasado a mayúscula, retornando así el resultado hacia el HTML.

El código sería el siguiente.

```

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

@Pipe({
  name: 'capitalize'
})
export class CapitalizePipe implements PipeTransform {

  transform(value: string): string {
    //opcion 1
    //return value[0].toUpperCase() + value.slice(1)

    //opcion 2
    let arr = value.split("");
    let palabra = "";
    arr.forEach((letra, index) => {
      if (index === 0) {
        palabra += letra.toUpperCase()
      } else {
        palabra += letra
      }
    })
  }
}

```

```
    return palabra

}
```

```
}
```

Y en al HTML se visualizaría de esta forma.

```
<p>{{apellido | capitalize }}</p> //Perez
```

Como veis los pipes son muy útiles para la transformación visual de contenido y su implementación no es nada complicada, te animo a que crees tus propios pipes para tus desarrollos.

Tema 32. Bootstrap en Angular

32.1. Introducción y objetivos

El objetivo de este tema es hacer una introducción al uso de librerías externas en Angular.

Angular es un framework muy potente que tiene muchas funcionalidades por defecto al que se le pueden añadir muchísimas más a través de la instalación de dependencias a través de gestores de paquetes como puede ser npm.

En este tema vamos a instalar y usar uno de los frameworks de **HTML y CSS** más usados de panorama profesional y con el que podremos dar estilo a nuestros componentes de forma fácil y rápida.

32.2. Instalación y configuración

Para instalar **Bootstrap** vamos a usar nuestra terminal de comandos y a realizar la instalación a través del gestor de paquetes **npm** que instalamos a través de **Node JS** al principio de este módulo, y con el que instalamos, por ejemplo, **angular/CLI**.

Lo que tenemos que hacer es crear un proyecto de Angular y una vez dentro de la carpeta del proyecto vamos a introducir el siguiente comando en nuestra terminal.

```
npm install bootstrap
```

Cuando acabe todo el proceso, antes de arrancar nuestro servidor, tenemos que hacer algunas configuraciones dentro del fichero **angular.json**. Básicamente lo que

tenemos que hacer es configurar la ruta de los CSS de Bootstrap y de los JS de Bootstrap dentro del fichero para que nuestra aplicación sepa de donde tiene que cargar dicha información.

```
"build": {
  "builder": "@angular-devkit/build-angular:browser",
  "options": {
    "outputPath": "dist/breakingbad",
    "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",
      "node_modules/bootstrap/dist/css/bootstrap.min.css"
    ],
    "scripts": [
      "node_modules/bootstrap/dist/js/bootstrap.min.js"
    ]
  },
}
```

Dentro de la parte de styles tendremos que colocar la ruta de node_modules de bootstrap con carga la hoja de estilos de minificada bootstrap.min.css.

Y haremos lo mismo para la carga del fichero js colocando lo misma ruta, pero cambiando css por js.

Cada vez que modifiquemos el fichero angular.json, al ser un fichero de configuración necesitamos reiniciar el servicio con el comando ng-serve. O bien parar el servidor antes de modificarlo. Dado que sus modificaciones afectan a la aplicación y solo se tiene en cuenta en al arranque de este.

Una vez que hallamos levantado el servidor podremos usar Bootstrap dentro de nuestros componentes.

32.3. Uso de Bootstrap en Angular

El uso de Bootstrap esta solo disponible dentro del HTML del componente, y nos sirve para prototipar rápido o tener una maquetación más o menos profesional de nuestros componentes.

Bootstrap trabaja con el concepto de clases y utilidad y componentes, nos aporta una cantidad de estilos que podemos usar a través de la imposición de sus clases dentro de nuestro código.

Un ejemplo de código HTML en un componente con Bootstrap podría ser el siguiente.

```
<section class="container-fluid mt-4" *ngIf="characters.length <= 0">
  <div class="row">
    <article class="col-12 mb-3">
      <div class="card">
        <div class="card-body">
          <h2 class="card-title">No hay personajes</h2>
        </div>
      </div>
    </article>
  </div>
</section>
```

Bootstrap además de tener clases de utilidad tiene componentes que podemos usar tal cual y multitud de plantilla que nos pueden ayudar a dar estilos a nuestros desarrollos.

Todo esto lo puede encontrar en la página del framework, que actualmente se encuentra en su versión 5.2.

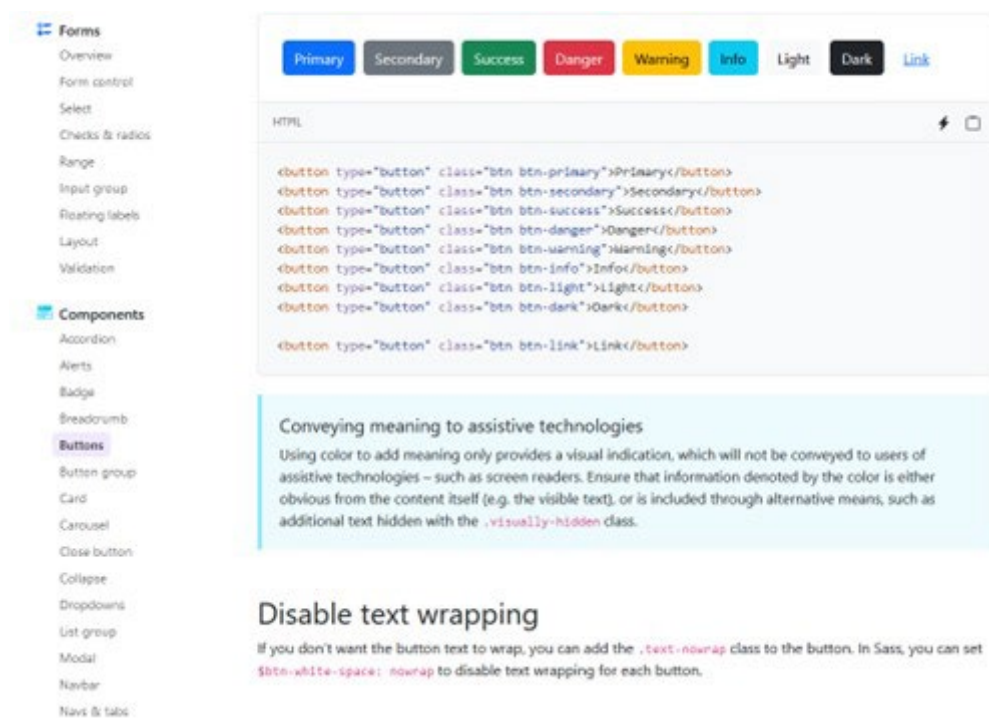


Figura 35. Documentación de Bootstrap. Fuente: elaboración propia.

Te anima a echarle un vistazo a todo el material que tienes de Bootstrap por internet he intentar darles estética a tus componentes a través de este potente framework.

A continuación, puedes ver el siguiente vídeo, *Cómo usar Bootstrap dentro de Angular*.



Accede al vídeo

Tema 33. Angular y Firebase.

Gestión de Librerías Externas con Angular. Google Maps

33.1. Introducción y objetivos

Este último tema de Angular cumple dos objetivos fundamentales, el primero ya iniciado en el tema anterior es como instalar librerías de terceros a través del gestor de paquete NPM, en este caso con la necesidad de configurar unas variables de entorno que nos permitan usar la librería de terceros y sus API's de desarrollo.

En el caso de esta librería vamos a usar Google Maps a través de la librería AGM-Maps de Angular, para lo cual tendremos que configurarnos una cuenta de Google Developer para poder hacer uso de la Api de mapas de Google.

El segundo punto importante en este caso es construir y publicar la aplicación, una vez hayamos acabado nuestros desarrollos tenemos que convertirlos a algo que nuestros servidores comprendan, en este caso archivos HTML y JavaScript, pues en este tema vamos a aprender los conceptos más importantes para realizar esa tarea.

Con estos dos puntos damos por finalizado nuestro módulo de Angular en el cual hemos aprendido desde lo más básico de creación de componentes hasta realizar aplicaciones que se sitúen y geolocalicen en un mapa.

33.2. ¿Cómo publicar una aplicación de Angular?

La publicación de una aplicación de Angular no tiene ningún misterio, simplemente debemos hacer unas configuraciones previas a la publicación en nuestro archivo `angular.json`.

Estas publicaciones tienen como objetivo simplemente asegurar que el proceso no se va a quedar bloqueado por falta de memoria o procesos de nuestra aplicación.

Abrimos el archivo `angular.json` y buscamos el apartado `configurations`, en ese json buscamos cuatro propiedades que son `MaximumWarning`, `MaximumError`, tanto en `initial` como en `anyComponentStyle`.

Estas propiedades marcan el volumen de datos que puede soportar la aplicación durante la compilación, si son muy bajos, como aparecen inicialmente, y la aplicación es muy pesada, corremos el riesgo de que el proceso se pare y no acabe con éxito. Yo recomiendo los siguientes valores.

```
"configurations": {
  "production": {
    "budgets": [
      {
        "type": "initial",
        "maximumWarning": "1mb",
        "maximumError": "2mb"
      },
      {
        "type": "anyComponentStyle",
        "maximumWarning": "10kb",
        "maximumError": "20kb"
      }
    ],
    "fileReplacements": [
      {
```

```

    "replace": "src/environments/environment.ts",
    "with": "src/environments/environment.prod.ts"
  }
],
"outputHashing": "all"
},

```

Una vez que hayamos configurado este fichero solo tenemos que ir a nuestra consola, meternos dentro de la carpeta de nuestra aplicación y escribir el siguiente comando.

```
ng build
```

Este comando desencadenará un proceso que generará una carpeta **dist** dentro de nuestro repositorio de código y la carpeta del proyecto, de la siguiente manera.

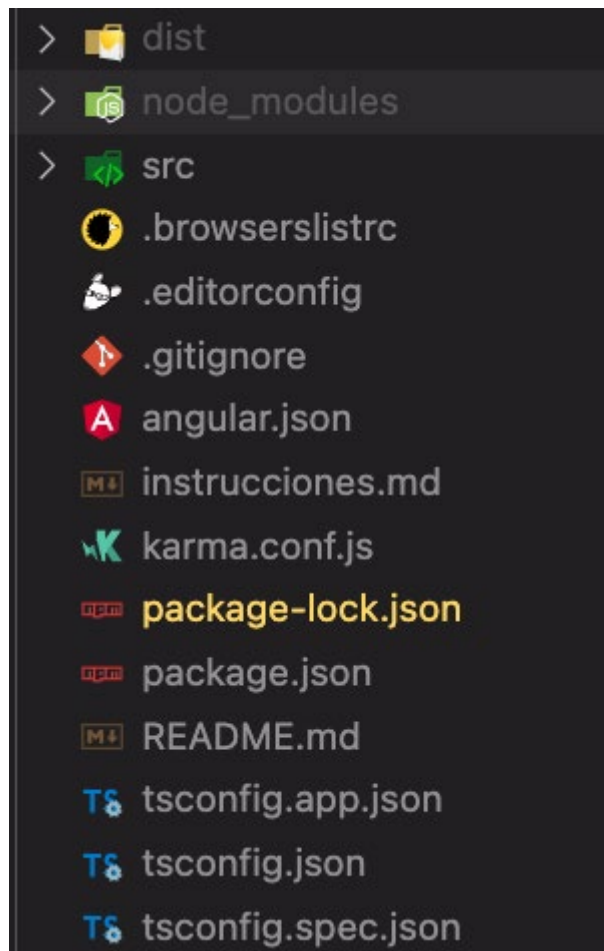


Figura 36. Arbol de carpetas con dist. Fuente: elaboración propia.

En esa carpeta se encuentran todos los archivos necesarios para hacer correr nuestra aplicación, en este caso una carpeta con el nombre de la aplicación y dentro puros HTML y JavaScript fruto de la transformación de los ficheros Typescript en ficheros de JavaScript nativo, que son los únicos ficheros que puede entender un navegador.

En el tema de despliegue de aplicaciones, en un próximo módulo, veremos como subir esta carpeta a un servidor y publicarlo en internet.

En el caso de Angular es tremendamente fácil porque solo es poseer esta carpeta y publicarla dentro de nuestro hosting.

```
npm install
```

33.3. Instalación de librerías de terceros en Angular

En esta parte de tema vamos a introducirnos más dentro de lo que es la instalación de librerías de terceros dentro de nuestra aplicación de Angular. En este caso vamos a instalar esta librería.



Figura 37. Angular Google Maps AGM. Fuente: elaboración propia.

Esta librería no solo me permite instalar mapas de Google Maps dentro de mis aplicaciones, sino que también me permite posicionar elementos a través de la Api de Google Maps, para situar elementos en el mapa a través de latitud y longitud.

Para instalar esta librería simplemente, y tal y como hemos hecho otras veces, tenemos que salir a nuestra terminal, a la carpeta de proyecto e instalar la librería a través de nuestro gestor de paquetes. Con el siguiente comando.

```
npm i @agm/core@1.1.0
```

La versión 1.1.0 no es la última versión de la librería, pero es que la última versión de la librería hoy en día está en versión beta y da algunos fallos, cuando la depuren un poco iremos a una más moderna.

Todo esto lo puedes ver en la página de npmjs.com, donde puedes buscar cualquier dependencia que quieras instalar a través de **npm**.

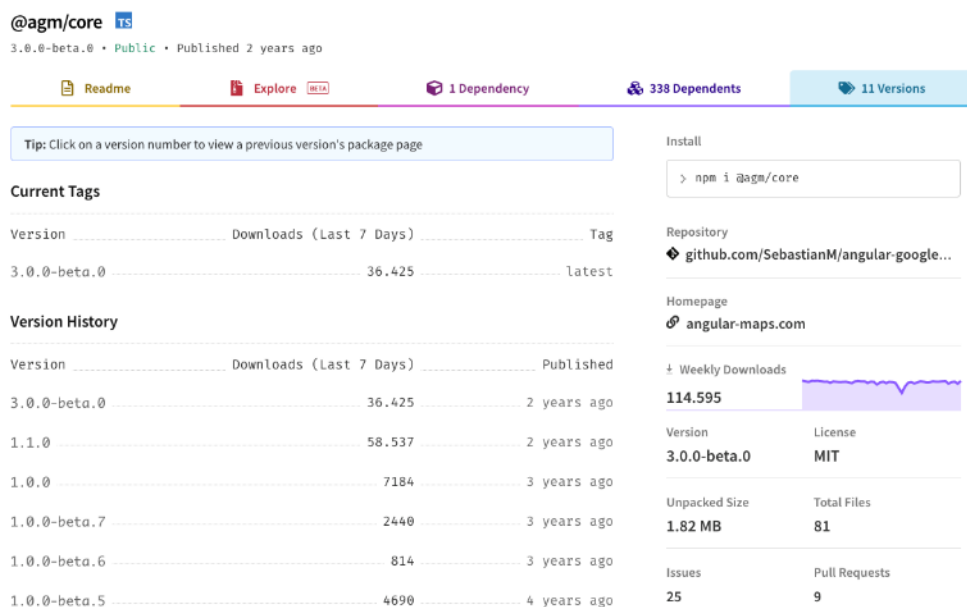


Figura 38. npmjs, captura de librería agm/core. Fuente: elaboración propia.

Una vez que tenemos nuestra librería instalada, podremos comprobarlo en nuestro fichero `packages.json`, en dependencias que aparecerá la librería instalada.

```
"dependencies": {  
  "@agm/core": "^1.1.0",
```

El siguiente paso es crearse una cuenta en Google Cloud Platform a través de la siguiente dirección web, y siendo poseedor de una cuenta de Gmail gratuita.

<https://console.cloud.google.com/>

Una vez que te has registrado debes crear un token de acceso creando una aplicación de mapas dentro de la gestión de la aplicación, dando de alta una aplicación de mapas en JavaScript.

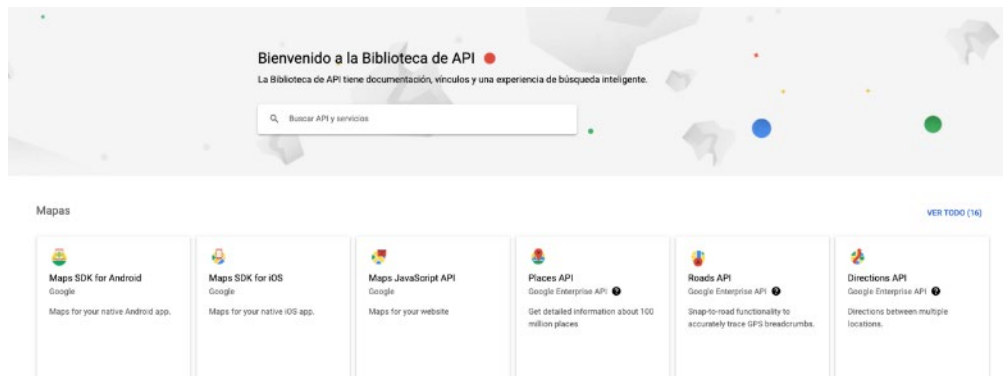


Figura 39. Google Platform elección de mapas de Google con Javascript. Fuente: Elaboración propia.

Una vez des de alta la aplicación, deberás crearte una Api-Key para que Google interprete tus peticiones como legales y te las permita realizar.

Crear una api-key es muy sencillo, pero si te atascas en la sección de a fondo te he dejado una publicación donde explica el proceso paso a paso.

Una vez que la hayas instalado y configurado simplemente tienes que hacer dos cosas.

Configurar la api-key dentro de nuestras variables de entorno dentro de la carpeta enviroments, tanto en el fichero de producción como en desarrollo. En los fichero

environment.prod.ts y envioment.ts. De la siguiente forma. ¡OJO! El aki-key que aparece en este código no es real, tenéis que sustituir por el vuestro.

```
export const environment = {  
  production: false,  
  googleMaps: {  
    apiKey: 'AIzaSyDasdfasdfaMZwseWl2S4'  
  }  
};
```

Cuando tengamos los ficheros de entorno configurados, iremos al fichero app.module.ts y ahí importaremos tanto los ficheros de entorno como la librería para que todos nuestros componentes pueden usarla sin ningún tipo de problema.

```
import { NgModule } from '@angular/core';  
import { BrowserModule } from '@angular/platform-browser';  
import { environment } from 'src/environments/environment';  
import { AgmCoreModule } from '@agm/core';  
  
@NgModule({  
  declarations: [],  
  imports: [  
    BrowserModule,  
    AppRoutingModule,  
    AgmCoreModule.forRoot(environment.googleMaps)  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```

Una vez importado nuestras librerías simplemente deberemos pasarle al AgmCoreModule los datos de las variables de entorno para que funcione. De la manera que hemos puesto en el código anterior.

```
AgmCoreModule.forRoot(environment.googleMaps)
```

Siendo googleMaps la clave que le pongo al fichero json de environment.

Y con esto ya tenemos configurado nuestra aplicación para que la podemos usar en nuestros componentes. Ahora vamos a ver un ejemplo sencillo de cómo se usa.

33.4. Uso de Google Maps en Angular a través de la librería AGM Maps

Para usar nuestro mapa de Google Maps dentro de nuestro componente y ver un mapa de Google con un elemento posicionado a través de la latitud y longitud, simplemente debemos crear unas propiedades del componente latitud y longitud y asignarle una posición válida, ya sea a mano o a través de una API.

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

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

  lat: number = 40;
  lng: number = -3;

  constructor() { }

  ngOnInit {
  }

}
```


Una vez que tengas la latitud y longitud dadas de alta simplemente creas el mapa a través del siguiente código dentro de la parte del HTML del componente.

```
<agm-map [latitude]="lat" [longitude]="lng">
  <agm-marker [latitude]="lat" [longitude]="lng">
    <agm-info-window>
      <div class="infowindow">
        <h4> título del popup</h4>
        <hr>
      </div>
    </agm-info-window>
  </agm-marker>
</agm-map>
```

Además de este código deberás asignarle en el css de tu componente una altura a tu mapa para que se visualice.

```
agm-map {
  width: 100%;
  height: 100vh;
}
```

Esto creará un mapa que te ocupe toda la pantalla y que nos posicione un mapa con un marker situando nuestra posición y que tiene un popup con un título cuando hacemos click en nuestro marker.

A través de esta librería podrás crear varios market y posicionar productos, inmuebles, lo que te dé la gana.

A continuación, puedes ver el siguiente vídeo, *Instalar AGM Map y hacer un mapa sencillo*.



Accede al vídeo

Documentación oficial de Angular

Angular.io. (2010-2022). Página web oficial. Recuperado de <https://angular.io/>

Aquí tienes la documentación oficial de Angular.

Qué es una SPA

Desarroloweb. (29 de noviembre de 2016). *Qué es una SPA*. Recuperado de: <https://desarrolloweb.com/articulos/que-es-una-spa.html>

Este artículo explica el concepto de SPA (Single Page Application), un tipo de aplicación web cada vez más usado por la agradable experiencia de usuario que aporta.

Documentación oficial de Typescript

Typescript. (s.f.). *TypeScript is JavaScript with syntax for types*. Recuperado de: <https://desarrolloweb.com/articulos/que-es-una-spa.html>

Aquí tienes la documentación oficial de typescript, la tienes en inglés y en español.

TypeScript para Principiantes

Shokeen, M. (25 de octubre de 2017). *TypeScript para principiantes, Parte 1: Comenzando*. Recuperado de: <https://code.tutsplus.com/es/tutorials/typescript-for-beginners-getting-started--cms-29329>

Buen tutorial de typescript con varias partes y en castellano.

TYPESCRIPT - Aprendiendo desde CERO con curso oficial de Microsoft

Midulive. (20 de marzo de 2022). *Aprende TypeScript desde cero con este curso oficial de Microsoft* [Archivo de vídeo]. YouTube. Recuperado de: <https://www.youtube.com/watch?v=YKclM8lxkfl>

Vídeo muy interesante sobre TypeScript en castellano.

Manual de Angular

Desarrollo web. <https://desarrolloweb.com/manuales/manual-angular-2.html>

Monográfico de angular en español donde podréis afianzar concepto.

Documentación oficial del framework

Angular documentación oficial. <https://angular.io/docs>

Documentación oficial del Angular.

Web interesante sobre Angular en español

Aprendiendo Angular. <https://ngchallenges.gitbook.io/project/componentes>

Aquí podrás encontrar la explicación de que es un componente desde el punto de vista de alguien que lo está aprendiendo, es una muy buena práctica cuando uno está aprendiendo enseñar a otros para fijar conceptos.

Entendiendo los componentes en Angular

Ro. (31 de julio de 2019). *Entendiendo los componentes en Angular - Guía de iniciación* [Blog]. <https://www.acontracorrientech.com/entendiendo-los-componentes-en-angular/>

Un blog interesante donde se explican cómo funcionan los componentes en angular.

Componentes en Angular

Aristizabal, V. (16 de septiembre de 2019). *Componentes en Angular* [Blog]. <https://medium.com/notasdeangular/componentes-en-angular-f25138b00c83>

Publicación en castellano que trata con dibujos y ejemplos el funcionamiento de los componentes.

Interpolación `{{}}` en Angular al detalle

Desarrollo web. (30 de octubre de 2017). *Interpolación `{{}}` en Angular al detalle* [Blog]. <https://desarrolloweb.com/articulos/binding-interpolacion-angular.html>

Interesante explicación de cómo se realiza la interpolación de propiedades en Angular.

Interpolación `{{}}` en Angular al detalle

Desarrollo web. (30 de octubre de 2017). *Interpolación `{{}}` en Angular al detalle* [Blog]. <https://desarrolloweb.com/articulos/binding-interpolacion-angular.html>

Interesante explicación de cómo se realiza la interpolación de propiedades en Angular.

Uso de Sass en Angular

Chris on Code. (21 de septiembre de 2020). *Using Sass with the Angular CLI* [Blog]. <https://www.digitalocean.com/community/tutorials/using-sass-with-the-angular-cli>

Ejemplo práctico del uso de SCSS en Angular.

Property Biding. Angular documentación oficial

Angular. <https://angular.io/guide/property-binding>

Esta es la documentación oficial de los property biding. Aquí puedes también consultar todas las novedades de las siguientes versiones si es que ocurriera.

Guía de iniciación a la data binding en Angular

Ro. (14 de agosto de 2019). *Guía de iniciación al data binding en Angular*. <https://www.acontracorrientech.com/guia-practica-del-databinding-en-angular/>

Ejemplo práctico. Este post es una guía de iniciación donde explica paso a paso y de forma muy clara lo que es el data biding y como se implementa de forma correcta.

Event Biding in Angular 8

Gestión de eventos desde Angular 8. <https://www.geeksforgeeks.org/event-binding-in-angular-8/>

En esta publicación explica de forma bastante detallada los entresijos del evento biding con Angular.

Documentación oficial de eventos Angular

Event binding. <https://angular.io/guide/event-binding>

Muy útil tener acceso a la explicación que hace la documentación de angular sobre la gestión de eventos en un interfaz hecho en Angular.

Vídeo explicativo de cómo se gestionan eventos en Angular en castellano

Píldoras informáticas. (26 de febrero de 2021). *Curso Angular. Event Binding. Vídeo 10* [Vídeo]. YouTube <https://www.youtube.com/watch?v=FPjFXQf1pgM>

Vídeo interesante de cómo se gestionan los eventos en Angular.

Input y Output en Angular

Bastidas, W. (2 de agosto de 2020). *Angular decoradores @Input y @Output* [Blog]. <https://medium.com/williambastidasblog/angular-decoradores-input-y-output-70af5f43a04>

Interesante artículo donde se explican los decoradores input y output.

Documentación oficial de input y outputs Angular

Angular. <https://angular.io/guide/inputs-outputs>

Explicación extensa y oficial de cómo se gestionan los input y outputs en angular.

Directivas en Angular

Aristizabal, V. (3 de mayo de 2020). *Directivas en Angular* [Blog]. <https://medium.com/notasdeangular/directivas-en-angular-efb8a8cf78e0>

Interesante artículo donde se explican las directivas en Angular.

Documentación oficial de directivas en Angular

Angular. <https://angular.io/guide/attribute-directives>

Explicación extensa y oficial de cómo se gestionan las directivas en angular.

Creación de directivas personalizadas

Tutoriales de programación. (s.f.). *Directivas estructurales – creación de directivas personalizadas*.

<https://www.tutorialesprogramacionya.com/angularya/detalleconcepto.php?punto=81&codigo=81&inicio=80>

Explicación de cómo se crea una directiva personalizada.

Introducción al servicio e inyección de dependencias en Angular

Angular. <https://docs.angular.lat/guide/architecture-services>

Explicación extensa y oficial de cómo se gestionan la inyección de dependencias en angular.

Diferencias entre formularios de tipo template y model

Mugika, A. (14 de enero de 2021). *Formularios en Angular – Diferencias Template y Reactive Forms* [Blog]. <https://mugan86.medium.com/formularios-en-angular-diferencias-template-y-reactive-forms-e37af5e30b81>

Interesante artículo donde se explican las diferencias entre los dos formularios desde otro punto de vista.

Documentación oficial formularios en español Angular

Angular. <https://docs.angular.lat/start/start-forms>

Explicación extensa y oficial de cómo se gestionan formularios en Angular.

Documentación oficial en español de Angular manejo de rutas

Angular. <https://docs.angular.lat/tutorial/toh-pt5>

Conceptos de rutas en angular en castellano documentación oficial.

Documentación oficial formularios en español Angular

Coding Potions. (28 de agosto de 2021). *Cómo manejar el router de Angular* [Blog]. <https://codingpotions.com/angular-router>

Cómo crear rutas en Angular. Sistema de routing.

Servicios en Angular

Desarrollo web. (14 de mayo de 2020). *Servicios de Angular* [Blog].
<https://desarrolloweb.com/articulos/servicios-angular.html>

Veremos qué son los servicios en el framework JavaScript Angular, cómo crear services e inyectarlos en los componentes para acceder a ellos.

Documentación oficial formularios en español Angular

Angular. (s.f.). *Agregar servicios*. <https://docs.angular.lat/tutorial/toh-pt4>

Cómo agregar y crear servicios y cómo gestionarlos explicados en español y desde la documentación oficial del framework.

Http en Angular

Angular. (s.f.). *Communicating with backend services using HTTP*.
<https://angular.io/guide/http>

Explicación de cómo hacer peticiones http y gestionarlas a través de observables, también explica como enviar cabeceras en Angular.

Comunicaciones http en Angular

Academia Binaria. (18 de abril de 2020). *Comunicaciones http en Angular* [Blog].
<https://academia-binaria.com/comunicaciones-http-en-Angular/>

Las comunicaciones «http» son una pieza fundamental del desarrollo web, y en **Angular** siempre han sido potentes en esta página web tienes una extensa explicación de cómo se producen.

Como convertir un observable en promesas en Angular

Cloudhadoop. (s.f.). *How to convert Observable to Promise in angular – TypeScript* [Blog].
<https://www.cloudhadoop.com/angular-convert-observable-to-promise/>

Un post interesante de cómo gestionar un observable en forma de promesa.

Pipes en Angular - Guía completa

A ContracorrienTech. (16 de octubre de 2019). *Pipes en Angular* [Blog].
<https://www.acontracorrientech.com/pipes-en-angular-guia-completa/>

Guía completa de cómo usar los pipes en español da un repaso completo por todos los pipes propios de Angular y como se crean pipes.

Bootstrap 5 in Angular

Techiediaries. <https://www.techiediaries.com/angular-bootstrap/>

Una guía completa en ingles de cómo usar Bootstrap 5 en versiones de Angular superiores a la 12.

Bootstrap 5 en Angular explicación en vídeo

Domini Code, (29 de noviembre de 2021). *Instalar Bootstrap 5 en Angular* [Vídeo].
YouTube. <https://www.youtube.com/watch?v=rli1aiL0o48>

Vídeo muy interesante y explicativo de cómo se instala Bootstrap 5 en castellano de una manera muy sencilla.

Página oficial de la librería AGM Maps

Angular Google Maps (AGM). (s.f.). *Really easy to get started*. <https://angular-maps.com/>

Documentación oficial de la librería AGM, documentación y guías rápidas.

Como realizar una publicación de Angular en un hosting compartido

Mugika Ledo, A. (10 de enero de 2020). *Publicar una aplicación Angular en un Hosting compartido* [Blog]. <https://mugan86.medium.com/publicar-una-aplicaci%C3%B3n-angular-en-un-hosting-compartido-c6c934da3c36>

Pasos a seguir para publicar una app en Angular en un hosting que soporta HTML, PHP, CSS y JS

Como crear una aplicación Google Maps en Angular

10 minutos Programando. (18 de agosto de 2020). *Google Maps con Angular 10* [Video]. YouTube. <https://www.youtube.com/watch?v=DQZTeZZXYBk>

Pasos a seguir para crear una pequeña aplicación de mapas en Google Maps.

Guía rápida: ¿cómo obtener la API key de Google Maps?

Trafaniuc, V. (26 de febrero de 2020). *Guía rápida: ¿cómo obtener la API key de Google Maps?* [Blog]. <https://maplink.global/blog/es/como-obtener-google-maps-api-key/>

Si estás buscando aprender cómo obtener una API Key de Google Maps, probablemente ya sepas qué es y cómo utilizar esta herramienta, ¿verdad?

1. ¿Qué es Angular?
 - A. Una librería de dedicado al desarrollo de frontend.
 - B. Un framework de JavaScript dedicado al desarrollo de frontend.
 - C. Una evolución de JavaScript nativo.
 - D. Es un conjunto de funcionalidades realizadas en TypeScript que me permiten desarrollar páginas web completas.

2. ¿Qué comando ejecutamos en nuestra terminal para crear un nuevo proyecto de Angular?
 - A. ng generate.
 - B. ng build.
 - C. ng new.
 - D. ng test.

3. ¿Qué significa la etiqueta `<app-root></app-root>`?
 - A. La etiqueta que inicializa todo el proyecto de angular.
 - B. Es la etiqueta principal, carga el componente principal, dentro del index.html.
 - C. Es la etiqueta que marca donde se cargan los componentes definidos en las rutas.
 - D. Es la etiqueta que se carga en el componente principal del sistema.

4. ¿Cuál es el comando para generar un componente en Angular?
 - A. ng generate.
 - B. ng test.
 - C. ng generate componente /rutadel componente.
 - D. ng component.

5. ¿Para qué sirve un interfaz en Angular?
- A. Genera un archivo de clase donde se definen e importan elementos importantes para el desarrollo de una aplicación en Angular.
 - B. El interfaz es una clase de Angular que permite definir algo que se tiene que cumplir obligatoriamente en aquella clase donde se aplique.
 - C. Es la parte visual dentro de mi aplicación, y define la estética que va a tener nuestra aplicación como resultado final.
 - D. Es parte de los métodos que definen un componente dentro de una clase.
6. ¿Cuántas hojas de estilo puede tener un proyecto de Angular?
- A. Una, la que se carga inicialmente en la carpeta principal del proyecto.
 - B. Una por cada componente.
 - C. Depende de la cantidad de estilos que quieras implementar.
 - D. No existe límite.
7. ¿Qué finalidad tiene el servicio?
- A. Conectar componentes entre sí.
 - B. Conectar a los componentes entre sí y pasarles los datos conectándose a la API externas.
 - C. Conectarse con las API externas para poder recibir los datos que luego le pasaremos a los demás componentes.
 - D. Sirve para almacenar nuestros datos de una aplicación para poder trabajar con ellos.
8. ¿Dónde se definen las diferentes rutas de nuestra aplicación?
- A. En el menú de navegación de nuestra página web
 - B. Dentro de nuestro index.html que conecta con todos los elementos de nuestra aplicación.
 - C. Dentro del app-routing.
 - D. Puedes definir rutas dentro de cualquier componente a través de router.

9. ¿Cuántas directivas estructurales hay?

- A. Una.
- B. Dos.
- C. Tres.
- D. Cuatro.

10. ¿Para qué sirve la interpolación en Angular?

- A. Para intercalar el uso de clase dentro de nuestra aplicación y cargar los componentes dentro de otros componentes.
- B. Para poder ejecutar operaciones matemáticas dentro del html del componente
- C. Para inyectar texto dentro de una plantilla HTML de nuestro componente.