



Máster de Formación Permanente en Full Stack  
Developer

---

## Módulo 3. Framework de Front End Angular

# Índice

## Tema 1. Introducción a Angular

- 1.1. Introducción y objetivos
- 1.2. ¿Cómo funciona Angular?
- 1.3. Características principales de Angular
- 1.4. ¿Cómo trabajamos con Angular?
- 1.5. Herramientas para trabajar con Angular
- 1.6. ¿Cómo instalar Angular?

## Tema 2. TypeScript

- 2.1. Introducción y objetivos
- 2.2. Instalación y uso TypeScript
- 2.3. Tipado
- 2.4. Interfaces
- 2.5. Decoradores

## Tema 3. Arquitectura de la aplicación

- 3.1. Introducción y objetivos
- 3.2. Partes de una aplicación de Angular
- 3.3. Configuración básica
- 3.4. Arquitectura por componentes
- 3.5. Creación de módulos

## Tema 4. Componentes

- 4.1. Introducción y objetivos
- 4.2. ¿Qué es un componente?

- 4.3. Partes de un componente
- 4.4. Cómo crear un componente
- 4.5. Componente principal y componentes hijos. ¿Cómo se enlazan?
- 4.6. Funciones de ciclo de vida de un componente

## Tema 5. Templating

- 5.1. Introducción y objetivos
- 5.2. Sintaxis de plantillas
- 5.3. Hojas de estilos de componente
- 5.4. Angular signals

## Tema 6. Data Binding: One-way and Two-ways

- 6.1. Introducción y objetivos
- 6.2. Property Binding
- 6.3. Two way date biding: FormsModule y [(ngModel)].  
Doble comunicación

## Tema 7. Event Binding

- 7.1. Introducción y objetivos
- 7.2. Creación de eventos en Angular
- 7.3. \$event

## Tema 8. Input y Output

- 8.1. Introducción y objetivos
- 8.2. Comunicación entre componentes
- 8.3. Decorador @Input(). Del padre al hijo
- 8.4. Decorador @Output(). Del hijo al padre

## Tema 9. Control Flow

- 9.1. Introducción y objetivos
- 9.2. Bloque condicional con @if
- 9.3. Bloque de repetición con @for
- 9.4. Bloque condicional @switch

## Tema 10. Class y style. Directivas ngStyle y ngClass

- 10.1. Introducción y objetivos
- 10.2. Inserción y modificación dinámica de estilos
- 10.3. Inserción y modificación dinámica de clases
- 10.4. Conclusiones

## Tema 11. Inyección de dependencias

- 11.1. Introducción y objetivos
- 11.2. ¿Qué es y para qué se usa?
- 11.3. Ejemplo de inyección de dependencia

## Tema 12. Formularios y validaciones

- 12.1. Introducción y objetivos
- 12.2. Formulario de tipo Template. FormsModule
- 12.3. Formulario de tipo Model. Reactive Forms
- 12.4. Validaciones de Angular y validaciones personalizadas

## Tema 13. Rutas y navegación

- 13.1. Introducción y objetivos
- 13.2. Configuración básica. Módulo de rutas
- 13.3. RouterLink y RouterLinkActivate

13.4. Parámetros de ruta: Activated Route

13.5. Rutas hijas

13.6. LocalStorage y guards

#### Tema 14. Servicios y HTTP

14.1. Introducción y objetivos

14.2. ¿Qué es un servicio?

14.3. Comunicación del servicio con los componentes

14.4. HttpClientModule

14.5. Peticiones GET, POST, PUT, DELETE

14.6. Promesas y observables

14.7 Interceptors

#### Tema 15. Pipes

15.1. Introducción y objetivos

15.2. ¿Qué es un Pipe?

15.3. Pipes propios de Angular

15.4. Creación de pipe propio

#### Tema 16. Bootstrap en Angular

16.1. Introducción y objetivos

16.2. Instalación y configuración

16.3. Uso de Bootstrep en Angular

#### Tema 17. Gestión de librerías, publicación y mapas

17.1. Introducción y objetivos

17.2. ¿Cómo publicar una aplicación de Angular?

17.3. Instalación de librerías de terceros en Angular

## 17.4. Uso de Google Maps en Angular

### A fondo

Documentación oficial de Angular

Qué es una SPA

Documentación oficial de Typescript

TypeScript para principiantes

TypeScript: aprende desde cero con curso oficial de Microsoft

Manual de Angular

Documentación oficial del framework

Web interesante sobre Angular en español

Entendiendo los componentes en Angular – Guía de iniciación

Componentes en Angular

Interpolación {{}} en Angular al detalle

Uso de Sass en Angular

Property biding. Angular documentación oficial

Guía de iniciación a la data binding en Angular

Event biding in Angular

Documentación oficial de eventos Angular

Video explicativo sobre cómo se gestionan eventos en Angular, en castellano

Input y Output en Angular

Documentación oficial de input y outputs Angular

Qué son los signals en Angular

Control Flow en Angular

Class and style en Angular

NgClass and ngStyle en Angular

Introducción al servicio e inyección de dependencias en Angular

Explicación de la función inject()

Diferencias entre formularios de tipo template y model

Documentación oficial sobre formularios, en español, de Angular

Documentación oficial de Angular sobre el manejo de rutas. Angular 17

Servicios en Angular

Documentación oficial sobre Servicios, en español, de Angular

HTTP en Angular

Comunicaciones HTTP en Angular

Cómo convertir un observable en promesas en Angular

Pipes en Angular: guía completa

Bootstrap 5 in Angular

Bootstrap 5 en Angular: explicación en vídeo

Github oficial de la librería de Google Maps

Cómo realizar una publicación de Angular en un hosting compartido

Guía rápida: ¿cómo obtener la API Key de Google Maps?

API de Maps Javascript

Test

# Tema 1. Introducción a Angular

## 1.1. 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 17, 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 versiones posteriores, ya que AngularJS fue una primera versión que no tiene nada que ver con las versiones actuales ni futuras. Su desarrollo se descartó por ser complejo y por apartarse en gran medida del desarrollo por componentes que es hacia donde se mueve el mercado actual. Aquí deberéis tener especial cuidado con buscar información del framework y revisar cuando se esté hablando de Angular 1 (AngularJS) o Angular 2 y sucesivos, ya que podemos encontrar cambios significativos.

A partir de este momento siempre que hablemos de Angular estaremos hablando de la versión actual, que es la 17.

# Tema 1. Introducción a Angular

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).

# Tema 1. Introducción a Angular

## 1.2. ¿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 término 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.

# Tema 1. Introducción a Angular

Es un framework que se basa en la creación de aplicaciones a través de componentes. Este concepto es muy práctico ya que nos permite desarrollar aplicaciones complejas a partir de la unión de componentes mucho más simples.

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.

# Tema 1. Introducción a Angular

## 1.3. 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 desarrollo por componentes y la organización robusta que nos propone Angular me permite la **separación de tareas en grupos de trabajo**, con lo que asume una agilidad mucho mayor en el desarrollo y una reducción en los 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.

# Tema 1. Introducción a Angular

## 1.4. ¿Cómo trabajamos con Angular?

---

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

<https://angular.dev/>

---

Vamos a argumentar en unos pocos puntos cada una de las ventajas que me aporta 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.

# Tema 1. Introducción a Angular

## 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 integra las librerías más avanzadas del mercado para la creación de test unitarios, lo que favorece la creación y ejecución de estas pruebas sobre nuestro código.

# Tema 1. Introducción a Angular

## 1.5. 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 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 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.

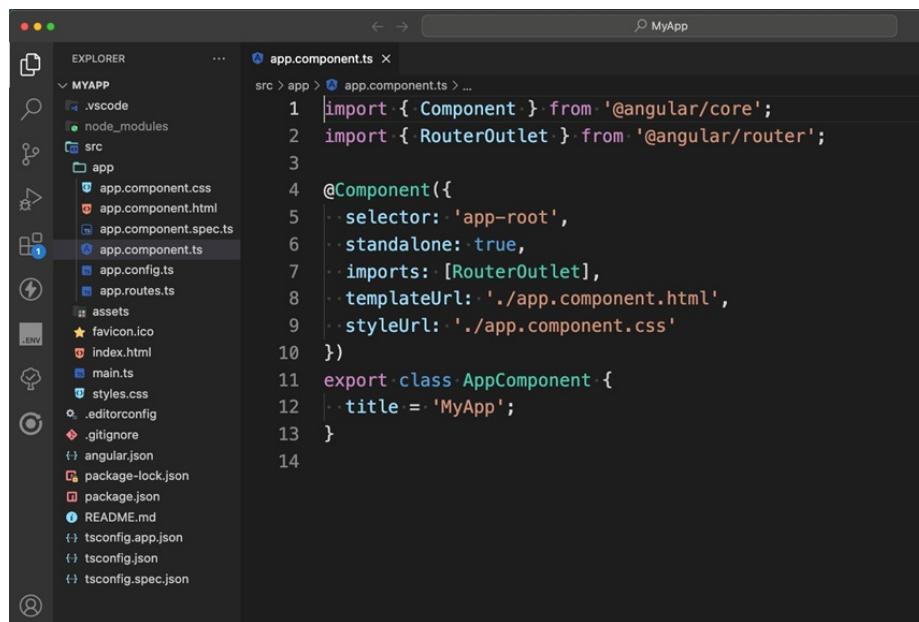


Figura 1. Árbol de ficheros de Angular. Fuente: elaboración propia.

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.

# Tema 1. Introducción a Angular

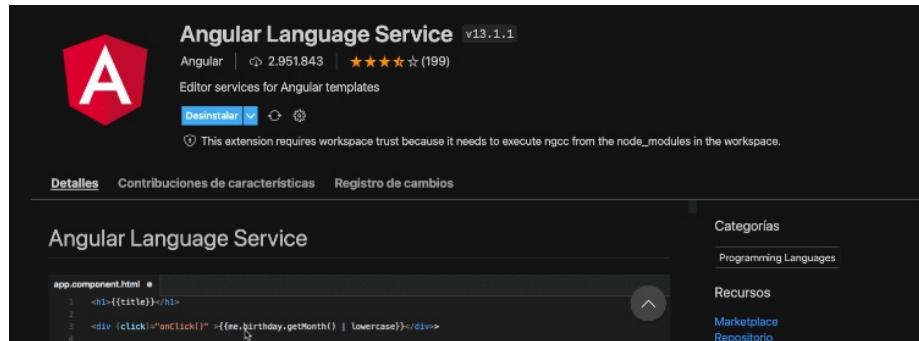


Figura 2. Plugin de Visual Studio Code. Fuente: elaboración propia.

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/>

---

# Tema 1. Introducción a Angular

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

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.

# Tema 1. Introducción a Angular

## 1.6. ¿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:

<b>ng build</b>	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.
<b>ng serve</b>	Me permite levantar el servidor de Desarrollo para arrancar la aplicación y visualizar los cambios que voy desarrollando en tiempo real.
<b>ng generate</b>	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.
<b>ng test</b>	Nos permite generar pruebas unitarias dentro de un proyecto determinado.

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

# Tema 1. Introducción a Angular

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
```

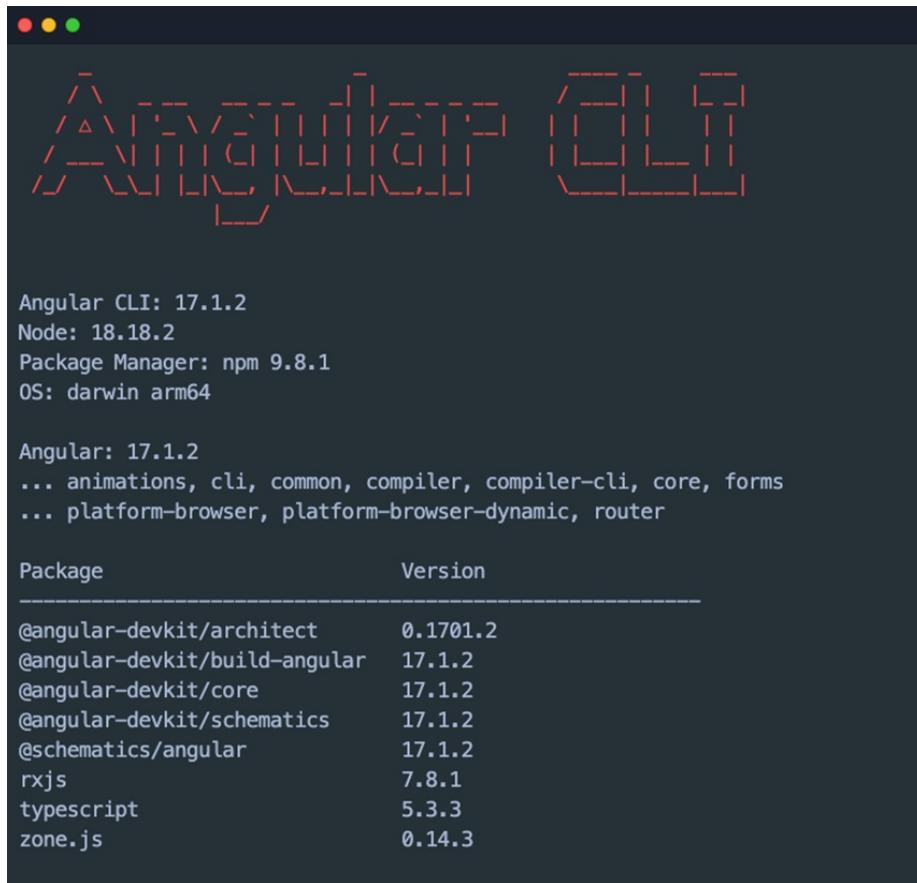
Este comando lo que me permite instalar el Angular CLI de forma global, de ahí el -g, dentro de nuestro equipo.

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.

# Tema 1. Introducción a Angular



A screenshot of a macOS terminal window. The title bar shows 'Angular CLI: 17.1.2'. The main pane displays the output of the 'ng -version' command. It includes the following text:  
Angular CLI: 17.1.2  
Node: 18.18.2  
Package Manager: npm 9.8.1  
OS: darwin arm64  
  
Angular: 17.1.2  
... animations, cli, common, compiler, compiler-cli, core, forms  
... platform-browser, platform-browser-dynamic, router  
  
Package Version  
-----  
@angular-devkit/architect 0.1701.2  
@angular-devkit/build-angular 17.1.2  
@angular-devkit/core 17.1.2  
@angular-devkit/schematics 17.1.2  
@schematics/angular 17.1.2  
rxjs 7.8.1  
typescript 5.3.3  
zone.js 0.14.3

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.

# Tema 1. Introducción a Angular

Este comando ejecutará un conjunto de órdenes que te realizará una serie de preguntas. La primera pregunta será qué motor de hoja de estilos queremos usar. Moviéndonos con las flechas, seleccionaremos en primera instancia CSS (igualmente más adelante realizaremos desarrollos con SCSS que habéis visto en capítulos anteriores).

```
Cursos/06_UNIR/Actualización_2024
→ ng new primeraApp
? 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.

Además de la pregunta anterior, antes de la instalación de la aplicación, el terminal nos pregunta si queremos activar la **renderización del lado del servidor (SSR)**. De momento respondemos que no, para poder centrarnos en los conceptos más simples, y dejaremos para más adelante las configuraciones complejas que nos ofrece Angular.

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.

# Tema 1. Introducción a Angular

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.

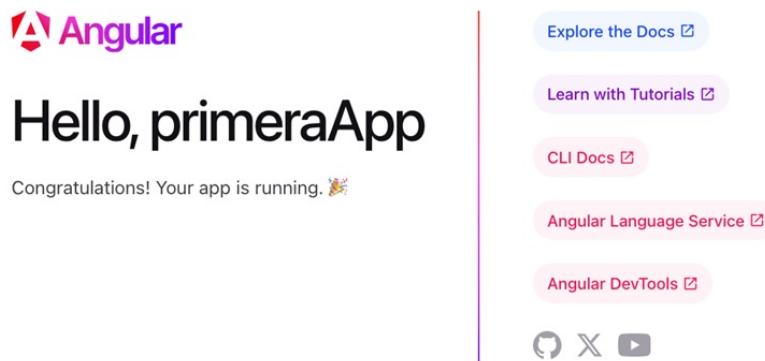
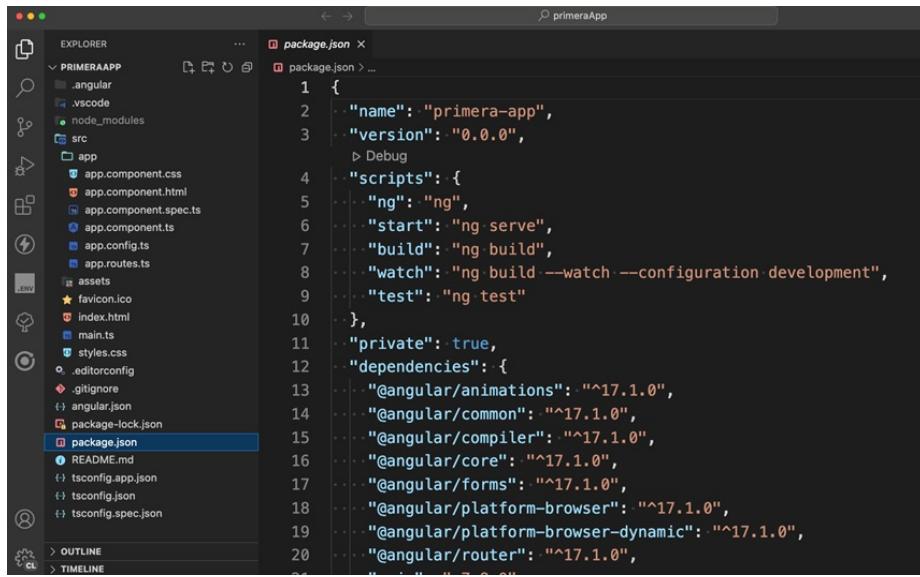


Figura 6. Pantalla inicial de un proyecto de Angular. Fuente: elaboración propia.

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.

# Tema 1. Introducción a Angular



```
1 {
2   "name": "primera-app",
3   "version": "0.0.0",
4   "scripts": {
5     "ng": "ng",
6     "start": "ng serve",
7     "build": "ng build",
8     "watch": "ng build --watch --configuration development",
9     "test": "ng test"
10   },
11   "private": true,
12   "dependencies": {
13     "@angular/animations": "^17.1.0",
14     "@angular/common": "^17.1.0",
15     "@angular/compiler": "^17.1.0",
16     "@angular/core": "^17.1.0",
17     "@angular/forms": "^17.1.0",
18     "@angular/platform-browser": "^17.1.0",
19     "@angular/platform-browser-dynamic": "^17.1.0",
20     "@angular/router": "^17.1.0",
21   }
22 }
```

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, iremos ampliando la mayoría de ellos.

## 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.

# Tema 1. Introducción a Angular

## 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 responsables del funcionamiento de nuestra aplicación. Dentro de esta carpeta cabe resaltar el archivo `app.config.ts`, dentro del cual encontramos la configuración general de la aplicación, así como la asignación del array en el que se encuentran las rutas a partir de las cuales generaremos nuestro flujo de trabajo.

## Directorio src/assets

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

Otro archivo es `angular.json`, este es esencial, ya que 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.

# Tema 1. Introducción a Angular

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

ng serve

Recuerda que si el servidor está 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:

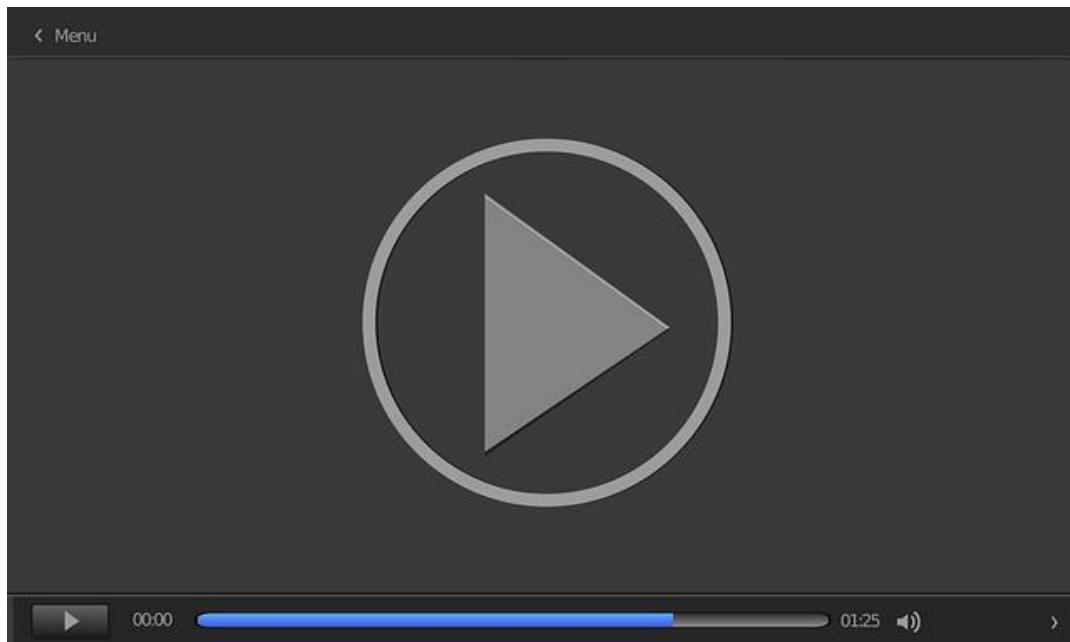


Figura 8. Combinación de teclas para parar un proyecto de Angular. Fuente: <https://necesitoayuda.eu/>

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.

# Tema 1. Introducción a Angular

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:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=e2eb6588-8dfa-440a-881b-b1270149baff>

---

## 2.1. 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.

## 2.2. 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
```

## 2.3. 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 creemos 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.

A parte 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!!")  
}
```

## Tema 2. TypeScript

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
```

## 2.4. 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.

## Tema 2. TypeScript

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.

## 2.5. 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{
```

# Tema 2. TypeScript

```
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 3. Arquitectura de la aplicación

## 3.1. Introducción y objetivos

El objetivo de este tema es presentarte la estructura básica a partir de la cual se genera la mayoría de las aplicaciones Angular. Aunque dispongamos de cierta flexibilidad a la hora de crear nuestras aplicaciones, necesitamos seguir una serie de normas y disponer de una serie de ficheros específicos para unir todos los elementos que vamos a ir generando.

Dicha estructura ya viene dada a partir de la creación de aplicaciones a través del CLI de Angular por lo que nosotros como desarrolladores solo tenemos que fijarnos en seguir unas buenas prácticas que se asemejen a lo que nos proponen los creadores del framework.

En las nuevas versiones de Angular los componentes son las piezas fundamentales de nuestras creaciones y es por eso por lo que conforman el núcleo central de todas las aplicaciones que vamos a crear.

# Tema 3. Arquitectura de la aplicación

## 3.2. Partes de una aplicación de Angular

Las últimas versiones de Angular han incluido un cambio fundamental en la arquitectura que usamos para poder desarrollar nuestras propias soluciones. Se ha colocado al **componente en el centro de todo**.

Cuando nos planteamos desarrollar cualquier aplicación con Angular debemos tener en cuenta que el resultado final será un puzzle conformado por una serie de componentes que serán capaces de comunicarse entre ellos y con el exterior. Es muy importante que mantengamos la independencia de dichos componentes para poder desarrollar así un código que se desacople de manera sencilla del contexto en el que lo creamos.

En versiones anteriores a la 17 observamos que la estructura fundamental de nuestras aplicaciones se asentaba en el **uso de módulos**. Podemos seguir trabajando con este concepto para poder agrupar diferentes funcionalidades y que «trabajen» bajo el mismo nombre. Durante los siguientes temas analizaremos los módulos más importantes que vamos a tener que seguir incluyendo dentro de nuestras aplicaciones.

Los **componentes** son lo que en programación denominamos las vistas. Están representados por clases Typescript, dentro de las cuales encontramos la lógica necesaria para poder crear las diferentes interfaces que componen nuestra aplicación.

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

**Componente = template + metadatos + clase**

# Tema 3. Arquitectura de la aplicación

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 o 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 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.

Existen cuatro tipos de **data binding**, los cuales iremos explicando con mayor profundidad durante el curso.

Veamos un poco por encima en qué consisten:

- ▶ **Interpolación:** nos permite pintar, dentro del HTML, cualquier valor que esté 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 la interfaz, en este caso haciendo click. Nos permite comunicar información desde el HTML hacia el TS.
- ▶ **Property binding:** es la forma que tenemos de asignarle a una propiedad de HTML el valor de una propiedad creada dentro de nuestro componente.

[src] = "propiedad"

# Tema 3. Arquitectura de la aplicación

Es como realizar una interpolación dentro de un atributo de HTML, lo que permite insertar cualquier sentencia JavaScript válida dentro de los atributos del HTML.

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

Las **directivas** permiten añadirle un **comportamiento dinámico** al HTML mediante un atributo o selector. Existen 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`.

En las nuevas versiones de Angular, aunque podamos seguir utilizando estas directivas, se han incluido herramientas más completas para poder controlar la creación de nuestras diferentes interfaces.

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

Los **inyectables**, o lo que se conoce como inyección de dependencias, permiten suministrarle 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.

# Tema 3. Arquitectura de la aplicación

## 3.3. Configuración básica

Uno de los cambios fundamentales de las últimas versiones de Angular es la capacidad que tienen ahora los componentes para poder **trabajar de manera independiente**, sin necesidad de estar asociados a un módulo.

Esto hace que, para poder crear una estructura de aplicación sólida y escalable, necesitemos un único componente que actuará como componente principal y a partir del cual colgarán el resto de los componentes. De esta manera estaremos generando una **jerarquía de componentes** que es fundamental conocer para poder establecer los diferentes flujos a través de los cuales movemos la información.

Para entender cómo se inician nuestras aplicaciones debemos revisar el fichero **main.ts**.

```
> main.ts > ...
1 import { bootstrapApplication } from '@angular/platform-browser';
2 import { appConfig } from './app/app.config';
3 import { AppComponent } from './app/app.component';
4
5 bootstrapApplication(AppComponent, appConfig)
6   .catch((err) => console.error(err));
7
```

Figura 9. Fichero main.ts. Fuente: elaboración propia.

Se trata del fichero inicial que se ejecuta cuando ponemos el servidor de nuestro proyecto a funcionar.

Lo más importante de este fichero es la ejecución del método **bootstrapApplication**. Como parámetros le estamos pasando, en primer lugar, la **referencia al componente** que va a actuar como componente **principal** (AppComponent) y, en segundo lugar, un **objeto** en el que tendremos reflejada la **configuración más global de nuestro proyecto** (app.config.ts).

# Tema 3. Arquitectura de la aplicación

La **diferencia fundamental** de esta arquitectura con respecto a la arquitectura modular es que, en esta segunda, la activación de todos los recursos contenidos en nuestro proyecto se realizaba en el interior de un módulo.

No es que vayamos a estar constantemente modificando el fichero **app.config.ts**, pero sí es importante que conozcamos su contenido, que sepamos cuándo tenemos que recurrir a él para poder establecer ciertas configuraciones globales dentro de nuestra aplicación.

```
1 import { ApplicationConfig } from '@angular/core';
2 import { provideRouter } from '@angular/router';
3
4 import { routes } from './app.routes';
5
6 export const appConfig: ApplicationConfig = {
7   providers: [provideRouter(routes)]
8 };
9 
```

Figura 10. Fuente: elaboración propia.

Como observamos en la imagen, dentro del fichero principal de configuración únicamente definimos e inicializamos la **variable appConfig**. Dentro de las diferentes opciones que incluimos en dicha variable encontraremos, sobre todo, las configuraciones más globales para nuestra aplicación.

En el caso de disponer de un proyecto creado desde cero, la configuración básica se limita a especificar dónde se encuentra el array de rutas que vamos a utilizar para poder saber qué interfaz se activa en función de la URL que dispongamos en el navegador (tema que ya profundizaremos más adelante).

# Tema 3. Arquitectura de la aplicación

## 3.4. Arquitectura por componentes

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 Typescript).

La **estructura general** parte de un conjunto de librerías, las obligatorias, que componen el núcleo central de Angular y una 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 en una jerarquía de componentes tal y como hacemos cuando se maqueta una página web normal y corriente.

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

A la hora de organizar nuestro código es recomendable pensar en la escalabilidad y la limpieza dentro de nuestra aplicación. Son conceptos que nos ayudan a mantener una buena organización y un correcto flujo de trabajo mientras nos encontramos en la etapa de desarrollo. Uno de los conceptos fundamentales que debemos tratar antes de ponernos a desarrollar es **cómo estructuramos los directorios** dentro de nuestra aplicación para que **nuestro trabajo sea óptimo**.

Partimos de la base de que cada proyecto es totalmente diferente a los anteriores y que, dependiendo de nuestra experiencia o nuestra forma de trabajar, podremos implementar una arquitectura u otra.

Existe una vertiente muy extendida en el desarrollo de aplicaciones Angular que **divide el código** generado **por funcionalidades**. Dependiendo de la labor que

## Tema 3. Arquitectura de la aplicación

vayan a cumplir nuestros componentes dentro del conjunto de la aplicación final, serán colocados en un directorio u otro.

Seguir esta forma de trabajar nos permite tener localizados en todo momento las partes fundamentales de nuestra aplicación y **facilita el trabajo en equipo sin incidir** en demasiados **conflictos de código** (cada persona o equipo puede encargarse de una funcionalidad distinta).

Durante los próximos temas nuestra tarea será analizar las diferentes herramientas que conforman nuestras aplicaciones y situarlas de manera óptima dentro de los proyectos que vayamos desarrollando.

# Tema 3. Arquitectura de la aplicación

## 3.5. Creación de módulos

Aunque los módulos ya no son una parte esencial de la creación de una aplicación con Angular, sí que podemos seguir utilizándolos para agrupar diferentes funcionalidades, clases o componentes que, por su naturaleza, deban trabajar de manera conjunta.

El comando del **CLI** que vamos a usar ahora para generar un módulo nuevo es “**generate**”. A continuación, tenemos que indicar qué 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
```

Una vez lanzado este comando en nuestro proyecto, dentro de la carpeta "src/app" se crea un subdirectorio con el mismo nombre del módulo generado. Dentro encontraremos además el archivo con el código del módulo.

# Tema 3. Arquitectura de la aplicación

La **estructura básica de un módulo** la podemos observar en el siguiente código:

```
1 import { NgModule } from '@angular/core';
2 import { CommonModule } from '@angular/common';
3
4 @NgModule({
5   declarations: [],
6   imports: [
7     CommonModule
8   ]
9 })
10 export class PruebaModule {}
```

Figura 11. Estructura básica de un módulo. Fuente: elaboración propia.

El decorador **@NgModule** nos permite disponer de una serie de opciones para poder agrupar los diferentes elementos que van a formar parte de este módulo.

La propiedad **declarations** es un array en el que colocaremos estos elementos (componentes, pipes, directivas...) que van a formar parte de este módulo que estamos creando y que, por tanto, podríamos utilizar en un futuro asociados a dicho módulo.

Dentro del array que colocamos como valor de la propiedad **imports** situamos la serie de módulos externos al nuestro que vamos a utilizar en el código que generemos para nuestro módulo. En este caso estamos incorporando el módulo **CommonModule** a los imports con todas las herramientas básicas para el trabajo con Angular.

# Tema 4. Componentes

## 4.1. 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. Además, se encargan de definir y ejecutar la lógica mediante la cual gestionaremos los eventos y las acciones externas.

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 cuál es la importancia del desarrollo de componentes en Angular y como plantearnos la creación de un sistema basado en componentes.

## 4.2. ¿Qué es un componente?

Un componente es una clase a la que le precede un decorador `@Component`. Mediante la configuración de este podemos crear nuevas etiquetas HTML mediante las cuales podemos añadir funcionalidad a nuestras aplicaciones controlando determinadas zonas de pantalla.

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.

## 4.3. 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 } from '@angular/core';

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

}
```

Las primeras líneas del fichero que representa un componente de Angular están reservadas para importar todas aquellas herramientas de la propia librería o externas que vamos a utilizar dentro del propio fichero.

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 el decorador **@Component** de la librería **@angular/core**, donde encontramos la base para la creación de componentes en Angular. Mediante este decorador vamos a poder configurar todos los metadatos que caracterizan a este componente.

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

# Tema 4. Componentes

Un componente se compone de tres partes:

**¿Qué son las anotaciones?** Son los metadatos que agregamos a nuestro código a través del decorador @Component y que permiten 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-prueba></app-prueba>
```

- ▶ **standalone:** se trata de un valor boolean que nos indica si el componente puede existir por sí mismo sin necesidad de pertenecer a un módulo (true) o, por el contrario, para poder usarlo, debemos incluirlo dentro de un módulo (false). La configuración por defecto cuando generamos componentes es colocar este valor como true, pero no debemos descartar la posibilidad de incluir ciertos componentes dentro de algún módulo en nuestras aplicaciones.
- ▶ **Imports:** esta propiedad está profundamente ligada con la anterior. Si nuestro componente trabaja de manera independiente y no está incluido en ningún módulo, necesitamos incluir, dentro de este array, todas las herramientas (módulos externos, componentes, directivas...) que vamos a incluir en el componente que estamos desarrollando. Al encontrarse aislado del resto, si queremos utilizar cualquier elemento, debemos incluirlo aquí.
- ▶ **template / templateUrl :** con estas propiedades especificamos el contenido que va a renderizar nuestro componente cuando en el DOM situemos el selector que lo define.

# Tema 4. Componentes

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-prueba',
  standalone: true,
  imports: [],
  template: `<h1>Hola Angular</h1>`,
  styles: [
    h1 { color: #fff; }
    .description {
      font-style: italic;
      color: green;
    }
  ]
})
export class PruebaComponent {
```

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.

# Tema 4. Componentes

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.

A parte 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.

## 4.4. 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` : contiene el código HTML mediante el cual fabricamos la pantalla (o vista).
- ▶ `app.component.css` : define los estilos que usaremos en la vista del componente.
- ▶ `app.component.ts` : está 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.

# Tema 4. Componentes

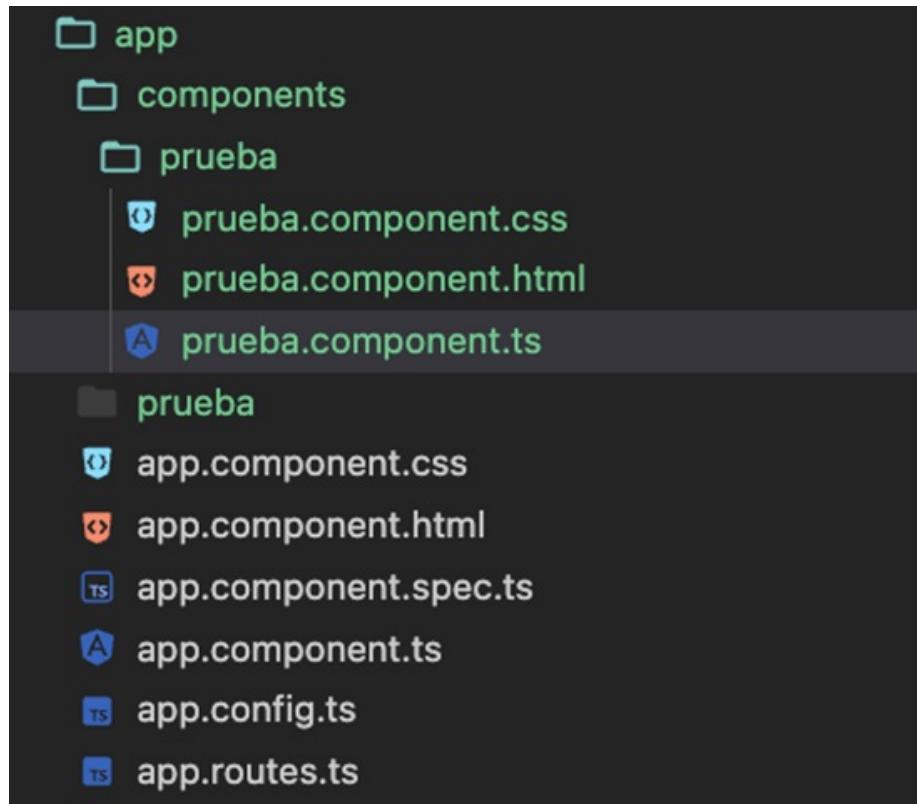


Figura 12. Árbol de carpetas de un componente. Fuente: elaboración propia.

Para crear un componente nos vamos a servir de la herramienta angular CLI que instalamos anteriormente y que, entre otras muchas cosas, nos va a servir para crear todos los elementos que utilizaremos dentro de nuestras aplicaciones Angular.

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

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

# Tema 4. Componentes

También tienes un formato corto para este script en terminal:

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

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

## 4.5. 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 incorporar el componente hijo dentro del array de imports de la configuración del componente padre.

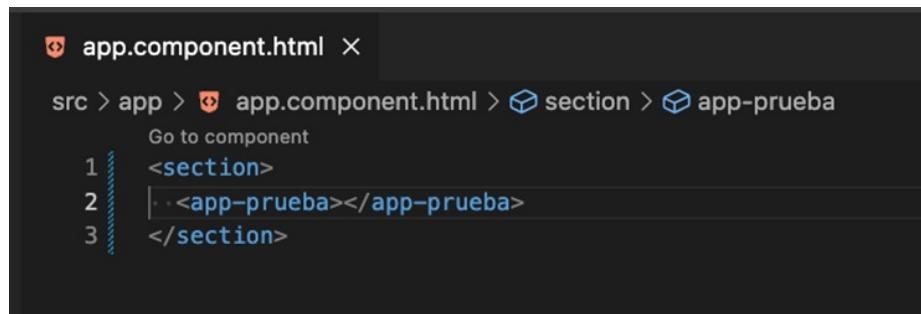
```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { PruebaComponent } from './components/prueba/prueba.component';

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

# Tema 4. Componentes

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:



```
app.component.html
src > app > app.component.html > section > app-prueba
1 <section>
2   <app-prueba></app-prueba>
3 </section>
```

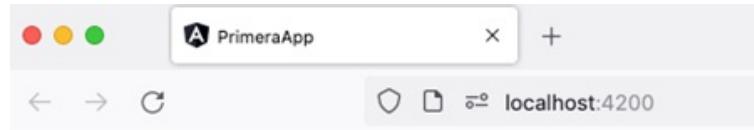
Figura 13. Cómo se enlazan dos componentes. Fuente: elaboración propia.

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 cargada el componente primer-componente visualizándose de la siguiente manera.

## Tema 4. Componentes



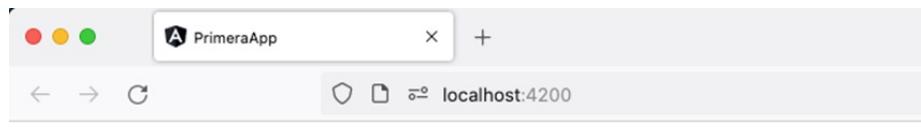
**prueba works!**

Figura 14. 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:



**Titulo del primer componente**

Primer parrafo

Segundo parrafo

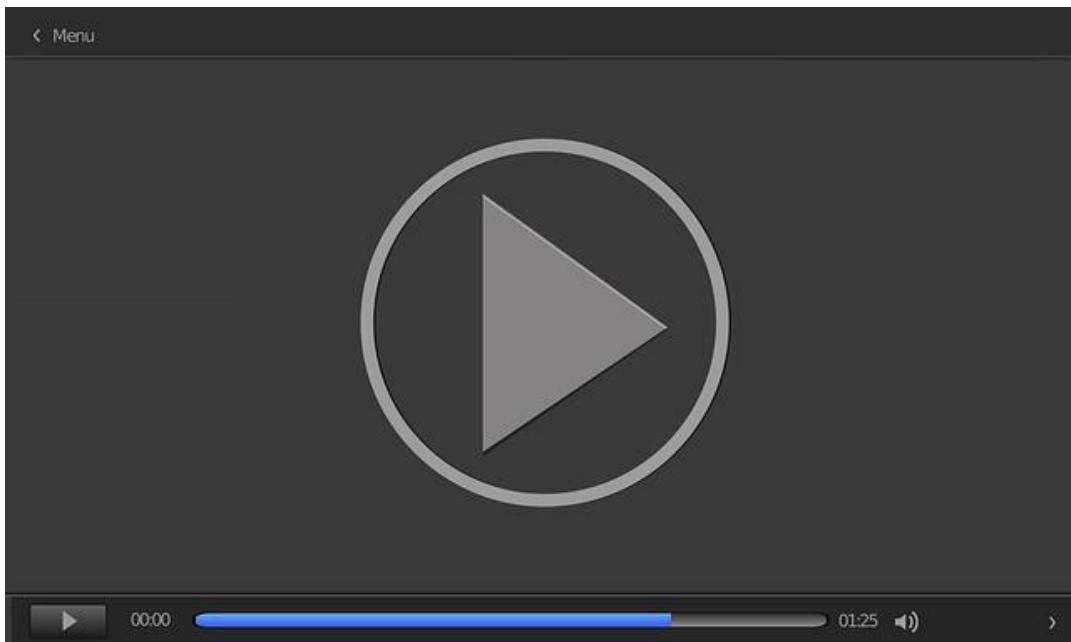
Figura 15. Resultado del navegador. Fuente: elaboración propia.

# Tema 4. Componentes

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:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=aff00d65-832f-42ba-abac-b127015156b0>

---

## 4.6. 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 16. 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.

# Tema 4. Componentes

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 hace inicializa las propiedades del componente.

### ngOnChanges

Este método es llamado cuando se aprecia algún cambio en cualquiera de las propiedades que recibe nuestro componente. Veamos 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 .

# Tema 4. Componentes

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 con 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

# Tema 4. Componentes

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.

## 5.1. Introducción y objetivos

Nos encontramos ante un tema corto, pero bastante visual, ya que en él 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.

## 5.2. 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 mucho 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.

# Tema 5. Templating

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 Binding (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.

# Tema 5. Templating

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 } from '@angular/core';

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

  //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 = 'Título del Componente'

  }
}
```

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 tratasesen, pero sin usar la palabra let, var, const que usábamos en JavaScript.

Dichas propiedades aparte de declararla y asignarles un tipo, tenemos la obligación

# Tema 5. Templating

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.

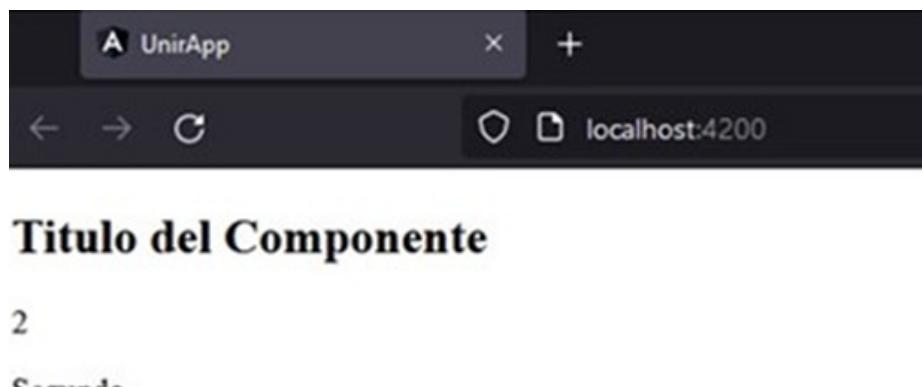


Figura 17. Renderizado de la interpolación de título y número. Fuente: elaboración propia.

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.

# Tema 5. Templating

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 } from '@angular/core';

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

  //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'
  }

  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.

# Tema 5. Templating



Figura 18. 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 } from '@angular/core';

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

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

  constructor() {}

  ngOnInit(): void {}

  getContenido(): string {
```

# Tema 5. Templating

```
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>{{ getContenido() }}</p>
```

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



Figura 19. 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.

## Tema 5. Templating

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]="getContenido()"></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.

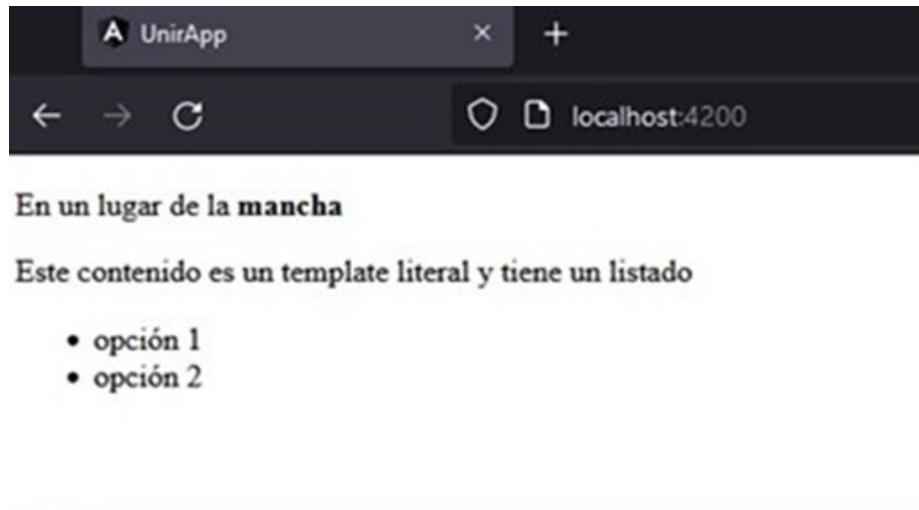
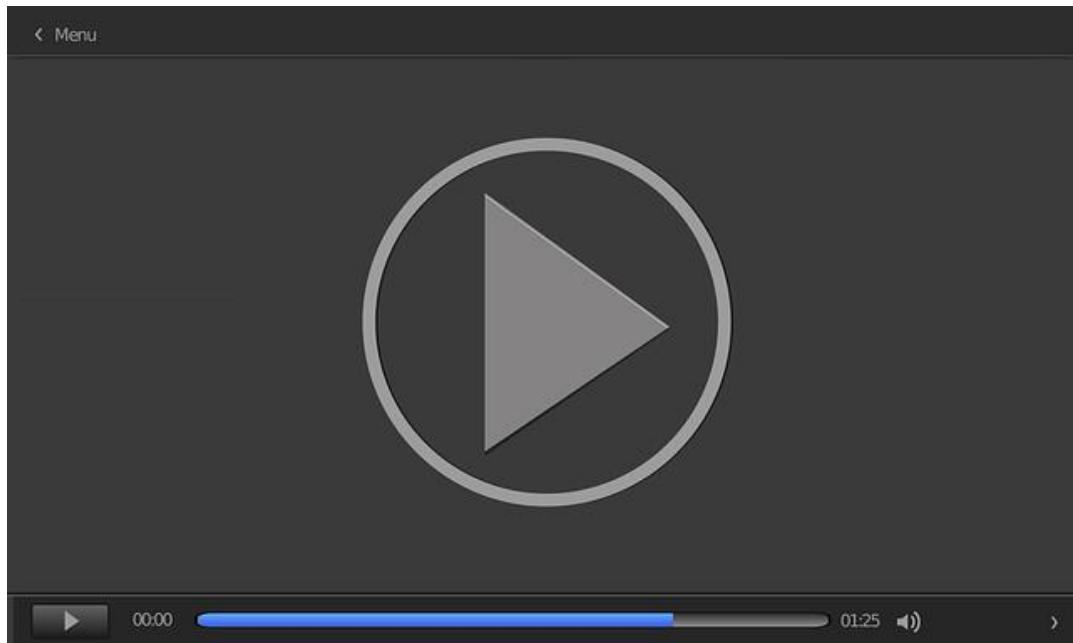


Figura 20. Renderizado correcto de contenido HTML dentro de la plantilla. Fuente: elaboración propia.

## Tema 5. Templating

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:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=73f3997e-e631-43a7-a08d-b127015dc02f>

---

## 5.3. 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.

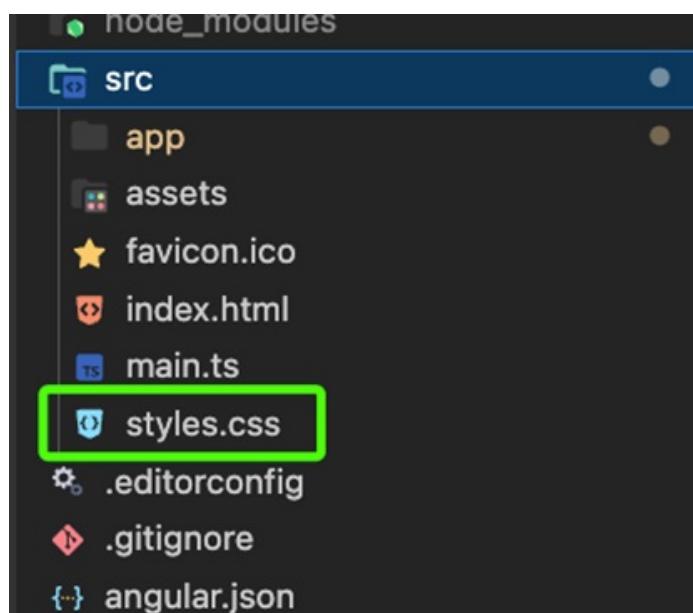


Figura 21. Style.css. Fuente: elaboración propia.

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.

```
Cursos/06_UNIR/Actualización_2024
→ ng new SassExample
? Which stylesheet format would you like to use?
CSS
➤ SCSS [ https://sass-lang.com/documentation/syntax#scss ]
Sass [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
Less [ http://lesscss.org ]
```

Figura 22. 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

# Tema 5. Templating

forma. Damos de alta justo debajo de script una propiedad stylePreprocessorOptions que contiene otra 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.

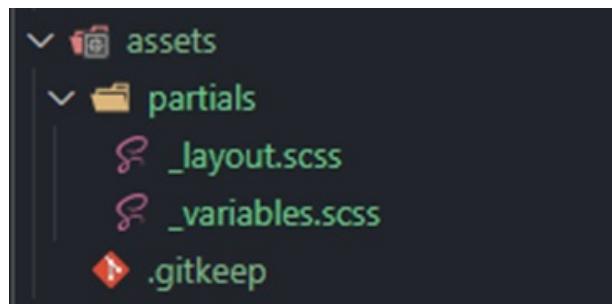


Figura 23. Sistema de carpetas SCSS dentro de un proyecto. Fuente: elaboración propia.

## 5.4. Angular signals

A partir de la versión 16 de Angular tenemos disponible la tecnología de **signals** mediante la cual podemos gestionar los diferentes estados dentro de nuestra aplicación. Sería una alternativa para tener en cuenta para gestionar cómo se modifican las propiedades de nuestros componentes que posteriormente visualizaremos dentro de las diferentes vistas asociadas.

A diferencia de las propiedades generadas dentro de la clase de un componente, un signal es un **wrapper en torno a un valor**, el cual es capaz de informar de los cambios ocurridos en dicho valor a todos los «interesados». Es decir, si estamos usando un signal para visualizar un valor dentro de nuestro HTML, cuando lancemos la acción para modificar el valor del signal, éste será capaz de avisar a la vista para poder incluir el nuevo valor.

Para poder visualizar el valor de un signal debemos utilizar su método `get` asociado.

En el siguiente ejemplo podemos ver cómo crear un componente que represente un simple contador.

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

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

  // Creamos el objeto signal asignándole un valor inicial
  cont = signal(0);
```

# Tema 5. Templating

}

Para inicializar el valor correspondiente llamamos a la función **signal** de la librería **@angular/core** pasándole el valor inicial que necesitemos en cada caso.

Para poder recuperar el valor del signal en la vista ejecutamos el método **get** asociado

```
<div>
  <p>Contador: {{cont()}}</p>
</div>
```

Podemos asignar un nuevo valor al signal a través de la función **set**:

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

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

  cont = signal(0);

  constructor() {
    this.cont.set(10);
  }
}
```

En el momento de realizar la asignación en el constructor, la vista recibirá el aviso de los cambios y actualizará el valor nuevo dentro del espacio que ocupa dentro del HTML.

Si el cambio que necesitamos realizar toma como base el valor anterior, podemos usar el método **update**.

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

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

  cont = signal(0);

  constructor() {
    setInterval(() => {
      // Modificamos el valor evitando efectos secundarios a partir del valor anterior
      // del signal
      this.cont.update(value => value + 1);
    }, 1000);
  }
}
```

## Computed signals

Podemos generar signals que solo sean de lectura y cuyo valor se calcule en función de los cambios que se producen en otro signal que sea de lectura-escritura.

En el ejemplo podemos ver cómo mantener actualizado el número de caracteres de una cadena de caracteres recogida de un campo de texto:

```
import { Component, Signal, WritableSignal, computed, signal } from '@angular/core';

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

  mensaje: WritableSignal<string> = signal("");
  numCaracteres: Signal<number> = computed(() => this.mensaje().length);
}
```

# Tema 5. Templating

```
onChange($event: any) {  
  // Modificando el valor del signal de lectura escritura  
  // modificamos también el signal de solo lectura  
  this.mensaje.set($event.target.value)  
}  
}
```

## Conclusión

La **gestión de las plantillas** dentro de nuestras aplicaciones de Angular es uno de los conceptos básicos que debemos manejar a la perfección. Conocer todas las posibilidades que tenemos disponibles para comunicar nuestras clases Typescript con las plantillas asociadas facilita muchísimo el trabajo de crear componentes complejos. Para esto es imprescindible manejar la interpolación sencilla y el uso de signals.

La **aplicación de estilos** sobre nuestros componentes también tiene su especial importancia ya que nos permite aislar lo máximo posible los estilos sobre un elemento concreto de nuestra aplicación. El uso de CSS o SASS es muy particular y, sobre todo, depende de la experiencia del equipo de desarrollo.

# Tema 6. Data Binding: One-way and Two-ways

## 6.1. 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 Binding**.

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.

# Tema 6. Data Binding: One-way and Two-ways

## 6.2. Property Binding

Cuando necesitemos **enlazar propiedades** del componente a las propiedades de los elementos del DOM podemos usar Property Binding como alternativa a la interpolación (que también podría usarse en estos casos). Para usarlo pondremos la propiedad del elemento del DOM entre “[...]”, con lo que le asignamos la propiedad del componente.

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 } from '@angular/core';

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

  ruta: string = 'http://www.google.es'

}
```

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

# Tema 6. Data Binding: One-way and Two-ways

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 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 {
  }
}
```

# Tema 6. Data Binding: One-way and Two-ways

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:

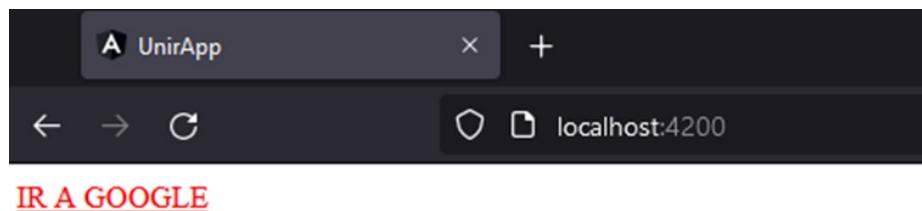


Figura 24. Resultado del Property Binding. Fuente: elaboración propia.

De igual forma, también nos permite especificar una clase de CSS para añadir a un elemento del DOM si una propiedad de un componente es true. Su sintaxis es similar a la de Property binding, con la diferencia de que hay que añadir class delante la clase CSS y que la propiedad del componente debe ser tipo booleano.

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'
```

# Tema 6. Data Binding: One-way and Two-ways

```
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  
}
```

# Tema 6. Data Binding: One-way and Two-ways

## 6.3. Two way date biding: 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 del componente donde vayamos a usarla, debemos importar la librería `FormsModule`.

El fichero con la lógica de nuestro componente quedaría de la siguiente manera:

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

@Component({
  selector: 'app-prueba',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './prueba.component.html',
  styleUrls: ['./prueba.component.css']
})
export class PruebaComponent {

  textoInput: string = 'Texto de prueba';
}
```

# Tema 6. Data Binding: One-way and Two-ways

Posteriormente podemos usar la directiva dentro de la plantilla de la siguiente manera:

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

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

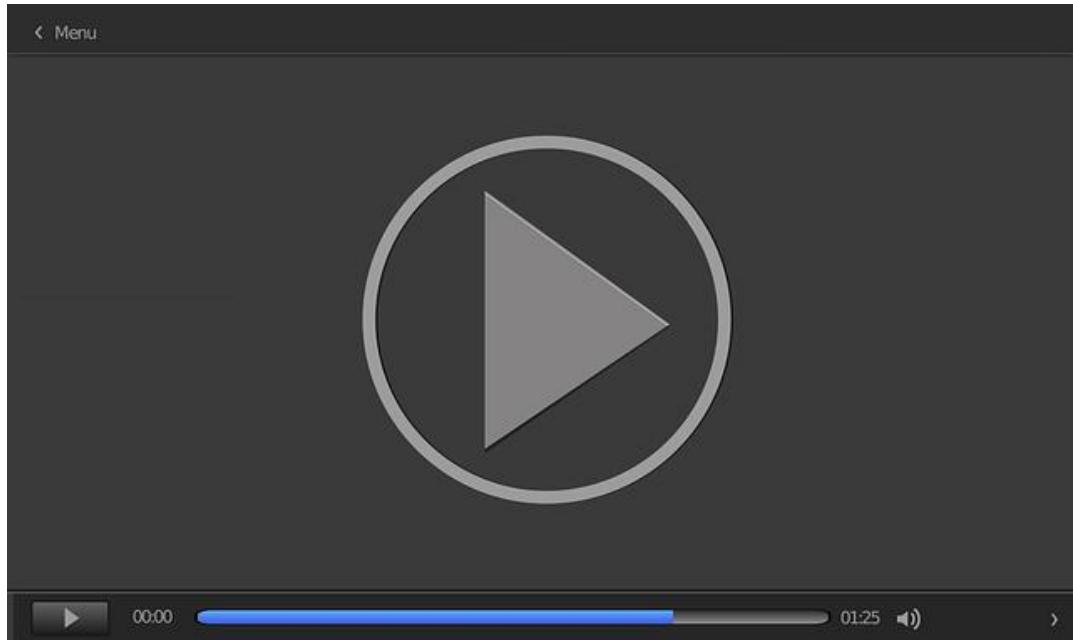
Si modificamos la variable, se modifica el valor del campo de texto y viceversa.

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.

# Tema 6. Data Binding: One-way and Two-ways

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:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=2d4538d9-7222-4dcf-85ab-b12801375e76>

---

## 7.1. 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.

## 7.2. Creación de eventos en Angular

Event Binding nos permite **capturar cualquier evento estándar** del DOM (hacer click en un botón, pasar el ratón por encima de determinado elemento, etc.) y **transmitirlo al componente** para que realice las acciones que convengan. Para hacer uso de Event Binding simplemente pondremos el evento que queramos controlar entre paréntesis, sin el “on” de delante y lo asignaremos a una función del componente.

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 comillas 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 } from '@angular/core';

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

## Tema 7. Event Binding

```
})  
export class PruebaComponent {  
  
    hacerClick() {  
        alert('has hecho click en el botón')  
    }  
  
}
```

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.

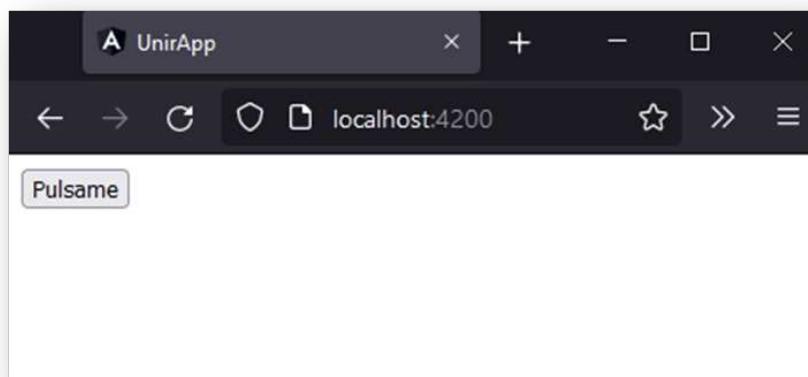


Figura 25. Resultado de la vista de componente antes de hacer click. Fuente: elaboración propia.

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.

# Tema 7. Event Binding

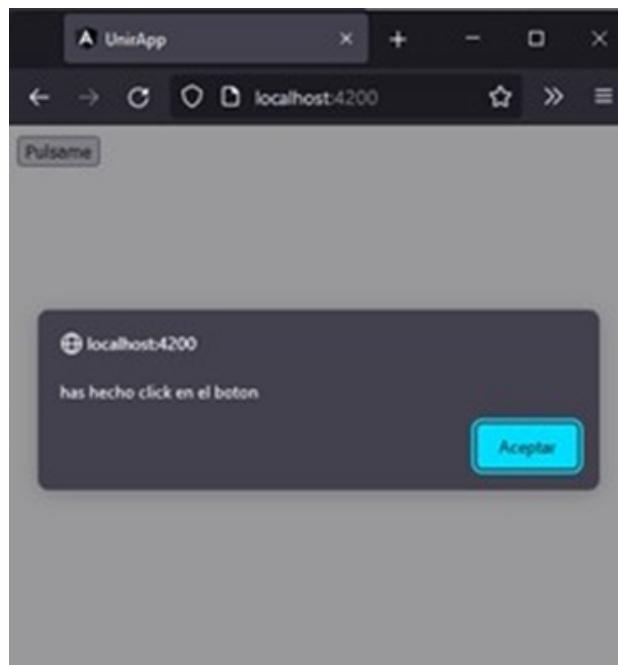


Figura 26. Salida por pantalla de la resolución del evento. Fuente: elaboración propia.

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 Binding. Os recuerdo los pasos para poder realizar esta técnica:

- ▶ Incluir **FormsModule** dentro de los imports del componente.
- ▶ 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.

# Tema 7. Event Binding

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.

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

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

  //Creamos e inicializamos la propiedad
  alumno: any = {
    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>
```

# Tema 7. Event Binding

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

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

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

  //Creamos e inicializamos la propiedad
  alumno: any = {
    nombre: '',
    edad: 0
  }

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

}
```

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

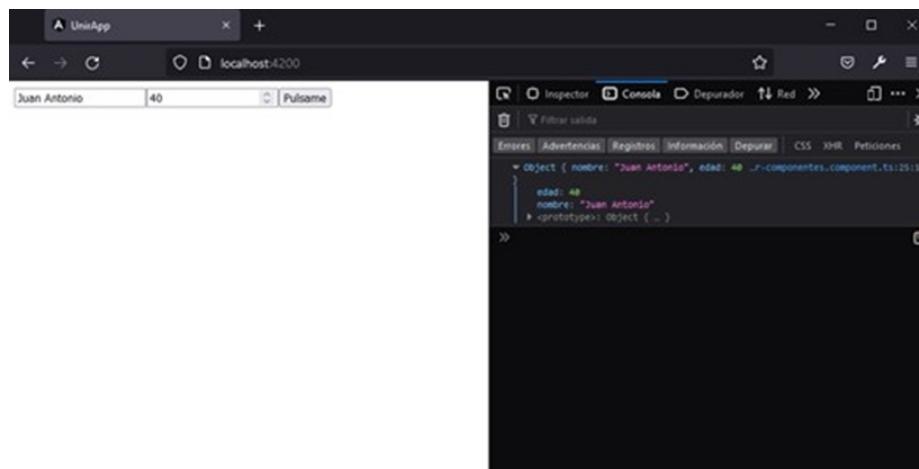


Figura 27. Captura de resultado del evento en consola al hacer click. Fuente: elaboración propia.

# Tema 7. Event Binding

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()>
```

## 7.3. \$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`.

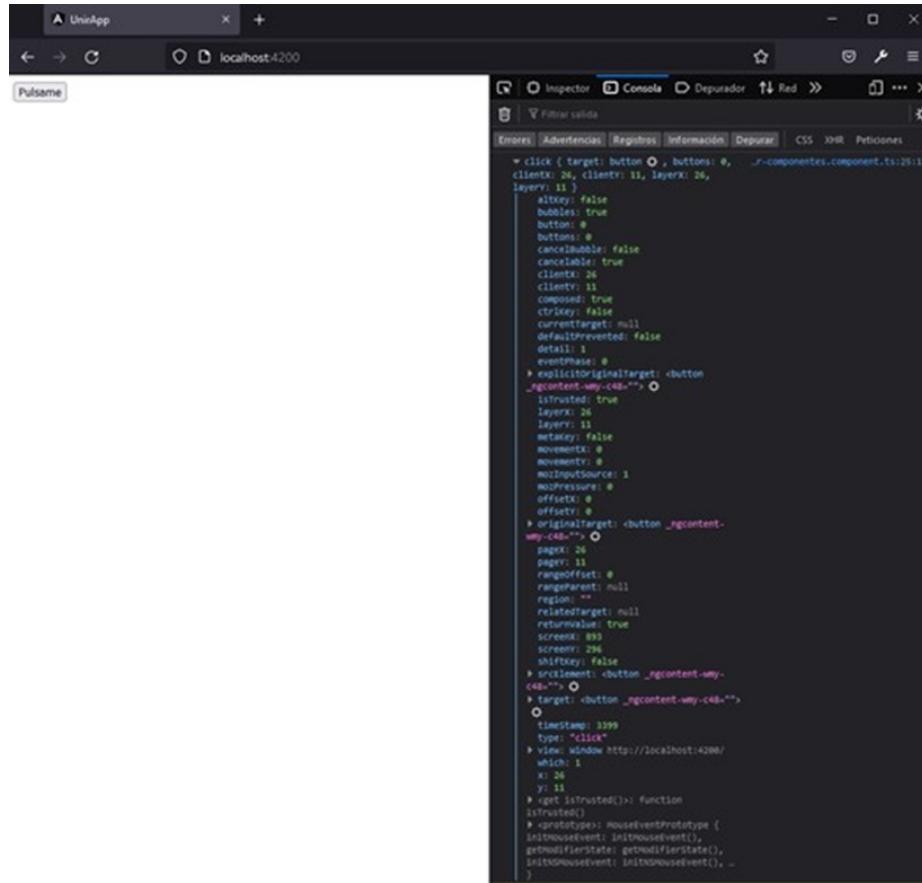
```
<button (click)="capturarDatos($event)">Pulsame</button>
```

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.

# Tema 7. Event Binding



A screenshot of a browser developer tools console window titled "UnirApp". The address bar shows "localhost:4200". The console tab is active, displaying the following JavaScript code and its corresponding event object properties:

```
clientX: 26, clientY: 11, layerX: 26, layerY: 11
altKey: false
ctrlKey: true
button: 0
buttons: 0
cancelable: false
cancelable: true
clientX: 26
clientY: 11
composed: true
ctrlKey: false
currentTarget: null
defaultPrevented: false
detail: 1
eventPhase: 0
explicitOriginalTarget: <button _ngcontent-wy-c48="">
isTrusted: true
layerX: 26
layerY: 11
metaKey: false
movementX: 0
movementY: 0
mozIsSource: 1
mozPressure: 0
offsetX: 0
offsetY: 0
originalTarget: <button _ngcontent-wy-c48="">
pageX: 26
pageY: 11
rangeOffset: 0
rangeParent: null
region: ""
relatedTarget: null
returnValue: true
screenX: 89
screenY: 26
shiftKey: false
timestamp: 3399
type: "click"
view: window http://localhost:4200/
which: 1
x: 26
y: 11
get isTrusted(): Function
isTrusted()
get prototype(): MouseEventPrototype {
  initMouseEvent: initMouseEvent(),
  get bubblesState: get bubblesState(),
  initMouseEvent: initMouseEvent(),
  ...
}
```

Figura 28. Captura de la salida de consola del navegador. Fuente: elaboración propia.

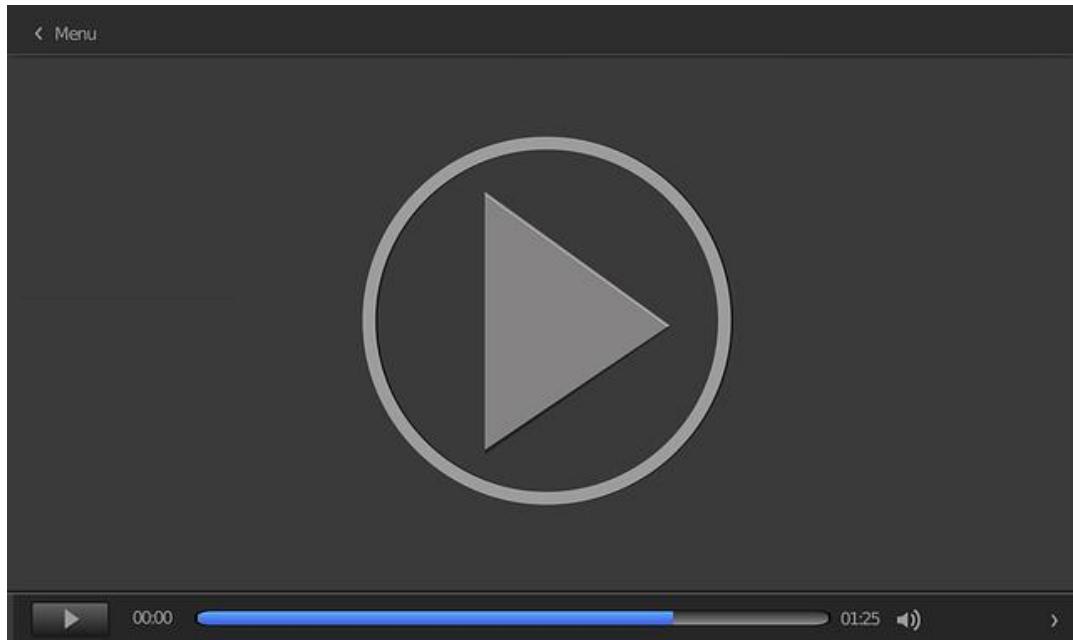
El target al igual que en JavaScript nativo representa el elemento del DOM que ejecuta el evento, y a partir de este elemento podemos capturar una propiedad del objeto, como por ejemplo el id, o el texto del botón con innerText, al igual que hacíamos en JavaScript. Por ejemplo \$event en caso de un evento 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 Biding o un texto a través de la interpolación.

# Tema 7. Event Binding

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.



---

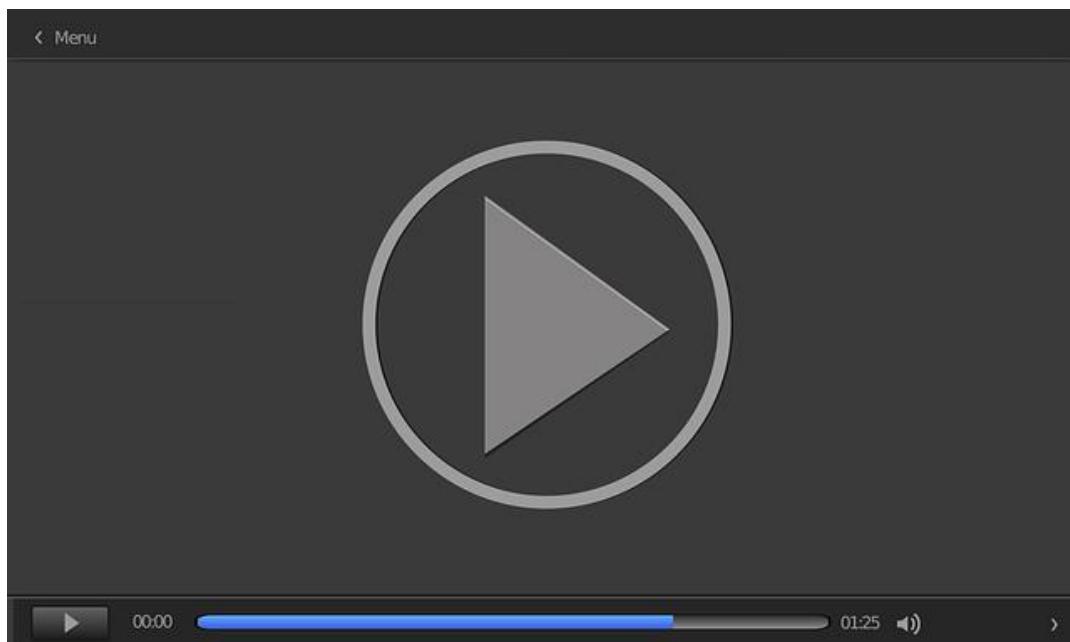
Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=9cdc41ba-d90f-4dc2-8d3c-b128013efaa5>

---

## Tema 7. Event Binding

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



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=44a7b734-4141-4d93-85c4-b128014c8316>

---

# Tema 8. Input y Output

## 8.1. 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.

# Tema 8. Input y Output

## 8.2. 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`.

# Tema 8. Input y Output



Figura 29. Explicación gráfica de un input. Fuente: elaboración propia.

Un componente que tiene 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 .

# Tema 8. Input y Output



Figura 30. 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.

# Tema 8. Input y Output

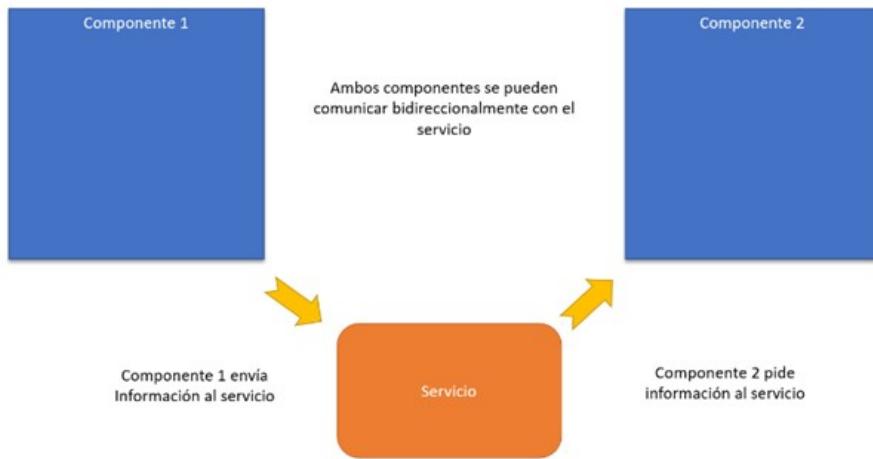


Figura 31. Comunicación a través del servicio. Fuente: elaboración propia.

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.

# Tema 8. Input y Output

## 8.3. 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, Input } from '@angular/core';
```

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

  @Input() texto: string = "";
```

# Tema 8. Input y Output

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-prueba [texto]=""Una cadena de texto que viene del padre"></app-prueba>
</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-prueba [miTexto]=""Una cadena de texto que viene del padre"></app-prueba>
</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.

# Tema 8. Input y Output

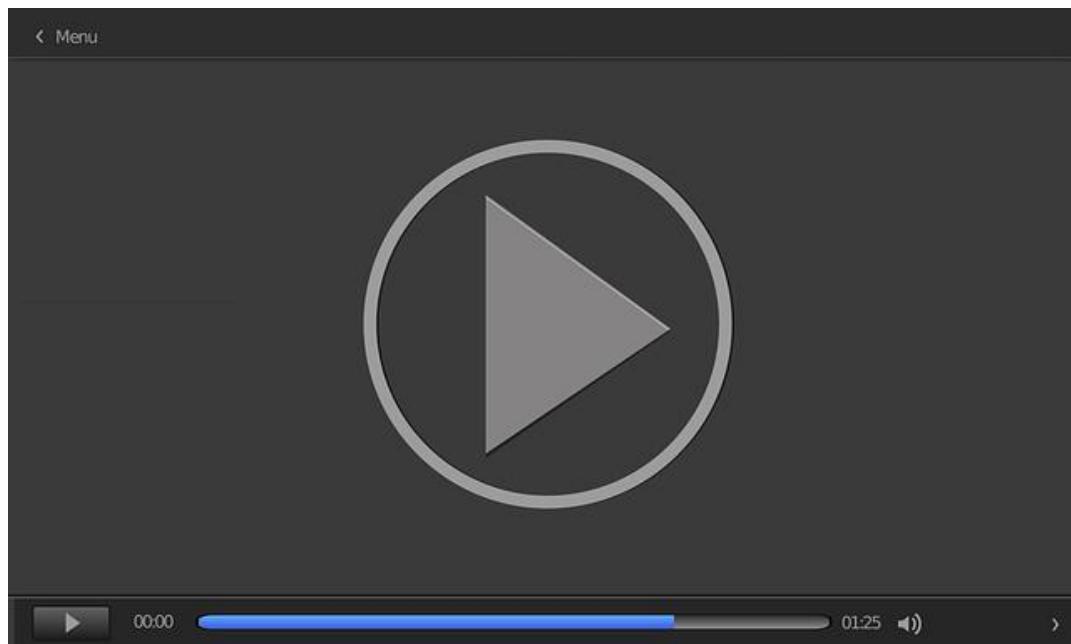
Si os acordáis de las funciones de ciclo de vida de los componentes lo input y los outputs se cargan con el `ngOnChanges` o con el `ngOnInit`.

```
//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 habréis 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:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=29adcdd6-aea2-46f7-9329-b12801522da2>

---

# Tema 8. Input y Output

## 8.4. 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, Input, Output } from '@angular/core';
```

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

  @Output() delete = new EventEmitter()

}
```

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

# Tema 8. Input y Output

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, Input, Output } from '@angular/core';

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

  @Output() delete = new EventEmitter()

  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.

# Tema 8. Input y Output

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-prueba (delete)="handleDeleteFather($event)"></app-prueba>
</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';
import { RouterOutlet } from '@angular/router';
import { PruebaComponent } from './components/prueba/prueba.component';

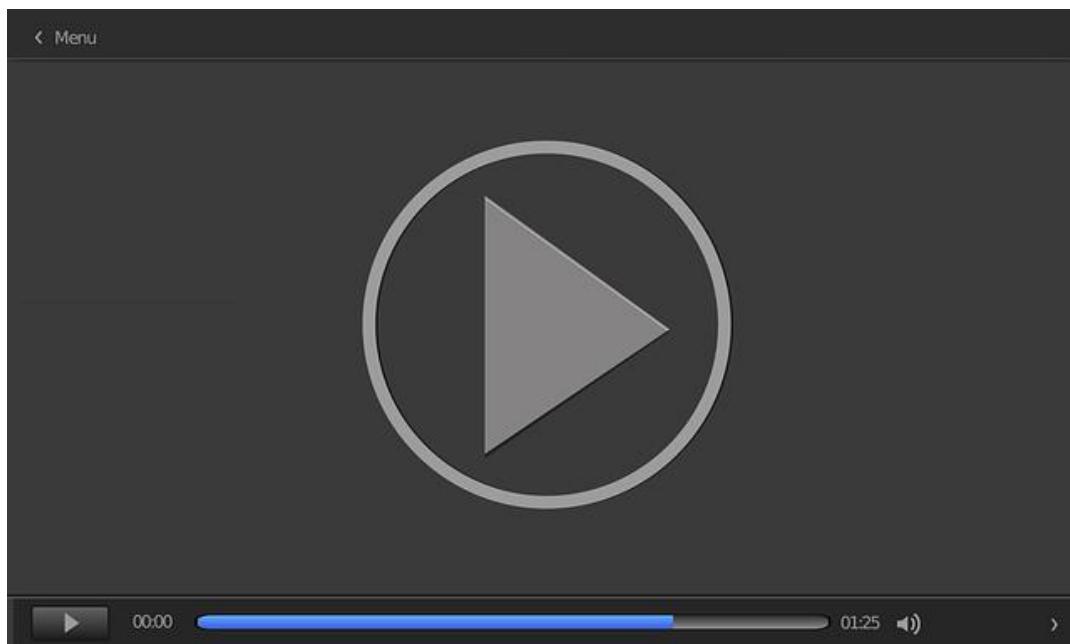
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, PruebaComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

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

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

# Tema 8. Input y Output

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



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=c5b0e6a8-365c-42d0-9d90-b1280169d9b5>

---

## 9.1. Introducción y objetivos

Con la gestión de la estructura de nuestras plantillas de manera dinámica alcanzamos uno de los puntos fundamentales del aprendizaje de Angular.

Desde la versión 17 del framework tenemos disponibles una **serie de herramientas** que nos van a permitir reorganizar los bloques de código HTML con los que vamos a crear nuestras interfaces de manera dinámica y de una forma mucho más simple que como veníamos haciendo hasta ahora.

Las posibilidades que nos ofrece esta serie de herramientas hacen que nuestro HTML deje de ser completamente estático y se vaya modelando en función de los datos disponibles en nuestros diferentes componentes, así como los que podamos recibir desde el exterior.

A partir de este momento cobra mucho más sentido el hecho de crear aplicaciones basadas totalmente en componentes reutilizables.

## 9.2. Bloque condicional con @if

Una de las acciones más habituales que vamos a necesitar como desarrolladores es la de mostrar u ocultar el contenido de cierta parte de nuestras plantillas en función de ciertas condiciones.

Estas condiciones van a permitirnos construir la interfaz que le mostraremos al usuario y además van a poder ser modificadas a través de los eventos que active el propio usuario.

La utilización de este bloque nos inhibe de tener que modificar los estilos de un componente concreto para mostrar u ocultar su contenido.

El uso del **bloque @if** es similar al que le damos en Javascript. En el siguiente ejemplo vemos cómo podemos mostrar u ocultar elementos de un componente a partir de una condición.

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

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

  edad: number = 23;

}
```

# Tema 9. Control Flow

Dentro del HTML del componente podemos utilizar la propiedad edad para crear nuestra condición.

```
<section>
  @if(edad > 18){
    <p>Eres mayor de edad</p>
  }
</section>
```

El párrafo contenido dentro del bloque condicional se muestra o se oculta dependiendo de si se cumple o no la operación lógica que hemos especificado.

Es **muy importante** que, a diferencia de la utilización de estilos CSS para ocultar ciertos elementos, sepamos que, con el bloque `@if`, el elemento sobre el cual estamos actuando **desaparece por completo del HTML**. Angular es capaz de saber, al analizar la condición, cómo y cuándo debe aparecer dicho elemento.

## ¿Por qué eliminamos el contenido en vez de ocultarlo?

Podríamos realizar el mismo trabajo que hace este bloque con la aplicación de ciertos estilos CSS que nos permiten mostrar/ocultar el HTML. Con `@if` 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 lo afecten y seguirá enviando los datos que se modifiquen.
- ▶ Sigue usando recursos de nuestra aplicación.
- ▶ Por otro lado, el cambio de oculto a no oculto es muy rápido.

# Tema 9. Control Flow

El trabajo con **@if** es diferente:

- ▶ El ocultar bloques de código afecta a su rendimiento, ya que se eliminan del DOM.
- ▶ Deja de recibir eventos.
- ▶ 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.

Junto con el bloque **@if** tenemos también la posibilidad de establecer qué elementos se van a visualizar en caso de que la condición sea negativa. Esta acción la logramos a través del bloque **@else**.

```
<section>
  @if(edad > 18){
    <p>Eres mayor de edad</p>
  }@else {
    <p>NO eres mayor de edad</p>
  }
</section>
```

En el ejemplo anterior siempre disponemos de un párrafo con la información. Lo que cambia dependiendo de la operación condicional es el contenido de este.

Dentro de estos bloques, como ejemplo, estamos creando párrafos simples, pero podríamos incluir cualquier bloque HTML válido o incluso componentes personalizados un poco más complejos. No existe ningún tipo de limitación a la hora de establecer los bloques contenidos en una expresión condicional.

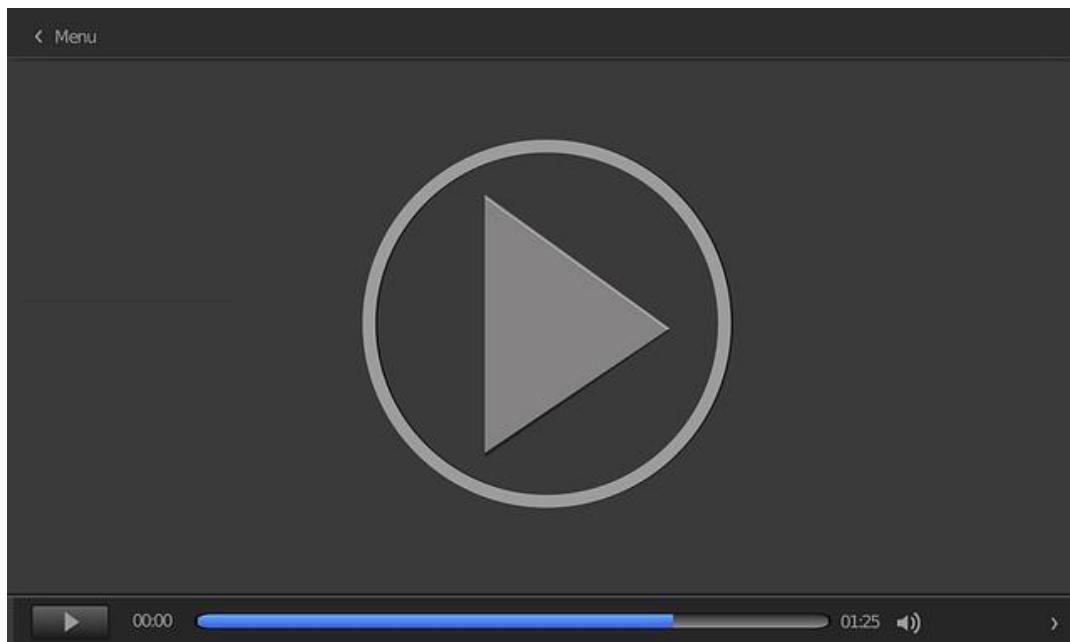
Completando el concepto de condicional semejante al que ya conocemos de Javascript, podemos incluir el uso de **@else if**. Vemos un ejemplo:

```
<section>
  @if (edad > 18) {
    <p>Eres mayor de edad</p>
```

# Tema 9. Control Flow

```
 } @else if (edad < 18) {  
     <p>Eres menor de edad</p>  
 } @else {  
     <p>Tiens justo 18 años</p>  
 }  
</section>
```

En este vídeo, *Bloque condicional @if*, veremos cómo se usa el bloque condicional @if.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=29449bca-1e8f-4533-b810-b1280170b791>

---

## 9.3. Bloque de repetición con @for

Abandonar el uso estático de los bloques HTML dentro de las plantillas de nuestros componentes es fundamental para agilizar la maquetación en nuestras aplicaciones. El bloque **@for** nos permite, a partir de una colección de elementos, repetir el bloque HTML que seleccionemos tantas veces como elementos tengamos en dicha colección.

Su uso es muy parecido al del **bucle for...of de Javascript**.

Analizaremos su uso a través de un ejemplo. Dentro de la clase del componente creamos e inicializamos un array con una serie de datos.

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

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

  estudiantes: { id: number, name: string }[] = [
    { id: 1, name: 'Luis García' },
    { id: 2, name: 'Inés Hernández' },
    { id: 3, name: 'Rocío Martín' },
    { id: 4, name: 'Ramón Romero' }
  ]
}
```

# Tema 9. Control Flow

La propiedad «estudiantes» la podemos utilizar dentro del HTML del componente para repetir cierto bloque de código:

```
<ul>
  @for (est of estudiantes; track est.id) {
    <li>{{ est.name }}</li>
  }
</ul>
```

Dentro del bucle estamos recorriendo el array «estudiantes» y en cada una de las iteraciones recibimos el valor correspondiente dentro de la variable **est**. Esta variable estará disponible únicamente dentro del ámbito del bucle.

Además, debemos incorporar de manera obligatoria un **valor para la propiedad track** del bloque @for. El valor que asignemos aquí debe ser un **identificador único** que le permita a Angular diferenciar cada una de las iteraciones del bloque. En caso de que el array se modifique, el framework será capaz de identificar qué elemento o elementos HTML son los que debe modificar, con lo cual dejará a los demás tal y como estaban.

## Variables asociadas al bloque @for

Además del elemento recuperado, en cada iteración del array disponemos de una serie de variables que nos ofrecen valores asociados a cada uno de los elementos de la colección que estamos recorriendo.

Los valores que podemos recuperar y utilizar son los siguientes:

- ▶ **\$count:** número de elementos dentro de la colección que iteramos.
- ▶ **\$index:** índice de la iteración actual.
- ▶ **\$first:** valor boolean que nos indica si nos encontramos o no en la primera iteración.
- ▶ **\$last:** valor boolean que nos indica si nos encontramos en la última iteración o no.

# Tema 9. Control Flow

- ▶ **\$even:** valor boolean que nos indica si la iteración actual es PAR.
- ▶ **\$odd:** valor boolean que nos indica si la iteración actual es IMPAR.

Podemos recuperar estos valores realizando una asignación sobre diferentes variables dentro del propio bloque FOR.

```
<ul>
  @for (estudiante of estudiantes; track estudiante.id; let cont = $count; let
    indice = $index; let primero = $first; let ultimo = $last; let par = $even;
    let impar = $odd) {
    <div>
      <p>{{ estudiante.name }}</p>
      <p>Número de elementos: {{ cont }}</p>
      <p>Índice actual: {{ indice }}</p>
      <p>¿Es la primera iteración? {{ primero }}</p>
      <p>¿Es la última iteración? {{ ultimo }}</p>
      <p>¿Es una iteración par? {{ par }}</p>
      <p>¿Es una iteración impar? {{ impar }}</p>
    </div>
  }
</ul>
```

En el ejemplo anterior hemos utilizado todas las variables para ver su funcionalidad, pero deberíamos incorporar solo aquellas que vayamos a utilizar en cada caso.

## Bloque @empty

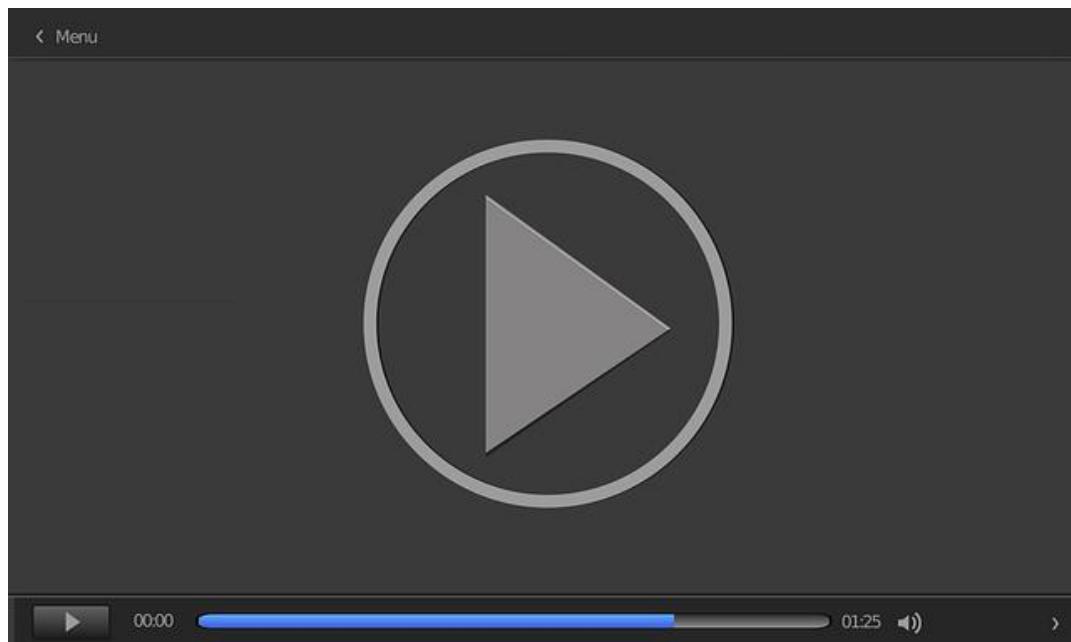
Una de las grandes mejoras asociadas al uso del bloque @for es la posibilidad de concretar un bloque de código HTML que se visualizará únicamente si el array con el que trabajamos está vacío. Esta acción la conseguimos a través del bloque @empty asociado a una iteración.

```
<ul>
  @for (estudiante of estudiantes; track estudiante.id;) {
    <li>{{ estudiante.name }}</li>
  }@empty {
    <li>No existen estudiantes</li>
  }
</ul>
```

# Tema 9. Control Flow

De esta manera evitamos tener que estar lanzando condicionales previos a la iteración del bucle para comprobar el número de elementos contenidos.

En este vídeo, *Bloque de repetición @for*, veremos cómo se usa el bloque de repetición @for.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=3529af72-5c7e-4904-9ceb-b12801750652>

---

## 9.4. Bloque condicional @switch

Mediante el uso del bloque @switch podemos evaluar una propiedad y, en función del valor de esta, mostrar un bloque u otro.

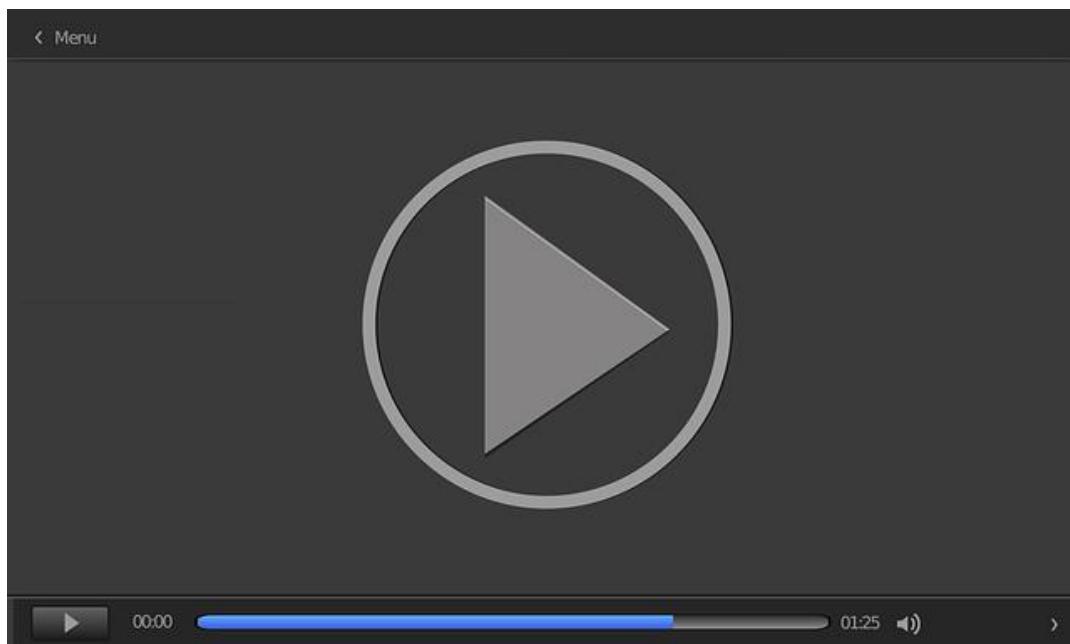
Se aproxima mucho al uso del condicional switch de Javascript. La gran diferencia con este es que no podemos evaluar condiciones complejas y nos tenemos que centrar en los valores que tengamos dentro de la clase del componente.

```
<div>
  @switch (role) {
    @case ('admin') {
      <p>Eres administrador. Todos los permisos</p>
    }
    @case ('moderador') {
      <p>Eres moderador. Permisos de escritura</p>
    }
    @case ('regular'){
      <p>Eres usuario. Permisos de lectura</p>
    }
    @default {
      <p>Role de usuario no admitido: {{role}}</p>
    }
  }
</div>
```

En este ejemplo estaríamos evaluando una variable llamada **role** de tipo string, definida dentro de la clase del componente. La modificación del valor de dicha variable provocaría que se viese dentro de la plantilla un párrafo u otro.

# Tema 9. Control Flow

En este vídeo, *Bloque condicional @switch*, veremos cómo se usa el bloque condicional *@switch*.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=9415fb23-913b-448a-9684-b1280179d434>

---

# Tema 10. Class y style. Directivas ngStyle y ngClass

## 10.1. Introducción y objetivos

La **gestión de estilos y clases** en Angular juega un papel crucial en la creación de interfaces de usuario dinámicas. Angular proporciona un conjunto de herramientas y directivas que les permite a los desarrolladores manipular estilos y clases de manera eficiente y condicional.

Esta **flexibilidad** es esencial para adaptar la presentación de los componentes a las interacciones de los usuarios, los cambios en los datos y las condiciones de la aplicación en tiempo real.

Como hemos visto en el tema anterior, Angular nos provee de unas herramientas que nos van a permitir cambiar la estructura de nuestra plantilla de forma fácil y cómoda, por lo que el cambio de estilos tiene que poder realizarse de la misma manera.

La capacidad de Angular para insertar y modificar dinámicamente estilos y clases directamente desde la lógica del componente ayuda a crear aplicaciones web que no solo son funcionales sino también estéticamente agradables.

Mediante el **uso de directivas específicas** como [style] , [ngStyle] , [class] , y [ngClass] , Angular simplifica la aplicación de estilos condicionales y la gestión de clases, lo que les permite a los desarrolladores controlar la apariencia de los elementos HTML de manera declarativa y reactiva.

Este enfoque no solo mejora la legibilidad y la mantenibilidad del código, sino que también abre las puertas a una personalización más profunda y dinámica de la interfaz de usuario.

# Tema 10. Class y style. Directivas ngStyle y ngClass

## 10.2. Inserción y modificación dinámica de estilos

Para modificar e insertar estilos de forma dinámica en tus componentes en Angular lo puedes hacer **mediante dos elementos** que, a pesar de ser muy parecidos, tienen algunas diferencias que los hacen útiles a la hora de trabajar con ellos. Una es la propiedad `[style]` y otra la directiva `[ngStyle]`. Digamos que puedes usar el enlace de propiedades `[style.nombre-propiedad]` para estilos específicos o la directiva `[ngStyle]` para múltiples estilos.

Veamos un ejemplo con `[style]`:

```
<!-- Cambia dinámicamente el color de fondo basado en una propiedad del componente -->
```

```
<div [style.backgroundColor]="isActive ? 'tomato' : 'transparent'">Contenido destacado</div>
```

```
<div [style.backgroundColor]="isActive ? 'tomato' : 'transparent'">Contenido destacado</div>
```

```
<!-- Podemos usar ambas sintaxis (dash-case o camelCase) -->
```

En este ejemplo, el color de fondo del `<div>` se cambia dinámicamente basándose en el valor de la propiedad `isActive` del componente.

De esta forma también podemos, por ejemplo, modificar la sintaxis para introducir las unidades de medida de los estilos aplicados.

```
export class PruebaComponent {  
  padding: number = 30;  
}
```

```
<p [style.padding.px]="padding">Párrafo 3</p>  
<!-- En este último ejemplo vemos cómo podemos agregar incluso las unidades  
que vamos a usar -->
```

Veamos ahora un ejemplo con `[ngStyle]` y cómo pueden modificarse varios estilos a la vez:

# Tema 10. Class y style. Directivas ngStyle y ngClass

```
<!-- Aplica múltiples estilos dinámicamente desde un objeto de estilos -->
```

```
<div [ngStyle]="{'font-size': fontSize + 'px', 'color': fontColor}">Texto estilizado</div>
```

Aquí, el tamaño de la fuente y el color del texto dentro del `<div>` se ajustan dinámicamente basándose en las propiedades `fontSize` y `fontColor` del componente.

Es **muy importante** que nos demos cuenta de que el primero sirve para modificar un estilo concreto y que podemos usar varias modificaciones `[style]` consecutivos para modificar de manera individual los estilos. Además de que la directiva `[ngStyle]` proporciona una forma de modificar estilos agrupados y en bloque, aunque, usando cierta lógica, podríamos hacer lo mismo con ambos elementos simplemente sabiendo manejar objetos de javascript de forma fluida.

Veamos algunos ejemplos con `[style]` en los que se modifican varios estilos a la vez:

```
export class PruebaComponent {  
  propiedades: any = {  
    backgroundColor: 'lightblue',  
    border: '1px solid black',  
    padding: '20px'  
  }  
}
```

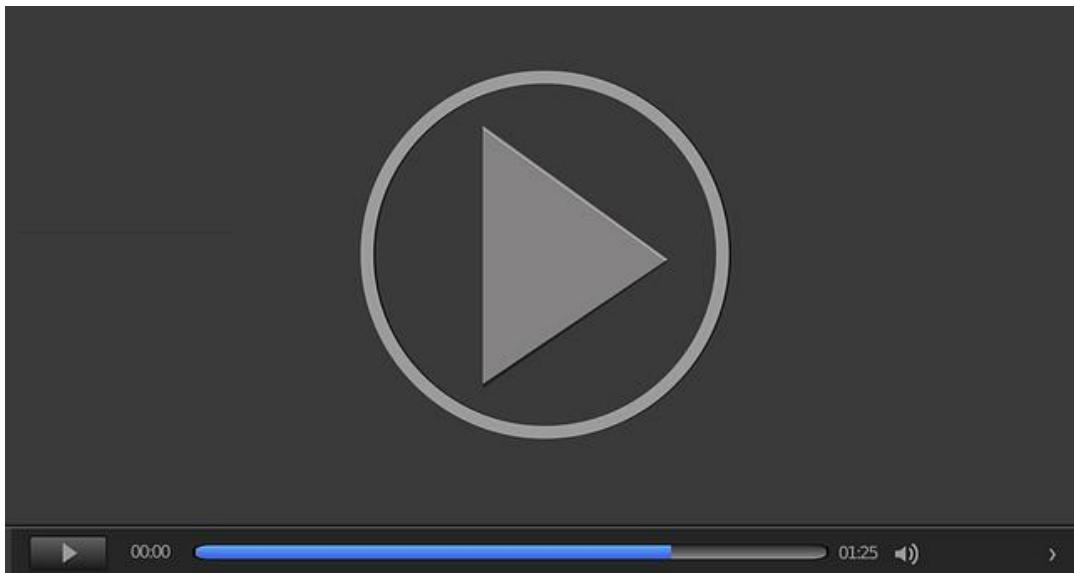
```
<div [style]="propiedades">  
  <p>Párrafo 1</p>  
</div>
```

Podemos modificar dinámicamente el objeto definido como propiedad de la clase para ver reflejados esos cambios en la interfaz.

En este vídeo, *Uso de propiedad style y ngStyle*, veremos cómo se usa la propiedad `style` y la directiva `ngStyle`.

< Menu

# Tema 10. Class y style. Directivas ngStyle y ngClass



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=83c605ea-e382-4367-96b7-b128017f162e>

---

# Tema 10. Class y style. Directivas ngStyle y ngClass

## 10.3. Inserción y modificación dinámica de clases

Para la gestión dinámica de clases, Angular proporciona [class.nombre-clase] para clases individuales y [ngClass] para múltiples clases.

Para poder aplicar una clase de manera condicional podemos enlazar los valores de la siguiente manera:

### Ejemplo de [class.nombre-clase]:

```
<article [class.oferta]="enOferta">Artículo a vender</article>
```

- ▶ En el caso anterior, cuando la propiedad `enOferta` sea «`true`» se aplica la clase `oferta` al elemento `article`.
- ▶ En caso contrario no se aplica la clase sobre el elemento.

Si necesitamos aplicar de manera dinámica varias clases sobre un elemento HTML lo podemos hacer a través de un **string**, **un array o un objeto**, lo vemos en el siguiente ejemplo:

### Archivo TS

```
export class PruebaComponent {  
  cadenaClases: string = 'article nuevo rojo';  
  arrayClases: string[] = ['article', 'antiguo', 'azul'];  
  objetoClases: any = {  
    article: true,  
    nuevo: false,  
    azul: true  
  }  
}
```

# Tema 10. Class y style. Directivas ngStyle y ngClass

## Archivo HTML

```
<article [class]="cadenaClases">Artículo 1</article>
<article [class]="arrayClases">Artículo 2</article>
<article [class]="objetoClases">Artículo 3</article>

<!--
  RESULTADO
<article class="article nuevo rojo"></article>
<article class="article antiguo azul"></article>
<article class="article azul"></article>
-->
```

Como las variables que asignan el valor de las clases para los artículos anteriores quedan definidas en el TS del componente, estas se pueden modificar en cualquier momento, con lo que se fuerza un cambio en las clases asignadas.

Ahora vamos a usar la directiva ngClass, en ella no solo podemos aplicar múltiples clases, sino que podemos crear objetos a los cuales aplicarles una condición por la cual activen o desactiven la propia clase dependiendo de si esta condición sea true o false.

### Ejemplo de [ngClass]:

```
<!-- Aplica múltiples clases dinámicamente desde un objeto, array o string -->
<div [ngClass]="{{'class1': true, 'class2': false}}>Contenido con múltiples clases</div>
```

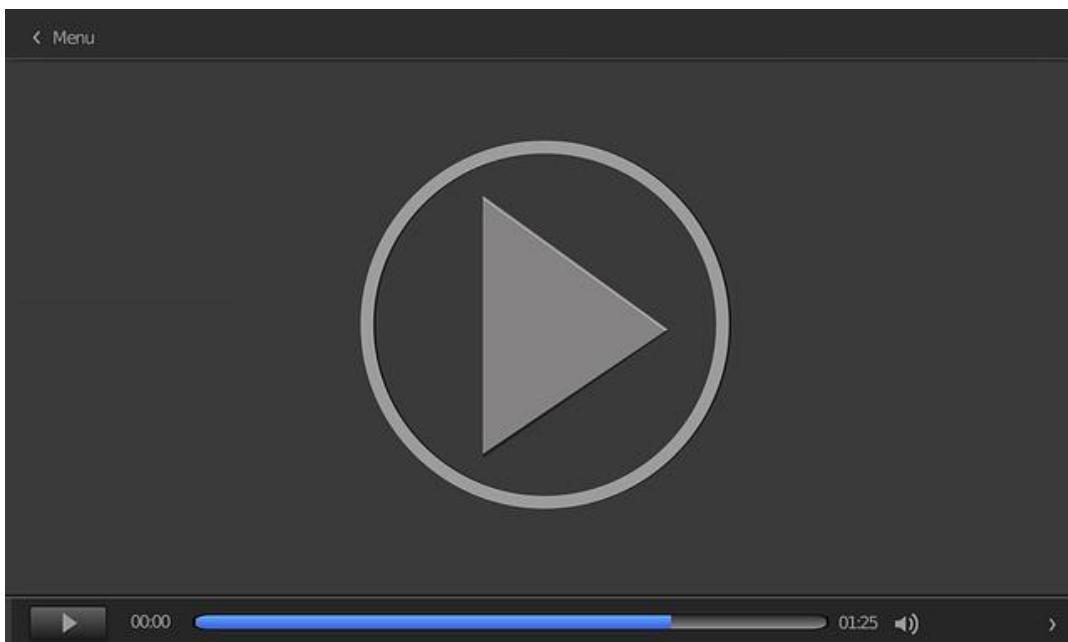
### O con un array o string:

```
<div [ngClass]="[class1, 'class2']">Contenido con múltiples clases</div>
```

Aquí, las clases class1 y class2 se aplican basándose en las condiciones condition1 y condition2 del componente. Si usas un array o string, simplemente aplicas las clases listadas sin basarte en condiciones.

# Tema 10. Class y style. Directivas ngStyle y ngClass

En este vídeo, *Uso de propiedad class y ngClass*, veremos cómo se usa la propiedad class y la directiva ngClass.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=ef671f31-119e-4e42-affa-b12900052ef7>

---

# Tema 10. Class y style. Directivas ngStyle y ngClass

## 10.4. Conclusiones

Angular nos brinda unas herramientas muy potentes para trabajar y modificar la apariencia de nuestras aplicaciones. Veamos un resumen rápido de las particularidades de todas ellas.

- ▶ **[style] y [ngStyle]:** permiten la inserción y la modificación dinámica de estilos en los componentes de Angular. Son adecuadas para los ajustes de estilos basados en las condiciones o propiedades del componente.
- ▶ **[class] y [ngClass]:** facilitan la gestión dinámica de las clases CSS en los componentes, lo que permite añadir o quitar clases basándose en la lógica de la aplicación.

# Tema 11. Inyección de dependencias

## 11.1. 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 encontramos 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.

# Tema 11. Inyección de dependencias

## 11.2. ¿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 externas 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.

# Tema 11. Inyección de dependencias

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. O través de la función inject(), la cual me permitiría crear un propiedad dentro de un componente para encapsular el servicio dentro.

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

Antes de nada, también hay que 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:

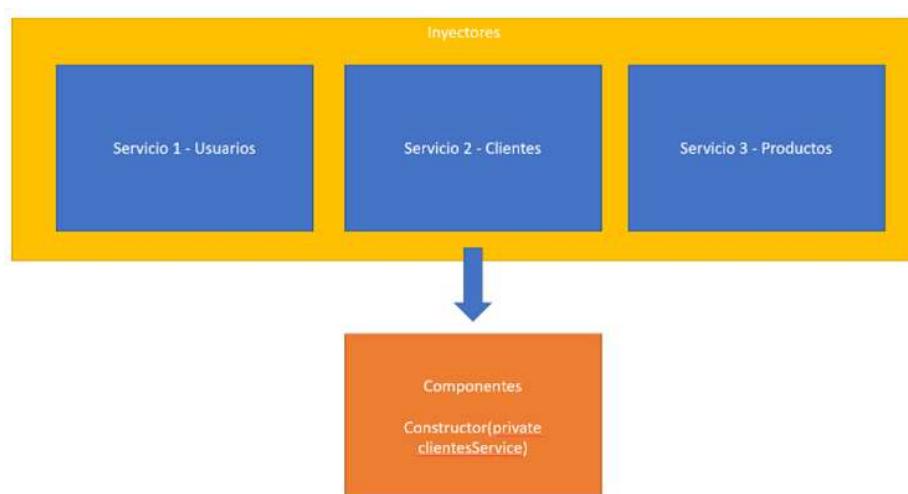


Figura 32. Explicación gráfica de inyección de dependencias. Fuente: elaboración propia.

# Tema 11. Inyección de dependencias

## 11.3. 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 capítulo pensado exclusivamente para la generación de servicios, vamos a usarlos 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 cuando generamos una directiva y un componente, tendremos que abrir nuestra terminal y situarnos dentro de la carpeta del 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 en la que se alojarán los diferentes servicios de nuestra aplicación

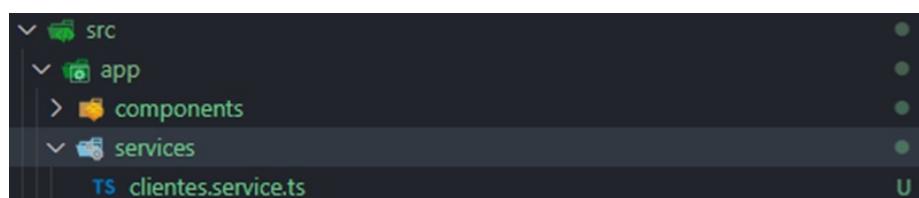


Figura 33. Árbol de carpetas tras la ejecución de la creación del servicio. Fuente: elaboración propia.

# Tema 11. Inyección de dependencias

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 el 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()`, la cual es un decorador que especifica que este servicio está disponible para el root de la aplicación, es decir, para todos los componentes 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 en **POO** se llama **GETTER**. De la siguiente forma.

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

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

# Tema 11. Inyección de dependencias

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

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

Esto que acabamos de crear es un inyectable 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 Dependency Injectable (DI) o inyección de dependencias.

Ahora, creamos un componente que consume el ClientesService para obtener datos. En el enfoque funcional de Angular 17 puedes inyectar servicios directamente en los componentes usando la función inject sin necesidad de especificarlos en el constructor.

Veamos cómo se lleva a cabo esta acción:

```
import { Component, OnInit } from '@angular/core';
import { ClientesService } from './clientes.service';
@Component({
  selector: 'app-mi-component',
  template: `<div>{{ data | json }}</div>`,
  standalone: true,
  imports: [],
})
export class MiComponent {
  data: any;
  private clientesService = inject(ClientesService);

  constructor() {}

  ngOnInit() {
    this.data = this.dataService.getData()
  }
}
```

# Tema 11. Inyección de dependencias

Dentro del componente hemos creado una propiedad para almacenar el array de colores y, por ejemplo, poder pintarlo dentro del HTML con un **@for**.

La **función inject** facilita la inyección de servicios directamente en las clases, lo cual es una característica poderosa del sistema de DI en Angular 17, lo que hace que el desarrollo sea más intuitivo y menos dependiente de la estructura de módulos.

Ahora podemos, por ejemplo, llenar el arrayColores cuando el componente está cargado, al llamar al servicio inyecto previamente y ejecutar 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, pueden 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. Es **flexible y escalable**.

Es importante saber que Angular maneja la inyección de dependencias desde prácticamente el inicio de su andadura como framework. Pues bien, la forma de injectar dependencias con la **función inject()** es relativamente moderna y responde al acercamiento por parte de Angular al uso no solo un modelo de programación en clase, sino también un enfoque más funcional.

Con lo que puede ser posible que nos encontremos código en producción donde la inyección de dependencias no se haga a través de la función inject(), sino pasando como parámetro el elemento por injectar en la función constructor de nuestro componente. Si bien esta forma está cada vez más en desuso, puede ser normal encontrar este tipo de formas de trabajo ya que las versiones de Angular las soportan.

# Tema 11. Inyección de dependencias

Veamos un ejemplo:

```
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()
  }
}
```

Como puedes observar en este fragmento, la inyección de `clientesService` se hace como parámetro privado dentro del método `constructor` de la clase. Este método está más orientado a la programación orientada a objetos en la que está basado principalmente Angular y, aunque si bien desde Angular se está promoviendo un poco la versión de la función `inject()`, es totalmente funcional y está disponible en todas las versiones. Puedes usar la que mejor te venga o mejor entiendas.

# Tema 12. Formularios y validaciones

## 12.1. Introducción y objetivos

Las dos acciones fundamentales que cumple cualquier aplicación web son:

- ▶ Mostrarle 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 Binding, 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 cuanto 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.

# Tema 12. Formularios y validaciones

- ▶ 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. Como hemos comentado en varios temas ya, Angular 17 está cambiando de la programación modular a la de desarrollo de componentes desacoplados. Eso no quiere decir que se abandone la creación de módulos, por eso hay algunas librerías, como es el caso de los formularios, que se encapsulan en módulos para luego importarlos dentro de los componentes que lo demanden, como venimos contando hasta este tema.

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.

# Tema 12. Formularios y validaciones

## 12.2. Formulario de tipo Template. FormsModule

Como ya hemos dicho en la introducción, se trata de módulos que se importan dentro de nuestro del componente en el que vayamos a usarlo dentro de su función decoradora `@Component()`.

Con lo que el primer paso en la creación de formularios, para poder usar esta potente característica dentro de nuestros componentes, es importar el módulo **FormsModule** dentro del TS del propio componente.

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 un componente y cómo se importa dicho módulo:

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

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

En las plantillas en las que 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.

# Tema 12. Formularios y validaciones

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

```
<h1>Formulario</h1>
<form novalidate #nuevaPersonaForm="ngForm"
(ngSubmit)="onSubmit(nuevaPersonaForm)">
  <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-template-form',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './template-form.component.html',
  styleUrls: ['./template-form.component.css']
})
export class TemplateFormComponent {

  onSubmit(pForm: NgForm) {
    console.log(pForm.value)
  }
}
```

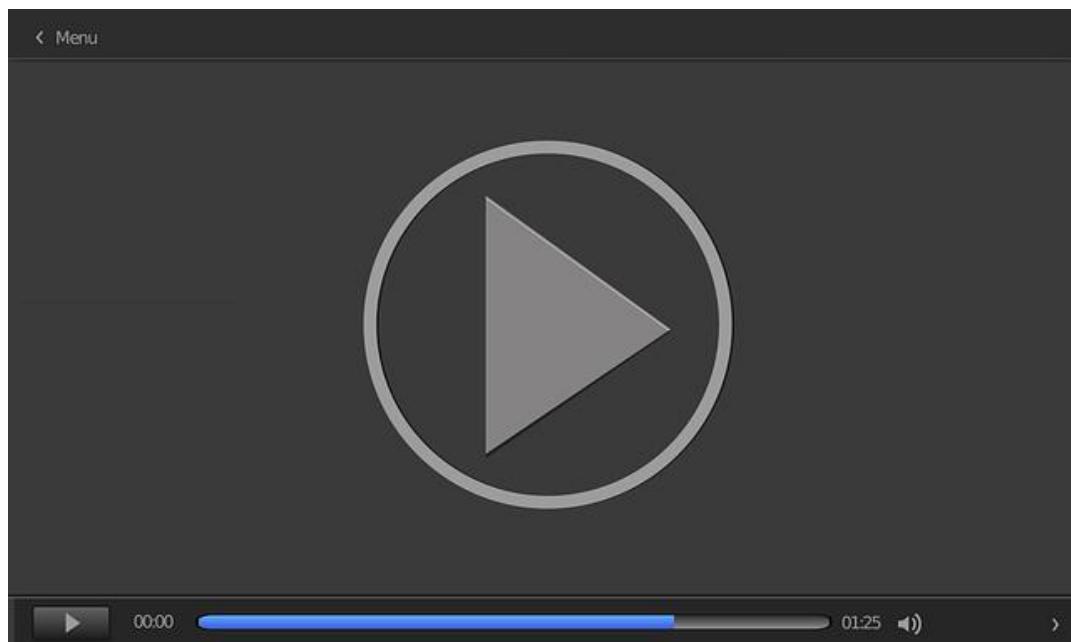
`pForm` representa un objeto JSON que me devuelve un objeto cuyas claves son los **name** que le hemos puesto a cada uno de los campos de nuestro formulario.

# Tema 12. Formularios y validaciones

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

Este tipo de formulario es muy sencillo y ofrece una forma fácil y rápida 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:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=b674f8d4-50be-4ea1-91df-b129015a77ec>

---

# Tema 12. Formularios y validaciones

## 12.3. 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 { Component } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-model',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './model.component.html',
  styleUrls: ['./model.component.css']
})
export class ModelComponent {
```

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

# Tema 12. Formularios y validaciones

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-model',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './model.component.html',
  styleUrls: ['./model.component.css'
})
export class ModelComponent {

  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`.

# Tema 12. Formularios y validaciones

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.

# Tema 12. Formularios y validaciones

## 12.4. 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 en el que el campo nombre es obligatorio y cuya longitud no puede ser menor que dos caracteres.

En este fragmento de código se muestra el FormControl del formulario anterior donde nos centramos en los validadores por defecto que tienen ReactiveFormsModule. Todas aplicaciones sobre el campo nombre.

```
nombre: new FormControl('nombre por defecto', [  
  Validators.required,  
  Validators.minLength(2)  
]);
```

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`.

# Tema 12. Formularios y validaciones

Es tarea del desarrollador asegurarse de 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.

A parte 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.

# Tema 12. Formularios y validaciones

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 que 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 de 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>
```

Los validadores más habituales incluidos con Angular son los siguientes:

- ▶ **Required:** determina si un campo es obligatorio.
- ▶ **minLength:** determina el tamaño mínimo de caracteres que debe tener el valor para ser válido.
- ▶ **maxLength:** determina el tamaño máximo de caracteres que debe tener el valor para ser válido.

# Tema 12. Formularios y validaciones

- ▶ **email:** evalúa si el string escrito en un campo concreto tiene formato de email.
- ▶ **pattern:** aplica una expresión regular para saber si el campo que validamos es correcto o no.

## Validadores Custom

Para crear nuestros propios validadores necesitamos generar un método que recibe como parámetro el control por validar y devuelve `null` si está correcto o un objeto que especifique 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.

# Tema 12. Formularios y validaciones

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">  
@if(form.controls.nombre.errors){  
<p>La edad es incorrecta</p>  
}
```

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">  
@if(form.controls.nombre.errors?.required){  
<p>El nombre es un campo requerido</p>  
}
```

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

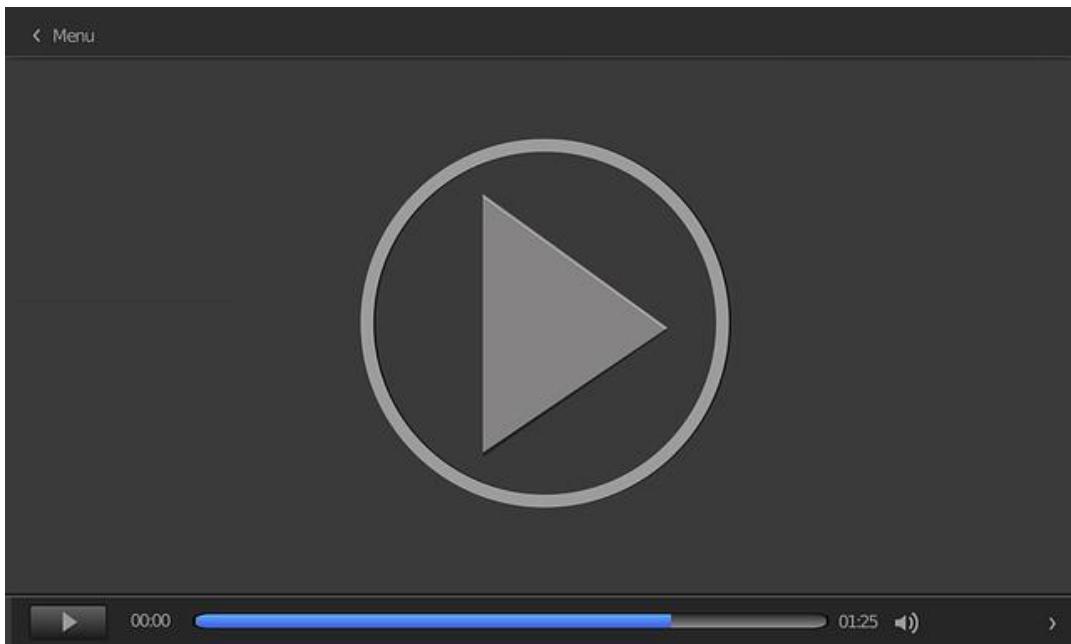
# Tema 12. Formularios y validaciones

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">
@if(form.controls.edad.errors){
<p>{{ form.controls.edad.errors?.message }}</p>
}
```

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.

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



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=2d0b627d-9722-403d-baae-b1290181dac3>

---

# Tema 13. Rutas y navegación

## 13.1. 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 la navegación web.

Siempre debemos tener en cuenta que estamos trabajando con aplicaciones de tipo Single Page Application (SPA), 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 los 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 los 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**.

# Tema 13. Rutas y navegación

El navegador en el que 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.

# Tema 13. Rutas y navegación

## 13.2. Configuración básica. Módulo de rutas

Para poder manejar las rutas de nuestra aplicación Angular, disponemos de un archivo de configuración llamado **app.config.ts**.

Recordamos que en la nueva versión de Angular se abandona el concepto de módulo para dar más libertad y desacoplar los componentes, con lo que este fichero viene a **sustituir al módulo principal** de la aplicación que había en las versiones anteriores.

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes)]
};
```

En dicho fichero se especifica dónde tenemos definido el array en el que concretamos todas las rutas de nuestra aplicación **app.routes.ts**.

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

Veamos una implementación sencilla de dicho fichero de configuración **app.routes.ts**.

```
import { Routes } from '@angular/router';
import { InfoComponent } from './components/info/info.component';
import { AboutComponent } from './components/about/about.component';
import { HomeComponent } from './components/home/home.component';

export const routes: Routes = [
  { path: 'inicio', component: HomeComponent },
  { path: 'informacion', component: InfoComponent },
  { path: 'sobre-mi', component: AboutComponent }
];
```

# Tema 13. Rutas y navegación

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.

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.

# Tema 13. Rutas y navegación

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.

```
export 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 especificar qué pasa si ninguno de los objetos definidos es capaz de responder a la ruta del navegador.

Podemos solucionarlo del siguiente modo:

```
Export const routes: Routes = [
  {path: "", redirectTo: 'home', pathMatch: 'full'},
  {path: 'home', component: HomeComponent},
  {path: 'hola', component: HolaComponent},
  {path: 'saludo', redirectTo: 'hola'},
  {path: "**", component: c404Component}
];
```

Para poder navegar entre las diferentes rutas que conforman nuestra aplicación no podemos usar el método tradicional (`a href`) .

# Tema 13. Rutas y navegación

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.

En Angular disponemos de diferentes formas para poder especificar la navegación dentro de nuestras aplicaciones.

La primera de ellas sería **inyectar el servicio Router** dentro de nuestros componentes.

Veamos 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, inject } from '@angular/core';
import { Router, RouterOutlet } from '@angular/router';

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

  gotoPage(path: any){
    this.router.navigate([path])
  }
}
```

El parámetro que se pasa a la función **navigate** es un poco extraño.

# Tema 13. Rutas y navegación

Se trata de un array que se **corresponde** con cada una de las **partes de la URL** a la que queremos navegar.

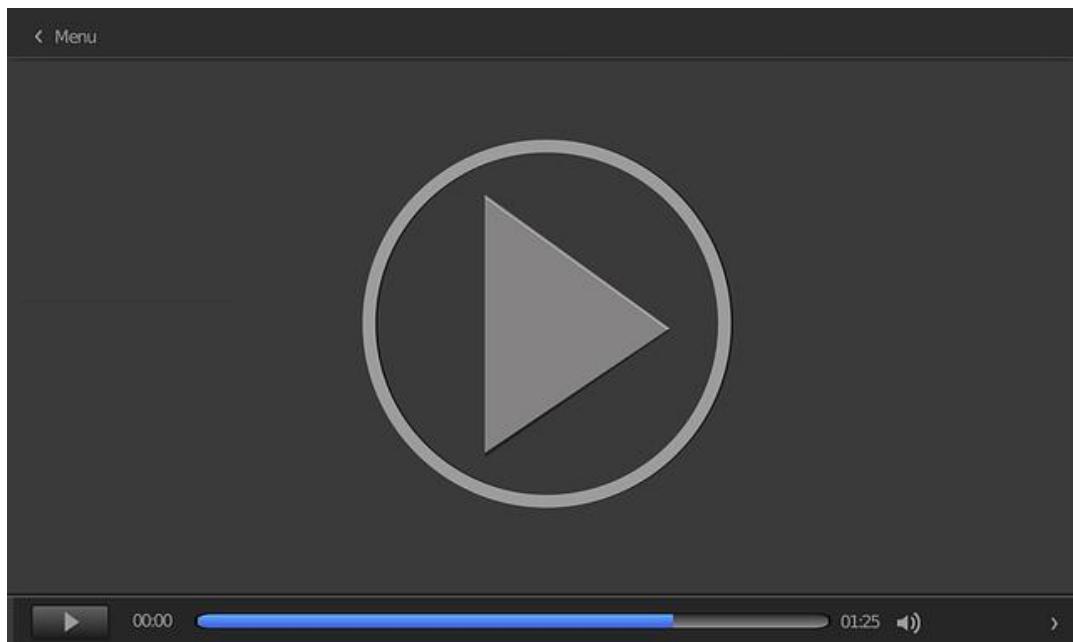
Por ejemplo, si tenemos definida la siguiente ruta:

```
{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:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=fde99b65-124f-48b3-90f6-b1290189a63e>

---

# Tema 13. Rutas y navegación

## 13.3. 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`. Para poder usarla tenemos que importarla en el componente en el que la vamos a usar, porque si no se hace, el HTML no reconocerá esta directiva.

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

Como ayuda en el trabajo con nuestros enlaces, disponemos de la directiva **routerLinkActive**, la cual, como hemos visto en el ejemplo de código anterior, también tenemos que importarla dentro del componente donde la vamos a usar.

# Tema 13. Rutas y navegación

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:

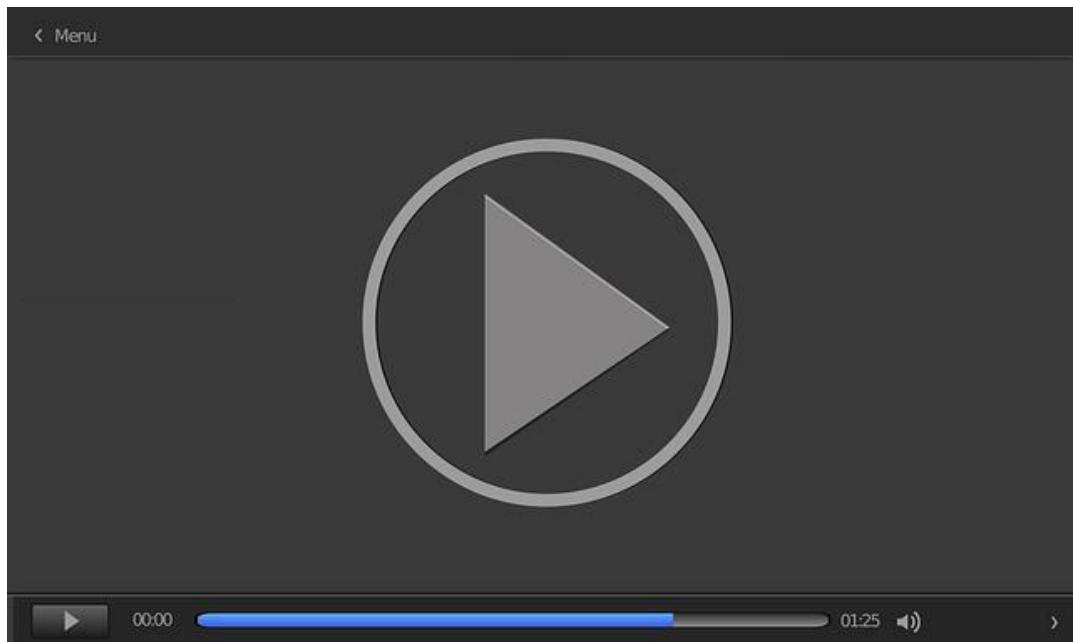
```
{ path: 'hola/:saludo', component: HolaComponent }
```

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.

# Tema 13. Rutas y navegación

En este vídeo, *Pasar parámetros de ruta*, vemos cómo se pasan los parámetros por la ruta.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=3e088042-b9c6-4b01-b9b5-b12f0170443b>

---

# Tema 13. Rutas y navegación

## 13.4. 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, inject } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

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

  activatedRoute = inject(ActivatedRoute);

  ngOnInit() {
    this.activatedRoute.params.subscribe(params => {
      // Dentro del objeto params tenemos una clave por cada una de las partes dinámicas
      // definidas en la ruta. Es decir, tantas claves como identificadores con dos puntos hayamos
      // especificado.
      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:

# Tema 13. Rutas y navegación

- ▶ /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 ampliación sencilla del código.

# Tema 13. Rutas y navegación

## 13.5. 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:

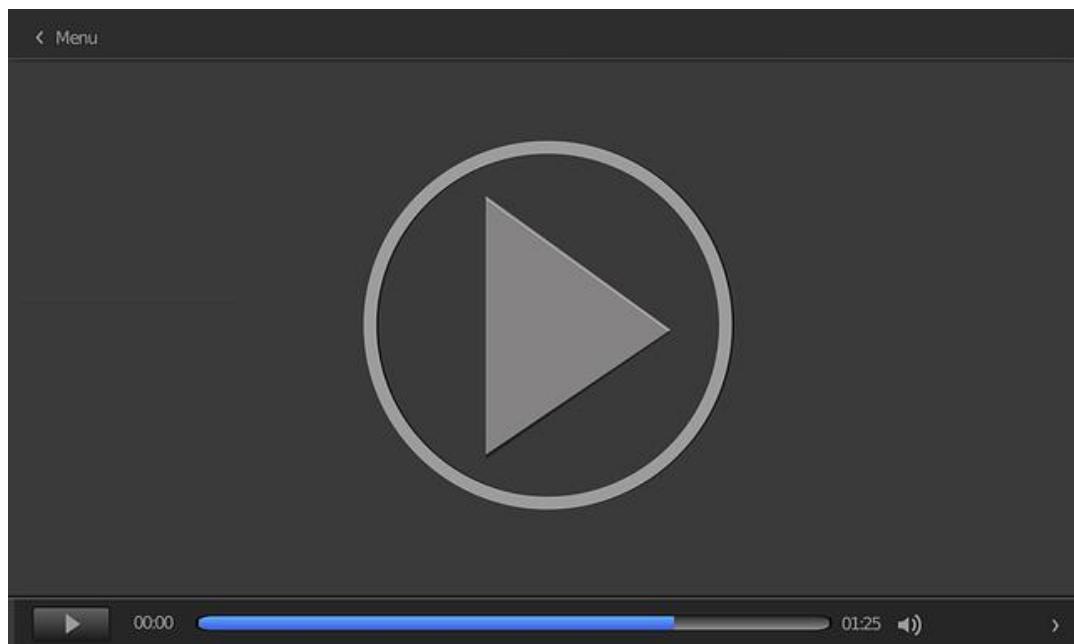
```
import { Component, inject } from '@angular/core';  
import { ActivatedRoute } from '@angular/router';  
  
@Component({  
  selector: 'app-autor',  
  standalone: true,  
  imports: [],  
  templateUrl: './autor.component.html',  
  styleUrls: ['./autor.component.css']  
})  
export class AutorComponent {  
  
  activatedRoute = inject(ActivatedRoute);  
  
  ngOnInit() {  
    this.activatedRoute.parent.params.subscribe(params => {  
      // para recoger el valor de aleatorio del padre usamos delante de params  
      // la palabra parent para referirnos a que el parámetro optativo es del padre.  
    })  
  }  
}
```

# Tema 13. Rutas y navegación

```
        console.log(params);
    });
}

}
```

A continuación, puedes ver el siguiente vídeo, *Creación de rutas hijas*.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=334b1516-f14c-4009-b6de-b12f017e1302>

---

# Tema 13. Rutas y navegación

## 13.6. LocalStorage y guards

En las aplicaciones tradicionales, cuando se accede a una ruta no permitida el servidor devuelve el estado 403 lo que le indica al cliente el error. Otra opción es que se redireccione al usuario hacia otra página que informe del posible error de acceso.

En nuestras aplicaciones de tipo SPA podemos lograr esta funcionalidad gracias a las **router guards**.

Con el uso de esta metodología podemos prevenir el acceso del usuario a zonas específicas de nuestra aplicación.

Existen varios **tipos de guards**:

- ▶ **CanActivate:** para decidir si una ruta puede ser activada.
- ▶ **CanActivateChild:** para decidir si las rutas hijas de una ruta pueden ser activadas.
- ▶ **CanDeactivate:** para decidir si se puede abandonar una ruta.
- ▶ **CanLoad:** para decidir si un módulo puede ser cargado de manera perezosa.
- ▶ **Resolve:** para resolver datos antes de activar una ruta.

Para una ruta podemos implementar el número de guards que necesitemos.

Puedes seleccionar una o varias de estas interfaces al implementar tu guard. Por ejemplo, si eliges **CanActivate**, Angular CLI generará un guard con el esqueleto necesario para implementar la lógica que determina si una ruta puede ser activada o no.

Cuando **generas un nuevo guard** en Angular, el sistema te proporcionará un archivo inicial que ya implementa una interfaz base. Esta interfaz sirve como punto de partida para que tú, como desarrollador, introduzcas la lógica necesaria para tu

# Tema 13. Rutas y navegación

aplicación. Por ejemplo, si estás creando un guard para verificar la autenticidad de un usuario, aquí es donde establecerías los criterios para determinar si el usuario puede o no acceder a determinadas áreas de tu aplicación.

La implementación de guards es un aspecto clave en el desarrollo de aplicaciones con Angular, ya que proporciona un mecanismo para **gestionar y restringir el acceso** a distintas secciones de la aplicación. Esto se hace comúnmente en función de ciertos criterios o reglas, tales como si un usuario ha sido autenticado o si tiene los permisos adecuados.

Con la versión 17 de Angular, que introduce una aproximación más autónoma o «*standalone*», es posible establecer guards que **operen de manera independiente**, sin la necesidad de estar vinculados a módulos específicos para su funcionamiento.

Esto simplifica la configuración y el registro de los guards, lo que los hace **más versátiles y fáciles de manejar**. Ahora te guiaremos paso a paso en la creación de un guard del tipo CanActivate, el cual es utilizado para decidir si una ruta particular puede ser activada en el contexto de una aplicación Angular que opera en modo *standalone*.

Para crear un guard en un proyecto Angular utilizando Angular CLI, puedes usar el siguiente comando:

```
ng generate guard
```

O su forma abreviada:

```
ng g guard
```

En el que **<nombre-del-guard>** es el nombre que deseas darle a tu guard. Por ejemplo, si quieres crear un guard llamado auth, el comando sería:

```
ng g guard auth
```

# Tema 13. Rutas y navegación

Al ejecutar este comando, Angular CLI te hará una serie de preguntas para determinar qué tipo de guard quieres crear.

```
○ > ng g guard guards/login --functional  
? Which type of guard would you like to create?  
➤● CanActivate  
○ CanActivateChild  
○ CanDeactivate  
○ CanMatch
```

Figura 34. Preguntas que te lanza la terminal. Fuente: elaboración propia.

Después de haber ejecutado el comando se genera el guard en modo funcional, esto hace que no sea una inyección de dependencia como en las versiones anteriores de Angular.

```
// auth.guard.ts  
import { CanActivate, Router } from '@angular/router';  
import { inject } from '@angular/core';  
  
export class AuthGuard implements CanActivate {  
  private router = inject(Router);  
  
  canActivate(): boolean {  
    const isAuthenticated = this.checkIfUserIsAuthenticated();  
    if (!isAuthenticated) {  
      this.router.navigate(['/login']);  
      return false;  
    }  
    return true;  
  }  
  
  private checkIfUserIsAuthenticated(): boolean {  
    // Implementación ficticia de comprobación de autenticación  
    return !!localStorage.getItem('authToken');  
  }  
}
```

# Tema 13. Rutas y navegación

## Usar el guard en la configuración de rutas

A continuación, te muestro cómo usar este guard en la configuración de las rutas de tu aplicación Angular en modo *standalone*. En Angular 17 puedes declarar rutas en un archivo específico como hemos explicado al principio de este tema.

```
// app.routes.ts
import { Route } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { LoginComponent } from './login/login.component';
import { AuthGuard } from './auth.guard';

export const APP_ROUTES: Route[] = [
  { path: '', component: HomeComponent, canActivate: [AuthGuard] },
  { path: 'login', component: LoginComponent },
];
```

De esta forma la ruta raíz te deja entrar en el componente HomeComponent siempre que el guard de autorización devuelva true, si devuelve false se hará una redirección normalmente a la pantalla de login.

## 14.1. Introducción y objetivos

En el tema «Inyección de dependencias» vimos la importancia que tenía para Angular que ciertos elementos fueran inyectados dentro de los componentes que se iban a usar.

Introdujimos 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 sino para hacer peticiones HTTP externas a servicios API-REST con la intención de pedir datos para posteriormente pasárselos a los diferentes componentes que maneja 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 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.

## 14.2. ¿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.

En breve resumen, 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 forman 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 componentes a través de la función `inject()` o pasándolos como parámetros privados dentro del método constructor de la clase del componente.

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.

# Tema 14. Servicios y HTTP

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, estarán disponibles 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 otros tipos de frameworks o desarrollos.

Se trata de **una clase** que contiene funcionalidad que nosotros podemos exportar dentro de nuestra aplicación.

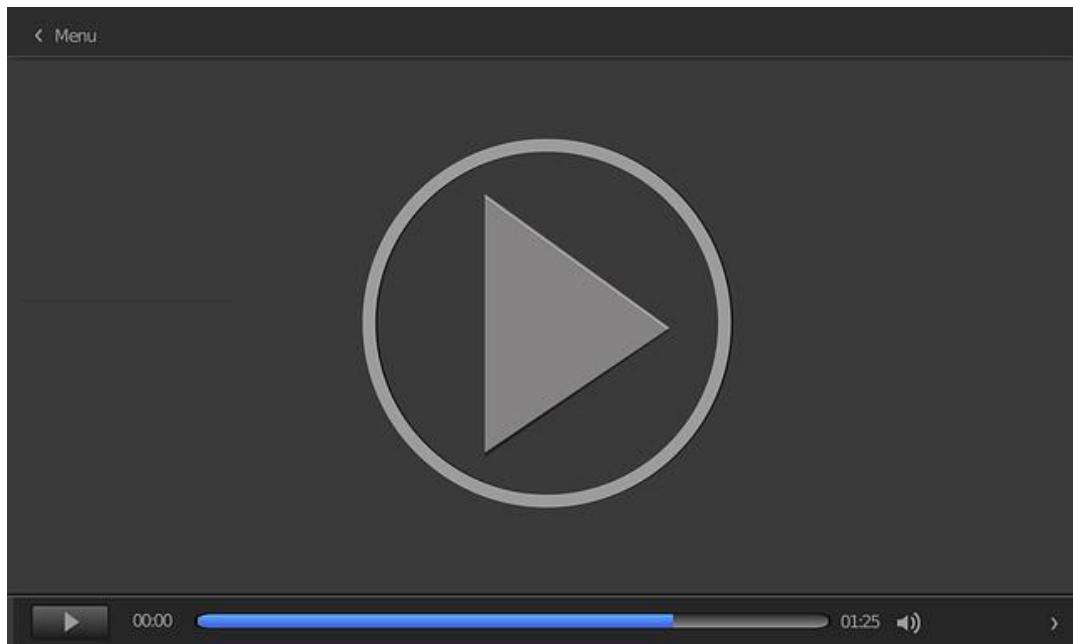
No necesitamos ningún código específico de Angular para trabajar con nuestros servicios, son clases normales.

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
```

# Tema 14. Servicios y HTTP

A continuación, puedes ver el siguiente vídeo, *Creación de servicios*.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=c0926934-b37a-424f-b439-b12f018a652f>

---

## 14.3. 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 recibirlas dentro de nuestra aplicación.

También para compartir métodos útiles que serán usados por la mayoría de los componentes de nuestra aplicación.

Para compartir métodos útiles que serán usados por la mayoría de los 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 vamos a necesitar que dicha información se represente dentro de nuestra aplicación de varias maneras diferentes.

# Tema 14. Servicios y HTTP

```
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).

# Tema 14. Servicios y HTTP

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**

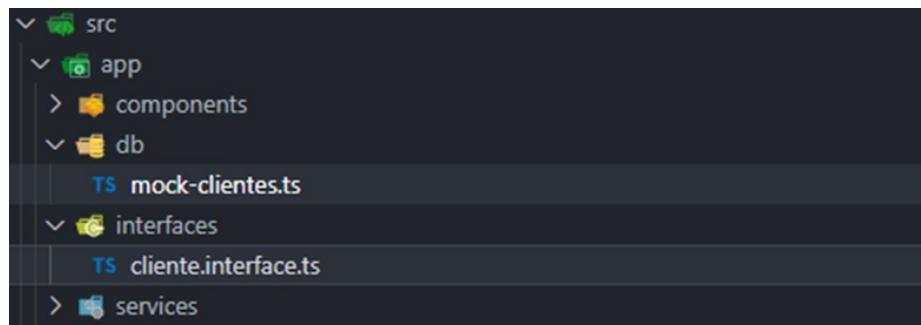


Figura 35. Árbol de carpetas. Fuente: elaboración propia.

# Tema 14. Servicios y HTTP

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 {
  private arrClientes: Cliente[] = Clientes
  constructor() { }

  getAll(): Cliente[]{
    return arrClientes
  }
}
```

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.

# Tema 14. Servicios y HTTP

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 injectado dentro del componente, llamando al método que hemos implementado antes `getAll()`.

```
import { Component, inject } from '@angular/core';
import { ClientesService } from './services/clientes.service';
import { Cliente } from './interfaces/cliente.interface';

@Component({
  selector: 'app-clientes-list',
  standalone: true,
  imports: [],
  templateUrl: './clientes-list.component.html',
  styleUrls: ['./clientes-list.component.css']
})
export class ClientesListComponent {
  //opcion 1: Metodo inject
  clientesServices = inject(ClientesService);
  alumnos: Cliente[] = [];
  //opcion 2: parametro dentro de la funcion constructor
  // constructor(private clientesServices: clientesService) {}

  ngOnInit() {
    this.clientes = this.clientesServices.getAll()
    //console.log(this.clientes)
  }
}
```

# Tema 14. Servicios y HTTP

Una vez que tengamos la propiedad array de clientes llena podremos ir a nuestra plantilla HTML y pintar nuestro resultado a través del control flow con el uso del **bucle @for** (la maquetación pueden realizarla como vosotros queráis).

```
<ul>
@for(cliente of clientes) {
  <li>{{cliente.nombre}} {{cliente.apellidos}}</li>
}
</ul>
```

Esta sintaxis implica que se ha creado, dentro del componente, una nueva propiedad para almacenar el resultado retornado por la función inject() que carga el ClientesServices.

Como podemos observar, el componente es opaco a la obtención de los datos (mock, localstorage, server...).

## 14.4. HttpClientModule

Una de las capacidades básicas que deben tener nuestras aplicaciones web es la de **poder conectarse con HTTP API externas**. A partir de estas conexiones podremos recuperar la mayoría de los datos para 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**, lo que facilita 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 como, por ejemplo, la cantidad de facilidades para realizar pruebas en la aplicación.

# Tema 14. Servicios y HTTP

Para poder **utilizar el servicio HttpClient**, debemos incluir el provider dentro del fichero app.config.ts:

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes),
    provideHttpClient()
  ]
};
```

Una vez incluido en el app.config.ts, 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 {

  httpClient= inject(HttpClient)

}
```

## 14.5. Peticiones GET, POST, PUT, DELETE

El protocolo HTTP dispone de una serie de verbos para cada una de las URL, los cuales nos permiten especificar qué acciones vamos a llevar a cabo.

Los verbos HTTP más utilizados son get, put, post y delete. Forman parte de lo que se conoce como **CRUD de peticiones**, con las cuales 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.

## 14.6. 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:

<a href="#">/posts</a>	100 posts
<a href="#">/comments</a>	500 comments
<a href="#">/albums</a>	100 albums
<a href="#">/photos</a>	5000 photos
<a href="#">/todos</a>	200 todos
<a href="#">/users</a>	10 users

Figura 36. JSONPlaceholder Resources. Fuente: elaboración propia.

# Tema 14. Servicios y HTTP

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	<a href="#">/posts</a>
GET	<a href="#">/posts/1</a>
GET	<a href="#">/posts/1/comments</a>
GET	<a href="#">/comments?postId=1</a>
POST	<a href="#">/posts</a>
PUT	<a href="#">/posts/1</a>
PATCH	<a href="#">/posts/1</a>
DELETE	<a href="#">/posts/1</a>

Figura 37. JSON PLACEHOLDER. Captura de pantalla. Fuente: elaboración propia.

# Tema 14. Servicios y HTTP

## 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 creemos 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 {

  httpClient = inject(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.

# Tema 14. Servicios y HTTP

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 cómo 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';
import { Cliente } from './interfaces/cliente.interface';

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

  //inicializamos un array de tipo cliente vacio
  clientes: Cliente[] = [];
  clientesServices = inject(ClientesService)

  //dentro del metodo de carga del componente llamamos al servicios y llenamos
  el array de clientes
  ngOnInit() {
    this.clientesServices.peticionGetObservable().subscribe(data => {
      this.clientes = data;
    })
  }
}
```

# Tema 14. Servicios y HTTP

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 del HTML de nuestro componente a través de un bucle **@for**, 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 necesitemos.

## 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 {

  httpClient = inject(HttpClient)

  peticionGetPromesa(): Promise<any[]> {
    return lastValueFrom(this.httpClient.get<any[]>('https://jsonplaceholder.typicode.com/users'))
  }
}
```

# Tema 14. Servicios y HTTP

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 `peticionGetPromesa()`. 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';
import { Cliente } from './interfaces/cliente.interface';

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

export class PrimerComponentesComponent {

  //inicializamos un array de tipo cliente vacio
  clientes: Cliente[] = [];
  clientesServices = inject(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 gestionarlas es de esta forma.

# Tema 14. Servicios y HTTP

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 bucle @for, 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 sobre 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 le vamos a enviar la petición y debemos concretar que nos devuelve el método y que 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 = {
        param1: 'value1',
        param2: 'value2'
    }
    return lastValueFrom(this.httpClient.post<any>('https://jsonplaceholder.typicode.com/posts',body))
}
```

# Tema 14. Servicios y HTTP

De la misma manera podemos lanzar peticiones de tipo PUT a través del **método PUT**, el cual se usa de la misma manera que POST.

```
peticionPut(): Promise<any> {
  const body = { param1: 'value1' };
  return lastValueFrom(this.httpClient.put<any>('https://jsonplaceholder.typicode.com/post', body))
}
```

Si necesitamos lanzar una **petición DELETE**, podemos hacerlo a través del método homónimo.

Cualquiera de los métodos anteriores, cuando los necesitemos ejecutar los tratamos con los métodos **then y catch** o con **async-await**.

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 adjunta** a la petición enviada al servidor. Enviamos elementos como la IP desde la cual se manda la petición o el navegador que utilizamos.

Determinadas API requieren que especifiquemos ciertas cabeceras personalizadas.

Podemos implementar las cabeceras de la petición con la siguiente estructura:

# Tema 14. Servicios y HTTP

Aquí ponemos un ejemplo de una petición desde el servicio que necesita un token de validación para que el servidor acepte la 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 implementa un objeto **HttpHeaders** que previamente hay que haber importado en nuestro servicio.

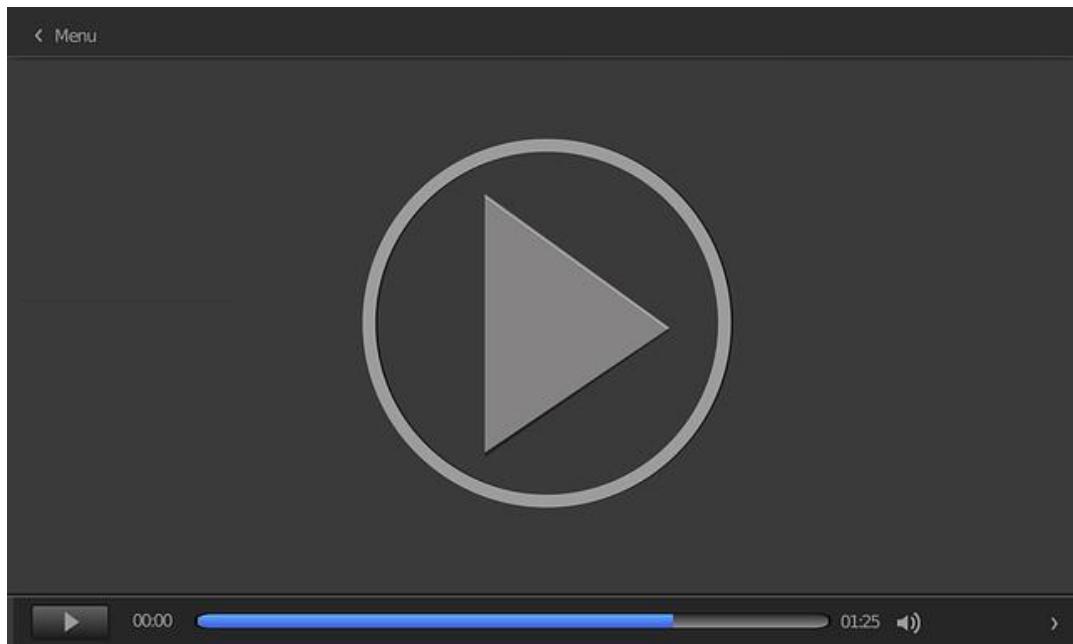
```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { lastValueFrom } from 'rxjs';
```

Esta es la forma de hacer peticiones a API externas en Angular y los diferentes métodos y casuísticas que hay que tener en cuenta.

Como con 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 una dificultad.

# Tema 14. Servicios y HTTP

A continuación, puedes ver el vídeo *Peticiones HTTP*.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=b2ef2fb8-e667-420c-afc3-b13000108748>

---

## 14.7 Interceptors

Un interceptor en Angular es una clase que **implementa la interfaz HttpInterceptor** y se utiliza para interceptar y modificar peticiones HTTP. Los interceptores son parte del módulo **HttpClientModule** y ofrecen una manera gestionar aspectos globales de la comunicación HTTP, tales como la adición de cabeceras (headers) a todas las peticiones, inclusión de tokens en las peticiones, logging, caché de respuestas, etc.

### ¿Qué hace un interceptor?

Un interceptor puede:

- ▶ **Modificar peticiones HTTP:** puede alterar las peticiones antes de que sean enviadas al servidor. Esto incluye modificar headers, cambiar el cuerpo de la petición, añadir tokens de autenticación, etc.
- ▶ **Modificar respuestas HTTP:** antes de que las respuestas sean entregadas a quien las solicita, puede interceptarlas para modificarlas, lo cual es útil para manejar formatos de datos, actualizar cachés, entre otros.

Para crear un interceptor en un proyecto Angular utilizando Angular CLI, puedes usar el siguiente comando:

```
ng generate interceptor
```

O su forma abreviada:

```
ng g interceptor <nombre-del-interceptor>
```

# Tema 14. Servicios y HTTP

Donde <nombre-del-interceptor> es el nombre que deseas darle a tu interceptor. Por ejemplo, si quieres crear un interceptor llamado auth , el comando sería:

```
ng g interceptor auth
```

Este comando generará dos archivos en tu proyecto: **auth.interceptor.ts** y **auth.interceptor.spec.ts**. El primero contiene el archivo con la lógica para el interceptor y el segundo es el fichero para las pruebas y los test unitarios.

El **archivo** que se crea al **definir un interceptor** en Angular incluirá una estructura elemental que cumple con la interfaz HttpInterceptor. Esta base te servirá como punto de partida para incorporar la funcionalidad específica que requieras, como puede ser la interceptación de las solicitudes HTTP para añadir un token de autenticación, entre otras acciones.

Con la llegada de Angular 17 y su capacidad para funcionar en modo *standalone*, ahora es posible desarrollar componentes y servicios **sin la obligación de utilizar NgModules**. Esto representa una evolución significativa en la forma de estructurar proyectos en Angular, al permitir que declares y exportes componentes, servicios, guards e interceptores de manera directa. Esta novedad no solo contribuye a simplificar la arquitectura de tus proyectos, sino que también facilita su comprensión y gestión.

En las siguientes secciones exploraremos cómo puedes aprovechar estas innovaciones para implementar un interceptor de autenticación, para adaptarnos a este renovado enfoque que propone Angular 17.

# Tema 14. Servicios y HTTP

## Crear un interceptor

En este primer paso creamos el interceptor **AuthInterceptor**. Este archivo es una clase de Angular que **implementa el interfaz HttpInterceptor**. Este archivo se colocará de forma automática entre nuestros servicios y nuestras peticiones externas para añadir las cabeceras, en este caso de autenticación, a todas nuestras peticiones.

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

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor() { }

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    //console.log('paso el interceptor');
    //la logica del interceptor
    const cloneRequest = request.clone({
      setHeaders: {
        'Content-Type': 'application/json',
        'Authorization': localStorage.getItem('miToken') || ""
      }
    })

    console.log(cloneRequest);
    return next.handle(cloneRequest);
  }
}
```

# Tema 14. Servicios y HTTP

## Proporcionar un interceptor

Cuando trabajamos con Angular en su modo *standalone*, los interceptores son integrados directamente dentro de los componentes o servicios donde se requieran. No obstante, en el caso de **interceptores** que deseamos que tengan un **alcance global**, es decir, que estén operativos a través de toda la aplicación, es necesario efectuar un paso adicional de registro. Incluso en Angular 17, es esencial indicar explícitamente que deseamos utilizar un interceptor particular para que Angular lo reconozca y lo incluya en el ciclo de procesamiento de peticiones HTTP. Este proceso de registro se lleva a cabo en el archivo **main.ts**, el cual actúa como punto de partida de nuestra aplicación. Esto es debido a que, con la arquitectura en modo *standalone*, no contamos con un módulo raíz de la manera tradicional.

```
// main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
import { AppComponent } from './app/app.component';
import { AuthInterceptor } from './app/auth.interceptor';

bootstrapApplication(AppComponent, {
  providers: [
    HttpClientModule,
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AuthInterceptor,
      multi: true,
    }
  ]
}).catch(err => console.error(err));
```

En este caso, usamos **bootstrapApplication** para iniciar la aplicación. Hemos designado **AppComponent** como el componente principal y hemos suministrado una serie de configuraciones adicionales como parte del segundo parámetro. Es en este paso en el que se lleva a cabo el registro del AuthInterceptor como un proveedor a nivel de aplicación, de esta forma se garantiza que cada petición HTTP emitida por la aplicación lleve consigo el token de autenticación necesario.

## 15.1. 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.

## 15.2. ¿Qué es un Pipe?

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 uno de los pipes más sencillos, el nombre está escrito en letra capital, es decir *Alberto*, al ponerle el pipe uppercase el dato aparecerá como ALBERTO.

```
<p>{{ nombre | uppercase }}</p>
```

## 15.3. 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>{{ rational | 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 aplicaciones 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 podemos 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.

```
<!--
```

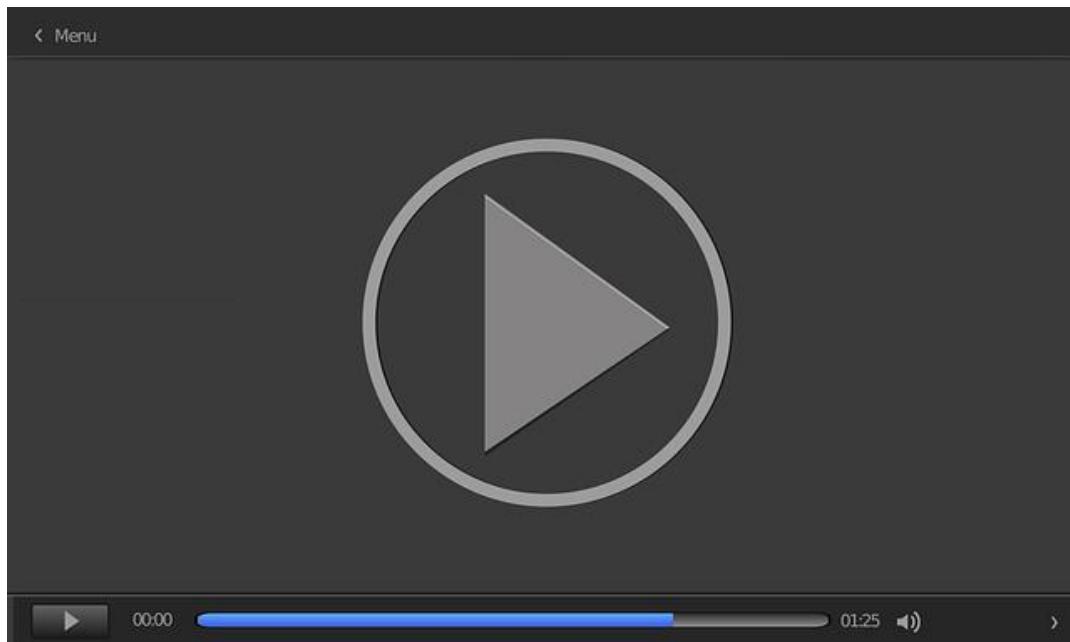
Formato	Equivale	Salida
short	M/d/yy h:mm a	6/15/15 9:03 AM
medium	MMM d y h:mm:ss a	Jun 15 2015 9:03:01 AM
long	MMMM d y h:mm:ss a z	June 15 2015 at 9:03:01 AM GMT+1
shortDate	M/d/yy	6/15/15
mediumDate	MMM d y	Jun 15 2015
longDate	MMMM d y	June 15 2015
fullDate	EEEE MMMM d y	Monday June 15 2015
shortTime	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 h:mm:ss a zzzz	Monday June 15 2015 at 9:03:01 AM GMT+01:00

```
-->
```

# Tema 15. Pipes

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:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=973f8bcc-531d-488b-8a2c-b130001a28e8>

---

## 15.4. 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()`.

Actualmente, con la nueva versión de Angular, con el modo *standalone*, los módulos ya no cobran tanta importancia y se les da mayor protagonismo a los componentes.

# Tema 15. Pipes

Si quieres usar un pipe, tienes que importarlo en el componente en el cual lo vas a usar. Veamos un ejemplo de importación de un pipe dentro de un componente.

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { CapitalizePipe } from './pipes/capitalize.pipe';

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

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.

# Tema 15. Pipes

El código sería el siguiente:

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

@Pipe({
  name: 'capitalize',
  standalone: true
})
export class CapitalizePipe implements PipeTransform {

  transform(value: string): string {
    let arr = value.split("");
    let palabra = "";
    arr.forEach((letra, index) => {
      if (index === 0) {
        palabra += letra.toUpperCase()
      } else {
        palabra += letra.toLowerCase()
      }
    })
    return palabra
  }
}
```

Y en el 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 16. Bootstrap en Angular

## 16.1. 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.

# Tema 16. Bootstrap en Angular

## 16.2. 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"  
    ]  
  },  
},
```

# Tema 16. Bootstrap en Angular

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.

# Tema 16. Bootstrap en Angular

## 16.3. Uso de Bootstrap en Angular

El uso de Bootstrap está 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.

```
@if(characters.length <= 0){  
<section class="container-fluid mt-4">  
  <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 una multitud de plantillas 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 5.3.

# Tema 16. Bootstrap en Angular

En la parte de documentación de la página puedes encontrar layouts, componentes, clases de utilidad y todo lo necesario para poder trabajar con un proyecto web completo.

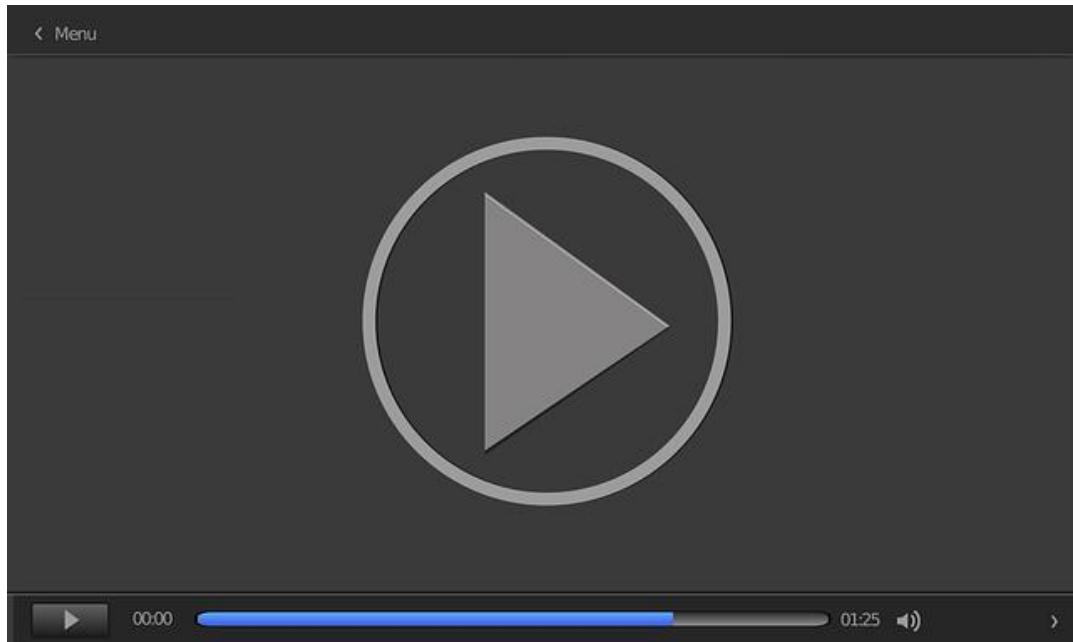
The screenshot shows the Bootstrap documentation for the 'Buttons' component. On the left, there's a sidebar with categories like 'Forms' (Overview, Form control, Select, Checks & radios, Range, Input group, Floating labels, Layout, Validation) and 'Components' (Accordion, Alerts, Badge, Breadcrumb, Buttons, Button group, Card, Carousel, Close button, Collapse, Dropdowns, List group, Modal, Navbar, Navs & tabs). The 'Buttons' section is currently selected. In the main content area, there's a row of colored buttons labeled 'Primary', 'Secondary', 'Success', 'Danger', 'Warning', 'Info', 'Light', 'Dark', and 'Link'. Below this, an 'HTML' code block shows the corresponding button tags with their class names. A callout box titled 'Conveying meaning to assistive technologies' explains that color is used to add meaning, which may not be conveyed to users of assistive technologies like screen readers. It suggests using additional text hidden with the `.visually-hidden` class. Another callout box titled 'Disable text wrapping' explains that if you don't want the button text to wrap, you can add the `.text nowrap` class to the button. In Sass, you can set `$btn-white-space: nowrap` to disable text wrapping for each button.

Figura 38. Documentación de Bootstrap. Fuente: elaboración propia.

Te animo a echarle un vistazo a todo el material que tienes de Bootstrap en Internet he intentar darles una estética a tus componentes a través de este potente framework.

# Tema 16. Bootstrap en Angular

A continuación, puedes ver el vídeo *Cómo usar Bootstrap dentro de Angular*.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=f1e3948a-54a3-4e75-8b37-b129014ef627>

---

# Tema 17. Gestión de librerías, publicación y mapas

## 17.1. Introducción y objetivos

Este tema de Angular cumple dos objetivos fundamentales. El primero, que ya inició en el tema anterior, es como instalar librerías de terceros a través del gestor de paquete NPM.

En el caso de esta librería vamos a usar Google Maps a través de la librería oficial de Google Maps para 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, por lo que 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 me sitúen y geolocalicen en un mapa.

# Tema 17. Gestión de librerías, publicación y mapas

## 17.2. ¿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 nuestros 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"  
            }  
        ],  
        "outputHashing": "all"  
    },  
}
```

# Tema 17. Gestión de librerías, publicación y mapas

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.

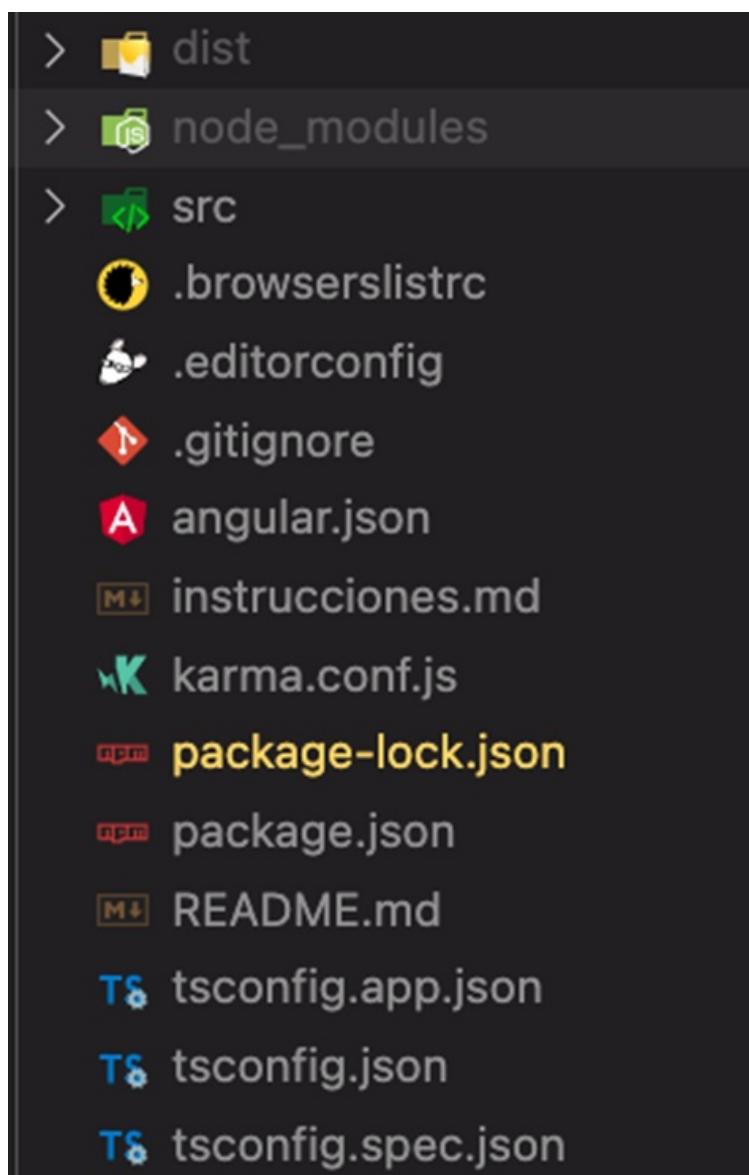


Figura 39. Árbol de carpetas con dist. Fuente: elaboración propia.

# Tema 17. Gestión de librerías, publicación y mapas

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 sobre 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 tremadamente fácil porque solo es poseer esta carpeta y publicarla dentro de nuestro hosting.

```
npm install
```

# Tema 17. Gestión de librerías, publicación y mapas

## 17.3. Instalación de librerías de terceros en Angular

En este capítulo vamos a hablar de la instalación de librerías externas que no pertenecen a Angular, pero que funcionan dentro de su ecosistema, y de sus ventajas de cara al desarrollo de nuestros proyectos.

### Ventajas de usar librerías externas

La inclusión de librerías externas en tu proyecto Angular aporta varias ventajas:

- ▶ **Reutilización de código:** aprovechas el trabajo que otros desarrolladores han hecho, lo que te ahorra tiempo y esfuerzo al no tener que reinventar la rueda.
- ▶ **Estandarización:** las librerías suelen seguir prácticas comunes y patrones de diseño que pueden ayudarte a mantener tu código organizado y estandarizado.
- ▶ **Eficiencia:** las librerías externas a menudo están optimizadas y probadas para rendir de la mejor manera posible, lo que puede mejorar el rendimiento de tu aplicación.
- ▶ **Funcionalidades mejoradas:** puedes extender las capacidades de tu aplicación fácilmente, al incorporar funcionalidades que no vienen incluidas de forma nativa en Angular.
- ▶ **Mantenimiento:** al depender de librerías que tienen una comunidad activa, te beneficias de las actualizaciones regulares que incluyen mejoras y correcciones de seguridad.
- ▶ **Compatibilidad:** las librerías suelen estar diseñadas para ser compatibles con diversas plataformas y dispositivos, lo que amplía el alcance de tu aplicación.
- ▶ **Soporte comunitario:** las librerías populares cuentan con el soporte de comunidades grandes, en las que puedes encontrar ayuda, documentación extensa y recursos de aprendizaje.

# Tema 17. Gestión de librerías, publicación y mapas

Antes de añadir una librería externa a tu proyecto, es importante **considerar** su **tamaño, rendimiento, compatibilidad**, frecuencia de **actualizaciones** y la **comunidad** que hay detrás. Esto te ayudará a evitar librerías que podrían estar descontinuadas, no mantenidas o que podrían introducir vulnerabilidades en tu aplicación.

Para ayudarte a comprender este proceso vamos a usar una **librería de Google Maps** para Angular que nos permite no solo colocar un mapa dentro de nuestras aplicaciones, sino también configurar otras acciones como geolocalizarnos y mostrarnos áreas dentro de nuestro mapa. También 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, tal y como hemos hecho otras veces, tenemos que salir a nuestra terminal, a la carpeta del proyecto e instalar la librería a través de nuestro gestor de paquetes con el siguiente comando:

```
npm i @angular/google-maps
```

Esta librería pertenece a un conjunto de herramientas externas a Angular, pero que nos ofrecen una cantidad de posibilidades muy interesantes para nuestros desarrollos. La librería se llama **Angular Material** y tiene no solo la parte de Google Maps, sino también una interfaz de componentes UI muy interesantes y algunos otros desarrollos como la API de YouTube para implementar desarrollos con dicha API dentro de nuestras webs.

# Tema 17. Gestión de librerías, publicación y mapas

Todo esto lo puedes ver en la página de [npmjs.com](https://www.npmjs.com), donde puedes buscar cualquier dependencia que quieras instalar a través de **npm**.

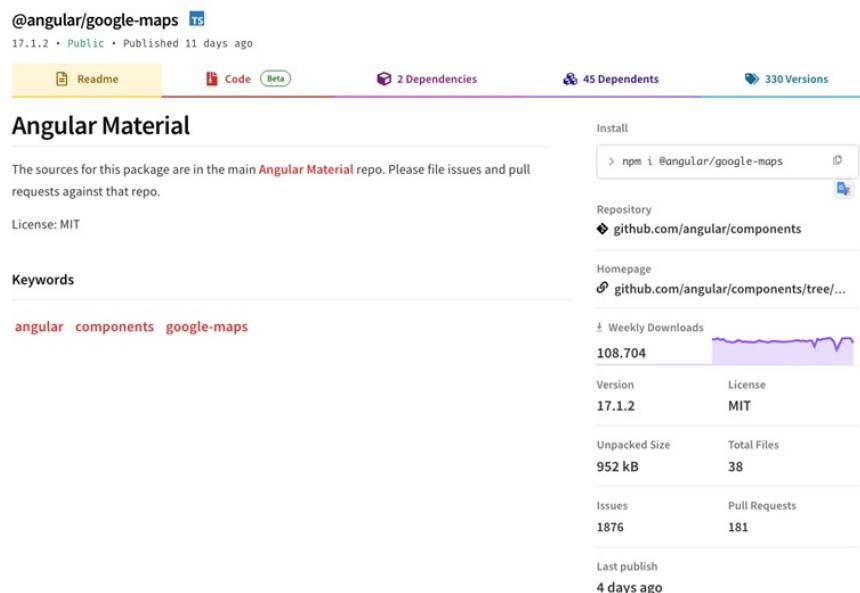


Figura 40. npmjs, captura de librería Angular Material. Fuente: elaboración propia.

Una vez que tenemos nuestra librería instalada, podremos comprobarlo en nuestro fichero packages.json, en las dependencias que aparecerá la librería instalada.

```
"dependencies": {  
  "@angular/google-maps": "^17.1.2",
```

El siguiente paso es crearse una cuenta en Google Cloud Platform a través de la siguiente dirección web, con una cuenta de Gmail gratuita:

---

Página web de Google Cloud Platform: <https://console.cloud.google.com/>

---

# Tema 17. Gestión de librerías, publicación y mapas

Una vez que te has registrado debes crear un token de acceso creando una aplicación de mapas dentro de la gestión de la aplicación y dar de alta una aplicación de mapas en JavaScript.

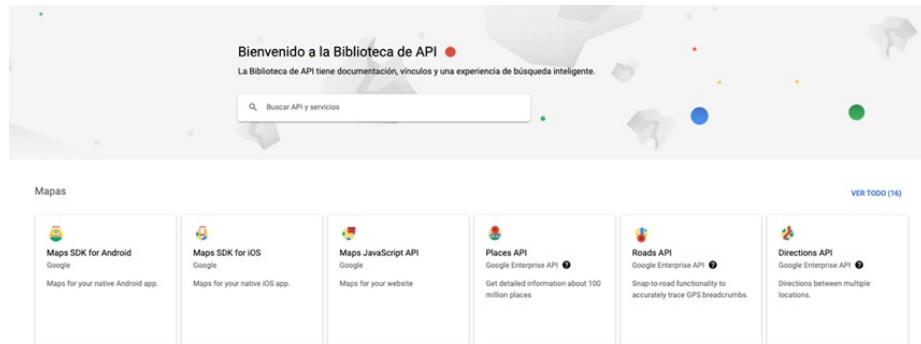


Figura 41. Google Platform elección de mapas de Google con Javascript. Fuente: elaboración propia.

Una vez que 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 que explica el proceso paso a paso.

---

Una vez que la hayas instalado y configurado simplemente tienes que hacer dos cosas:

- ▶ Instalar el API-Key creado para la API de Google Maps en el fichero index.html de esta forma:

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Mapas</title>
<base href="/">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="icon" type="image/x-icon" href="favicon.ico">
<script src="https://maps.googleapis.com/maps/api/js?>
```

# Tema 17. Gestión de librerías, publicación y mapas

```
key=GOOGLE_API_TOKEN&libraries=places,directions"></script>
</head>
<body>
<app-root></app-root>
</body>
</html>
```

- ▶ Cuando tengamos la API-Key instalada en nuestro index.html, ya estamos listos para usar dicha librería en nuestros componentes de la siguiente manera:

```
import { Component } from '@angular/core';
import { GoogleMap, GoogleMapsModule } from '@angular/google-maps';

@Component({
  selector: 'app-mapa',
  standalone: true,
  imports: [GoogleMapsModule, GoogleMap],
  templateUrl: './mapa.component.html',
  styleUrls: ['./mapa.component.css']
})
export class MapaComponent {
```

Como puedes observar, importaremos el módulo de GoogleMapsModule y la librería GoogleMap al componente en el que la queramos usar, al colocarlas en el array de imports de dicho componente.

Y con esto ya tenemos configurada nuestra aplicación para que la podemos usar en nuestros componentes. Ahora vamos a ver un ejemplo sencillo de cómo se usa.

# Tema 17. Gestión de librerías, publicación y mapas

## 17.4. Uso de Google Maps en Angular

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. También tendremos que definir una propiedad para el zoom del mapa para ver con qué amplitud queremos colocar dichas coordenadas.

```
import { Component, ViewChild } from '@angular/core';
import { GoogleMap, GoogleMapsModule, MapInfoWindow, MapMarker } from '@angular/google-maps';

@Component({
  selector: 'app-mapa',
  standalone: true,
  imports: [GoogleMapsModule, GoogleMap],
  templateUrl: './mapa.component.html',
  styleUrls: ['./mapa.component.css']
})
export class MapaComponent {

  zoom: number = 8;
  center: google.maps.LatLng = new google.maps.LatLng(40, -3);
  myposition: google.maps.LatLng | any;
}
```

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:

```
<google-map [center]="center" [zoom]="zoom">
  <map-marker #miMarker="mapMarker" [position]="myposition" >
    </map-marker>
  </google-map>
```

Además de este código deberás asignarle en el CSS de tu componente una altura a tu mapa para que se visualice.

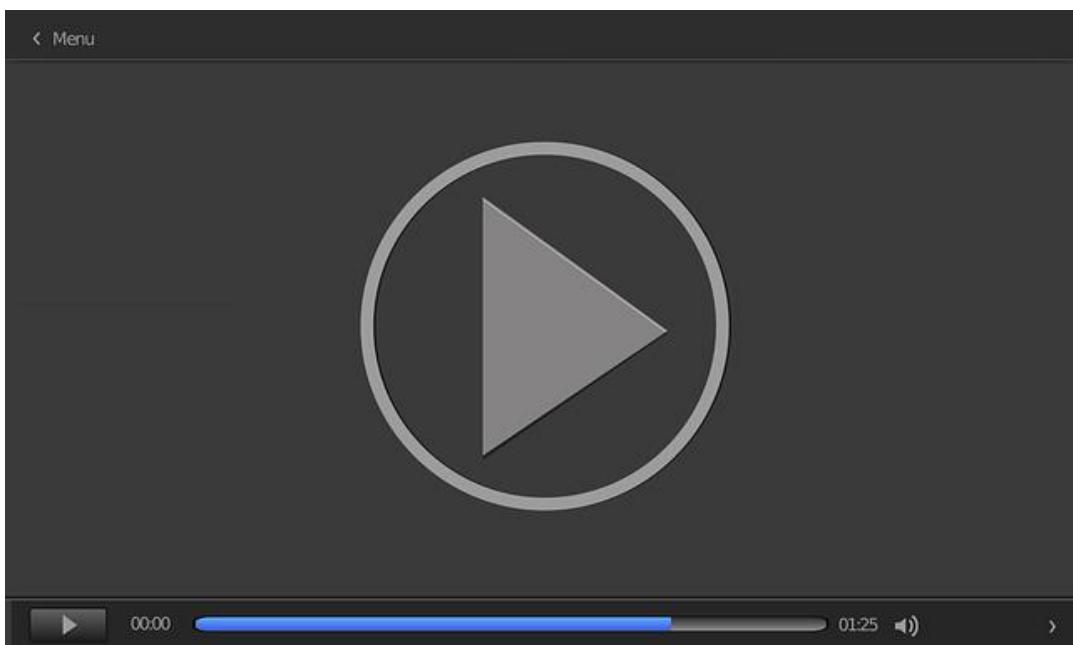
# Tema 17. Gestión de librerías, publicación y mapas

```
google-map {  
    width: 100%;  
    height: 100vh;  
}
```

Esto creará un mapa que te ocupará toda la pantalla y que nos posicionará con un marker que sitúe nuestra posición.

A través de esta librería podrás crear varios marker usando un **bucle @for** en el que repitas las marker, por ejemplo, desde un array de posiciones y posicionar productos, inmuebles, lo que te dé la gana.

A continuación, puedes ver el vídeo *Crear una aplicación de Angular con Google Maps*.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=29f7b3e3-1545-4a82-867e-b13000259b39>

---

## Documentación oficial de Angular

Página web de Angular: <https://angular.dev/>

Aquí tienes la documentación oficial de angular.

## Qué es una SPA

Alvarez, M. Á. (2016, noviembre 29). Qué es una SPA. *Desarrollo Web*.

<https://desarrolloweb.com/articulos/que-es-una-spa.html>

Este artículo explica el concepto de Single Page Application (SPA), un tipo de aplicación web que es cada vez más usado por la agradable experiencia de usuario que aporta.

## Documentación oficial de Typescript

Página web de Typescript: <https://www.typescriptlang.org/>

Aquí tienes la documentación oficial de typescript, la tienes en inglés y en español.

## TypeScript para principiantes

Shokeen, M. (2022, marzo 14). TypeScript para principiantes. Parte 1: Comenzando.

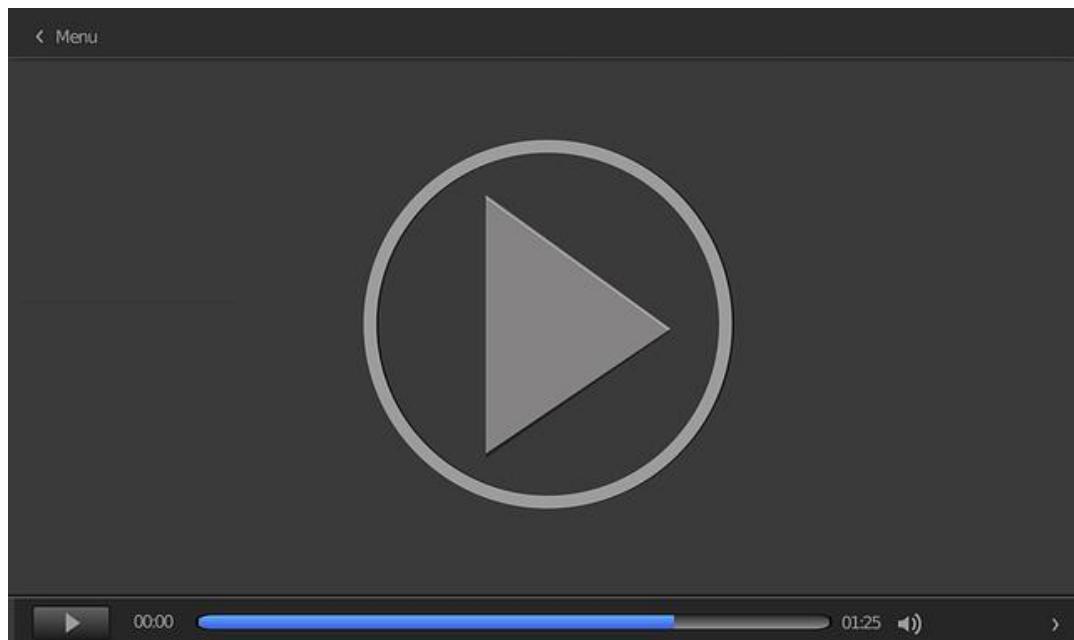
*Envato Tuts+.* <https://code.tutsplus.com/es/tutorials/typescript-for-beginners-getting-started--cms-29329>

Es un buen tutorial de TypeScript con varias partes y en castellano.

## TypeScript: aprende desde cero con curso oficial de Microsoft

Midulive. (2022, marzo 20). *Aprende TypeScript desde cero con este curso oficial de Microsoft [vídeo]*. YouTube. <https://www.youtube.com/watch?v=YKclM8Ixkfl>

Este es un vídeo muy interesante sobre TypeScript en castellano.



Accede al vídeo:

<https://www.youtube.com/embed/YKclM8Ixkfl>

## Manual de Angular

Desarrollo web. (s. f.). *Manual de Angular.*

<https://desarrolloweb.com/manuales/manual-angular-2.html>

Monografía sobre Angular en español con la que podréis afianzar este concepto.

## Documentación oficial del framework

Angular. (s. f.). *Introduction. What is Angular?* <https://angular.dev/overview>

Documentación oficial de Angular.

## Web interesante sobre Angular en español

---

Aristizabal Angel, V. M. (2020). Componentes. *Aprendiendo Angular*.  
<https://ngchallenges.gitbook.io/project/componentes>

---

Aquí podrás encontrar la explicación de qué es un componente desde el punto de vista de alguien que está aprendiendo. Es una muy buena práctica, cuando uno está aprendiendo, el enseñarles a otros para fijar los conceptos.

## Entendiendo los componentes en Angular – Guía de iniciación

Rocío. (2019, julio 31). Entendiendo los componentes en Angular. Guía de iniciación.

A *ContracorrienTech*. <https://www.acontracorrientech.com/entendiendo-los-componentes-en-angular/>

Un blog interesante en el que se explica cómo funcionan los componentes en Angular.

## Componentes en Angular

Aristizabal Angel, V. M. (2019, septiembre 15). Componente en Angular. *Medium*.

<https://medium.com/notasdeangular/componentes-en-angular-f25138b00c83>

Es una breve, pero muy efectiva, explicación de lo que es un componente. Publicación en castellano que trata, con dibujos y ejemplos, el funcionamiento de los componentes.

## Interpolación {{}} en Angular al detalle

Alvarez, M. Á. (2017, octubre 30). Interpolación {{}} en Angular al detalle. *Desarrollo web*. <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. (2020, septiembre 14). Using Sass with the Angular CLI. *DigitalOcean*.

<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. (s. f.). *Property biding*. <https://angular.io/guide/property-binding>

Esta es la documentación oficial sobre los property biding. Aquí también puedes consultar todas las novedades de las siguientes versiones si es que ocurriera.

## Guía de iniciación a la data binding en Angular

Rocío. (2019, agosto 14). Guía de iniciación al data binding en Angular. *A ContracorrienteTech.* <https://www.acontracorrientech.com/guia-practica-del-databinding-en-angular/>

Ejemplo práctico. Este post es una guía de iniciación que explica paso a paso y de forma muy clara lo que es el data binding y cómo se implementa de forma correcta.

## Event binding in Angular

Taran910. (s. f.). Event Binding in Angular 8. Geeks for geeks.

<https://www.geeksforgeeks.org/event-binding-in-angular-8/>

## Documentación oficial de eventos Angular

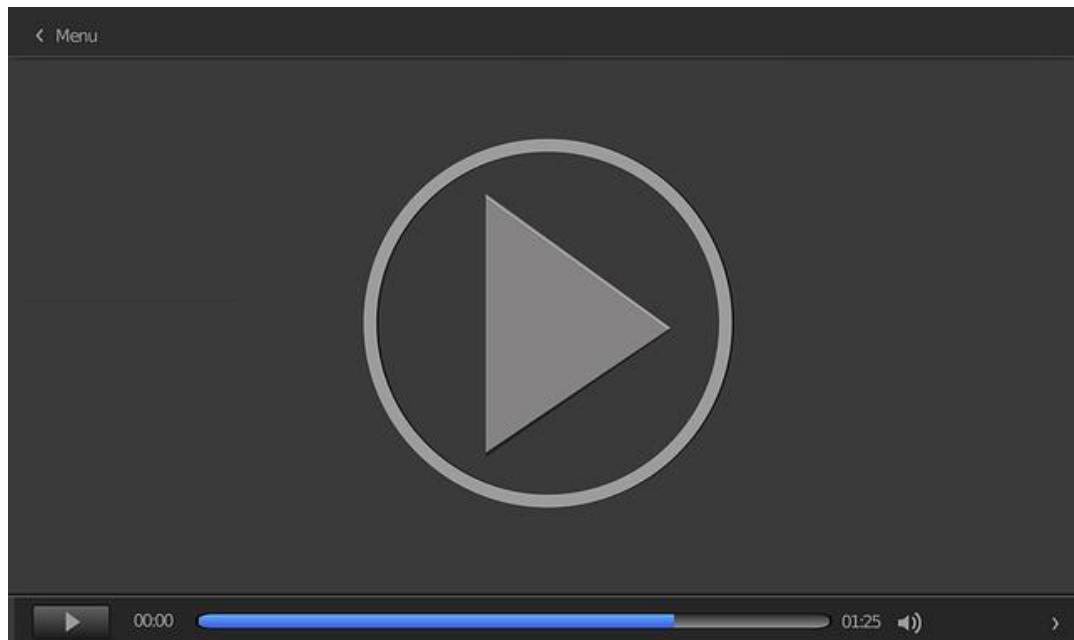
Angular. (s. f.). *Event binding*. <https://angular.dev/guide/templates/event-binding>

Es muy útil tener acceso a la explicación que hace la documentación de Angular sobre la gestión de eventos en un interfaz hecho en Angular.

## Video explicativo sobre cómo se gestionan eventos en Angular, en castellano

Píldorasinformáticas. (2021, febrero 26). *Curso Angular. Event Binding.* Vídeo 10 [vídeo]. YouTube. <https://www.youtube.com/watch?v=FPjFXQf1pqM>

Es un vídeo interesante sobre cómo se gestionan los eventos en Angular.



Accede al vídeo:

<https://www.youtube.com/embed/FPjFXQf1pqM>

## Input y Output en Angular

Bastidas, W. (2020, agosto 2). Angular decoradores @Input y @Output. *Medium*.  
<https://medium.com/williambastidasblog/angular-decoradores-input-y-output-70af5f43a04>

Es un artículo interesante en el que se explican los decoradores input y output.

## Documentación oficial de input y outputs Angular

Angular. (s. f.). *Accepting data with input properties.*

<https://angular.dev/guide/components/inputs>

Angular. (s. f.). *Custom events with outputs.*

<https://angular.dev/guide/components/outputs>

Explicación extensa y oficial de cómo se gestionan los input y outputs en Angular.

## Qué son los signals en Angular

Angular University. (2024, enero 26). *Angular Signals: Complete Guide*. <https://blog.angular-university.io/angular-signals/>

Angular. (s. f.). *Angular Signals*. <https://angular.io/guide/signals>

Explicación extensa y oficial sobre cómo se gestionan los signals en Angular.

## Control Flow en Angular

Angular. (s. f.). *Built-in control Flow*. <https://angular.dev/guide/templates/control-flow>

Explicación oficial sobre el uso de los bloques de control del flujo.

## Class and style en Angular

Angular. (s. f.). *Class and style binding*. <https://angular.dev/guide/templates/class-binding>

Explicación oficial sobre el uso de class y style en Angular.

## NgClass and ngStyle en Angular

Angular. (s. f.). *Built-in directives*. <https://angular.dev/guide/directives>

Explicación oficial sobre el uso de ngClass y ngStyle.

## Introducción al servicio e inyección de dependencias en Angular

Angular. (s. f.). *Introducción a servicios e inyección de dependencias.*  
<https://docs.angular.lat/guide/architecture-services>

Explicación extensa y oficial sobre cómo se gestiona la inyección de dependencias en Angular.

## Explicación de la función inject()

Angular. (s. f.). *Inject*. <https://docs.angular.lat/api/core/inject>

Explicación extensa y oficial sobre cómo se gestionan la función inject() en Angular.

## Diferencias entre formularios de tipo template y model

Mugika Ledo, A. (2021, enero 14). Formularios en Angular – Diferencias Template y Reactive Forms. *Medium*. <https://mugan86.medium.com/formularios-en-angular-diferencias-template-y-reactive-forms-e37af5e30b81>

Un artículo interesante en el que se explican las diferencias entre los dos formularios desde otro punto de vista.

## Documentación oficial sobre formularios, en español, de Angular

Angular. (s. f.). *Inténtalo: Usa formularios para capturar la información del usuario.*

<https://docs.angular.lat/start/start-forms>

Explicación extensa y oficial sobre cómo se gestionan los formularios en Angular.

## Documentación oficial de Angular El maneo de ruta Angular

Angular. (s. f.). *Angular Routing*. <https://angular.dev/guide/routing>

Documentación oficial sobre los conceptos de rutas en Angular.

## Servicios en Angular

---

Alvarez, M. Á. (2020, mayo 14). Servicios en Angular. *Desarrollo web.*  
<https://desarrolloweb.com/articulos/servicios-angular.html>

---

Veremos qué son los servicios en el framework Javascript Angular, cómo crear services e injectarlos en los componentes para acceder a ellos.

## Documentación oficial sobre ~~Servicios~~ en español de Angular

Angular. (s. f.). *Agregar servicios*. <https://docs.angular.lat/tutorial/toh-pt4>

Documentación sobre cómo agregar y crear servicios, cómo gestionarlos. Está explicado en español y desde la documentación oficial del framework.

## HTTP en Angular

Angular. (s. f.). *Understanding communicating with backend services using HTTP.*

<https://angular.dev/guide/http/>

Documentación oficial de Angular 17 con toda la información del módulo HTTP.

## Comunicaciones HTTP en Angular

Academia Binaria. (2020, enero 12). Comunicaciones HTTP en Angular. *Studocu*.  
<https://www.studocu.com/pe/document/universidad-catolica-san-pablo/sistemas-optoelectricos/comunicaciones-http-en-angular-academia-binaria/16215161>

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 sobre cómo se producen.

## Cómo convertir un observable en promesas en Angular

Cloudhadoop. (2023, diciembre 31). *How to convert Observable to Promise in angular/TypeScript.* <https://www.cloudhadoop.com/angular-convert-observable-to-promise/>

Un post interesante sobre cómo gestionar un observable en forma de promesa.

## Pipes en Angular: guía completa

Rocío. (2019, octubre 16). Pipes en Angular | Guía completa. *A ContracorrienteTech.*

<https://www.acontracorrientech.com/pipes-en-angular-guia-completa/>

Guía completa sobre cómo usar los pipes en español. Da un repaso completo por todos los pipes propios de Angular y cómo se crean.

## Bootstrap 5 in Angular

---

Bouchefra, A. (2023, septiembre 27). Add Bootstrap 5 to Angular 17 with example & tutorial. *Techiediaries*. <https://www.techiediaries.com/angular-bootstrap/>

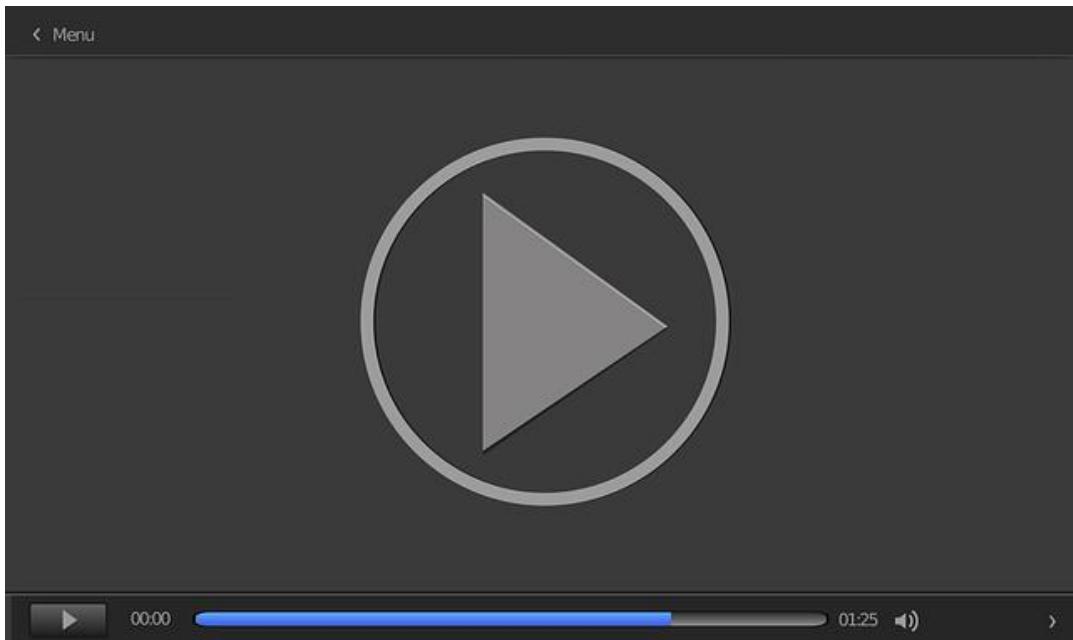
---

Una guía completa en inglés de cómo usar Bootstrap 5 en versiones de Angular superiores a la 12.

## Bootstrap 5 en Angular: explicación en vídeo

Domini Code. (2021, noviembre 29). *Instalar Bootstrap 5 en Angular 13* [vídeo]. YouTube. <https://www.youtube.com/watch?v=rl1aiL0o48>

Un vídeo muy interesante y explicativo de cómo se instala Bootstrap 5, en castellano, de una manera muy sencilla.



Accede al vídeo:

<https://www.youtube.com/embed/rli1aiL0o48>

## Github oficial de la librería de Google Maps

Angular (s. f.). Components. GitHub. <https://github.com/angular/components>

Documentación oficial de la librería, documentación y guías rápidas.

## Cómo realizar una publicación de Angular en un hosting compartido

Mugika Ledo, A. (2020, enero 10). Publicar una aplicación Angular en un Hosting compartido. *Medium*. <https://mugan86.medium.com/publicar-una-aplicaci%C3%B3n-angular-en-un-hosting-compartido-c6c934da3c36>

Pasos por seguir para publicar una app en Angular en un hosting que soporta HTML, PHP, CSS y JS.

## Guía rápida: ¿cómo obtener la API Key de Google Maps?

Trafaniuc, V. (2020, febrero 26). Guía rápida: ¿cómo obtener la API key de Google Maps? *Maplink*. <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?

## API de Maps Javascript

Google Maps Platform. (s. f.). *Descripción general.*

<https://developers.google.com/maps/documentation/javascript/overview?hl=es-419>

Documentación en castellano de la API de Google Maps para Javascript de Google.

**1.** ¿Qué es Angular?

- A. Una librería dedicada 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 permite 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. Es 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 que se usa para generar un componente en Angular?

- A. ng generate.
- B. ng test.
- C. ng generate component /ruta del componente.
- D. ng component.

5. ¿Para qué sirve una interfaz en Angular?

- A. Genera un archivo de clase en el que se definen e importan los elementos importantes para el desarrollo de una aplicación en Angular.
- B. La interfaz es una variable de Angular que permite definir algo que se tiene que cumplir obligatoriamente en aquella clase en la que 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 los componentes entre sí.
- B. Conectar los componentes entre sí y pasarles los datos al conectarse a las 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 se conecta con todos los elementos de nuestra aplicación.
  - C. Dentro del app-routes.ts.
  - D. Puedes definir rutas dentro de cualquier componente a través de router.
  
- 9.** ¿Cuántas estructuras usa el control flow?

  - A. Una.
  - B. Dos.
  - C. Tres.
  - D. Cuatro.
  
- 10.** ¿Para qué sirve la interpolación en Angular?

  - A. Para intercalar el uso de una 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 injectar texto dentro de una plantilla HTML de nuestro componente.