



ANGULAR

by Google

Contenido

Introducción a Angular	3
Características de Angular	3
Primeros pasos.....	4
AngularCLI.....	4
NodeJS	4
Instalación de AngularCLI usando NodeJS	5
Introducción a TypeScript	6
Primer proyecto en Angular: ¡¡¡HOLA MUNDO!!!	7
Crear el proyecto Hola Mundo	7
Ejecutar el proyecto Hola Mundo	8
Estructura básica de un proyecto en Angular	10
Componentes	12
Entendiendo los componentes	12
<i>Creando un nuevo componente</i>	13
Usando los componentes	15
Refuerzo 1	18
Vistas.....	19
Tipos de binding.....	19
Esquema de binding entre vista-modelo.....	19
Propiedades [...]	20
Expresiones {{...}}.....	20
Eventos (...).....	21

Binding [...]	21
Ejercicios (binding y componentes)	23
Decoradores	24
Módulos.....	25
Creando un nuevo módulo.....	25
Añadir componentes al módulo nuevo	26
Esquema módulos y componentes	27
Sumario	28
 Proyecto Héroes.....	29
Directivas estructurales	33
*ngIf	33
*ngFor.....	34
ngSwitch	34
ngContainer	35
ngClass	35
Ejercicios (ngIf, ngFor, ngClass)	36
 Listado de Héroes. ngFor	37
Modelos	39
 Usando Modelos en Héroes	40
Servicios.....	42
Crear un servicio	42
Declaración implícita de propiedades en TypeScript.....	47
Comunicación entre componentes	48
@Input. Paso de información de padres a hijos.....	48
@Output. Paso de información de los hijos a los padres	50
Interfaces en TypeScript.....	54
Definición de una interfaz	54
Implementación de una interfaz	55
Usando la interfaz como un nuevo tipo	56
Clases o interfaces	57
Usar interfaz en un modelo.....	57
Pipes.....	58
Cambiar Locale (configuración regional)	58

Pipes personalizadas	60
 Héroes 2.0	61
Routing en Angular	63
<base href>	63
Añadir routing manualmente	63
Usar el routing creado por AngularCLI	64
RouterLink.....	67
 Cambio de la clase Heroes a la interfaz	68
Observables	69
Peticiones HTTP (Ajax)	70
 Novedades antes del CRUD	77
CRUD en Angular usando Firebase de Google.....	79
Crear el proyecto en Firebase	79
Agregar Firebase al proyecto	85
Read (Leer registros)	89
Create (Guardar registros).....	94
Delete (Borrar registros)	101
Update (Actualizar registros).....	102
Notificaciones con SweetAlert2	107
Testing (Pruebas unitarias con Karma)	111
Ejecutar las pruebas	112
Archivos de tests	115
Creando una calculadora.....	115
Elaborando pruebas unitarias (it).....	117
Expect	120
Testing usando el DOM.....	122
BeforeEach y AfterEach	124
Repositorio en GitHub.....	125
Desplegar en producción.....	126
Desplegando desde GitHub en Netlify.com	127
Créditos y menciones	132

Angular

Frameworks para frontend

Introducción a Angular

Angular, es un framework para aplicaciones web desarrollado **en TypeScript**, de código abierto, mantenido por Google, **que se utiliza para crear** y mantener **aplicaciones web de una sola página (SPA, single page application)**.

Su objetivo es aumentar las aplicaciones basadas en navegador con capacidad de Modelo Vista Controlador (MVC), en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles.

Angular se basa en clases tipo "Componentes", cuyas propiedades son las usadas para hacer enlace de los datos (data binding). En dichas clases tenemos propiedades (atributos) y métodos (funciones).

Angular es la evolución de AngularJS, el cual fue lanzado en 2010 por Google y se desarrolló en JavaScript. Debido a las limitaciones del lenguaje y el cambio radical de funcionamiento se volvió a escribir desde 0, naciendo un framework nuevo más que una nueva versión de este. AngularJS no es compatible con las versiones de Angular2 en adelante.

Características de Angular

Velocidad y rendimiento

- **Generación de código:** Angular convierte tus plantillas de código altamente optimizado para las máquinas virtuales de JavaScript de hoy en día, ofreciéndote todas las ventajas del código escrito a mano con la productividad de un framework.
- **Universal:** Ejecuta la primera vista de tu aplicación en node.js, .net, PHP, y otros servidores para renderizado de forma casi instantánea obteniendo sólo HTML y CSS. También abre posibilidades para la optimización del SEO del sitio.
- **División del código:** Las aplicaciones Angular se cargan rápidamente gracias al nuevo enrutador de componentes. Éste ofrece una división automática de códigos para que los usuarios sólo carguen el código necesario para procesar la vista que lo solicitan.

Productividad

- **Plantillas:** Permite crear rápidamente vistas con una sintaxis de plantilla simple y potente.
- **Angular CLI:** Las herramientas de la línea de comandos permiten empezar a desarrollar rápidamente, añadir componentes y realizar test, así como previsualizar de forma casi instantánea la aplicación.

- **IDEs:** Obtén sugerencias de código inteligente, detección de errores y otros comentarios en la mayoría de IDEs.
- **Integración:** Permite la integración fácil de otras librerías y frameworks más conocidos, como Bootstrap, Angular Material, etc.

Historia completa del desarrollo

- **Testing:** Utiliza las últimas herramientas disponibles para realizar pruebas unitarias y otros tipos de pruebas (e2e) de forma rápida y estable.
- **Animación:** Permite crear animaciones complejas y de alto rendimiento con muy poco código a través de la intuitiva API de Angular.
- **Accesibilidad:** Posee características para crear aplicaciones accesibles con los componentes disponibles para ARIA.

Primeros pasos

AngularCLI

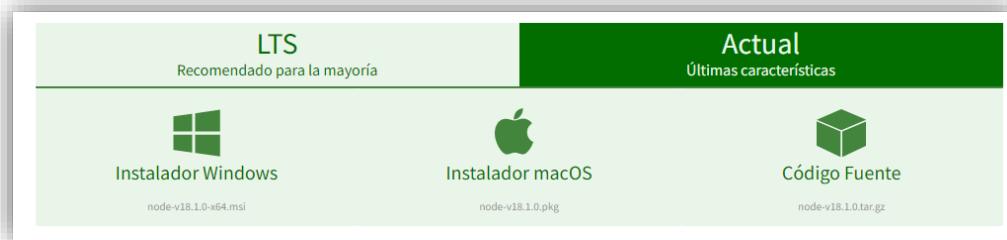
Una de las principales características de Angular respecto a AngularJS es la incorporación de una interfaz de línea de comandos (*CLI – Command Line Interface*).

Su principal ventaja es que **nos va a facilitar** el proceso de **creación de una aplicación**, el **añadir componentes**, compilar la aplicación y dejarla preparada para producción, preparar los archivos que deben subirse al servidor en la etapa de Testing, entre otras muchas opciones.

NodeJS

Node.js es un entorno en tiempo de ejecución, de código abierto y multiplataforma, para la capa de servidor (aunque no exclusivamente) basado en JavaScript. Su propósito es similar al de Apache en JavaEE o Tornado en Python. Nos servirá para poder crear, ejecutar, depurar y compilar aplicaciones de Angular, creando un servidor en caliente que se reinicia en cada cambio que se aplique al código, pudiendo ver de forma instantánea cada cambio en la aplicación sin necesitar de reiniciar el proyecto o recargar el navegador.

AngularCLI es una herramienta de NodeJS, por lo que para poder instalarla necesitaremos contar con NodeJS instalado en nuestro sistema operativo. Podemos descargarla gratuitamente de <https://nodejs.org/es>, eligiendo la versión más nueva disponible si es para probar o aprender, o la última LTS (*Long Term Support*) si es para desarrollo de una aplicación para producción.

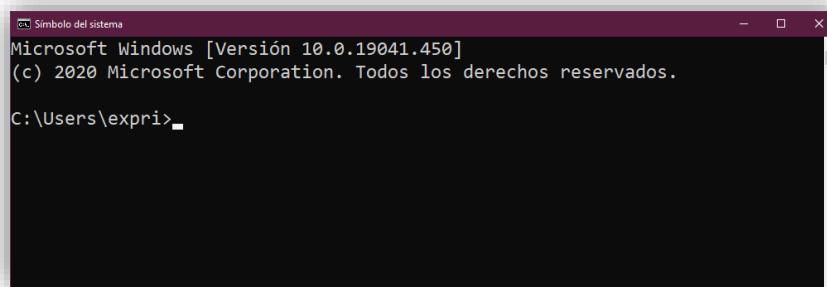


La instalación es muy simple, pulsamos siguiente, siguiente y listo.

 *No instalar las "Tools for Native Modules" durante la instalación. No es necesario.*

Instalación de AngularCLI usando NodeJS

Una vez que hemos instalado NodeJS, nuestro siguiente paso será instalar [AngularCLI](#) en NodeJS. Para ello abriremos la consola de comandos de nuestro sistema operativo. Para abrirla en Windows, pulsamos tecla Windows + R, y escribimos cmd y pulsamos intro. Nos aparecerá una ventana como la siguiente:



Donde pone "expri" aparecerá tu nombre de usuario de Windows. Y a partir de ahora podremos escribir directamente ahí los siguientes comandos de NodeJS.

 **Nota:** De aquí en adelante, todos los comandos que deban ser introducidos por consola, se representarán en un cuadro como el siguiente. Los comandos se pueden copiar y pegar directamente.

```
npm install -g npm@latest
npm set audit false
npm uninstall -g @angular/cli
npm cache clean -force
npm install -g @angular/cli@latest
ng --version
```

Instalamos el instalador de paquetes
 Desactivamos las auditorías (seguridad)
 (Opcional) Desinstalamos versión previa
 Limpiamos caché
 Instalamos la última versión de CLI
 Para comprobar la versión instalada

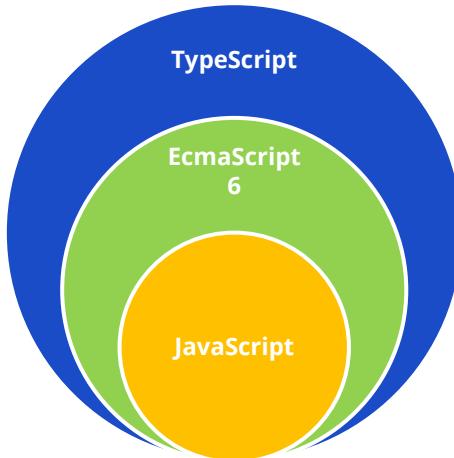
 **Nota:** Todos los pasos que se explican a lo largo del tutorial pertenecen a la versión de AngularCLI 13.3.5, actualizadas al 16/05/2022. Pueden existir diferencias en próximas actualizaciones de Angular.

Una vez instalado NodeJS y AngularCLI, ya tenemos las herramientas necesarias para poder crear un proyecto Angular con su estructura funcional en cuestión de segundos. Vamos al turrón.

Introducción a TypeScript

Hemos comentado que Angular está desarrollado en TypeScript, pero ¿podré enterarme de todo si no conozco TypeScript? La respuesta es sí. El curso está pensado para aprender Angular conociendo sólo JavaScript e iremos explicando de forma transversal las novedades y mejoras que trae TS respecto a JS.

TypeScript es un super set de JavaScript, por lo que podemos decir que es JavaScript con novedades y añadidos. Si conoces JavaScript tienes el 80% del camino andado. Veamos algunas de las características más destacables.



- Es un super set de JavaScript.
- Está mantenido por Microsoft. Y Angular por Google 😊.
- Es un lenguaje fuertemente tipado y flexible a la vez. Todos los tipos de datos [aquí en la documentación oficial](#).
- Permite las características de POO más modernas, como clases, objetos, constructores, métodos, interfaces, etc.
- Permite inyección de dependencias, como Spring Framework de Java.
- Permite los decoradores.
- La mayoría de IDEs soportan autocompletado inteligente de código.

Toda la documentación oficial y actualizada de TypeScript la [tienes disponible en este enlace](#).

Primer proyecto en Angular: ¡¡¡HOLA MUNDO!!!

Crear el proyecto Hola Mundo

Para crear nuestro primer proyecto usando AngularCLI, es tan fácil como escribir un par de comandos en la consola.

En primer lugar, nos situaremos en la carpeta donde queramos crear nuestro proyecto sin crear la carpeta contenedora, ya lo hará el asistente. Es decir, si queremos crear un proyecto llamado *mi-proyecto* en una carpeta llamada *ProyectosGuays*, nos colocaremos en la carpeta ya creada de *ProyectosGuays* y entramos en la línea de comandos.

💡 Nota: Una forma fácil de entrar en la línea de comandos desde una carpeta concreta en Windows, es crear y situarnos en esa carpeta desde el explorador de archivos y hacer clic en la barra de direcciones, borrar su contenido y escribir "cmd" y pulsar intro. Se abrirá la ventana de la línea de comandos directamente en esa carpeta.

Desde la línea comandos, escribimos lo siguiente:

```
ng new
```

Con esto se iniciará el asistente que nos preguntará lo siguiente:

```
? What name would you like to use for the new workspace and initial project? hola-mundo
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
CREATE hola-mundo/angular.json (3598 bytes)
CREATE hola-mundo/package.json (1254 bytes)
...
...
The file will have its original line endings in your working directory
Successfully initialized git.
>
```

⚠️ Nota: Estos pasos son actualizados continuamente en cada versión de Angular.

- ▶ Primero nos preguntará el nombre del proyecto. También creará una carpeta y dentro es donde colocará todos los archivos y carpetas que se descargarán.
- ▶ Después nos pregunta si queremos añadirle el *routing*. Es algo que podemos crear manualmente después, pero si lo hacemos ahora nos ahorraremos el trabajo. Ya veremos detenidamente lo que es exactamente.
- ▶ Por último, nos pregunta que estilo de CSS queremos usar. Por defecto usa CSS, pero si queremos aplicar otras hojas de estilo, éste el momento de cambiarlo.
- ▶ Después empezará a descargar la estructura básica del proyecto hasta que aparezcan las dos últimas líneas que sale en el ejemplo, y nos devolverá a la línea de comandos.

¡¡Y listo!! Ya tendremos nuestro proyecto creado. Tan sólo nos queda abrir la carpeta con nuestro IDE favorito y a programar nuestro proyecto, aunque antes de tocarlo, vamos a ejecutarlo y ver su estructura básica de carpetas y archivos.

💡 Si al final del todo salen avisos con el mensaje "warning: LF will be replaced by CRLF", se soluciona escribiendo en la consola lo siguiente: `git config --global core.autocrlf false`

Ejecutar el proyecto Hola Mundo

Ahora que ya tenemos el proyecto creado, nos queda ejecutarlo. Para ello tendremos que escribir la siguiente línea de comandos:

```
cd hola-mundo Para entrar en la carpeta creada en el paso anterior. En el ejemplo le hemos llamado 'hola-mundo'. Sustitúyelo por el tuyo.  

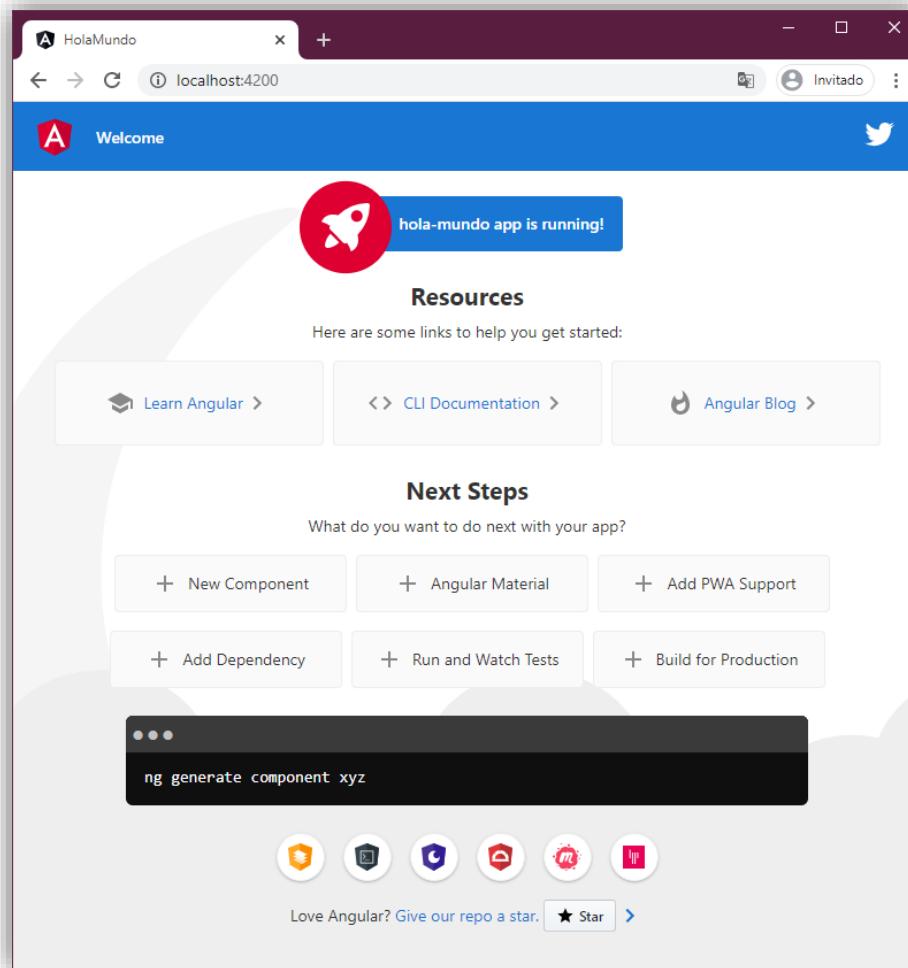
ng serve Así se arrancará el servidor de desarrollo ejecutando el proyecto.
```

```
Compiling @angular/core : es2015 as esm2015
Compiling @angular/compiler/testing : es2015 as esm2015
...
...
** Angular Live Development Server is listening on localhost:4200, open your browser on
http://localhost:4200/**
: Compiled successfully.
```

Ya tenemos el proyecto ejecutándose en un servidor para desarrollo, y para verlo sólo tendremos que abrir nuestro navegador preferido y escribir la url que nos ha indicado. En nuestro caso <http://localhost:4200/>.

 **Muy importante:** No cerrar esta ventana mientras queramos seguir ejecutando la aplicación.

Al abrir el navegador en esa url, nos mostrará una aplicación como ésta:



La aplicación no hace nada, son un conjunto de enlaces a la documentación oficial y un poco de ayuda sobre como añadir componentes, material, dependencias, etc. Ya lo veremos todo más adelante. Ahora mismo nos sirve para **comprobar que todo funciona correctamente** y ver la aplicación ejecutándose en vivo.

Ahora nos queda echarle un vistazo a lo que hay debajo del capó y ver la estructura de proyecto que nos creó AngularCLI y los archivos que contiene. Para ello, abriremos la carpeta *hola-mundo* en el IDE que queramos. En el tutorial usaremos *Visual Studio Code* (<https://code.visualstudio.com/download>).

 Se recomienda instalar la extensión [Angular Language Service](#), ya que nos proporciona una mejor experiencia en la edición de código y plantillas con Angular.

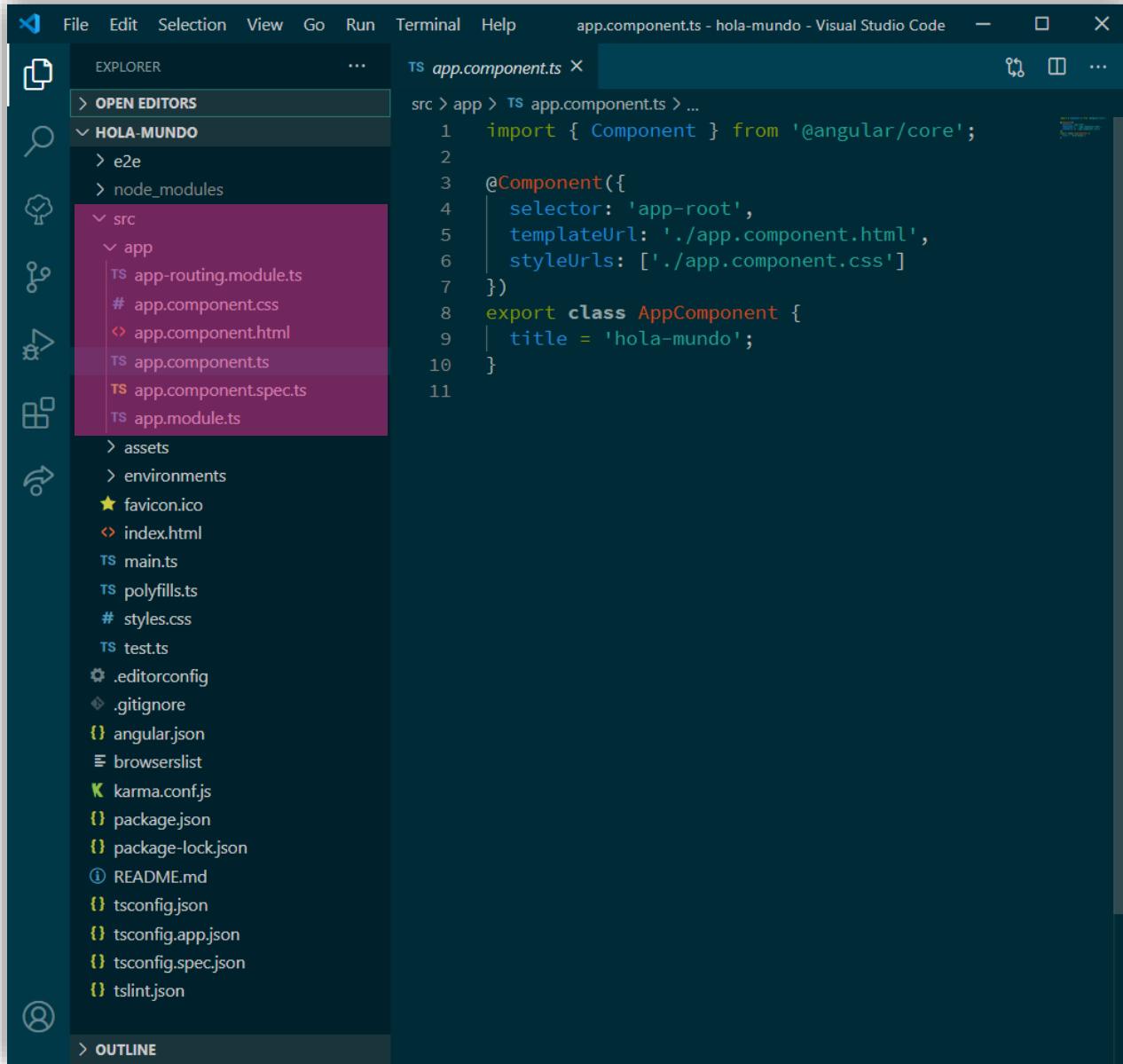
Puedes abrirlo también con la herramienta Stackblitz, editarlo, ver su contenido, juguetear, hacerle modificaciones en caliente y ver instantáneamente el resultado. Y todo sin miedo a romperlo, ya que no se guardarán los cambios. Lo usaremos a menudo en el tutorial 😊.



<https://stackblitz.com/edit/ng-ejemplo1-holamundo>

Estructura básica de un proyecto en Angular

Al abrir el proyecto con VSC verás que tiene la siguiente estructura básica:



The screenshot shows the Visual Studio Code interface with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Terminal:** app.component.ts - hola-mundo - Visual Studio Code.
- Explorer:** Shows the project structure under 'HOLA-MUNDO' (expanded) and 'src' (selected).
- Editor:** The file 'app.component.ts' is open, displaying the following code:

```

1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'hola-mundo';
10 }
11

```

Verás una gran cantidad de archivos y carpetas, así que vamos a centrarnos en los pocos archivos principales que forman parte de la aplicación y ya se irán viendo poco a poco el resto según nos vaya haciendo falta.

Ahora sólo nombraremos algunos aspectos básicos, como los módulos o los componentes, pero ya los detallaremos más adelante.

/: Carpeta raíz. La mayoría son json que definen propiedades de nuestro proyecto y configuran el entorno para diversas herramientas. Los archivos más representativos son:

- ▶ **angular.json**: Descriptor del proyecto.
- ▶ **package.json**: Descriptor de dependencias npm.
- ▶ **.gitignore**: Indica qué archivos y carpetas serán ignorados al subir el proyecto a GitHub.
- ▶ **.editorconfig**: Indica una configuración común que usará cualquier IDE que abra este proyecto.

/src: Es la carpeta más interesante, ya que será donde irá todo el código fuente del proyecto. No obstante, no será el resultado final ya que, al desplegar el proyecto, los resultados de compilar todos los archivos se llevarán a una carpeta llamada /dist.

- ▶ **index.html**: Es la página de inicio del proyecto. Aquí es donde empieza todo, y se irán cargando los componentes de la app.
- ▶ **styles.css**: La hoja de estilos global que se aplicará en la aplicación. Después veremos que cada componente tendrá su propio css.
- ▶ **main.ts**: El archivo principal y global de TypeScript que hace que cargue todo el resto de código. Aquí podremos fácilmente indicar cuando llegue el momento, que nuestro proyecto ya está en producción y no en desarrollo. No tenemos que modificar nada en principio.

/src/app: Contiene los archivos de los componentes de nuestra aplicación. Inicialmente nos viene creado un componente llamado **AppComponent**.

- ▶ **app-module.ts**: Es el módulo raíz de la app, el cual por convención tiene este nombre. Aquí es donde importaremos todos los demás módulos que vaya a usar nuestra app y donde se declararán los componentes, además del **AppComponent**,
- ▶ **app-routing.module.ts**: Este archivo se creará si elegimos la opción de “add Angular Routing” cuando hicimos el “ng new”. Aquí indicaremos las rutas de nuestra aplicación, es decir, le diremos al enrutador que cuando el navegador vaya a una determinada url, tendremos que mostrar un determinado componente. Por ejemplo: cuando vayamos a /hola-mundo/contacto, nos mostrará el componente de contactos.
- ▶ **app-component.***: Son los archivos HTML, CSS y TS del componente **AppComponent** que nos creó AngularCLI al construir el proyecto.

/src/assets: En esta carpeta es donde colocaremos todas las carpetas y archivos que queremos que estén en la app tal cual, sin compilar o minimificar. Podrán ser accesibles desde el navegador usando su ruta (por ejemplo miapp.com/assets/img/portada.jpg). Suelen ser recursos estáticos, como imágenes, sonidos, videos, documentos, etc. Un error común es colocar aquí la hoja de estilos css, pero su lugar correcto es en /src/styles.css.

/node_modules: Son los archivos descargados de las dependencias que hemos añadido con npm. Por lo que todas las librerías declaradas como dependencias en **package.json** estarán en esta carpeta, incluidas las propias de Angular. No tocaremos nada aquí. Esta carpeta rondará los 500mb en disco (tenerlo en cuenta si se ubica en una carpeta sincronizada con la nube).

Para más información sobre la estructura completa de un proyecto, consulta el [siguiente enlace](#).

Componentes

Una aplicación que usa Angular basa su desarrollo en base a componentes. Hasta la aplicación más básica deberá tener al menos un componente, que será el componente padre o raíz, del cual se le irán añadiendo más y más componentes hijos.

De la misma forma que en HTML existe la etiqueta `<body>` sobre la que vamos añadiendo más y más etiquetas, pero todas dentro del contenido de `<body> </body>`.

De hecho, lo que conseguiremos al crear un nuevo componente, será una “nueva” etiqueta HTML, que tendrá su propio código HTML, su propio CSS y un archivo TS que hará el papel de controlador y es dónde irá la lógica y el modelo de datos del componente, siguiendo el patrón MVC.

Los componentes son la pieza fundamental dentro de las aplicaciones en Angular, y por su estructura nos ofrecen las siguientes ventajas:

- ▶ **Organización:** Cada componente va en una carpeta, con sus archivos HTML, CSS y TS por separado.
- ▶ **Mantenibilidad:** Cuanto más organizado y separado esté el código, más fácil será encontrar errores, mejorarlo y hacerlo escalable.
- ▶ **Reutilización de código:** En cada componente elegimos su selector (nombre de la etiqueta), y podremos usar ese selector donde queramos en la app, tantas veces como queramos.

Entendiendo los componentes

En lugar de diseccionar el componente `AppComponent` que ya viene en nuestro proyecto `hola-mundo`, vamos a modificar el componente raíz y a crear un nuevo componente desde 0, en un nuevo proyecto un poco más simple para centrarnos en los componentes y abstraernos un poco del resto.

Nota: Al final de todo el proceso, tienes el enlace a stackblitz con el código fuente del ejemplo completo, pero intenta hacerlo poco a poco siguiendo las instrucciones.

1. Creamos un proyecto nuevo con `ng new`. Ahora sin routing.
2. Nos movemos a la carpeta y lo arrancamos con `ng serve`.
3. Lo abrimos con Visual Studio Code.
4. Abrimos el archivo `app.component.html`, borramos todo su contenido y colocamos algo más simple, como lo siguiente:

```
<!-- app.component.html -->
<div>
  <h1>Ejemplo 2 – Probando componentes</h1>
</div>
```

5. Comprobamos que, en lugar de la aplicación completa, nos aparecerá en el navegador solamente el contenido anterior. Esto es para simplificar un poco la aplicación anterior y centrarnos en los componentes. Nuestra aplicación ahora mismo consta de un componente inicial (o raíz) llamado “`AppComponent`” y que tiene como contenido esas líneas de HTML. También podríamos modificar el CSS en el archivo `app.component.css`.

Creando un nuevo componente

Hemos modificado un poco el componente raíz de toda aplicación Angular. Ahora vamos a ver los pasos para crear un nuevo componente.

1. Tendremos que salir a la línea de comandos y situarnos en la carpeta de nuestro proyecto.
2. Escribimos lo siguiente:

```
ng g component
```

Así iniciaremos el asistente para generar (g de generate) un nuevo componente, mostrando lo siguiente:

```
? What name would you like to use for the component? saludo
CREATE src/app/saludo/saludo.component.html (21 bytes)
CREATE src/app/saludo/saludo.component.spec.ts (628 bytes)
CREATE src/app/saludo/saludo.component.ts (275 bytes)
CREATE src/app/saludo/saludo.component.css (0 bytes)
UPDATE src/app/app.module.ts (407 bytes)
```

>

3. Ya tenemos el componente creado. Sólo hace una pregunta, y será el nombre del componente. Introducimos `saludo` y pulsamos intro.

Nota: Es una buena idea colocar todos los componentes dentro de una misma carpeta, por lo que se puede incluir como parte del nombre y AngularCLI nos creará esa carpeta en caso de no existir. Podríamos responder que el nombre será `components/saludo`. De esta forma me crea una carpeta llamada `components`, dentro de ella otra llamada `saludo`, y dentro los archivos que forman el componente. Todos los sucesivos componentes lo colocaremos en la misma carpeta y así estará nuestro proyecto más organizado. Lo haremos en proyectos más grandes con muchos componentes, pero por ahora no. Todo esto es subjetivo y obviamente cada uno es libre de organizar su aplicación como vea oportuno.

4. Podemos observar que ha creado 4 archivos en `src/app/saludo`:

- ▶ `saludo.component.html`: La plantilla de la vista HTML.
- ▶ `saludo.component.css`: El css del HTML.
- ▶ `saludo.component.ts`: Dónde irá la lógica del componente.
- ▶ `saludo.component.spec.ts`: Para el testing. No lo tocaremos.

5. Por último, también actualizó el `app.module.ts` para registrar el nuevo componente.

 **Nota:** Cuando se incluya código como en el cuadro siguiente, se incluirá el nombre del archivo como un comentario del lenguaje que sea, para que no exista errores al copiar y pegar 😊.

```
//app.module.ts
import { SaludoComponent } from './saludo/saludo.component';
@NgModule({
  declarations: [ AppComponent, SaludoComponent ],
  ...
})
```

Ya podemos cerrar la consola y volver a VSC. Vamos a ver y a entender el código de los archivos nuevos creados. Obviaremos el CSS (que está vacío) y el spec.ts (ya que no haremos pruebas por ahora).

```
<!-- saludo.component.html -->
<p>saludo works!</p>
```

```
//saludo.component.ts
import { Component, OnInit } from '@angular/core';

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

  constructor() { }
  ngOnInit(): void { }
}
```

En todos los archivos TS de un componente veremos unas partes comunes, que son:

- ▶ **Imports.** Aquí se añadirán librerías y otras funciones que vayamos necesitando.
- ▶ **@Component:** Es un decorador (lo veremos más adelante) para indicar:
 - **selector:** El selector HTML que usaremos para poner el componente en la plantilla HTML. Aquí se indica que es `app-saludo`, por lo que en el HTML tendremos que poner `<app-saludo> </app-saludo>`.
 - **templateUrl:** La plantilla HTML en un archivo externo (lo más normal).
 - **styleUrls:** Un array que contiene una lista con todos los archivos css externos para esta vista. Los estilos se aplican exclusivamente a esta vista y no afecta a las demás.
- ▶ **Clase NombreComponent:** Aquí se define la clase, con sus atributos y métodos. Esta clase tendrá la función del controlador de MVC. Ya hay un método por defecto que es el `ngOnInit()`. Este método se lanzará automáticamente al iniciar el componente y podemos eliminarlo si no lo necesitamos (el método, el `import` y el `implements`).

El código visto es el generado por el AngularCLI, y será usado en la mayoría de las ocasiones, pero hay muchas variantes, pudiendo quedar el código un poco más simplificado como en el siguiente ejemplo, en el que hemos creamos un nuevo componente llamado “*verificacion*”. Este componente sólo tiene el archivo TS, ya que al tener incluido el HTML y CSS, hemos borrado sus archivos.

```
//verificacion.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-verificacion',
  template: ` <h1>Petición aceptada</h1>
    <p>Su correo ha sido eliminado de nuestra base de datos</p>
    <p>No volverá a recibir más spam</p> `,
  styles: ['h1 { color:green; }']
})
export class VerificacionComponent {}
```

Usando los componentes

Hemos entendido lo que es un componente, y hemos creado y modificado uno. Ahora nos queda saber "colocar" el componente en nuestro proyecto.

Si nos fijamos, en el `src/index.html`, el código fuente es el siguiente:

```
<!-- index.html -->
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Ejemplo2Componentes</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Este el documento inicial que carga la app, no hay etiquetas `<link>` para los CSS ni `<script>` para JS. Pero ya hemos visto donde han de situarse tanto los estilos como el código. De inyectarlos ya se encarga el **core** de Angular y es algo que no nos tenemos que preocupar, es más, va cambiando la forma de hacerlo en las distintas versiones de Angular.

De esta forma tenemos un código mucho más limpio y fácil de entender. Fíjate que en `<body>` sólo hay una etiqueta `<app-root>`, y es el selector del componente raíz incluido en la app.

Al incluir el selector, Angular ya se encarga de inyectar ahí la plantilla HTML de dicho componente usando los estilos propios si los hubiese. Nunca escribiremos nada en el contenido de las etiquetas de los componentes, aunque si veremos más adelante que podremos hacer más cosas añadiéndole atributos.

Veamos el código HTML del componente raíz (`app.component`) y saludo (`saludo.component`), una vez añadimos el código a ambos.

```
<!-- app.component.html -->
<div>
  <h1>Ejemplo 2 - Probando componentes</h1>
</div>

<hr/>
<app-saludo></app-saludo>
```

```
<!-- saludo.component.html -->
<p>saludo works!</p>
```

Por lo que en nuestro servidor de desarrollo la app se muestra así:



Hasta aquí es bastante simple, son como HTML y CSS separados en etiquetas modulables. Pero vamos a añadir un poquito de código y salsa al asunto.

Le añadiremos un botón al componente **saludo** con un texto asignado desde código que muestre una alerta simple con un texto variable. Para simplificar, pegaremos sólo la clase, lo demás sigue intacto (importaciones y el @Component).

```
//saludo.component.ts
export class SaludoComponent implements OnInit {
    //Atributos
    idioma = "EN";
    textoBoton = "Click";

    //Métodos
    constructor() { }
    ngOnInit(): void { }

    pulsarBoton() {
        if (this.idioma === "ES") {
            alert("He hablado.");
        } else {
            alert("I have spoken.");
        }
    }
}
```

Y al HTML le añadimos el botón y traducimos el texto que venía por defecto:

```
<!-- saludo.component.html -->
<p>Componente saludo funciona!!!</p>
<p>
    <button (click)="pulsarBoton()">{{textoBoton}}</button>
</p>
```

Si lo ejecutamos, veremos que, al pulsar el botón, se ejecuta la función `pulsarBoton()` y ésta es la que decide si mostrar el texto en inglés o español.

Ahora toca comentar ciertas partes del código:

⚠ En el HTML, así es como Angular enlazamos eventos a una función, con la propiedad `(click)`. Muy parecido al JavaScript nativo. Ya explicaremos esto más adelante.

⚠ En el HTML, el texto del botón es una variable que hemos definido en el código del componente, y para poder usarla desde HTML, usamos la doble llave `{{ }}`. Esta sintaxis especial lo que hace es **evaluar la expresión** que hay entre los signos, como si estuviésemos con código, por lo que se pueden usar variables definidas en la clase y en tiempo de ejecución se sustituye la expresión por su valor. `{{1+1}}` es una expresión, y sería sustituido por `2`. Una variable es una expresión, que al evaluarse en sustituido por su valor. Por lo que si ponemos `{{textoBoton}}`, se sustituye por `"Click"`. Es muy importante entender que lo que va entre los dobles corchetes es una expresión. Se pueden usar en cualquier parte del HTML, también como valores de los atributos normales (ej: ``).

Las expresiones son tan potentes que podríamos hacer esto:

```
<button>{{idioma==='ES' ? "Púlsame": "Click me" }}</button>
```

⚠ En las clases, intentaremos poner primero los atributos (variables de la clase), y después los métodos (incluido el constructor y demás métodos que vengan por defecto). Aunque se pueda hacer en cualquier orden, es una mala práctica hacerlo de cualquier otra forma.

⚠ Como se hace en Java, no deberíamos inicializar los atributos directamente en su declaración, si no usar su constructor.

⚠ Para ser más estrictos y seguir con la notación de Java, se puede y debe especificar el tipo de datos de las variables, y del valor devuelto por los métodos (funciones de la clase).

⚠ Como se hace en Java, los atributos y métodos también tienen sus modificadores de acceso. Por ahora usaremos `public` y `private` según nos convenga.

⚠ Para hacer referencia a cualquier miembro (atributo o método) dentro de la propia clase, usaremos el identificador `this`. Aunque a veces no sea necesario, siempre queda el código más legible y menos confuso (y si venimos de Java, mucho mejor, ya que seguimos con la costumbre).

Antes de hacer caso a las recomendaciones anteriores y ver el código final, vamos a observar las ventajas de todo lo que acabamos de hacer:

💡 Cada componente tiene sus propios atributos y métodos, perfectamente agrupados, de una forma muy parecida a como se agrupan las clases Java, sólo que aquí además tienen su propio HTML y CSS.

💡 Podemos acceder fácilmente a los atributos y métodos desde HTML usando `{{...}}`.

💡 Nos “obliga” a usar el patrón de diseño MVC, no mezclando el código con la vista. Si tenemos algo que hacer con código, lo haremos desde el archivo TS, aunque tengamos la posibilidad de hacerlo desde HTML.

 Podemos reutilizar cada componente en cualquier otra parte de la aplicación, tan sólo escribiendo su selector.

Siguiendo las recomendaciones, la clase anterior quedaría de la siguiente forma:

```
//saludo.component.ts
export class SaludoComponent implements OnInit {
    //Atributos
    public idioma: string;
    public textoBoton: string;

    //Métodos
    constructor() {
        this.idioma = "EN";
        this.textoBoton = (this.idioma === "ES") ? "Púlsame": "Click me";
    }

    ngOnInit(): void { }

    public pulsarBoton(): void {
        if (this.idioma === "ES") {
            alert("He hablado.");
        } else {
            alert("I have spoken.");
        }
    }
}
```

TypeScript es muy flexible y dinámico y aunque siempre hagamos el código de la misma forma (o parecida), deberemos tener siempre en cuenta que otros compañeros, documentación o ejemplos que encontremos por internet pueden hacerlo de una forma distinta a la nuestra. Nosotros siempre intentaremos hacer uso de las buenas prácticas.



<https://stackblitz.com/edit/ng-ejemplo2-componentes>

Refuerzo 1

Hacer un proyecto nuevo, con un componente que muestre un cuadro de texto de tipo `password` y un botón que, al pulsarlo, haga que se muestre el contenido del cuadro de texto. Al volverlo a pulsar se volverá a ocultar.

 **Nota:** Hazlo sin mirar el resultado del código final. Si te atascas pide ayuda. Como dijo alguien sabio: "Transmite lo que has aprendido, fuerza, maestría... pero insensatez, debilidad y fracaso, sobre todo. El mejor profesor el fracaso es".



<https://stackblitz.com/edit/ng-refuerzo1-password>

Vistas

Los componentes de Angular forman la estructura de datos de una aplicación. La plantilla HTML asociada a un componente proporciona los medios para mostrar esos datos en el contexto de una página web. Juntos, la clase y la plantilla de un componente forman **una vista** de los datos de nuestra aplicación. Se muestran los datos al usuario y se vinculan controles HTML (botones, cuadro de texto, etc.) de la plantilla a las propiedades (atributos y funciones) de la clase del componente.

El proceso de combinar los valores de los datos con su representación en la página se denomina enlace de datos (binding).

Tipos de binding

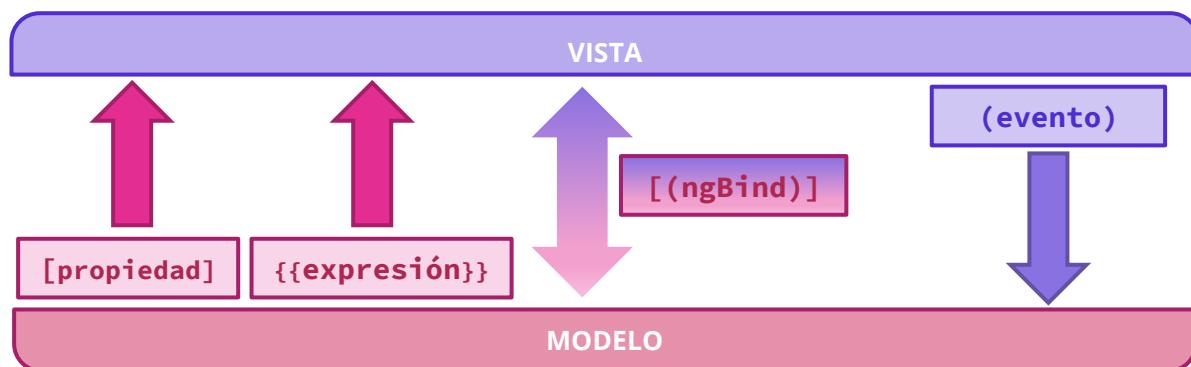
Vemos una primera aproximación de los tipos de enlaces que pueden existir entre el modelo y la vista y después procederemos a verlas con más detenimiento. También se incluye el nombre y enlace a la documentación oficial de Angular:

- ▶ **Propiedad (*property binding*)**: Es cualquier valor que podemos asignar por medio de un atributo HTML, ya sean estándar, o del propio Angular.
- ▶ **Expresión (*interpolation*)**: Es un “volcado” de información en el texto de la página. Angular evaluará el resultado de dicha expresión y será sustituida por su valor final.
- ▶ **Eventos (*event binding*)**: Es un suceso que ocurre (normalmente por acciones del usuario, como pulsar un botón o una tecla) y para el cual se suele asociar a un disparador, que puede ser una función que se ejecutará al lanzarse el evento. Hay otros eventos que no los produce el usuario, como un temporizador, cuando la página termina de cargarse, o cuando se reciben los datos de una petición AJAX.
- ▶ **Doble enlace (*two-way binding*)**: Es un enlace doble entre el modelo y la vista. Al establecer dicho enlace, si un dato cambia en el modelo ese cambio se refleja automáticamente en la vista y viceversa.

Esquema de binding entre vista-modo

El problema del doble binding es que es costoso para el sistema, y puede llegar a ralentizar una app compleja, por lo que ahora los desarrolladores tienen más opciones para enlazar datos en una sola dirección ahorrando recursos del sistema y ganando en rendimiento.

Veamos un diagrama con un resumen del flujo de información entre la vista y el modelo, y la sintaxis para realizarlos:



Ya que tenemos una idea de las partes, y el tipo de enlace que tienen entre el modelo y la vista, veremos con más detenimiento cada uno de ellos, con ejemplos y sintaxis:

Propiedades [...]

Las **propiedades** tienen un flujo desde el modelo hacia la vista. Una información que tengamos en el modelo se puede asignar a una propiedad del HTML, usando los corchetes. Ejemplos:

```
<a [href]="homeURL">Pulsa aquí para volver</a>
```

El enlace nos llevará a la url que haya en la variable `homeURL` que estará en el componente

```
<p [class]="estado">¡Todo correcto!</p>
<p [class.success]="esCorrecto">¡Todo correcto!</p>
<p [class]="arrayClases">¡Todo correcto!</p>
```

Al primer ejemplo, le aplica como clase, la variable `estado`, que hemos decidido desde código que su valor puede ser "correcto" o "error", dependiendo de cómo queramos mostrar el mensaje.

Segundo ejemplo, se le aplica la clase `success` al párrafo, cuando la expresión enlazada `esCorrecto` sea verdadera, y la elimina si es falsa.

En el tercer ejemplo aplicamos una variable con valor `arrayClases=[“btn”, “btn-danger”, “btn-sm”]` y se aplican las 3 clases al párrafo. Más opciones todavía en la [documentación oficial](#).

```
<button [hidden]="!visible">Ver/Ocultar</button>
```

Al botón, le damos una propiedad `hidden` y su valor será el que tenga la variable `visible` del modelo (pero negado, ya que tiene el operador NOT delante).

Las propiedades anteriores `class` y `href` son de HTML estándar, sin embargo, `hidden` es propia de Angular. Ya veremos más adelante como crear nuestras propias propiedades para personalizar los componentes.

Expresiones {{...}}

Las **expresiones** viajan desde el modelo hacia la vista. La diferencia respecto a las propiedades es que se usarán en mayor medida como contenido de los elementos HTML, no como propiedad o su valor. Se expresan con doble llaves. Ejemplos:

```
<h1>{{titulo}}</h1>
```

El contenido del encabezado `h1`, se sustituirá por el valor de la variable `titulo`.

```
<p>Precio total: {{ precio + iva }} €</p>
```

Se sustituye las `{{}}` por el resultado de sumar las dos variables. El resto se queda tal cual.

```
<a href="{{homeURL}}>Pulsa aquí</a>
```

Mismo resultado que usando la propiedad `[href]`. Aunque más adelante veremos cómo en algunos casos es más conveniente el uso de las propiedades en lugar de las expresiones. No hay razones técnicas para elegir uno frente a otro. Sin embargo, cuando queramos establecer a un elemento HTML propiedades que NO SEA de tipo String, deberás usar el binding de propiedad en lugar de expresiones.

Eventos (...)

Los **eventos** viajan desde la vista hacia el modelo. Se indica entre paréntesis el nombre del evento y como valor la función que se debe ejecutar entre comillas. Al lanzarse el evento, Angular se encarga de hacer una llamada a la función definida. Ejemplos:

```
<button (click)="interruptor()">Ver/Ocultar</button>
```

Al hacer `click` sobre el botón, se hará una llamada a la función `interruptor()`.

```
<p #parrafo (click)="procesar(parrafo)" />
```

Además, también podemos pasar un elemento HTML como parámetro a una función usando su referencia (`#parrafo`). En el código de la función tendremos un objeto llamado `parrafo` y que tendrá todos los atributos del HTML (`parrafo.style.color`, `parrafo.className`, `parrafo.innerHTML`, etc.).

```
<button (dblclick)="contador=0">Reset</button>
```

Se puede escribir directamente una instrucción básica, una asignación, o algo simple, pero se recomienda encapsular la lógica en una función, para hacer la aplicación más escalable.

```
<input type="text" (keyup)="comprobar($event)" />
```

Al levantar una tecla en el cuadro de texto, se lanza la función `comprobar()` pasándole el evento como parámetro, si es que lo necesitamos. `$event` es un objeto que tiene toda la información del evento que se produjo, por ejemplo, la tecla que se pulsó, su código ASCII, etc. La información cambiará dependiendo del evento que se produzca, por lo que se recomienda imprimirlo por consola y comprobar su estructura antes de trabajar con él.

Binding [...]

De esta forma, los datos viajan desde el modelo de datos hacia la vista y viceversa, por lo que cualquier cambio en un lado se reflejará en el otro. Por ejemplo: Si se enlaza una variable (definida en la clase del componente) con un cuadro de texto (definido en la plantilla HTML del componente), al modificar la variable desde código, su valor se verá actualizado en el cuadro de texto, y si además modificamos el valor desde el cuadro de texto, el cambio se asignará a la variable.

La notación es curiosa, porque realmente hace referencia a que se usa a la vez el enlace de propiedades (con los corchetes), y el de eventos (con los paréntesis) conjuntamente.

 `[]()` = Banana in a box

Angular te aconseja que visualices una banana en una caja, para que recuerdes que primero van los corchetes (caja) y dentro los paréntesis (banana).



Antes de poder usar el doble binding, o también llamado two-way data binding, debemos importar el módulo **FormsModule** en nuestro archivo `app.module.ts`, dejando el resto como estaba:

```
// app.module.ts
...
import { FormsModule } from '@angular/forms'; // ----- Aquí...
@NgModule({
  ...
  imports: [
    BrowserModule,
    FormsModule // ----- ...y aquí
  ],
  ...
})
export class AppModule { }
```

Ahora con el **FormsModule** importado, ya tenemos disponible la directiva **ngModel**, y si podemos usar el two-way data binding en cualquier parte de nuestra aplicación, siguiendo la sintaxis del siguiente ejemplo:

```
// clase en el código del componente
export class DobleEnlaceComponent {
  contador = 0;
  suma() { this.contador++; }
  resta() { this.contador--; }
}
```

```
<!-- HTML del componente -->
<input type="text" [(ngModel)]="contador" />
<button (click)="suma()">+</button>
<button (click)="resta()">-</button>
<button (click)="contador=0">Reset</button>

<h3>El valor de contador es {{contador}}</h3>
```

Si probamos el ejemplo, podremos modificar el atributo `contador` de dos formas, mediante código (usando los botones), o escribiendo directamente en el `<input>`.

Al usar la expresión `{{contador}}` comprobaremos que cualquier modificación se ve reflejado en la variable instantáneamente, sea de una forma o de otra.

💡 No siempre es necesario importar el módulo `FormsModule` para usar el doble binding, pero es una forma "simple" de ver esta capacidad interesante de Angular antes de conocer los `@Input` y `@Output`.



<https://stackblitz.com/edit/ng-ejemplo3-banana-in-a-box>

Ejercicios (binding y componentes)

 **Demo1:** Hacer un proyecto en Angular, que muestre en el app.component.html un párrafo. El contenido de ese párrafo será una variable string definida en el app.component.ts. Añade más elementos h1, h2, h3, h4, h5, h6 y un botón. Debe salir el mismo texto en todas las etiquetas. Prueba a modificar los estilos, añadiendo un color de texto distinto al negro.

 **Demo2:** Hacer un proyecto en Angular, que muestre en el app.component.html 3 elementos <div>. En app.component.css le aplicamos los estilos siguientes:

```
div {
    background-color: cornflowerblue;
    width: 100px;
    height: 100px;
    float: left;
    margin: 1rem;
    transition: all 300ms;
}
```

En app.component.ts crear lo siguiente:

- Una variable (atributo) para el ancho y otra para el alto.
- Hacer dos métodos, masAlto() y masAncho(). Al ejecutar estos métodos, sumaremos 10 al valor correspondiente.

En app.component.html hacer lo siguiente:

- Enlazar las variables del componente con el atributo `style.width` y `style.height` para que al modificar la variable, se aplique el estilo en las vistas. [Consulta aquí](#) para ver como se hace. Es importante consultar frecuentemente la documentación oficial de Angular.
- Crear dos botones, llamados "Más Alto" y "Más Ancho" que, al hacer clic en ellos, hagamos una llamada a cada uno de los métodos anteriores, respectivamente.
- Hacer otro botón que ponga el tamaño de los divs a su valor original, pero ahora, sin usar ningún método (no es lo correcto, pero para practicar todas sus formas).
- ¿Cuántos tipos de binding hemos usado?
- 😊 ¿Te atreves a generar un color aleatorio para cada div?

 **Demo3:** Hacer un proyecto en Angular que además del app.component, tenga un nuevo componente con los siguientes elementos HTML:

- Un cuadro de texto para introducir un peso.
- Un cuadro de texto para introducir una altura.
- Un párrafo dónde se colocará el resultado de calcular el índice de masa corporal (IMC) con los valores introducidos.
- Un botón que iniciará una función para hacer los cálculos anteriores.

Tener en cuenta:

- El componente tiene un estilo distinto del resto de la aplicación. Aplicarle un color de fondo y texto distinto desde el sitio correcto.

Decoradores

Cuando vimos los componentes, vimos por encima lo que era un decorador. `@Component` es un decorador.

Un decorador es una implementación de un patrón de diseño que sirve para ampliar una función mediante otra función, pero sin tocar la original. El decorador recibe una función como argumento (la que queremos "decorar") y devuelve la misma función, pero con alguna funcionalidad adicional.

Las funciones decoradores empiezan por una `@` y le sigue un nombre. El nombre debe hacer referencia a lo que queramos decorar, ya que debe existir previamente. Podemos decorar una función, una propiedad de una clase, una misma clase, etc.

Detengámonos un poco en el decorador que conocemos dentro de `app.component.ts`:

```
//app.component.ts
import { Component } from '@angular/core';

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

Con el `import`, importamos la clase `Component` de una librería (`core`), y con el `@Component`, le indicamos unos parámetros al decorador, como el selector del componente, y la ubicación de los archivos HTML y CSS. Puede recibir más parámetros como vimos (`template`, `styles`, etc.).

Una de las razones por las que Angular tomó TypeScript como lenguaje es por permitir el uso de decoradores. En JavaScript es algo que pronto podrá hacerse ya que es una de las propuestas para formar parte del estándar ECMAScript 2016 (*aka* ES7), pero que ya están disponibles en TypeScript.

 En Angular, escribimos todo el código en archivos TypeScript (*.ts), pero con "ng serve" se compila y transforma todo a código JavaScript (*.js), que es el único que entienden los navegadores por ahora.

Módulos

Un módulo es una herramienta para organizar mejor el código y los componentes de una aplicación. Se puede decir que un módulo agrupa uno o varios componentes.

La aplicación creada por AngularCLI ya tiene un módulo raíz creado (`app.module.ts`), por lo que podemos añadir los componentes que necesitemos y listo. Pero si nuestra aplicación empieza a crecer demasiado y empieza a acumular decenas de componentes, importaciones de librerías, pipes, directivas, etc., el módulo principal puede ser demasiado grande y estará todo declarado en el mismo archivo.

Así que, para aplicaciones más complejas, se pueden crear módulos y agregarles los componentes a los módulos creados, en lugar de todos al principal. El orden se realizará de la manera más lógica posible, siempre dependiendo del modelo de negocio, o de las preferencias de cada organización.

Para ver como se integra todo (e ir mecanizando los procesos por repetición), **vamos a crear un proyecto nuevo** como hemos visto hasta ahora, y sigue los pasos para crear el módulo (listados) y añadirle nuevos componentes (cabecera, cuerpo y pie).

Creando un nuevo módulo

Siempre tenemos un módulo raíz en nuestra app. Será la que llamamos `app.module`. Vamos a ver los pasos para crear nuevos módulos para organizarnos mejor.

1. Hay que parar la ejecución del servidor si estuviese en marcha.
2. Tendremos que salir a la línea de comandos, situarnos en la carpeta de nuestro proyecto y escribir lo siguiente:

```
ng g module
```

Así iniciaremos el asistente para generar un nuevo módulo, mostrando lo siguiente:

```
? What name would you like to use for the NgModule? listados
CREATE src/app/listados/listados.module.ts (194 bytes)
>
```

3. Ya tenemos el módulo creado. Sólo hace una pregunta, y será el nombre del componente. Introducimos `listados` y pulsamos intro.
4. Ya podemos iniciar el servidor de desarrollo de nuevo (`ng serve`) y volver a VSC.

💡 No importa si ponemos mayúsculas o minúsculas. El CLI usará la nomenclatura correcta para cada caso, usando las minúsculas para las carpetas y la primera en mayúscula para el nombre de la clase. Hasta incluso si a un componente le indicamos en camelCase (p.e: `miComponente`), AngularCLI llamará a sus archivos `mi-componente.*` y la clase se llamará `MiComponente` (PascalCase).

💡 Cuando creamos un módulo o un componente, podemos indicarle en la CLI el nombre que queremos, y así no nos lo pedirá el asistente (`ng g module listados`).

💡 Escribiendo `ng serve --open` abrirá automáticamente el explorador al terminar 😊.

💡 `g` es un alias de `generate`. También puedes hacer "`ng g m`". Puedes consultar la lista completa de comandos y sus alias disponibles para AngularCLI en <https://angular.io/cli>

Nos ha creado en nuestro proyecto una carpeta `/src/app/listados` y dentro, el archivo `listados.module.ts`. Vemos el contenido completo a continuación:

```
//listados.module.ts
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
@NgModule({
  declarations: [],
  imports: [CommonModule]
})
export class ListadosModule { }
```

Es prácticamente igual que el módulo raíz que incluye toda app creada con AngularCLI. Una vez creado el nuevo módulo, tendremos que importarlo en el módulo raíz para poder usar todos los componentes que después creemos. Para ello añadimos lo siguiente al archivo `app.module.ts` (lo que estaba lo dejamos):

```
//app.module.ts
import { ListadosModule } from './listados/listados.module'; // <- Esto
@NgModule({
  declarations: [...],
  imports: [
    BrowserModule,
    ListadosModule // <- Y esto.
  ],
})
```

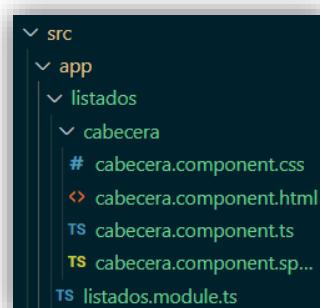
Añadir componentes al módulo nuevo

Hasta ahora todos los componentes lo añadíamos directamente al módulo principal. Para agregar un componente a un módulo, sólo tenemos que indicarle la carpeta del módulo, y seguidamente el nombre del componente.

```
ng g component listados/cabecera
```

Esto nos creará una carpeta llamada `cabecera` dentro de la carpeta `listados` con los archivos propios del componente. También añadirá ese componente en el código de `listados.module.ts`.

Ahora todos los componentes que coloquemos ahí pertenecerán al módulo de `listados` y todas las dependencias o librerías de terceros que vayamos a usar en esos componentes, se declararán en `listados.module.ts` en lugar de `app.module.ts` como veníamos haciendo hasta ahora.



Para cuando la aplicación crece más de la cuenta, separarla en módulos es indispensable.

AngularCLI hace algunas modificaciones en los archivos ya existentes, pero otras tendremos que hacerlo nosotros. Para que podamos usar el componente nuevo en el módulo raíz (app), tendremos que añadir unas líneas al decorador de `listados.module.ts`:

```
//listados.module.ts
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
//Este import lo ha injectado CLI al crear el componente
import { CabeceraComponent } from './cabecera/cabecera.component';

@NgModule({
  declarations: [CabeceraComponent], //Esto también lo puso CLI
  imports: [CommonModule],
  exports: [CabeceraComponent]    //----- Esta línea debemos añadir!!
})
export class ListadosModule { }
```

Al indicarle que podemos exportar el componente, ya me permitirá usarlo desde otros módulos. Esto es sólo si queremos usarlo en otros módulos, si no queremos, pues no lo exportamos.

Para usar el componente, tan sólo tendremos que usar su selector como siempre, pero el componente `cabecera` pertenece al módulo de `listados.module.ts`, y no a `app.module.ts`.

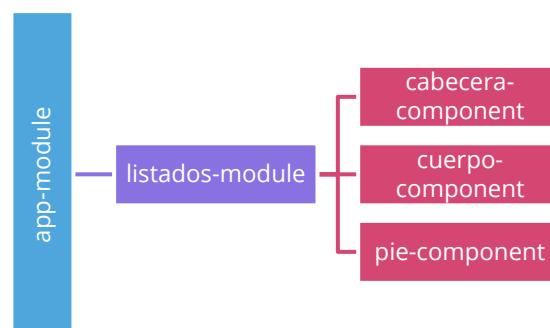
```
<!-- app.component.html -->
<h1>Encabezado de App</h1>
<app-cabecera></app-cabecera>
```



<https://stackblitz.com/edit/ng-ejemplo4-modulos>

Esquema módulos y componentes

Toda aplicación en Angular parte de un módulo base llamado `app-module`. Ahí podremos añadir más módulos, y a cada módulo más componentes. Los componentes a su vez pueden estar separados en carpetas.



Sumario

Repetimos resumidamente los pasos para crear un módulo nuevo, añadirle un componente, y poder usar el módulo nuevo en el `app.module`:

1. Crear el nuevo módulo.
2. Crear el/los componente(s) dentro del módulo.
3. Repetimos el punto 2 si queremos añadir más componentes al mismo módulo.
4. Importar el módulo nuevo en el `app.module.ts`, junto con los otros módulos ya existentes (`BrowserModule` y `NgModule`) y en el `imports: []` del decorador.
5. Si queremos usar los componentes del módulo nuevo en otros, hacer el `exports: []` en el `nuevo-modulo.module.ts` de los componentes a compartir.

Para que entendamos mejor el concepto de modularidad de Angular, imagina que necesitamos usar [Bootstrap](#) y la librería [SweetAlert2](#), pero sólo para los componentes que añadamos al módulo de **listados**, porque no los necesitamos en el resto de módulos. Pues todos esos import de las librerías se hacen en `listados.module.ts` en lugar del `app.module.ts` (el cual afecta a toda la app), teniendo nuestro código mucho más organizado y seguro.

Imagina que el día de mañana, queremos usar la siguiente versión de Bootstrap, o la nueva de SweetAlert, la ventaja es que sólo estaremos manipulando el módulo de listados, y el resto de la app queda intacto y libre de nuevos errores que podamos cometer.

Proyecto Héroes

Hasta ahora, hemos aprendido conceptos teóricos, como las partes de un proyecto, estructura de archivos o *scaffolding* (andamiaje), binding, componentes, etc.

Ahora empezaremos a ver cosas más entretenidas y para ello no crearemos proyectos nuevos para probar cada cosa, sino que lo iremos viendo todo sobre un mismo proyecto.

La idea es crear un proyecto simple capaz de gestionar una lista de superhéroes con sus habilidades y ya veremos poco a poco en lo que se convertirá.

Al turrón:

1. Para empezar, creamos un proyecto nuevo, lo llamamos "heroes", y ahora si le añadiremos el routing, que pronto empezaremos a darle uso.
2. **Vamos a añadir Bootstrap** (junto con sus dependencias) al proyecto, para ello, **entramos en la carpeta del proyecto** y escribimos lo siguiente en la CLI:

```
npm install bootstrap jquery @popperjs/core
```

 Con "bootstrap" se descargará la última versión estable que haya en el momento. Para instalar versiones beta, usar bootstrap@next.

Una vez descargado, ya podemos abrir el proyecto con VSC y abrimos `angular.json` y le añadimos el archivo .css a los objetos "**styles**", y los .js a "**scripts**", respectivamente:

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"],
"scripts": [
  "node_modules/jquery/dist/jquery.min.js",
  "node_modules/@popperjs/core/dist/umd/popper.min.js",
  "node_modules/bootstrap/dist/js/bootstrap.min.js"]
```

Importante: en el objeto styles, primero se cargará bootstrap, y después nuestros estilos, para que los nuestros tengan prioridad sobre bootstrap.

3. Creamos la carpeta `src/app/componentes`. Y creamos en ella 3 componentes. **Cabecera**, **Cuerpo** y **Pie**. (Recuerda, `ng g component componentes/cabecera`, etc.).
4. Ahora ya podemos arrancar el proyecto con "`ng serve`". Le añadiremos más librerías en un futuro, pero por ahora suficiente.
5. Creamos una carpeta para las imágenes en `src/assets/img` y nos descargamos alguna para poder en la cabecera de la página.
6. Para empezar con el HTML, primero modificamos el componente base (`app.component.html`) sobre el que pondremos la llamada de los 3 componentes que forman la página. Lo hacemos así para poder ir viendo el resultado mientras vamos modificando los componentes. Nos queda una página principal bastante simple.

```
<!-- app.component.html -->
<app-cabecera></app-cabecera>
<app-cuerpo></app-cuerpo>
<app-pie></app-pie>
```

7. Y ahora seguimos con el HTML de cada uno de los componentes, y si abrimos el servidor de desarrollo podemos ir viendo en tiempo real como va quedando la página, sin recargas.

```
<!-- cabecera.component.html -->
<header>
  <h1 class="jumbotron text-center bg-dark text-light">Superhéroes</h1>
</header>
```

Nota: la clase jumbotron no existe en bootstrap 5.0. Puedes usar display-1 para mostrar el texto más grande en su lugar

```
<!-- cuerpo.component.html -->
<main>
  <section class="container">
    <!-- 1 fila, 3 columnas -->
    <div class="row">
      <!-- columna izquierda -->
      <div class="col"></div>

      <!-- columna central -->
      <div class="col-4">

        <!-- Tarjeta -->
        <div id="tarjeta" class="card sombra">
          
          <div class="card-body">
            <h5 class="card-title">Lista de héroes</h5>
            <div class="card-text">
              <p>Aquí mostraremos el contenido de la lista de héroes</p>
            </div>
          </div>
        </div>

      </div>

      <!-- columna derecha -->
      <div class="col"></div>
    </div>
  </section>
</main>
```

```
<!-- pie.component.html -->
<footer id="pie" class="fixed-bottom text-center bg-dark text-light">
  <div>
    <p>&copy; Superhéroes en Angular</p>
  </div>
</footer>
```

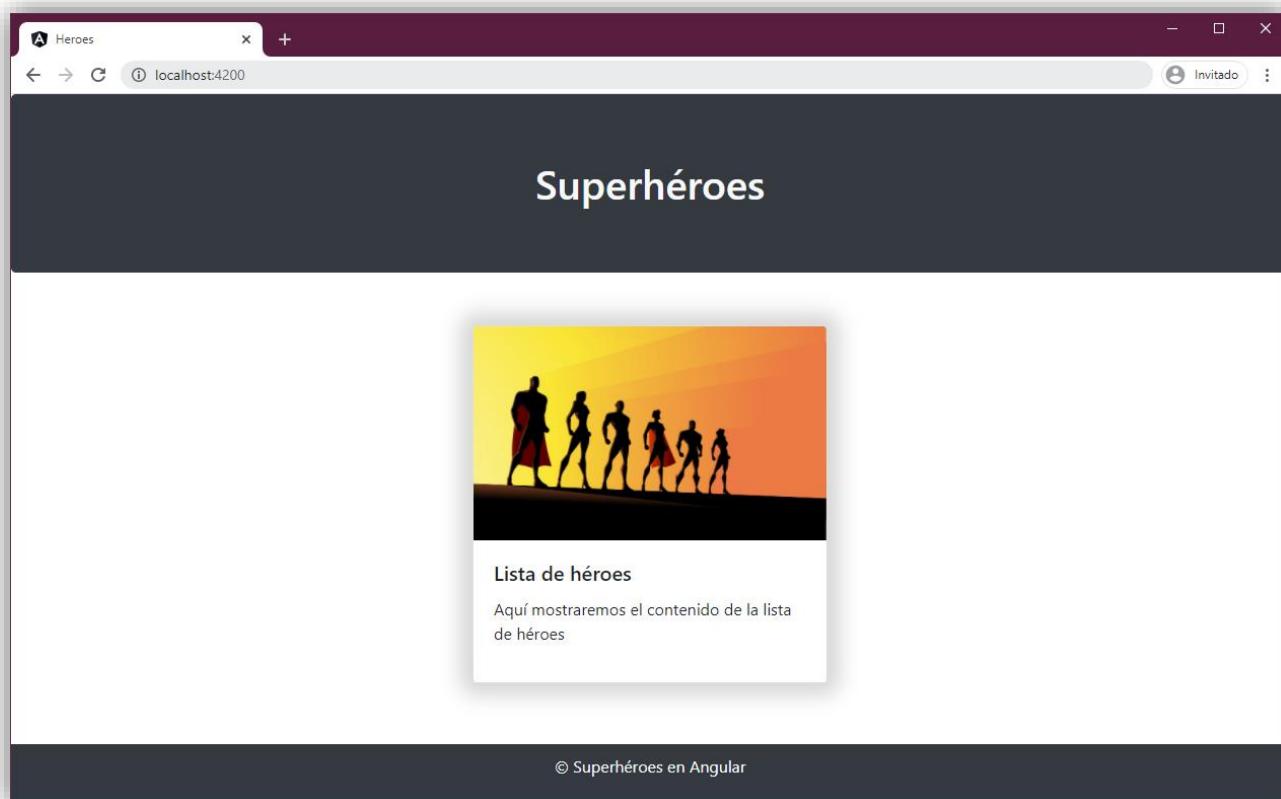
8. Todos los estilos se han aplicado usando exclusivamente bootstrap. Sólo se ha creado la clase sombra, en `styles.css` para tenerlo globalmente.

```
/*styles.css*/
.sombra {
  box-shadow: 0px 0px 30px 5px rgba(0,0,0,0.75);
}
```

No hemos tocado el `app.module`, ni el `css` o `ts` de ninguno de los componentes.

Es una guía, te aconsejo que no copies y pegues, si no que hagas una página más o menos con la misma estructura, pero con tu estilo y a tu forma. Con tener un párrafo en el cuerpo de la página donde mostrar un contenido nos valdrá.

La página debería verse de la siguiente manera:



9. Lo siguiente es crear un array con algunos héroes en la clase de CuerpoComponent (`cuerpo.component.ts`), para después mostrarlos en la página principal:

```
export class CuerpoComponent implements OnInit {
  public listadoHeroes: Array<any>;
  constructor() {
    this.listadoHeroes = [
      {
        nombre: "Superman",
        imagen: "",
        ataque: 95,
        defensa: 93,
        poder: 350,
        vida: 100
      },
      {
        nombre: "Thor",
        imagen: "",
        ataque: 92,
        defensa: 75,
        poder: 275,
        vida: 100
      }
    ];
  }
}
```

Declaramos el atributo que contiene el array de héroes y lo inicializamos en el constructor. Realmente todavía no es un array de héroes, si no de objetos. Cada objeto tiene sus atributos con sus valores (nombre, imagen, ataque, defensa, poder y vida).

El tipo de datos es un `Array<any>`. Con esto declaramos que es un array, pero de un tipo de datos no definido (ya que los héroes no tienen todavía una clase). Si queremos un array de números o cadenas, sería `Array<number>` o `Array<string>`, respectivamente.

10. Para terminar, si queremos mostrar el contenido en la página, usamos `{{...}}` en el HTML:

```
<!-- cuerpo.component.html -->
...
<div class="card-text">
  <p>{{listadoHeroes}}</p>
</div>
```

En la tarjeta nos mostrará tantos elementos `[object Object]` como objetos hayamos colocado en el array. Si queremos un dato concreto tendremos que usar `{{listadoHeroes[0].nombre}}` y nos mostrará el nombre del primer objeto del array. Veremos cómo mejorarlo en el siguiente apartado 😊.



<https://stackblitz.com/edit/ng-heroes1-inicial>

Directivas estructurales

Angular posee algunas directivas para manipular el DOM. Con ellas podemos dar forma o remodelar la estructura del DOM, agregando, quitando o manipulando elementos.

Al igual que con otras directivas, las estructurales también se aplican a un elemento HTML, y la directiva se aplicará a ese elemento y a sus descendientes.

Son fáciles de reconocer ya que les precede un asterisco (*) antes del nombre. Veamos las más comunes.

***ngIf**

Es la más simple y la más fácil de entender. Le indicamos una expresión booleana y hará que ese elemento **aparezca** o **desaparezca**. Es importante entender que no oculta los elementos mediante CSS, si no que los pone o elimina físicamente del DOM (un elemento oculto sigue existiendo en el DOM).

Veamos su sintaxis con un ejemplo:

```
<img *ngIf="heroe.imagen" [src]= "heroe.imagen" />

```

Se evalúa la expresión que hay en ***ngIf**, y si es **true**, el elemento se incluye en el DOM, y si da **false**, no se incluye. Por lo que, aunque existan dos elementos ****, sólo uno estará en el DOM, mostrando la imagen del héroe si la tenemos, y si no mostrando una por defecto.

Recordemos que, en JavaScript, se puede usar una variable cadena en una expresión booleana, devolviendo **false** si está vacía ("") o nula, y **true** en caso contrario. Por eso nos ahorraremos hacer ***ngIf="heroe.imagen != ''"**.

Si queremos hacer algo parecido a un **else**, la mejor forma es repetir el elemento y poner la condición al contrario (como el ejemplo anterior). Hay una forma de usar **else** [en la documentación oficial](#).

En el ejemplo anterior lo hemos usado sólo en un **img**, el cual no puede contener elementos en su interior, pero si se hace en elementos que si pueden (**p**, **div**, **a**, **button**, etc.), se ocultarían o mostrarían todos los elementos contenidos. En el siguiente ejemplo evitamos mostrar una tabla vacía.

```
<div class="listado" *ngIf="heroes.length > 0">
    ... un listado completo con datos, tablas, etc...
</div>
```

*ngFor

En nuestro proyecto ya habíamos llegado a “mostrar” el contenido de un array en la página principal. Aunque para mostrar bien el contenido deberíamos de repetir el elemento que representa a cada héroe (tarjetas) y cambiar en cada uno su posición del array (`listadoHeroes[0].nombre`, `listadoHeroes[1]`, etc.)

Eso en programación lo arreglamos fácilmente con un bucle, ya que siempre se repite lo mismo, sólo cambia un número que indica la posición del array. Pues podemos hacer bucles de tipo `forEach` en HTML usando Angular. La sintaxis es la siguiente:

```
<div class="card-text">
  <p *ngFor="let heroe of listadoHeroes">
    Nombre: {{heroe.nombre}} Poder: {{heroe.poder}}
  </p>
</div>
```

Lo anterior me creará siempre una etiqueta `<div>`, y en su interior, tantas etiquetas `<p>` como elementos haya en el array de héroes. En cada iteración del bucle me da el siguiente objeto del array `listadoHeroes` en un nuevo objeto llamado `heroe`, al cual podemos acceder a todas sus propiedades de forma normal.

Si necesitáramos el índice por el cual va el bucle, tan sólo tendríamos que hacer:

```
*ngFor="let heroe of listadoHeroes; let i=index;"
```

Y ya tendríamos una variable local llamada `i`, que empieza en 0 y termina en el número de elementos del array -1.

```
Número: {{i}} Nombre: {{heroe.nombre}}
```



Lista de héroes

Nombre: Superman	Poder: 350
Nombre: Batman	Poder: 100
Nombre: Spiderman	Poder: 150
Nombre: Linterna Verde	Poder: 200
Nombre: Ironman	Poder: 150
Nombre: Capitán América	Poder: 135
Nombre: Thor	Poder: 275

ngSwitch

Como en cualquier lenguaje de programación, si cuando vamos a hacer un `if` hay muchas opciones, lo más cómodo es una instrucción de tipo `switch`. Veamos un ejemplo:

```
<ul [ngSwitch]="heroe.nombre">
  <li *ngSwitchCase="'Groot'">Marvel</li>
  <li *ngSwitchCase="'Thor'">Marvel</li>
  <li *ngSwitchCase="'Spiderman'">Marvel</li>
  <li *ngSwitchDefault>DC</li>
</ul>
```

Se evalúa la variable que indiquemos en `[ngSwitch]`. Y sólo se mostrará el elemento que coincide con lo indicado en `*ngSwitchCase`

En este caso, se mostrará un `` con un único ``. Si `heroe.nombre` vale Groot, saldrá Marvel y así sucesivamente. Si no es ninguno de los 3 que hay definidos, saldrá DC.

ngContainer

La introducción de un elemento contenedor, generalmente un `` o `<div>`, para agrupar los elementos en una sola etiqueta raíz, generalmente es inofensivo. Por lo general ... pero no siempre. Hay casos en los que sí puede verse afectado el estilo si agrupamos unos elementos dentro de otros.

El `<ng-container>` de Angular es un elemento de agrupación que no interfiere con los estilos o el diseño porque Angular no lo coloca en el DOM. Por lo que, si no tenemos claro en qué elemento contenedor colocar otros elementos, `ng-container` será la mejor opción.

```
<p>
  Levantó la mano, y dijo...
  <ng-container *ngIf="spoilerFree">
    ...YO SOY IRONMAN !!!
  </ng-container>
</p>
```

ngClass

Existen muchas formas de aplicar estilos css con clases a elementos HTML en Angular. Veremos las comunes y útiles. Vamos a suponer que hemos creado dos estilos para mostrar las tarjetas, uno llamado `.tarjeta-villano` y otro `.tarjeta-hero` y además nuestros heroes tienen un atributo llamado `honor` (con valores -100 hasta 100, que determinan si son héroes o villanos). Veamos unos ejemplos de cómo aplicar clases css dinámicamente con Angular:

💡 Con `[class]`. Esta es la forma más simple de aplicar estilos. Se le aplica un estilo mediante una variable de tipo `string` que su valor se lo daremos en la clase del componente (llamada `estiloTarjeta` en el ejemplo).

```
<!-- *.component.html -->
<div [class]="estiloTarjeta" >
  <!--contenido de la tarjeta-->
</div>
```

💡 Con `[class]` recibiendo un array de clases a aplicar. Se puede definir el array directamente en el HTML o en el archivo TS (preferiblemente).

```
<div [class]=["'tarjeta', 'tarjeta-villano']">
  <!--en html: sin definir el array en ts-->
  <!--contenido de la tarjeta-->
</div>

<div [class]="arrayClases">
  <!--en ts: var arrayClases=['tarjeta', 'tarjeta-villano']-->
  <!--contenido de la tarjeta-->
</div>
```

💡 Con `[class]` recibiendo un JSON. Es el más complejo y potente. Las claves (*keys*) del objeto son la(s) clase(s) a aplicar si la condición expresada en el valor (*value*) da true. En caso contrario no se aplica.

```
<!-- *.component.html -->
<div [class]="
  {
    'tarjeta-heroe' : heroe.honor > 0,
    'tarjeta-villano' : heroe.honor <=0
  }
">
  <!--contenido de la tarjeta-->
</div>
```

Como siempre hay variantes y cambios entre una versión y otra de Angular, te recomiendo SIEMPRE vigilar la documentación oficial de Angular. Aquí puedes consultar esta parte de [bindind attribute class](#).

Ejercicios (ngIf, ngFor, ngClass)

Para practicar un poco y asentar estas directivas importantes, vamos a hacer lo siguiente:

📝 **Demo1: (ngFor)** En la tarjeta donde mostramos el array de héroes tal cual, vamos a imprimir una lista no numerada (un elemento `` y un `` por cada héroe, donde se vea el nombre y la vida.). Así veremos parte del contenido del array en condiciones. Mira la sintaxis del ejemplo e intenta aplicarlo a nuestro proyecto.

📝 **Demo2: (ngClass)** Algunos héroes han caído en la batalla. Así que en `cuerpo.component.ts`, vamos a poner el atributo vida de algunos héroes a 0. Vamos a mostrar en rojo los caídos y en verde los que todavía aguantan. Puedes usar las clases `bg-success` y `bg-danger` de bootstrap o bien crear las tuyas propias.

📝 **Demo3: (ngIf)** En el listado anterior, vamos a eliminar de la lista los “héroes caídos en batalla”, es decir, sólo mostramos a los héroes que tengan la vida > 0. Quizás tengas que usar el `<ng-container>`.

Como ahora vamos a hacerlo todo en una tabla HTML en lugar de dejar el `` que hemos construido en estos 3 ejercicios, vamos a eliminar el `` y todo su contenido y lo dejamos como estaba al principio `<p>{{listadoHeroes}}</p>`. Que no se te olvide también revivir a los caídos.

Listado de Héroes. ngFor

Vamos a aplicar algunas de las nuevas directivas aprendidas a nuestra aplicación para mostrar finalmente el listado de la siguiente forma:

Lista de héroes					
Nombre	Poder	Honor	Ataque	Defensa	Vida
Superman	350	94	80	93	100
Batman	50	75	60	75	100
Capitana Marvel	300	70	79	80	100
Catwoman	50	-5	63	72	100
Spiderman	150	85	86	89	100
Viuda Negra	35	37	88	83	100
Thanos	250	-92	90	95	100
Linterna Verde	200	80	82	83	100
Ironman	150	69	83	85	100
Capitán America	135	95	64	95	100
Thor	275	60	92	75	100

- Hemos añadido unos cuántos héroes y villanos a la lista, y dotado de un nuevo atributo llamado honor (para poder determinar si son héroes o villanos y en qué medida).
- Mostramos en amarillo (*warning*) la celda de la tabla (no la fila) de los héroes más poderosos (poder > 200).
- Mostramos en azul (*info*) la fila completa de los héroes con honor > 70.
- Mostramos en rojo (*danger*) la fila completa de los héroes con honor < -70.

1. Para ello, creamos una tabla normal de HTML, donde la primera fila es la cabecera de la tabla, y la siguiente fila es la tendremos que repetir tantas veces como héroes tengamos (elementos tenga el array), por lo que usaremos ***ngFor** en el segundo **<tr>**.
2. Dentro del elemento **<tr>** que se repetirá, mostraremos el valor los atributos del héroe en cada **<td>**, y lo hacemos usando la interpolación **{...}**.
3. Por último, para darle un poco de estilo a la tabla, además de bootstrap normal, aplicamos una serie de clases de forma condicional con **class** siguiendo las normas anteriores

No queda muy estético, ya lo arreglaremos más adelante, pero nos sirve para probar las nuevas directivas.

Hemos modificado el archivo **cuerpo.component.ts** añadiendo el atributo honor y unos cuantos elementos más al array. También hemos modificado el archivo **cuerpo.component.html** con el siguiente código:

```
<!-- cuerpo.component.html -->
<!-- vamos directamente al texto de la tarjeta, el resto sigue igual -->
<div class="card-text">
  <table class="table table-hover table-sm">
    <thead>
      <tr class="table-dark">
        <th>Nombre</th>
        <th>Poder</th>
        <th>Honor</th>
        <th>Ataque</th>
        <th>Defensa</th>
        <th>Vida</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let heroe of listadoHeroes"
          [class]={
            'table-info': heroe.honor > 70,
            'table-danger' : heroe.honor < -70
          }">
        <td>{{heroe.nombre}}</td>
        <td [class.table-warning]="heroe.poder > 200">
          {{heroe.poder}}
        </td>
        <td>{{heroe.honor}}</td>
        <td>{{heroe.ataque}}</td>
        <td>{{heroe.defensa}}</td>
        <td>{{heroe.vida}}</td>
      </tr>
    </tbody>
  </table>
</div>
```

En el siguiente enlace puedes consultar el código completo, verlo en funcionamiento, cambiar lo que quieras, etc.



<https://stackblitz.com/edit/ng-heroes2-listado>

Modelos

Un modelo en Angular es una clase JavaScript que representa un tipo de objeto que va a existir en la aplicación. La sintaxis JSON (lo que hemos usado hasta ahora) para definir objetos es flexible, no obliga de definir los tipos de las variables, ni tampoco permite definir métodos.

Por ejemplo, un `Array<any>` permite crear un array de cualquier cosa:

```
let listadoHeroes: Array<any>;
listadoHeroes = [
  {
    nombre: "Superman",
    imagen: "assets/img/avatars/superman.jpg",
    ataque: 80,
    defensa: 93,
    poder: "Guanzazos como panes",
  },
  {
    nombre: ["Batman", "Caballero Oscuro"],
    imagen: "",
    ataque: 60,
    defensa: 75,
    poder: 50,
    honor: true,
    vida: 100
  }
];
```

Podemos observar que en el array hay dos objetos, pero no están definidos. Uno tiene más atributos que otro y hay diferencias entre los tipos de algunos atributos. **En ningún momento esto provoca un error** (ni de compilación ni nos avisa el editor) ya que la definición del array lo permite, por lo que nos podemos equivocar y arrastrar un error que es difícil detectarlo.

Para evitar eso, si hubiese una estructura fija que vamos a usar mucho en nuestra aplicación (por ejemplo, un héroe) es conveniente crear ese modelo de datos y después reusarlo todas las veces que nos haga falta. Los modelos se pueden crear a partir de **clases** o de **interfaces**. Se suele usar más las interfaces puesto que no inicializan los valores del objeto (como si hacen las clases con su constructor), ya que es una tarea que se suele delegar en los **servicios**, que veremos más adelante.

 Ambas formas son válidas para declarar Arrays de un tipo concreto de objetos.

```
let lista1: Array<Heroe>;
let lista2: Heroe[];
```

Usando Modelos en Héroes

Para usar un modelo, tenemos que crear un archivo .ts que defina una clase normal en JavaScript, y después hacer el correspondiente **import** en cada componente dónde usemos dicha clase. Esta clase es recomendable colocarla en una carpeta nueva para ir colocando el resto de los modelos que vayamos creando.

1. Creamos la carpeta `src/app/modelos` y dentro, el archivo `heroe.model.ts`. (En singular, ya que representa UN héroe. No es necesario poner `model` en el nombre del archivo, pero así queda más homogéneo con el resto de los archivos.)
2. *Definimos la clase de una forma casi idéntica a como se hace en Java:*

```
//heroe.model.ts
export class Heroe {
    //Atributos
    public nombre: string;
    public poder: number;
    public honor: number;
    public ataque: number;
    public defensa: number;
    public vida: number;
    public imagen: string;

    //Constructor
    constructor (nombre:string, poder:number, honor:number, ataque:number,
                defensa:number, vida:number, imagen:string) {
        this.nombre = nombre;
        this.poder = poder;
        this.honor = honor;
        this.ataque = ataque;
        this.defensa = defensa;
        this.vida = vida;
        this.imagen = imagen;
    }
}
```

 Existe una versión abreviada permitida en TypeScript para la definición anterior, pero por ahora usaremos ésta ya que es la que estamos acostumbrados de Java.

3. Una vez definida la clase, como le hemos indicado `export`, ya podemos hacer `import` en donde queramos, en concreto en `cuerpo.component.ts`, que es el único sitio donde hacemos referencia por ahora:

```
//cuerpo.component.ts
import { Heroe } from 'src/app/modelos/heroe.model';
```

4. Y para definir el array no utilizaremos `Array<any>`, si no nuestro nuevo modelo:

```
public listadoHeroes: Array<Heroe>;
```

⌚ Visual Studio Code, si intentamos usar el modelo (para definir el array, por ejemplo) sin haberlo importado antes, nos hará automáticamente la importación.

Es posible que, al cambiar el tipo del array, nos de errores si no hemos mantenido la estructura respetando número y tipo de datos (el orden no importa). Es decir, si hay más atributos de los definidos, si falta alguno o si no son del tipo definido.

```
constructor() {
  this.listadoHeroes = [
    {
      nombre: "Superman",
      poder: 350,
      honor: 94,
      ataque: 80,
      (property) Heroe.imagen: string
      Type 'true' is not assignable to type 'string'. ts(2322)
    }, heroemodel.ts(10, 12): The expected type comes from property 'imagen' which is declared here
    on type 'Heroe'
    Peek Problem (Alt+F8) No quick fixes available
    imagen: true,
    ataque: 60,
    defensa: 75,
    poder: 50,
    honor: 75,
    vida: 100
  ],
}
```

Para comprobarlo, podemos probar a cambiar un tipo de datos por otro, y el propio IDE nos avisará que para la propiedad `imagen` esperaba un ‘`string`’ y le hemos asignado un booleano.

5. Podemos mantener el formato JSON como antes, o crear objetos con el operador `new`, pasándole los valores directamente al constructor, tal y como hacemos en Java.

```
this.listadoHeroes = [
  new Heroe("Superman", 350, 94, 80, 93, 100, ""),
  new Heroe("Batman", 50, 75, 60, 75, 100, ""),
  new Heroe("Capitana Marvel", 355, 70, 85, 80, 100, ""),
  new Heroe("Catwoman", 50, -5, 63, 72, 100, ""),
  new Heroe("Spiderman", 150, 85, 86, 89, 100, ""),
  new Heroe("Viuda Negra", 35, 37, 88, 83, 100, ""),
  new Heroe("Thanos", 250, -92, 90, 95, 100, ""),
  new Heroe("Linterna Verde", 200, 80, 82, 83, 100, ""),
  new Heroe("Ironman", 150, 69, 83, 85, 100, ""),
  new Heroe("Capitán América", 135, 95, 64, 95, 100, ""),
  new Heroe("Thor", 275, 60, 92, 75, 100, "")
];
```

Con JSON quizás se entiende mejor, y con `new` queda más compacto. Con JSON no hace falta mantener el orden de las propiedades, con `new` sí. Para gustos colores.

Y así de fácil hemos creado un modelo de datos que usaremos en nuestra aplicación y le seguiremos dando uso cuando queramos guardarlo en una base de datos.

Servicios

Actualmente, nuestros ejemplos están creando y mostrando datos “falsos”. Nos referimos a que es un array (*listadoHeroes*) que se crea directamente en el componente “cuerpo” de nuestra aplicación (*cuerpo.component.ts*), y no es obtenido por una base de datos o una consulta Ajax. A este tipo de datos se les llama *mock*.

Los componentes no deben obtener ni guardar los datos directamente. Deben centrarse en presentar esos datos, y delegar el acceso de los datos a un servicio.

Aunque no tengamos todavía una base de datos, ni hayamos visto cómo hacer consultas Ajax a una API Rest, podemos crear un mecanismo que separa el código mediante el que se obtienen y se guardan los datos, del mecanismo que los prepara y los muestra. De esta forma nos podemos abstraer de toda la lógica del acceso a datos, y si en un futuro cambia, sólo habrá que modificar esa parte (por ejemplo, pasamos de usar mongoDB a MySQL). A ese mecanismo le llamaremos ***servicios***.

Resumiendo, un **servicio será un proveedor de datos**, que se ocupa de la lógica del acceso a los mismos y del tratamiento que se haga de ellos dentro de la aplicación. Los servicios **serán usados por los componentes**, los cuales delegarán la responsabilidad de acceder a la información y realizar operaciones con ella (consultar, guardar, modificar y borrar).

Imagina un servicio como la cocina de un Burger King™. El dependiente (componente) pide un menú a la cocina (servicio). La cocina puede optar por prepararla en el momento o tener las hamburguesas más solicitadas ya preparadas, en cualquier caso, decide la cocina. La cocina pone en la bandeja de entrega, la hamburguesa, las patatas y la bebida (imagínate que sí). El menú serían los datos. El dependiente coge la comida y se la entrega al cliente. Fíjate que en ningún momento el dependiente sabe que ocurrió para crear la comida, si se hizo de una forma o de otra o los procesos que intervinieron en la creación. No necesita saberlo. Y si mañana esos procesos cambian y se mejoran, el dependiente sigue sin necesitar saber las mejoras del proceso. Básicamente, el componente pide unos datos y se le dan, la lógica del cómo se obtiene es cosa del servicio.

Lo único que se debe tener en cuenta es que se le han de entregar los datos de la misma forma que los espera. Por ejemplo, si espero un objeto, no puedo entregar un array, porque el componente estará preparado para recibir y mostrar un objeto y no un array y viceversa.

Crear un servicio

Como hemos visto anteriormente, para crear un servicio usaremos también AngularCLI:

1. Desde la línea de comandos, en la carpeta raíz de nuestro proyecto y escribir lo siguiente:

```
ng g service
```

Así iniciaremos el asistente para generar un nuevo servicio, mostrando lo siguiente:

```
? What name would you like to use for the service? servicios/heroes
CREATE src/app/servicios/heroes.service.spec.ts (357 bytes)
CREATE src/app/servicios/heroes.service.ts (135 bytes)
>
```

2. Ya tenemos el servicio creado. Sólo hace una pregunta, y será el nombre del servicio. Podemos indicarle una carpeta y creará el archivo dentro de la carpeta. Introducimos `servicios/heroes` y pulsamos intro.

💡 Esto nos creará un servicio llamado "HeroesService", y creará un archivo llamado "`heroes.service.ts`". AngularCLI se encarga de seguir la nomenclatura creada por Angular.

💡 Si tenemos distintos módulos y queremos tener separados los servicios de cada módulo, tan sólo tenemos que indicar la ruta de dónde queremos el servicio cuando lo estamos creando. No es habitual, ya que los servicios suelen usarse desde varios componentes y módulos por lo que se suelen crear en el módulo raíz y en una carpeta llamada `servicios`, `services`, `shared` o algo parecido.

💡 Lo que no hace AngularCLI es añadir el servicio a ningún módulo concreto. Crea los archivos físicamente dónde le indiques, pero después tendrás que añadirla manualmente en el módulo que se use. Ahora veremos cómo.

3. Nos queda declarar el servicio en el módulo dónde quieras usarlo. Una vez declarado, lo podrán usar todos los componentes del módulo. Es importante no olvidarse de este paso, ya que no lo hace AngularCLI como si hacía con los componentes. Para declararlo, tendremos que añadir al decorador del módulo en la parte de `providers`, el nombre del servicio. Lo añadiremos a nuestro `app.module.ts` y quedaría así:

```
//app.module.ts
//Módulos
import ...
//Componentes
import ...
//Servicios
import { HeroesService } from './servicios/heroes.service';

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

💡 En VSC, podemos añadir el provider primero, y él nos hará la importación automática.

4. El *provider* del servicio se lo hemos añadido al módulo para usarlo en todos sus componentes, pero se puede añadir a un componente concreto.
5. Ahora echemos un vistazo al código del servicio (`src/app/servicios/heroes.service.ts`):

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

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

  constructor() { }
}
```

Parece una clase normal, excepto por el decorador `@Injectable`. Sirve para indicarle a Angular que la clase que se decora (`HeroesService`), puede necesitar dependencias que pueden ser entregadas por inyección de dependencias, que es un patrón de desarrollo también usado en Spring de Java.

Su funcionamiento normal será el de ir añadiendo métodos a la clase que reciban lo que necesite y devuelvan lo que se le pide. También puede contener atributos que tengan información relativa a los datos, como la url de la API Rest o de la ubicación de las imágenes.

Lo más seguro es que en el servicio usemos el modelo de datos, por lo que casi con total seguridad habrá que hacer un import del modelo sobre el que trate el servicio (o modelos, si se tratan más).

Por defecto, el comando AngularCLI `ng generate service (ng g s)` registra un proveedor con el inyector raíz para su servicio incluyendo los metadatos del proveedor, que es `providedIn: 'root'` en el `@Injectable`.

Cuando creamos un servicio con el inyector raíz, Angular crea una única instancia compartida (patrón *singleton*) del servicio y lo inyecta en cualquier clase que lo solicite. El registro del proveedor también permite a Angular optimizar la aplicación eliminando el servicio si ya no va a usarse más.

Antes de ver cómo sería el uso del servicio desde el componente, veamos el contenido completo final sin cortes del archivo `heroes.service.ts`:

```
//heroes.service.ts
import { Injectable } from '@angular/core';
import { Heroe } from '../modelos/heroe.model';

@Injectable({
  providedIn: 'root'
})
export class HeroesService {
  //Atributos

  public urlImagenesInt:string = "assets/img/";
  public urlImagenesExt:string = "https://borilio.github.io/curso-angular/img/";

  //Constructor
  constructor() {}

  /**
   * Devuelve un Array<Heroe> con todos los héroes
   */
  public getHeroes(): Array<Heroe> {
    let lista = [
      new Heroe("Superman", 350, 94, 80, 93, 100, ""),
      new Heroe("Batman", 50, 75, 60, 75, 100, ""),
      new Heroe("Capitana Marvel", 355, 70, 85, 80, 100, ""),
      new Heroe("Catwoman", 50, -5, 63, 72, 100, ""),
      new Heroe("Spiderman", 150, 85, 86, 89, 100, ""),
      new Heroe("Viuda Negra", 35, 37, 88, 83, 100, ""),
      new Heroe("Thanos", 250, -92, 90, 95, 100, ""),
      new Heroe("Linterna Verde", 200, 80, 82, 83, 100, ""),
      new Heroe("Ironman", 150, 69, 83, 85, 100, ""),
      new Heroe("Capitán América", 135, 95, 64, 95, 100, ""),
      new Heroe("Thor", 275, 60, 92, 75, 100, "")
    ];
    return lista;
  }
}
```

Se han añadido 2 atributos para las url de las imágenes, y un método llamado `getHeroes()`, que no recibe ningún parámetro, y se encarga de devolverme un array completo de héroes. Una vez montado el servicio, el día de mañana cuando queramos hacer una consulta real a una base de datos o a una API Rest, sólo tendremos que modificar lo que hace el método `getHeroes()` del servicio. El componente seguirá funcionando igual y no habrá que modificarlo.

Por otro lado, las vistas seguirán mostrando la información de la misma forma, pero en lugar de usar datos simulados (*mocks*) estará usando los datos reales.

6. Ahora sólo nos queda hacer uso del servicio desde el componente `cuerpo.component.ts`, veremos cómo incluirlo y hacer la llamada a su método.

```
//cuerpo.component.ts
import { Component, OnInit } from '@angular/core';
import { Heroe } from 'src/app/modelos/heroe.model';
import { HeroesService } from 'src/app/servicios/heroes.service';

@Component({
  selector: 'app-cuerpo',
  templateUrl: './cuerpo.component.html',
  styleUrls: ['./cuerpo.component.css']
})
export class CuerpoComponent implements OnInit {
  public heroesService: HeroesService;
  public listadoHeroes: Array<Heroe>;

  constructor(heroesService:HeroesService) {
    this.heroesService = heroesService;
    this.listadoHeroes = [];
  }

  ngOnInit(): void {
    this.listadoHeroes = this.heroesService.getHeroes();
  }
}
```

Las modificaciones son, los `imports` del modelo y el servicio, hemos añadido como atributo el servicio y lo hemos inicializado en el constructor como cualquier atributo. Después en el `ngOnInit()` hemos hecho usado el servicio para obtener la lista de héroes, y hemos inicializado la lista que ya teníamos antes.

Puede parecer que podríamos inicializar `listadoHeroes` en el constructor, y como poder se puede, pero no es una buena práctica. El constructor sólo debe hacer inicializaciones simples, como conectar los parámetros recibidos a los atributos, pero no debería hacer nada más. No debería llamar a una función que realiza solicitudes HTTP (Ajax) a un servidor remoto ni a una base de datos. Eso lo hacemos mejor en el `ngOnInit()`. Puedes ver más detalles sobre el ciclo de vida de los componentes en la [documentación oficial](#).

Y ya funcionaría correctamente nuestro servicio. En un futuro, en el servicio es donde irán las funciones para guardar nuevos héroes, borrar, o hacer consultas con filtros (p.ej: `getHeroes("man")` -> Devolvería una lista con los héroes que tengan "man" en su nombre).

A continuación, el código de Proyecto Heroes, implementado un modelo y un servicio, con el código anterior. Se añadió además una columna en la tabla para incluir una imagen.



<https://stackblitz.com/edit/ng-heroes3-servicios-modelos>

Declaración implícita de propiedades en TypeScript

TypeScript ofrece algo simplicidad en la creación de atributos en las clases. Estas facilidades en la sintaxis de algunos procedimientos se le denominan también versiones sintácticamente azucaradas (*syntactic sugar*).

La clase del modelo de datos (`heroe.model.ts`) está definida así:

```
//heroe.model.ts
export class Heroe{
    public nombre: string;
    public poder: number;
    public honor: number;
    public ataque: number;
    public defensa: number;
    public vida: number;
    public imagen: string;

    constructor (nombre:string, poder:number, honor:number, ataque:number,
                defensa:number, vida:number, imagen:string) {
        this.nombre = nombre;
        this.poder = poder;
        this.honor = honor;
        this.ataque = ataque;
        this.defensa = defensa;
        this.vida = vida;
        this.imagen = imagen;
    }
}
```

Y se puede simplificar así:

```
//heroe.model.ts
export class Heroe{
    constructor (public nombre:string, public poder:number,
                public honor:number, public ataque:number,
                public defensa:number, public vida:number,
                public imagen:string) {
    }
}
```

Al detectar en el constructor los modificadores `private`, `public` o `protected`, TypeScript entiende que lo que quieras es DECLARAR un atributo en el objeto recién construido e inicializarlo con el valor recibido por parámetro. Con lo que nos ahorramos definirlo, pasarle un parámetro al constructor e inicializarlo con ese valor dentro del constructor.

Comunicación entre componentes

Hasta ahora, nuestros componentes han funcionado de forma independiente, pero ¿cómo haríamos si tuviésemos que enviar información entre ellos? Por ejemplo, queremos un componente que muestre una ficha detallada del héroe, y para mostrarla tendremos que enviar UN héroe concreto como dato desde el componente cuerpo (donde está el listado) al componente ficha para que la pueda mostrar.

Además de los servicios como proveedores de datos, existen diferentes casos mediante los cuales los componentes se comunicarán con otros. Según la estructura en árbol de los componentes, la comunicación puede ir de padres a hijos, o bien, de los hijos hacia los padres. Y cada una se hace de una forma diferente.

@Input. Paso de información de padres a hijos

Esta comunicación se realiza por medio de propiedades personalizadas en los componentes. La información se añade desde la vista, usando atributos en el selector del componente padre.

Para ello, las propiedades del componente hijo se deben decorar mediante `@Input`, y así Angular sabrá que esa(s) propiedad(es) pueden inicializarse o modificarse desde fuera (el padre).

Vamos a verlo directamente en un ejemplo ya que es más simple de lo que parece. Por ejemplo, al componente para el pie de página, queremos enviarle el texto que queremos usar en él y el año para el copyright. Lo enviaremos desde el componente padre (`app.component`) y lo recibirá el hijo (`pie.component`).

1. Primero declararemos la(s) propiedad(es) en el hijo (el texto y el año), con la diferencia que usaremos el decorador `@Input` (una vez por cada propiedad). Se pueden indicar valores por defecto, que se usarán en caso de no recibir nada desde el padre.

```
//pie.component.ts
import { Component, OnInit, Input } from '@angular/core';

@Component({
  selector: 'app-pie',
  templateUrl: './pie.component.html',
  styleUrls: ['./pie.component.css']
})
export class PieComponent implements OnInit {
  @Input()
  public texto:string;

  @Input()
  public fecha:number=2020; //valor por defecto

  constructor() { }
  ngOnInit(): void { }
}
```

💡En las versiones más recientes de TypeScript (o si se crea el proyecto como strict), nos obligará a inicializar todos los atributos siempre. Así que es mejor ponerle un valor por defecto a `fecha` y a `texto`.

2. Y ahora en nuestro componente padre (`app.component`), le pasamos los valores desde la vista a los atributos del hijo de la siguiente forma:

```
<!-- app.component.html -->
<...></...>

<app-pie
  [texto]="'Super Pie de página nuevo &copy;'" 
  [fecha]="2021"
></app-pie>
```

Hay que tener en cuenta que el valor del atributo del HTML es una expresión, por lo que tendríamos que escribir entre las comillas dobles como si de código se tratara, es decir, si es un literal de cadena (como el ejemplo), tendríamos que entrecollarlo otra vez con comillas simples ya que está todo entre comillas dobles (se puede a la inversa), si es un número u otras variables no haría falta.

La fecha se inicializará al valor indicado desde el padre (2021) pero en caso de no indicar la propiedad en la vista, se inicializaría con el valor por defecto indicado antes (2020).

3. Una vez enviado (paso 2) y recibido (paso1) los valores que queremos, tan sólo nos quedaría mostrar esos datos recibidos en el componente hijo (`pie.component.html`) ya de la forma conocida usando la interpolación.

```
<!-- pie.component.html -->
<footer id="pie" class="fixed-bottom text-center bg-dark text-light">
  <div>
    <p>
      {{texto}} {{fecha}}
    </p>
  </div>
</footer>
```

Para mostrar el ejemplo, hemos usado algo muy simple para que se vea más claro el mecanismo de comunicación. Se han enviado 2 datos por separado (una cadena y un número) pero se podría haber enviado los dos juntos en un mismo objeto (`{texto:'blah', fecha:2021}`), o en otros casos, un array de objetos más complejos, cada uno con más atributos y funcionaría igual.

Recuerda que el binding de propiedades (`[]`) viajan en una dirección, por lo que si el padre modifica los valores enviados (texto o fecha), se modificará sus valores en el hijo (pie), pero si por el contrario, desde el `pie.component` modificamos esas variables, el nuevo valor no viajará hacia el padre.



<https://stackblitz.com/edit/ng-heroes4-input>

@Output. Paso de información de los hijos a los padres

Cuando la comunicación es de hijos a padres, Angular en lugar de propiedades, usa eventos. Es decir, cuando el hijo tiene un dato que quiere que llegue al padre, genera un evento que puede ser capturado desde el padre, y así realizar las acciones que necesite.

Para definir los eventos que van de los hijos a los padres, Angular usa el decorador `@Output`.

De forma que la comunicación funcionará de la siguiente manera:

- ▶ El **componente hijo** será el encargado de escalar el evento hacia el parent, para avisarle del suceso. Así podrá comunicarle un dato que el parent necesite saber sobre el suceso.
- ▶ El **componente parent** deberá ser capaz de capturar el evento emitido por el hijo y recuperar el dato que envió (que estará en el evento).

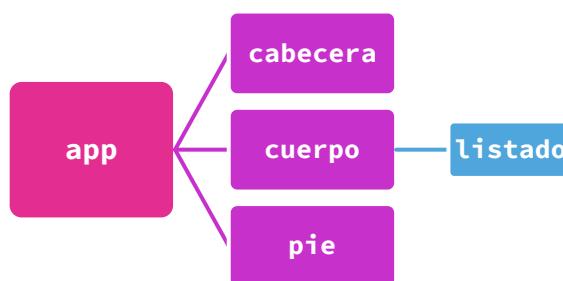
Para que se todo se entienda mejor (es más complejo que con `@Input`), antes de nada, haremos unos cambios en la estructura de nuestro proyecto. Crearemos un nuevo componente llamado “`listado`”, donde moveremos todo lo referente a la tarjeta que muestra el listado (el `div#tarjeta`). Dejaremos en el componente “`cuerpo`” la estructura de una fila con 3 columnas (`<main>`). (Resumiendo, hemos extraído la tarjeta del listado a un componente nuevo). Estas modificaciones son para diferenciar claramente un componente parent y dentro de él, un hijo (`cuerpo` es el parent y tiene a `listado` como hijo).

Te dejamos el código con el que partimos, antes de empezar con el `@Output`.



<https://stackblitz.com/edit/ng-heroes5-listado-component>

Lo que queremos hacer: Queremos seleccionar un héroe como favorito. Al hacer clic en la tabla, se mostrará un `div` simple mostrando el héroe, nombre e imagen del héroe seleccionado.



En el diagrama vemos la estructura actual de componentes de nuestra aplicación. La tabla está en el componente `listado`, pero el `div` que vamos a mostrar está en el componente `cuerpo`. Por lo que tendremos que emitir un **evento** desde el hijo, que capturará el parent. En el evento irá el dato que queremos enviar (como si de un paquete se tratara). El dato que enviaremos será el héroe sobre el que hicimos clic.

💡 Con fines didácticos, **se explicarán los pasos en el orden más fácil** para comprender el proceso, **pero no es el orden idóneo**, ya que iremos haciendo referencia a métodos antes de crearlos, con lo cual pueden salir temporalmente errores en la compilación. Una vez entendido todo, es mejor crear los métodos primero y después referenciarlos ya que el IDE ayuda autocompletando los métodos, eventos e imports, además de evitar los errores de compilación.

Una vez expuesto lo que queremos hacer, veremos punto por punto los pasos a seguir para conseguir lo que queremos:

- Declarar el evento y el método a ejecutar:** Todo empezará por hacer clic en una fila de la tabla (podría ser cualquier otro elemento) que mostramos en `listado.component.html`. Con `(click)` le indicamos que ejecutaremos un método llamado `onSeleccionarFavorito()` hacer click con el ratón, y le pasaremos el objeto `heroe` de esa fila.

```
<!-- listado.component.html -->
...
<tbody>
  <tr ... (click)="onSeleccionarFavorito(heroe)">
    <td> ... </td>
  </tr>
...
...
```

- Crear ese método** en `listado.component.ts`. Recuerda que este método se ejecutará al hacer clic en la fila. Por ahora sólo mostrará datos por consola. Pruébalo y así sabemos que vamos bien.

```
//listado.component.ts
//Se ejecuta al hacer clic en una fila, recibiendo el héroe de la fila
public onSeleccionarFavorito(heroeFavorito: Heroe) {
  console.log("Hemos seleccionado a:", heroeFavorito);
}
```

- Crear el evento y emitirlo.** Lo que tenemos que hacer en el método `onSeleccionarFavorito()` es EMITIR un evento nuestro con la información que queremos enviar. El héroe en nuestro caso. Así que en la clase `ListadoComponent`, le hacemos los siguientes cambios:

```
import { ..., EventEmitter, Output } from '@angular/core';
...
@Output()
public eventoFavorito = new EventEmitter<Heroe>();

public onSeleccionarFavorito(heroeFavorito: Heroe) {
  console.warn("El hijo dice: se seleccionó a:", heroeFavorito);
  this.eventoFavorito.emit(heroeFavorito);
}
```

Expliquemos detalladamente lo que hacemos en el hijo para emitir el evento con la información:

- ▶ Creamos un atributo en la clase que será un `EventEmitter<Heroe>` con el decorador `@Output`. Con `EventEmitter`, tenemos que indicar el tipo de dato que enviaremos en el evento, en nuestro caso queremos enviar un `Heroe`, si fuese un `string`, pondríamos `EventEmitter<string>`.
- ▶ Tenemos que importar `EventEmitter` y `Output`. Ambos desde `@angular/core`. VSC nos ayuda y lo hace por nosotros (pero siempre vigilar que se haga bien).
- ▶ El nombre del atributo “`eventoFavorito`” es importante porque será el nombre del evento para cuando lo tengamos que capturar desde el padre. Si queremos enviarlo con otro nombre de evento, podríamos indicarlo con `@Output("otroNombreDistinto")`.

4. **Capturar el evento desde el padre.** Desde `cuerpo.component.html`, en el selector del hijo `<app-listado>` tenemos que indicarle el método que se lanzará cuando se produzca el evento (`eventoFavorito`).

```
<!--cuerpo.component.html-->
<main>
  ...
  <!-- columna central -->
  <div class="col-10">
    <app-listado [eventoFavorito]="procesarFavorito($event)"></app-listado>
  </div>
  ...
</main>
```

```
//cuerpo.component.ts
...
export class CuerpoComponent implements OnInit {
  //Lo inicializaremos en el evento procesarFavorito
  public heroeFavorito?:Heroe;

  constructor() {}

  ngOnInit(): void {}

  public procesarFavorito(heroe: Heroe){
    console.warn("El padre dice: 'eventoFavorito' recibió ", heroe);
    this.heroeFavorito = heroe;
  }
}
```

Explicamos con detalle lo que hacemos en el padre:

- ▶ En la vista, le indicamos que método debe ejecutar al producirse el evento (cuando se produzca el evento `eventoFavorito`, ejecutaremos el método `procesarFavorito()`, pasándole el evento como parámetro (recordemos que ahí va el objeto enviado, es decir, el héroe seleccionado). El nombre del evento debe ser el nombre del objeto de la clase `EventEmitter` creado con anterioridad (`eventoFavorito` en nuestro caso)).
 - ▶ En la clase, creamos un atributo en la clase (`heroeFavorito`) donde guardaremos el dato recibido para tenerlo disponible en la vista.
- NOTA: Fíjate que el atributo `heroeFavorito` tiene un signo de cierre de interrogación (?) en su declaración. Esto indica que el atributo es opcional y permite que no sea inicializado. En antiguas versiones de TS permitía no inicializar el atributo o bien asignar un null directamente.*
- ▶ Definimos el método que se ejecutará al producirse el evento (`procesarFavorito`) que recibe un `heroe`. Dentro hacemos lo que necesitemos, en nuestro caso, pasar el valor al atributo para poder mostrarlo en la vista.

5. **Mostrar el/los dato(s) recibidos en el padre.** Una vez recibida la información (en nuestro caso el objeto `heroe`) desde el hijo al padre, ya podemos hacer lo que queramos con ella. Nosotros queremos mostrar en un `div` el héroe que seleccionamos previamente. Así creamos el `div` en `cuerpo.component.html` dónde veamos. Por ejemplo, en una de las columnas vacías que hay en el cuerpo.

```
<!-- cuerpo.component.html -->
<main>
  ...
  <!-- columna izquierda -->
  <div class="col">
    <div id="favorito" *ngIf="heroeFavorito" class="alert alert-danger">
      <h4>Favorito</h4>
      <hr/>
      <p>{{heroeFavorito.nombre}}</p>
      <img *ngIf="heroeFavorito.imagen" [src]="heroeFavorito.imagen"
           alt="avatar" height="150px"/>
    </div>
  </div>

  <!-- columna central -->
  <div class="col-8">...</div>
  ...
</main>
```

Explicamos con detalle que hacemos con el dato recibido:

- ▶ En una de las columnas que había vacía, colocamos un nuevo `div`, el cual gracias a `*ngIf`, mostraremos sólo cuando exista un valor en el atributo `heroeFavorito` (existirá cuando se haga clic en la tabla).
- ▶ Mostramos los datos que queramos. Al recibir el objeto completo, tenemos todas sus propiedades y elegimos qué mostrar y qué no. En nuestro caso hemos decidido mostrar el nombre y la imagen. La imagen sólo la mostramos en caso de que tengamos un valor con `*ngIf`. Se puede hacer que, si no hay imagen, se muestre otra por defecto. Hemos simplificado para centrarnos en el problema (en stackblitz si está hecho).

Y ya estaría. Es cierto que el proceso es un poco complicado, pero es cuestión de mecanizarlo haciéndolo una y otra vez y tras varias repeticiones, ya verás que no parecerá tan difícil.



Miniconsejo: Elige sabiamente tus identificadores. Te ayudarán o te complicarán.
Pensarlos muy bien deberías.



<https://stackblitz.com/edit/ng-heroes6-output>

Interfaces en TypeScript

Una diferencia característica entre TypeScript y JavaScript, es el tipado de datos. Podemos usar tipos primitivos en variables como siempre, pero también podemos definir tipos de datos más complejos usando clases e interfaces.

Las interfaces son un mecanismo de la programación orientada a objetos, que trata de arreglar el problema de la herencia múltiple. Las clases pueden extender (heredar) de otra, pero no de varias, mientras que las interfaces sí.

Las interfaces no tienen sus métodos implementados, por lo que la clase que implemente ("herede") una interfaz, *deberá* hacer su propia implementación de los métodos obligatoriamente. Con las clases no pasa eso, heredas también la implementación del método y si no te conviene del todo, *puedes* sobrescribirlo.

Se pueden entender las interfaces como un contrato, en el que se indica los atributos y métodos que debe tener la clase para que sea posible y se deberán cumplir.

Esos son los conceptos de forma genérica en programación. Cada lenguaje lo aplica con ligeras diferencias. Por ejemplo, en TypeScript una interfaz puede definir propiedades, mientras que en otros lenguajes sólo métodos. En Java también pueden definir atributos, pero han de ser constantes (**static final**).

Definición de una interfaz

Las interfaces en TypeScript se definen casi igual que las clases. Sólo tener en cuenta que:

- ▶ Se definen usando la palabra clave **interface**, en lugar de **class**.
- ▶ Si tienen atributos, sólo se define el tipo, no puede contener valores.
- ▶ Los métodos no contendrán código en su implementación.

Veamos un ejemplo de definición de una interfaz llamada **Volador**:

```
interface Volador {
    tieneCapa:boolean;
    alturaMaxima: number;
    puedeVolarEspacio: boolean;
    velocidadMaxima: number;

    volar(): void;
    volarMach(numero:number): void; //Mach1, Mach2...
    aterrizar(esHeroico:boolean): void; //Puede aterrizar normal, o no
}
```

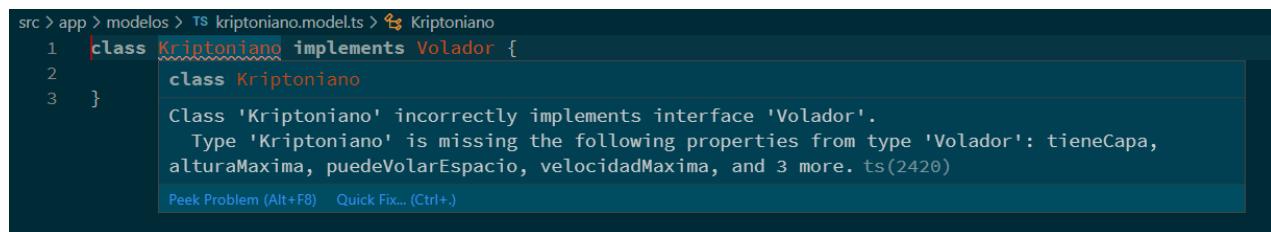
Es prácticamente igual que las clases y no hay mucha diferencia respecto a otros lenguajes como Java. Ahí estamos "*definiendo los límites del contrato*". La clase que implemente la interfaz deberá cumplirlos.

 Los identificadores de las interfaces usan las mismas reglas que los de las clases, y suelen usarse **adjetivos**, para que quede claro que lo que estamos "añadiendo" a la clase es una característica (que vuelan, que tienen súper fuerza, que son alienígenas, etc.)

Implementación de una interfaz

Una vez definida la interfaz, se podrá implementar en todas las clases que deseemos mediante la palabra **implements** en la definición de la clase. Nosotros hemos definido una interfaz Volador, para que los Héroes que tengan esa capacidad, puedan implementarla. Veamos un ejemplo:

Al crear una clase que implementa la interfaz, el propio VSC nos avisa que le faltan propiedades a la clase para poder implementar correctamente la interfaz.



src > app > modelos > kryptoniano.model.ts > Kryptoniano

```

1  class Kryptoniano implements Volador {
2      class Kryptoniano
3  }
  Class 'Kryptoniano' incorrectly implements interface 'Volador'.
    Type 'Kryptoniano' is missing the following properties from type 'Volador': tieneCapa,
  alturaMaxima, puedeVolarEspacio, velocidadMaxima, and 3 more. ts(2420)
  Peek Problem (Alt+F8) Quick Fix... (Ctrl+.)

```

Y si pulsamos en “[Quick Fix](#)” nos propone añadir lo que le falta. Nos rellenará las propiedades con valores por defecto y los métodos con código neutral para que reescribamos el código.

Le damos unos valores apropiados y rellenamos los métodos con algo conceptual.

```

class Kryptoniano implements Volador {
    tieneCapa: true;
    alturaMaxima: 100; // 100km de altura se considera espacio
    puedeVolarEspacio: true;
    velocidadMaxima: 1062167040; //Dpto. de Física de la Univ. de Leicester

    volar(): void {
        console.log("Estoy volando...");
    }
    volarMach(numero: number): void {
        console.log("Estoy volando a mach " + numero);
    }
    aterrizar(esHeroico: boolean): void {
        if (esHeroico) {
            console.log("Aterrizo de forma molona");
        } else{
            console.log("Aterrizo normalito");
        }
    }
}

```

Y ya tendríamos definida la clase **Kryptoniano** que implementa la interfaz **Volador**. Todos los objetos creados con la clase deberán tener de forma obligatoria los atributos y métodos definidos en la interfaz, como si de un contrato se tratara.

Usando la interfaz como un nuevo tipo

Además, TypeScript nos ofrece una característica adicional de las interfaces: Crear nuevos tipos de datos.

Por ejemplo, creamos una interfaz para definir los ataques:

```
interface Ataque {
    nombre: string,
    daño: number, //Evita la ñ!!
    acierto: boolean
}
```

Ahora podemos crear variables de ese tipo. El editor no se quejaría en ningún momento... por ahora.

```
let ataque1: Ataque;
```

Si tratamos de inicializar la variable con un valor que no es el que hemos indicado en la interfaz, VSC nos informará del error:

```
let ataque1: Ataque
Type '"Onda Vital"' is not assignable to type 'Ataque'. ts(2322)
Peek Problem (Alt+F8) No quick fixes available
ataque1 = "Onda Vital";
```

Para que no provocara un error, tendríamos que inicializarla correctamente tal y como se ha descrito, por ejemplo:

```
ataque1 = {
    nombre: "Kame Hame Ha",
    acierto: true,
    daño: 60
};
```

Ahora sí que tenemos todas las propiedades definidas con los mismos tipos de datos (el orden no importa), por lo que VSC nos dice que todo está correcto. En primera instancia nos avisa el IDE, pero si lo ignoramos y compilamos el código, el propio compilador de TypeScript también fallará.

Puede parecer más complejo ya que nosotros mismos nos ponemos limitaciones, pero es una buena forma de evitar errores que serían difíciles de detectar (no provocarán ningún error en el IDE), ya que al declarar las variables nos obligará a inicializarlas de una determinada forma.

Clases o interfaces

Ambas opciones para crear objetos sirven más o menos para lo mismo. Entonces, ¿Cuándo usar una interfaz o cuando una clase para crear un objeto? **Pues depende de cómo vayamos a generar los datos.**

Es muy habitual usar simplemente interfaces, que no tienen inicialización (constructores) ni funcionalidad (métodos) ya que esas partes serán delegadas en los servicios.

Pero si usamos clases, los nuevos objetos se crearán con el operador `new`, con lo que tendremos que usar sus constructores.

El “problema” es que los objetos no siempre se crean en el frontend. Lo habitual es que sean creados en algún *web service* (API REST) y los obtengamos mediante llamadas HTTP (AJAX). En éstos casos no haremos “new” ya que los objetos estarán ya creados, y nosotros sólo tendremos que volcar esos datos en JSON en alguna variable. En estos casos es mucho más útil definir una interfaz que coincida con los datos que nos va a devolver el servidor y colocar esos valores en una variable que implemente esa interfaz.

Usar interfaz en un modelo

Anteriormente vimos cómo crear modelos de datos, y para ello usábamos una clase. También podemos crear modelos basados en una interfaz.

Por ejemplo, queremos crear un modelo para crear mensajes, definidos por una interfaz. Crearíamos un archivo `mensaje.model.ts`, pero en lugar de una clase, definiríamos una interfaz:

```
//mensaje.model.ts
export interface Mensaje {
  fecha: Date;
  nombre: string;
  entregado: boolean;
}
```

Para importarlo en otras partes del programa sería exactamente igual que con una clase. Y para crear objetos que sean de esa interfaz, sería igual que lo anteriormente descrito en el apartado [Usando la interfaz como un nuevo tipo](#).

Pipes

Las *pipes* son pequeñas funcionalidades que podemos usar dentro de nuestras vistas, como una función que se aplica a un string y hace una pequeña tarea.

Se pueden usar *pipes* para transformar cadenas, cantidades de moneda, fechas y cualquier otro dato que queramos mostrar. Son funciones sencillas que se pueden usar en las expresiones para aceptar un valor de entrada y devolver un valor transformado. Son muy útiles porque se pueden usar en toda la aplicación y sólo hay que declararlas una vez.

Angular ya incluye muchas *pipes* y también podemos crear las nuestras propias. Veamos primero algunos ejemplos de *pipes* ya existentes y como usarlas.

```
<!-- Convierte a mayúscula desde la vista -->
<p>{{ favorito.nombre | uppercase }}</p>
```

```
<!-- Muestra el objeto en formato JSON, como JSON.stringify de JS -->
<pre>JSON = {{ favorito | json }}</pre>
```

```
<!-- Mostraría la fecha distintos formatos -->
<h5>{{ pelicula.fechaEstreno | date: 'short' }}</h5>
<h5>{{ pelicula.fechaEstreno | date: 'dd/MM/YYYY' }}</h5>
```

```
<!-- Cambiaría al formato moneda y la mostraría así: €1,234,567.89
<h5>{{ pelicula.recaudacion | currency:'EUR' }}</h5>
```

```
<!-- También se pueden encadenar pipes -->
<h5>{{ pelicula.fechaEstreno | date: 'short' | uppercase }}</h5>
```

Usarlas es bastante simple, sólo hay que escribir la expresión que queramos y a continuación aplicar el símbolo de tubería (`|`) seguido del nombre del *pipe*.

Puedes consultar una lista de todas las pipes incluidas en Angular en el siguiente enlace <https://angular.io/guide/pipes>.

Cambiar Locale (configuración regional)

En ciertos pipes, se usa configuración regional de la zona, como la separación de miles para los números, símbolo de moneda, y en los pipes de fechas se muestran en inglés. Esto es porque se usa la configuración regional *en-US* por defecto (inglés – United States). Se puede cambiar la localización a *es-ES* y así nos mostrará la configuración regional en español de España.

Para ello, tendremos que irnos a nuestro *app.module.ts* (si queremos cambiarlo de forma global a toda la aplicación, si no al módulo que queramos establecerlo a ese idioma):

```

import { LOCALE_ID, NgModule } from '@angular/core';
import { AppComponent } from './app.component';

...
//Cambiar el local a es-ES (o es-AR si queremos español de Argentina, p. ej.)
import localeES from '@angular/common/locales/es';
import { registerLocaleData } from '@angular/common';
registerLocaleData(localeES);

@NgModule({
  declarations: [AppComponent, ...],
  imports: [...],
  providers: [
    { provide: LOCALE_ID, useValue: 'es' }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Así ya nos saldrán los textos de las fechas en español, la hora en formato 24h (en lugar de AM y PM), las comas y los puntos para los números tal y como se usan en España, y el Euro como formato de moneda por defecto en lugar del Dólar.

Se pueden registrar más `locale` (repitiendo las líneas de los import y el registerLocaleData, **la de provide no**, ya que ahí establecemos el idioma por defecto) y así podríamos pasarse el local que queramos por argumentos al pipe que deseemos en otro `locale`.

Por ejemplo, así cargaríamos el `locale` en francés, y ya tendríamos disponible '`fr`' para usarlo en los pipes.

```

import localeFR from '@angular/common/locales/fr';
registerLocaleData(localeFR);

```

Habría que tener en cuenta que ya hemos establecido el español por defecto, y que el inglés (en) ya está registrado también, por lo que podemos usarlo directamente. Ahora ya tendríamos los 3 locales disponibles para usar: '`es`', '`en`' y '`fr`'. Español si no indicamos nada, y en el idioma indicado si se pasa por parámetros al pipe.

```

{{fecha | date : 'long': 'GMT+3':'en'}} //zona horaria gmt+3 y en inglés
{{fecha | date : 'long': undefined:'fr'}} //zona horaria por defecto y en francés

```

Pipes personalizadas

También puedes crear tus propios *pipes* si los necesitas. Vamos a crear una *pipe* que nos mostrará por pantalla si un número “Es par” o “Es impar”. Los pasos serían los siguientes:

1. Creamos una carpeta para guardar nuestros pipes. Normalmente suele ser en `src/app/pipes` ya que se usarán en toda la aplicación.
2. Creamos un nuevo archivo en la carpeta anterior, con la sintaxis `<nombre>.pipe.ts`. Para la nuestra la llamaremos `esPar.pipe.ts`
3. Escribimos un código como el siguiente en el archivo `*.pipe.ts`

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

@Pipe({
  name: 'esPar' //Con este nombre invocamos la pipe desde la vista
})
export class EsParPipe implements PipeTransform {

  transform(value: number){
    //Y aquí haremos la lógica de la pipe,
    //devolviendo lo que quieras tras recibir un valor (value)
    return (value % 2) == 0 ? "Es par": "Es impar";
  }
}
```

El método siempre deberá llamarse `transform` y recibirá un `value` (que será el valor de la expresión que queremos transformar). Se debe cambiar el tipo de dato (en nuestro caso es `number`, pero puede ser cualquiera) para así restringir el dato que usemos en la *pipe*.

4. Ahora la cargamos en nuestro `app.module.ts` para poder usarla en toda la aplicación.

```
...
//Pipes
import { EsParPipe } from './pipes/esPar.pipe';

@NgModule({
  declarations: [
    ...
    EsParPipe
  ],
  ...
})
export class AppModule { }
```

5. Para usarla en nuestra vista, sólo tendremos que hacer:

```
<h5>{{ hero.poder | esPar }}</h5>
```

El resultado de la interpolación anterior será sustituido por el resultado de ejecutar la función que hicimos en `esPar.pipe.ts` pasándole el valor de “`hero.poder`”. Esto no nos imprimirá el número en ningún momento, si no un texto “Es par” o “Es impar”, pero porque así se decidió en la función de la *pipe*.

 ¿Sabías que...? También podemos crear un pipe con Angular CLI con el comando `ng g p esPar`

Héroes 2.0

Para poder añadir y gestionar nuevas funcionalidades nuestra aplicación necesita unas mejoras. Empezamos en orden desde arriba hasta abajo comentando los cambios:

 **¿Sabías que...?** Cuando escribes `ng generate component | module`, etc., puedes añadir el argumento `--skip-tests` para que evitar la creación de los archivos `*.spec.ts` para el testing. También `-is` para los archivos `css`.

- + Nuevo componente **navegación**. Será la barra de navegación para movernos entre las distintas partes de nuestra aplicación. Se mostrará dentro de la **cabecera**.



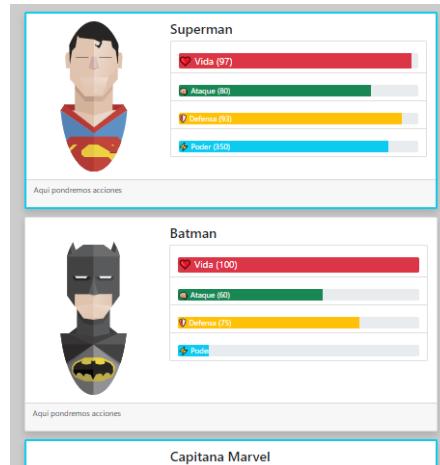
- + En lugar de mostrar directamente **Listado**, mostraremos un nuevo componente llamado **home**, que será como una pantalla de bienvenida.



- + Cambiamos el nombre al componente **Listado**, llámándolo **Listado-tabla**. Mostrará el antiguo listado, pero simplificando un poco su información. Eliminamos todo lo de los colores contextuales.

Lista de héroes		
Imagen	Nombre	Vida
	Superman	97
	Batman	100
	Capitana Marvel	100
	Catwoman	100
	Spiderman	0

- + Creamos un componente nuevo llamado *Listado-tarjetas*, que mostrará el mismo listado que antes, pero usando tarjetas individuales en lugar de una tabla HTML. Mostramos un borde azul a los más poderosos, y rojo a los villanos. Las barras están creadas con un nuevo componente llamado *barra*, el cual recibe los parámetros necesarios para mostrarla.



- + El componente *cuerpo*, lo simplificamos eliminando todo el contenido de la columna izquierda (dónde mostrábamos el héroe marcado como favorito). No te preocupes porque volveremos a practicar con comunicación entre componentes. Para probar que todo va bien, mostramos los 3 componentes directamente. Para ocultarlos, le añadimos comentarios y listo.

```
<!-- cuerpo.component.html -->
<main>
  <section class="container">
    <div class="row">
      <!-- Columna izquierda -->
      <div class="col"></div>

      <!-- Columna central -->
      <div class="col-8">
        <app-home></app-home>
        <app-listado-tabla></app-listado-tabla>
        <app-listado-tarjetas></app-listado-tarjetas>
      </div>

      <div class="col"></div>
    </div>
  </section>
</main>
```

Intenta hacer los cambios propuestos sin ayuda. Así reforzarás lo que hemos visto hasta ahora. Es importante hacer los cambios ya que son necesarios para poder aplicar las nuevas funcionalidades que veremos a continuación. Si te atascas, siempre puedes consultar todo el código en el siguiente enlace.



<https://stackblitz.com/edit/ng-heroes7-renew>

Routing en Angular

El enrutador o '*Router*' es el componente en Angular que gestiona la navegación de una vista a otra. Para ello, '[Router](#)' interpreta las URL del navegador como instrucciones para cambiar la vista.

Ejemplo: Si voy a la url `miapp.com/blog` desde el navegador, que se muestre un componente llamado blog. Y lo hará tanto escribo la url en un navegador, como si usamos un enlace con `href="/blog"`.

<base href>

La etiqueta HTML `<base href="/">` especifica la dirección URL base que se utilizará para todas las direcciones URL relativas contenidas dentro de un documento. Sólo puede haber un elemento `<base>` en un documento. Es importante que nuestro archivo `index.html` contenga esta etiqueta para indicar la carpeta raíz de nuestra aplicación, ya que el Router lo empleará para moverse por las rutas. Ejemplo: Si nuestra aplicación se ubica en `www.midominio.es/2021/miapp`, sería `<base href="/2021/miapp">`

Nuestra aplicación, si no hemos cambiado la estructura por defecto que crea AngularCLI, deberemos tener algo parecido a esto en el `index.html`:

```
<!-- index.html -->
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Nombre</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root>Cargando...</app-root>
</body>
</html>
```

 **Consejo del día:** Por defecto, `<app-root>` no trae ningún contenido. Es buena idea incluir un texto con "Cargando..." o "Loading..." porque antes de que Angular empiece a renderizar el contenido de las vistas, durante unos segundos mostraremos ese mensaje en lugar de una pantalla en blanco.

Añadir routing manualmente

Cuando creamos una aplicación usando AngularCLI nos pregunta si queremos añadir el routing. Si se lo indicamos, ya nos crea toda la estructura del Router necesaria lista para empezar a usarla. En el siguiente punto explicamos lo que nos crea.

En este manual nos centraremos en usar un Routing ya creado, si lo queremos crear manualmente o bien se lo queremos añadir con posterioridad a una aplicación ya creada, puedes seguir los pasos descritos en el siguiente punto para recrear el *Router*, o bien en la [documentación oficial](#).

Usar el routing creado por AngularCLI

Cuando creamos una app con AngularCLI, el asistente `ng new` nos pregunta si queremos añadir el routing. Se recomienda añadirlo siempre ya que nos ahorra muchos pasos en un futuro, aunque pensemos inicialmente que no vamos a necesitarlo.

Como punto de partida, vamos a explicar lo que ya tenemos, y después veremos su configuración:

Ya tenemos un módulo llamado `app-routing.module.ts`, el cual está añadido al módulo principal `app.module.ts`.

```
//app-routing.module.ts
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [];

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

```
//app.module.ts
//Módulos
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module'; //<-- Aquí

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

@NgModule({
  declarations: [
    AppComponent,
    . . .
  ],
  imports: [
    BrowserModule,
    AppRoutingModule //<-- y aquí
  ],
  . . .
})
export class AppModule { }
```

Una vez visto lo que tenemos, vamos a explicar lo que queremos. Queremos que nuestra aplicación, dependiendo de la ruta en la que nos encontremos, nos muestre automáticamente el componente que esté asociado a esa ruta. Por ejemplo, si estamos en www.midominio.com/miapp/listados-tarjeta, me muestre en la pantalla el componente de *Listado-tarjeta*.

Esa asociación la vamos a realizar en la constante llamada *routes* del módulo **AppRoutingModule**, de la siguiente forma:

```
//app-routing.module.ts
imports . . .

const routes: Routes = [
  { path: "", component: HomeComponent, pathMatch: "full" },
  { path: "listado-tabla", component: ListadoTablaComponent },
  { path: "listado-tarjetas", component: ListadoTarjetasComponent },
  { path: "**", redirectTo: "" }
];
...
```

La constante *routes* es un Array de objetos de tipo **Routes**, [que posee algunos atributos](#) pero por ahora nos centraremos en unos cuantos:

- ***path***: Es la URL que se comparará con la URL del navegador. Si coinciden, nos llevará al componente indicado.
- ***component***: Es el componente que se instanciará cuando la ruta coincide.
- ***redirectTo***: Es una URL a la que redirigir cuando la ruta coincide. Será absoluta si la ruta comienza por /. En caso contrario, la ruta se entenderá como relativa.
- ***pathMatch***: Este atributo determina la estrategia de comparación de rutas. Puede ser “prefix” (por defecto) o “full”. De forma predeterminada, el enrutador verifica los elementos de la url empezando por la izquierda y se detiene cuando encuentra una coincidencia. Por lo que /equipo/11/user coincidiría con /equipo/:id. Para evitar coincidencias no deseadas, es importante usar “full” cuando redirigimos a rutas vacías (una ruta vacía es un prefijo de cualquier URL, por lo que aplicaría la redirección cuando vayamos a cualquier destino, provocando un bucle infinito.) Resumiendo, deberemos usar “full” al ir a “”.

En la última posición del array, se puede indicar con **path : “**”**, que cualquier otra ruta que no coincida con ninguna de las anteriores, nos lleve a una url o a un componente concreto. Es bastante habitual crear un componente tipo **error404.component** al cual nos mostraría la aplicación si ponemos cualquier url que no hayamos indicado expresamente. **Es muy importante que este elemento sea el último** del array en *routes*.

Ya podemos probar la primera parte de nuestro Routing. Podemos ir a cualquier url de las mapeadas en el array de rutas, y veremos cómo nos deja ahí. Si por el contrario escribimos una ruta no contemplada, nos llevará a la raíz de nuestro programa. Si hacemos algún **console.log** en los componentes, veremos que son instanciados cada vez que intentamos ir a ellos. Se aconseja hacer esto para entender mejor lo que pasará después.

Sólo nos queda indicarle a Angular **DONDE** queremos que renderice esos componentes que se están instanciando al usar el sistema de rutas. Y eso lo hacemos colocando la etiqueta **<router-outlet>**.

Como nosotros mostramos los elementos en el **cuerpo.component**, ahí es donde usaremos la etiqueta:

```
<!-- cuerpo.component.html -->
<main>
  <section class="container">

    <div class="row">
      <!-- Columna izquierda -->
      <div class="col"></div>

      <!-- Columna central -->
      <div class="col-8">

        <!--
          La etiqueta router-outlet será sustituida por
          el componente que esté asociado a la url activa
        -->
        <router-outlet></router-outlet>

      </div>

      <!-- Columna derecha -->
      <div class="col"></div>
    </div>

  </section>
</main>
```

Ahora al irnos manualmente a las URLs indicadas en **routes**, ya sí que el componente asociado se renderizará en el lugar donde pongamos la etiqueta **<router-outlet>**.

Aunque todavía hay un problema. Comprobarás que al pulsar los enlaces vuelve a recargar completamente la ventana del navegador al pasar de una url a otra. Este problema no es sólo estético, sino que, si se vuelven a cargar los archivos de Angular, si teníamos algo en los servicios o cualquier otro dato en memoria, se perderán. Todo eso y algunas cosas se arreglarán usando **routerLink** que veremos a continuación.

Como práctica, puedes hacer el componente **error404** y que cualquier petición a una url no existente se redireccionara allí.

RouterLink

El **RouterModule** nos provee de un nuevo atributo que usar en las vistas HTML para poder hacer enlaces a otras rutas sin tener que recargar TODA la página, por lo que la aplicación dará la sensación de no moverse de la ruta actual y sólo recargará la parte nueva a renderizar por **<router-outlet>**.

En la barra de navegación, sólo tendríamos que cambiar los atributos **href** por **routerLink**, quedando de la siguiente forma:

```
<!-- navegacion.component.html -->
. . .
<li class="nav-item">
  <a class="nav-link active" aria-current="page" routerLink="home">Home</a>
</li>
<li class="nav-item">
  <a class="nav-link" routerLink="listado-tabla">Tabla</a>
</li>
<li class="nav-item">
  <a class="nav-link" routerLink="listado-tarjetas">Tarjetas</a>
</li>
. . .
```

Ahora la navegación funciona fluida, sin recargar el navegador. Si pusiste los **console.log** en el constructor de los componentes, podrás observar que ahora se van acumulando los mensajes al pasar de una vista a otra, cuando antes se perdían porque el navegador recargaba la página.

Todo funciona perfecto, pero todavía falta un último detalle. Ahora mismo SIEMPRE muestra el primer enlace como activo, ya que es el que tiene la clase **active**. Deberíamos hacer ahora que se pusiese activo el enlace en el que estoy de forma dinámica y que vaya cambiando cuando cambio. Pues es algo sumamente fácil usando el atributo **routerLinkActive**.

Es un atributo que recibe como valor la clase que deberá aplicarse al elemento en caso de que sea el elemento activo. Cuando deje de serlo, dejará de aplicarse esa clase.

Aplicamos el atributo **routerLinkActive** en cada enlace usado con **routerLink**. Quedaría así:

```
<ul class=". . .">
  <li class="nav-item">
    <a class="nav-link" routerLink="home" routerLinkActive="active">
      Home
    </a>
  </li>
. . .
```

Encontrarás toda la información adicional de **RouterLink** en la [documentación oficial](#).



<https://stackblitz.com/edit/ng-heroes8-router-module>



Cambio de la clase Heroes a la interfaz

Como ya vimos anteriormente, sería más apropiado utilizar Interfaces en lugar de clases para los modelos, ya que es habitual que los datos provengan de algún servicio web y nos los proporcione una llamada Ajax.

En estos casos no harás “new” instanciando una clase, si no que el API nos lo devolverá en JSON. En estos casos es especialmente idóneo definir una interfaz y declarar el objeto que nos devuelva el servidor con el tipo definido por esa interfaz. Las interfaces no poseen una inicialización o métodos, ya que lo normal es que esta tarea la realicen los servicios.

De todas formas, podríamos llegar a mapear cada objeto recibido por un API REST en un objeto instanciado de una clase, para poder usar los métodos. No nos merece la pena [hacer todo lo necesario](#) por usar dos métodos que no nos ahorran prácticamente nada, por lo que cambiamos a una interfaz y así vemos también como sería.

Así que modificaremos nuestro modelo *Heroe*, y lo cambiamos de clase a interfaz. Tendremos que eliminar los métodos `esVillano()` y `esPoderoso()`, así como eliminar de `listado-tarjetas.component.html` el uso que hacíamos de esos métodos.

Así quedarían las modificaciones:

```
// modelos/heroе.model.ts
export interface Heroe {
    nombre:string;
    poder:number;
    honor:number;
    ataque:number;
    defensa:number;
    vida:number;
    imagen:string;
}
```

```
<!-- listado-tarjetas.component.html -->
<div class="">

    <!-- Cada tarjeta individual -->
    <ng-container *ngFor="let heroe of heroes">
        <div class="card my-3"
            [class.shadow]="true"
            [class.border]="heroe.honor < -50 || heroe.poder > 200"
            [class.border-4]="heroe.honor < -50 || heroe.poder > 200"
            [class.border-info]="heroe.poder > 200"
            [class.border-danger]="heroe.honor < -50"
        >
        . . .
    </ng-container>
</div>
```

Observables

Antes de pasar al siguiente punto (consultas AJAX) es importante entender un nuevo tipo de dato. Los **observables**.

Los **observables** representan una colección de futuros valores, es decir, datos que probablemente aún no estamos recibiendo.

Las comunicaciones entre navegadores y servidores son bastante más lentas que las operaciones en memoria. Por tanto, deben realizarse de manera asíncrona para garantizar una buena experiencia de usuario.

El patrón Observable, fue implementado por Microsoft en la librería [Reactive Extensions](#), aka RxJs. El equipo de Angular decidió utilizarla para el desarrollo de las comunicaciones asíncronas.

Imagina una [máquina de hacer helados](#). Al principio de todo el proceso, antes de que la máquina arranque a fabricar los helados, sabemos que habrá un depósito con toda la mezcla lista para salir a través del molde devolvernos los helados uno a uno. No sabemos a priori cuántos helados nos dará la máquina, pero sabemos que serán muchos, así que habrá que estar preparado. Esta es la representación real de un **observable**, un conjunto de N cantidad de objetos que podremos recibir en un momento determinado, pero no sabemos exactamente cuándo.

La programación reactiva (como la de Angular, React, etc.) se centra en los **observables** porque tienen ciertas características que los diferencian de las funciones estándar.

Las funciones normales solo pueden retornar un valor una vez, mientras que los observables pueden retornar un valor múltiples veces. Las promesas no se pueden cancelar, mientras que los observables si podemos. Nos podemos suscribir y "unsuscribir", por lo que recibiremos los datos (y dejaremos de recibirlos) según nos convenga.

Ya veremos cómo definirlos y usarlos al realizar una petición HTTP a un servicio Rest en el siguiente punto.

Peticiones HTTP (Ajax)

AJAX, es un acrónimo de ***Asynchronous JavaScript And XML*** (JavaScript asíncrono y XML). Es una técnica de desarrollo web que se ejecuta en el cliente mientras se mantiene una comunicación asíncrona con el servidor en segundo plano, por lo que es posible realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones.

Angular posee un módulo con todas las herramientas necesarias para realizar peticiones y respuestas HTTP, llamado ***HttpClientModule***. Lo deberemos importar en la mayoría de las ocasiones en el módulo raíz de la aplicación (***app.module***) o bien en algún módulo dedicado a acceso a datos.

Dicho módulo, incluye un servicio llamado ***HttpClient***, que es el inyectaremos como clase en cada componente que vaya a realizar alguna petición Ajax.

En nuestra aplicación Héroes, los componentes le piden al servicio la lista de héroes, y el servicio es el encargado de “buscarse la vida” para obtener esa información. Ahora mismo genera la lista con un array creado directamente en duro o *hardcoded*, pero tan sólo deberemos modificar el servicio para que nuestra aplicación busque la información en cualquier servicio REST en lugar de generarla por ella misma.

Los cambios por aplicar en nuestra aplicación serían los siguientes:

1. Importar el ***HttpClientModule*** en nuestro ***app.module.ts*** como un módulo más:

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

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

2. Modificamos nuestro servicio ***heroes.service.ts*** para que haga una consulta Ajax y me devuelva un ***Observable*** (en lugar de crear “a mano” el ***Array<Heroe>***). Vemos el código y lo comentamos:

```
// servicios/heroes.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Heroe } from '../models/heroe.model';

@Injectable({
  providedIn: 'root'
})
export class HeroesService {
  //Este va a ser nuestro servidor REST
  private REST_API_URL: string = "https://heroes.mocklab.io";

  //Creamos un atributo y lo inicializamos directamente
  constructor(private httpClient: HttpClient) {}

  /**
   * Devuelve un Observable con un array de héroes
   * @returns Observable con la lista completa de héroes de nuestra app
   */
  public getHeroes(): Observable<Heroe[]> {
    let datos = this.httpClient.get<Heroe[]>(this.REST_API_URL + "/heroes");
    return datos;
  }
}
```

Empecemos por orden desde el principio:

- ▶ Hacemos los imports de `HttpClient` y `Observable`. Podemos dejar que lo importe automáticamente Visual Studio cuando los usemos directamente.
- ▶ Eliminamos el atributo `urlImagenesExt` ya que usaremos las rutas directamente en la vista. Sólo para usar esta variable nos veíamos obligados a inyectar el servicio completo, y no era eficiente. Aunque así aprendimos que se podía 😊.
- ▶ Creamos un nuevo atributo que contendrá la ruta base externa de nuestra API. Es un *mockable* ya configurado que simulará un *backend* al que realizaremos consultas. Está preparado para simular un retardo aleatorio entre 400 y 2000 ms. Así podremos usar barras de progreso en un futuro.
- ▶ Inyectamos directamente al constructor el atributo `httpClient` del tipo `HttpClient`, y ya lo tenemos disponible en todos los métodos que hagamos en el `heroes.service`.
- ▶ En nuestro método `getHeroes()`, ya no devuelve un `Array<Heroe>` (o `Heroe[]`), si no un `Observable<Heroe[]>`.
- ▶ La llamada Ajax en sí, es bastante simple. El objeto `httpClient`, tiene un método `.get()` el cual recibe como parámetro la url de nuestro REST y devuelve un Observable con el resultado de esa consulta.
- ▶ En nuestro caso usaremos [http://heroes.mocklab.io/heroes](https://heroes.mocklab.io/heroes) (puedes hacer clic para comprobar que es lo que recibirá el `Observable`).

- ▶ Hemos declarado una variable llamada datos, por si queremos imprimirla o juguetear con ella antes de retornarla.
 - ▶ Y nuestro servicio ya estaría listo. Ahora nos queda modificar el uso que hacemos de él, tanto en `listado-tabla` como en `listado-tarjeta`.
 - ▶ Podríamos pensar que en nuestro servicio podemos hacer la consulta, y devolver directamente el `Array<Heroe>` que necesitamos, pero al ser consultas asíncronas, no podemos hacer la petición y devolver la respuesta en el mismo servicio, ya que lo mismo cuando lo hagamos tarda en llegar y el flujo sigue su curso. El programa no se detiene hasta obtener una respuesta. Es por ese motivo por el que devuelve un `Observable`. Es un posible valor futuro, pero que tendremos que suscribirnos y esperar noticias de él.
3. Aquel componente que antes consumiera el servicio (`listado-tabla` y `listado-tarjeta`), antes obtenía un `Array<Heroe>`, ahora obtiene un `Observable<any>`. Vamos a comparar el código que tenía antes `listado-tabla.component.ts` y podemos compararlo con los nuevos cambios.

```
// listado-tabla.component.ts
import { Component, EventEmitter, OnInit, Output } from "@angular/core";
import { Heroe } from "../../modelos/heroe.model";
import { HeroesService } from "../../servicios/heroes.service";

@Component({
  selector: "app-listado-tabla",
  templateUrl: "./listado-tabla.component.html",
  styleUrls: ["./listado-tabla.component.css"]
})
export class ListadoTablaComponent implements OnInit {
  public heroesService: HeroesService;
  public listadoHeroes: Array<Heroe>;

  @Output() public eventoFavorito = new EventEmitter<Heroe>();

  constructor(heroesService: HeroesService) {
    this.heroesService = heroesService;
    this.listadoHeroes = new Array<Heroe>();
  }

  ngOnInit(): void {
    this.listadoHeroes = this.heroesService.getHeroes();
  }

  public onSeleccionarFavorito(heroeFavorito: Heroe) {
    console.warn("El hijo dice: se seleccionó a ", heroeFavorito);
    this.eventoFavorito.emit(heroeFavorito);
  }
}
```

Y ahora veremos los cambios que vamos a efectuar, resaltados en azul.

```
// listado-tabla.component.ts
import { Component, EventEmitter, OnInit, Output } from '@angular/core';
import { Heroe } from 'src/app/modelos/heroe.modelo';
import { HeroesService } from 'src/app/servicios/heroes.service';

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

  public heroes: Heroe[];

  @Output()
  public eventoFavorito = new EventEmitter<Heroe>();

  constructor(private heroesService: HeroesService) {
    this.heroes = new Array<Heroe>();
  }

  ngOnInit(): void {
    let datos = this.heroesService.getHeroes();
    //datos.subscribe(respuesta => {}, error => {}); //Y ahora la desarrollamos
    datos.subscribe(
      (respuesta) => {
        this.heroes = respuesta.heroes;
      },
      (error) => {
        console.error("Error = ", error);
      }
    );
  }

  public onSeleccionarFavorito(heroeFavorito: Heroe) {
    console.warn("El hijo dice: se seleccionó a ", heroeFavorito);
    this.eventoFavorito.emit(heroeFavorito);
  }
}
```

Expliquemos paso a paso cada uno de los cambios:

- ▶ Declarar el servicio como atributo, directamente en el constructor. Esto se podía haber quedado como estaba, pero así vamos usando la sintaxis azucarada de TypeScript 😊.

- ▶ Y en el **ngInit()** es donde está la mandanga. Al igual que antes, es donde hacemos la llamada al servicio y su resultado lo guardamos en una variable. La diferencia radica sólo en que antes directamente obteníamos el **Array<Heroe>** y ahora obtenemos el **Observable**, el cual lo guardamos en una variable llamada **datos**.

- ▶ El **Observable** es como un contrato que nos asegura que en cuanto la consulta asíncrona termine, ahí se guardarán los datos. Por eso tenemos que “suscribirnos” para ejecutar una función de tipo flecha en cuanto la petición HTTP se termine.

```
datos.subscribe( (respuesta) => {}, (error) => {} );
```

- ▶ Ahí estamos diciendo a **datos**, que cuando termine la petición GET a la url indicada en la consulta Ajax (la hicimos en el servicio), ejecute la función {} recibiendo el parámetro indicado entre los (). Así no necesitamos usar la palabra clave **function**.
- ▶ Esa línea la expandimos para poder escribir en el interior de las funciones más cómodamente:

```
datos.subscribe(
  (respuesta) => {
    //Aquí la petición habrá terminado bien
  },
  (error) => {
    //Y aquí habrá provocado algún error
  }
);
```

- ▶ Así que ya tenemos el sitio donde poder escribir las líneas de código que queramos en el momento en el que el Observable se “rellene”. Tan sólo debemos extraer lo que queramos de él y guardarlo dónde queramos (**this.heroes**).
- ▶ Lo normal sería imprimir el objeto obtenido para comprobar su estructura:

```
datos.subscribe(
  (respuesta) => {
    console.log("Respuesta = ", respuesta);
  },
  (error) => {
    console.error("Error = ", error);
  }
);
```

- ▶ Si ejecutamos la aplicación ahora, y, vamos a la opción de **Tabla**, veremos lo siguiente por la consola de depuración del navegador:

```

Res lista-de-tabla.component.ts:27
puesta =
  {origen: "/heroes", mensaje: "listado completo de Héroes", heroes: Array(11)} 1
    ▼ heroes: Array(11)
      ▼ 0:
        ataque: 80
        defensa: 93
        honor: 94
        imagen: "https://borilio.g...
        nombre: "Superman"
        poder: 350
        vida: 100
        ► __proto__: Object
      ▼ 1:
        ataque: 60
        defensa: 75
        honor: 75
        imagen: "https://borilio.g...
        nombre: "Batman"
        poder: 50
        vida: 100
        ► __proto__: Object
  
```

- Podemos comprobar que en la variable `respuesta` hemos recibido el objeto JSON que nos ha dado <https://heroes.mocklab.io/heroes>. Y que en el listado no mostramos nada todavía. Tan sólo debemos elegir la información que deseemos, ya que el REST API puede darnos la información de cualquier otra forma. En nuestro caso, nos devuelve un array de objetos. Para hacer referencia al array, usaremos `respuesta`. Ten en cuenta que siempre deberemos tener claro la estructura del JSON que recibamos para saber exactamente dónde y cómo están los datos. Por eso es una buena práctica investigar el objeto una vez recibido, antes de asignarlo a donde deba ir.
- Así que ya teniendo claro que el array está directamente en `respuesta`, ya sólo debemos asignarlo al array declarado como atributo en nuestro componente (`this.heroes`).

```

datos.subscribe(
  (respuesta) => {
    this.heroes = respuesta;
  },
  (error) => {
    console.error("Error = ", error);
  }
);
  
```

- Si lo recibido “encaja” con la interfaz definida en el modelo, todo irá bien y ya nos saldrá el listado de héroes en nuestra tabla.

Imagen	Nombre	Vida
	Superman	100
	Batman	100
	Capitana Marvel	100

- ▶ Si hay cualquier error en la consulta (p.ej.: url mal construida) podemos investigar igualmente el objeto error, extraer algo si queremos y mostrarlo en la página. (Podemos crear un atributo llamado error y hacer lo mismo que con los héroes, `this.error = error`. Y en el HTML mostrar `{{this.error.mensaje}}`, por ejemplo.)
 - ▶ La función del error es opcional y podemos omitirla, dejando sólo `(respuesta) => {}` como opciones del `subscribe()`.
4. En `listado-tabla.component.html` podemos mostrar un componente con una barra de progreso (o un texto), que sólo se muestra si `this.heroes.length === 0`, con lo que conseguimos que los segundos que dure el proceso de obtener los datos, se muestre una barra y una vez el observable se cargue con los datos, desaparecerá.

```
<!-- listado-tabla.component.html -->
<!-- tarjeta -->
<div class="card shadow my-3">
  <div class="card-body">
    <h3 class="card-title">Lista de héroes</h3>
    <app-barra-cargando *ngIf="this.heroes.length === 0">
      </app-barra-cargando>
    <table *ngIf="this.heroes.length > 0" ...>
      . . .
    </table>
  </div>
</div>
```

Si te animas, también puedes intentar agregar [Angular material](#), además de [barras de progreso muy chulas](#), encontrarás muchos otros componentes y aspectos gráficos que modificarán mucho el aspecto de tus aplicaciones. Es muy fácil de usar, merece la pena.

Por último, no podemos olvidar el [*Listado-tarjetas*](#), que deberemos hacer lo mismo que hemos hecho para el listado-tabla. La parte del servicio ya la tenemos lista, sólo tendremos que modificar `listado-tarjetas.component.ts`, que es donde usábamos el servicio, repitiendo los pasos que hicimos en `listado-tabla.component.ts`. La vista seguirá funcionando sin aplicar ningún cambio.

💡 **¿Sabías que...?** Puedes imprimir un objeto json directamente en las vistas usando una [pipe](#) incluida en Angular.

`<p>{{this.heroes | json }}</p>` nos mostrará el contenido del objeto json sin mostrarlo por la consola de depuración del navegador.



<https://stackblitz.com/edit/ng-heroes9>

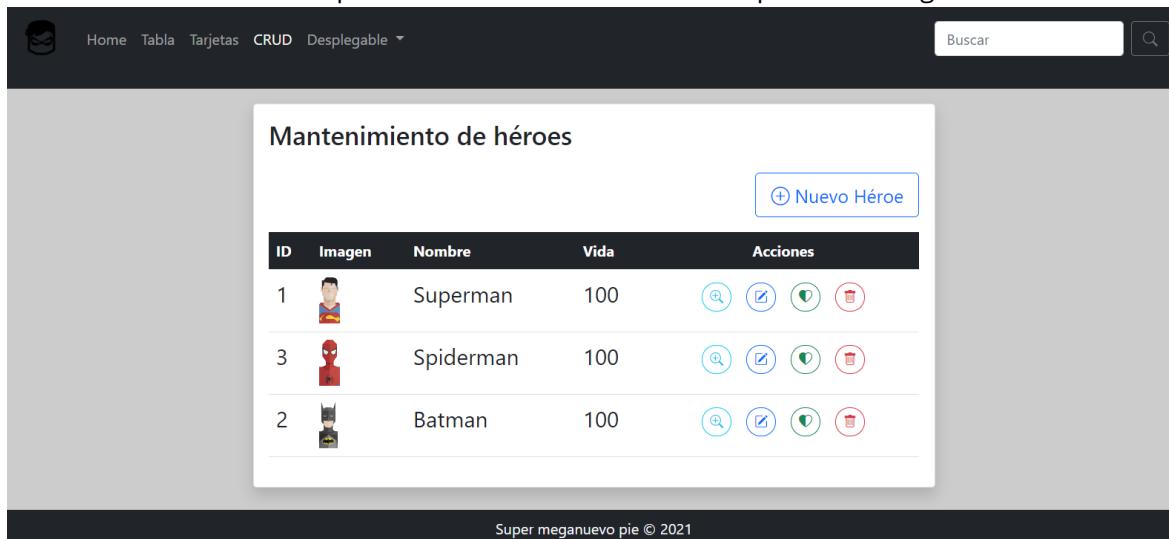
Novedades antes del CRUD

Antes de integrar Firebase a nuestro proyecto, vamos a dejar preparados una serie de cambios en nuestra aplicación.

Todo lo que hasta ahora funciona, no lo vamos a modificar, así podremos tener a modo de ejemplo, las consultas Ajax y todo lo nuevo que veremos con Firebase. Ambos por separado.

Vamos a explicar los cambios que hemos realizado, y al final encontrarás un enlace a stackblitz con todo el código aplicado. Puedes intentar realizar tú los cambios y así reforzarás lo visto hasta ahora:

1. Añadimos una nueva opción en un hueco que teníamos libre en la barra de navegación. **CRUD**. Ahí mostraremos un nuevo componente llamado *Listado-crud* parecido al siguiente:

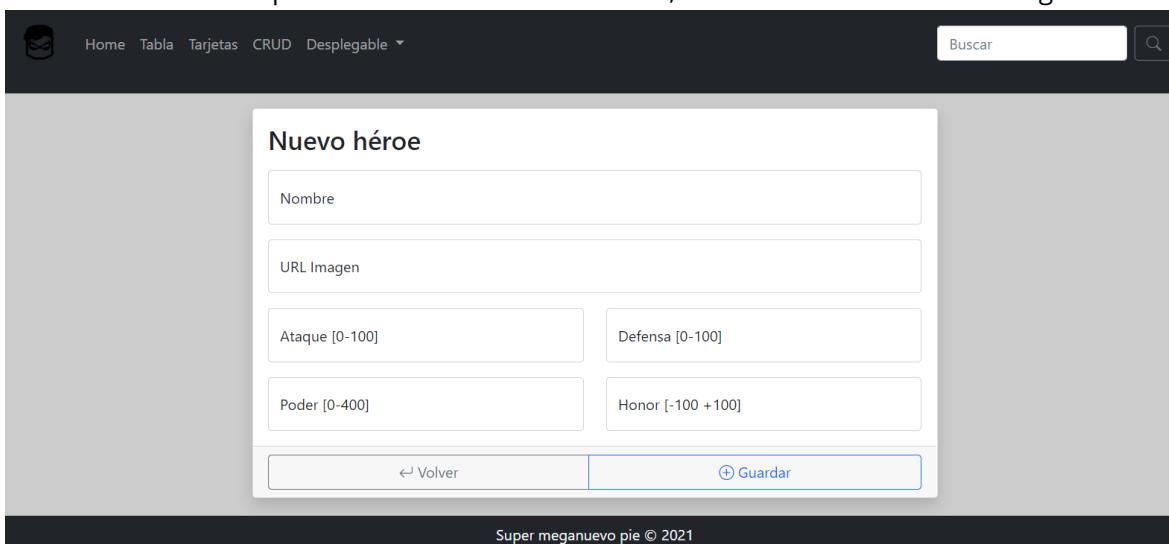


The screenshot shows a web application interface. At the top, there's a navigation bar with icons for envelope, Home, Tabla, Tarjetas, CRUD, Desplegable, and a search bar. Below the navigation is a title 'Mantenimiento de héroes'. Underneath is a table with columns: ID, Imagen, Nombre, Vida, and Acciones. The table contains three rows of superhero data:

ID	Imagen	Nombre	Vida	Acciones
1		Superman	100	
3		Spiderman	100	
2		Batman	100	

At the bottom of the component, it says 'Super meganuevo pie © 2021'.

2. Puedes reutilizar el componente *listado-tabla*, y añadir una columna a la tabla con varios botones: **info, editar, curar y borrar** (no hacen nada por ahora) y otra columna con la **id**.
3. Hemos incluido un botón para añadir un **Nuevo Héroe**. Al pulsar el botón de nuevo Héroe, me llevará a un nuevo componente llamado *nuevo-heroe*, con un formulario como el siguiente:



The screenshot shows a 'Nuevo héroe' (New Hero) form. At the top, there's a navigation bar with icons for envelope, Home, Tabla, Tarjetas, CRUD, Desplegable, and a search bar. Below the navigation is a title 'Nuevo héroe'. The form has several input fields and buttons:

- A large input field labeled 'Nombre'.
- A smaller input field labeled 'URL Imagen'.
- Two side-by-side input fields: 'Ataque [0-100]' and 'Defensa [0-100]'.
- Two side-by-side input fields: 'Poder [0-400]' and 'Honor [-100 +100]'.
- At the bottom are two buttons: ' Volver' and '+ Guardar'.

 At the very bottom, it says 'Super meganuevo pie © 2021'.

4. Si pulsamos **Volver**, nos llevará de nuevo a *Listado-crud*. **Guardar** no hace nada por ahora.

5. El contenido de la tabla se recibe como al principio, con un objeto **json** creado directamente en el componente. Próximamente haremos que cargue sus valores con la base de datos de Firebase.

Aquí tienes el enlace de stackblitz con el código completo del proyecto, antes de empezar a añadir Firebase y para realizar un CRUD.



<https://stackblitz.com/edit/ng-heroes10-pre-firebase>

CRUD en Angular usando Firebase de Google

Firebase es una plataforma para el desarrollo de aplicaciones web y móviles, desarrollada por Google en 2014. Ubicada en la nube e integrada con Google Cloud Platform, usa un conjunto de herramientas para la creación y sincronización de proyectos, haciendo posible el fácil crecimiento de usuarios y la monetización de la aplicación.

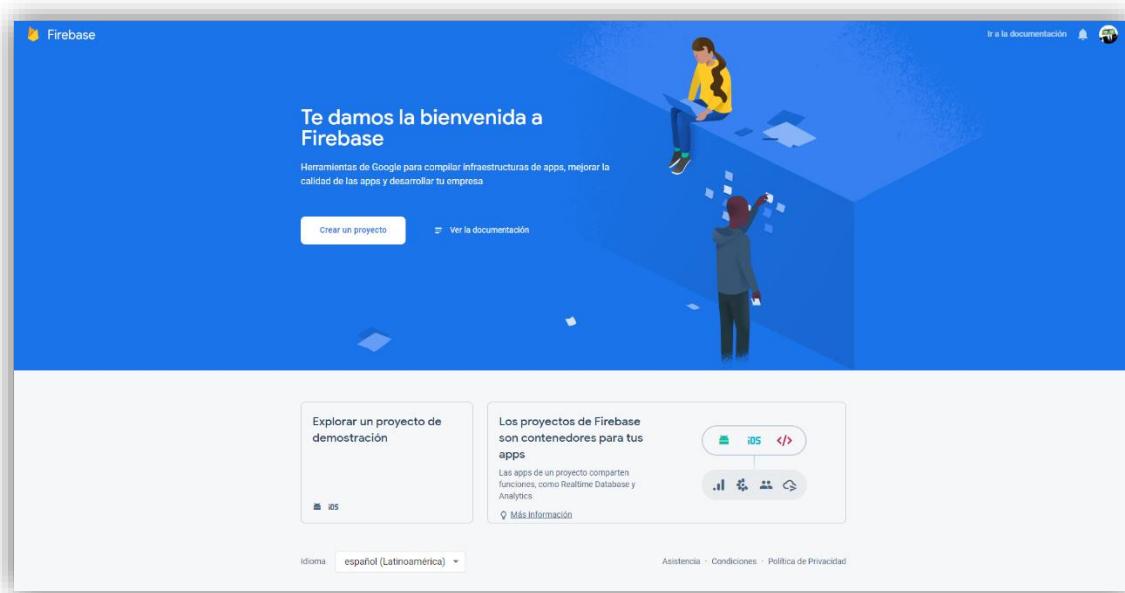
Posee una serie de ventajas para los desarrolladores, por ejemplo:

- Sincronizar fácilmente datos de los proyectos entre distintas apps, sin tener que administrar conexiones o escribir lógica de sincronización compleja.
- Uso de herramientas multiplataforma. Se integra con aplicaciones móviles (Android e iOS), web, Unity, Java, C++, etc.
- Crea proyectos sin necesidad de un servidor.
- Dotar a tus aplicaciones de logueo con cuenta de Google, Facebook, GitHub o Twitter fácilmente, o bien por método clásico de usuario y contraseña propios. Aporta funcionalidades como la recuperación y verificación de cuentas, por correo o SMS.
- Una gran documentación oficial, activos en GitHub y Stackoverflow. Tienen un canal de YouTube y soporte gratuito para sus usuarios.

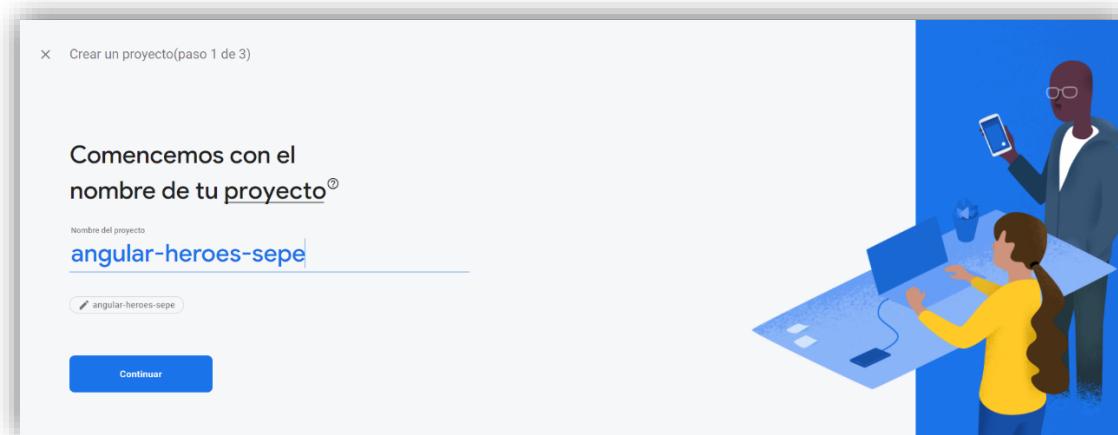
Crear el proyecto en Firebase

Para crear el proyecto, nos vamos a página de Firebase <https://firebase.google.com/>, iniciamos sesión con la cuenta de Google que queramos asociar a nuestros proyectos, y pulsamos comenzar ([Ir a la consola](#))

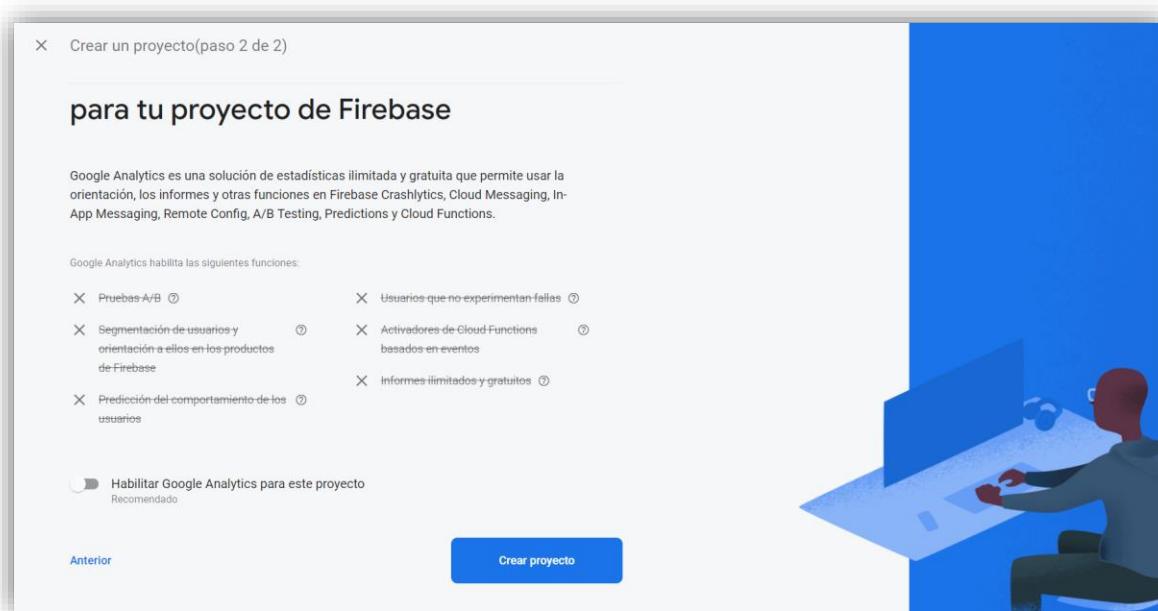
1. En la consola de Firebase nos aparecerán todos nuestros proyectos para poder administrarlos. Pulsaremos **Crear un proyecto**.

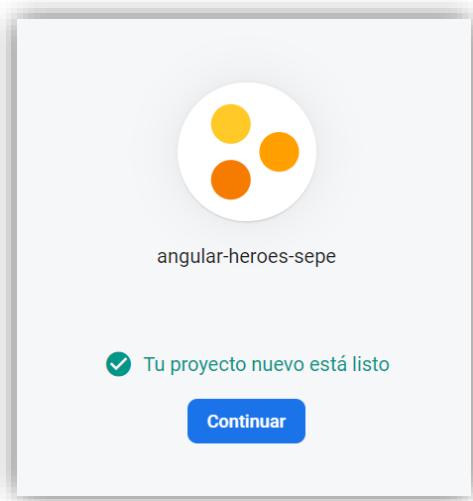


2. Introducimos un nombre para el proyecto. Podemos editar la id del proyecto y poner alguna más amigable.



3. Puedes habilitar Google Analytics si quieres hacer uso de esta herramienta.
 4. Pulsamos en **Crear proyecto**.

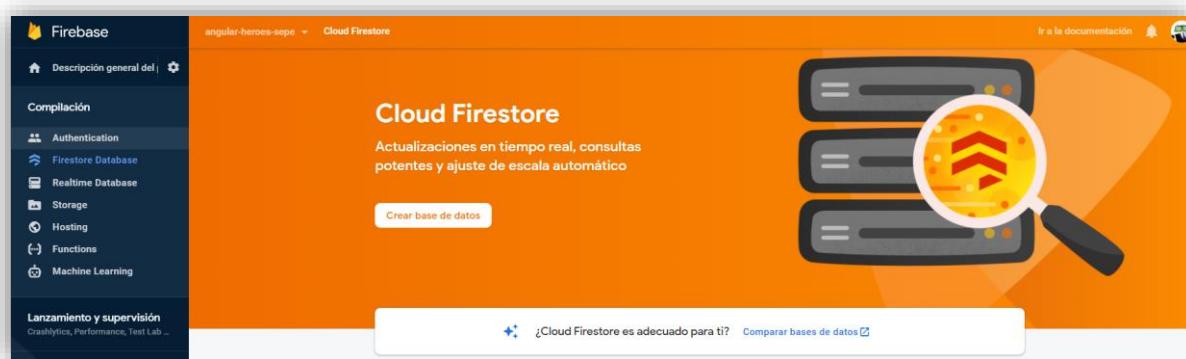




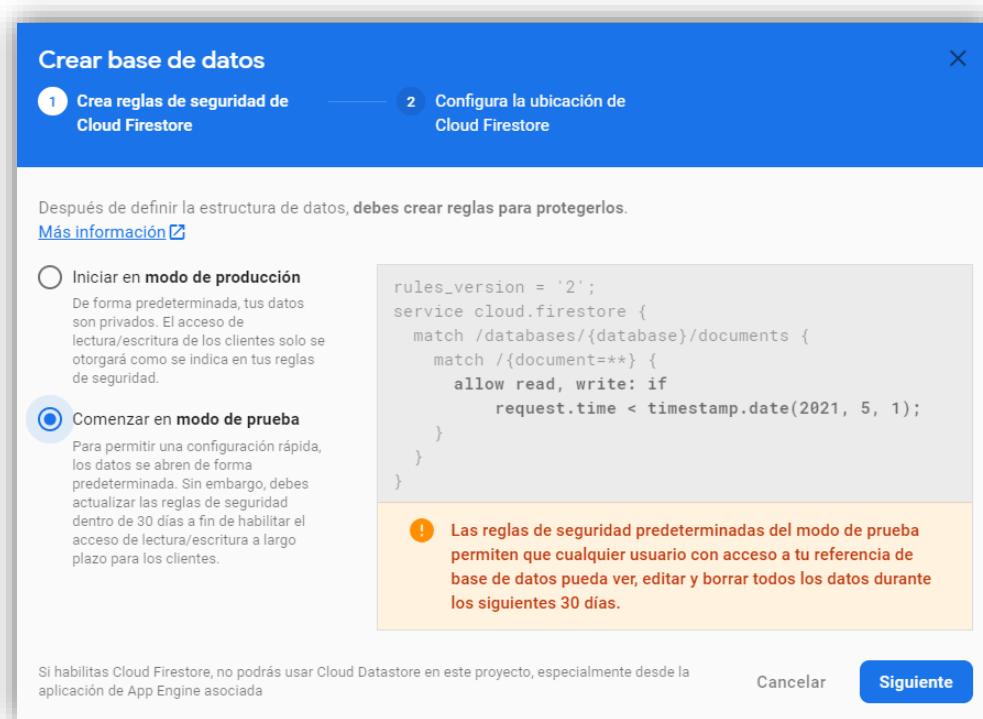
5. Ya tendremos nuestro proyecto creado. Si pulsamos **Continuar**, nos llevará al panel de control de nuestro proyecto.

Y ya tendremos creado nuestro proyecto en Firebase. Vamos a crearle una base de datos e insertar algunos registros para poder trabajar más tarde con ellos.

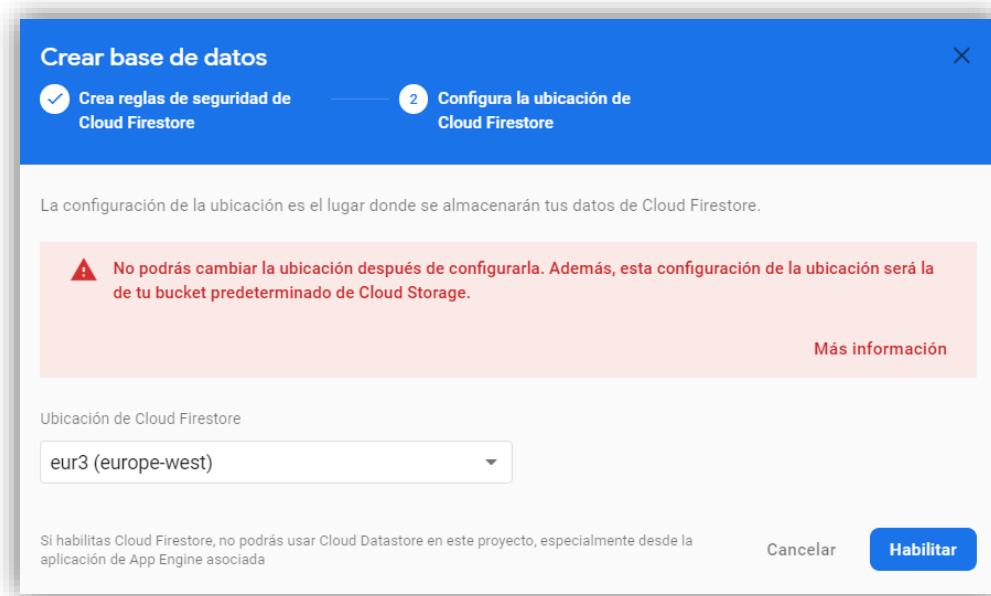
6. Pulsamos en la izquierda en **Firestore Database** y a continuación en **Crear base de datos**.



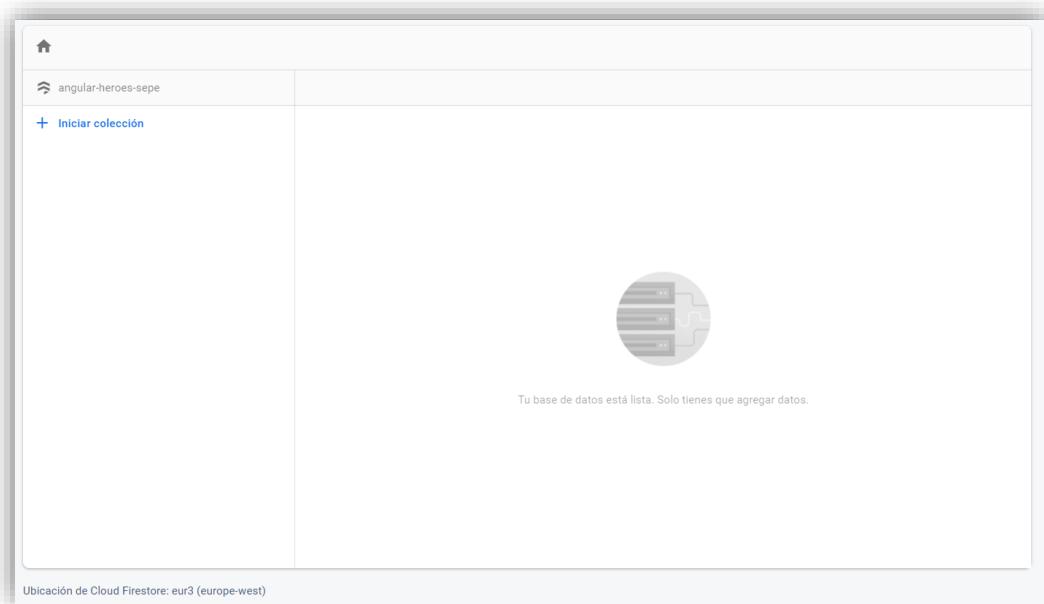
7. Después nos muestra las reglas de seguridad para Cloud Firestore. Podemos elegir Modo producción directamente, para que sean privados desde un principio, o empezar en modo prueba, para más tarde cambiarla a modo producción. Nosotros empezaremos en modo prueba para hacerlo más rápido inicialmente. Deberemos cambiar esto en los próximos 30 días.



8. En el siguiente paso seleccionamos la ubicación física dónde nuestros datos se almacenarán. Seleccionaremos eur3 (europe-west).



9. Ya tendremos la base de datos creada. Ahora le añadiremos algunos datos. Pulsaremos en **Iniciar colección**. A continuación, escribimos “heroes” como ruta superior del documento.



10. Ahora añadimos una colección llamada “heroes” (algo parecido a una tabla) y le añadimos un documento (algo parecido a un registro). Como los Firebase ya gestiona la id de cada documento, no crearemos un campo para la id, si no que dejaremos que genere las id automáticamente. El resto de los campos, los rellenamos tal y como ya conocemos:

Agrega un documento

Ruta superior
[/heroes](#)

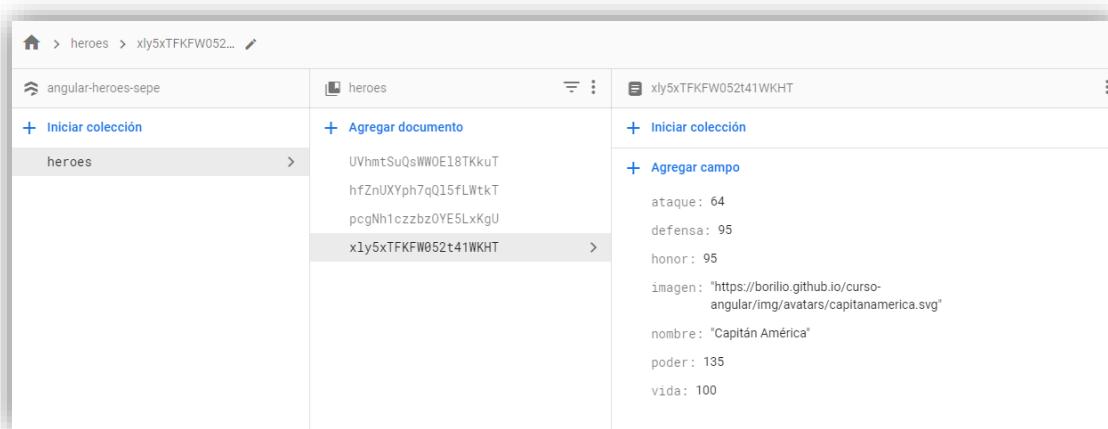
ID de documento [?](#)
xly5xTFKFW052t41WKHT

Campo	Tipo	Valor
nombre	string	Capitán América
imagen	string	https://borilio.git/
ataque	number	64
defensa	number	95
poder	number	135
honor	number	95
vida	number	100

+ Agregar campo

[Cancelar](#) [Guardar](#)

11. Y podemos repetir el punto anterior con un par de nuevos documentos, para cuando hagamos la conexión real, podamos recuperar estos "registros".

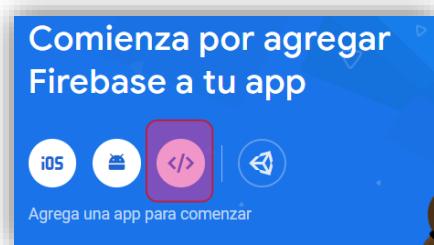


The screenshot shows the MongoDB Compass interface with three documents in the 'heroes' collection:

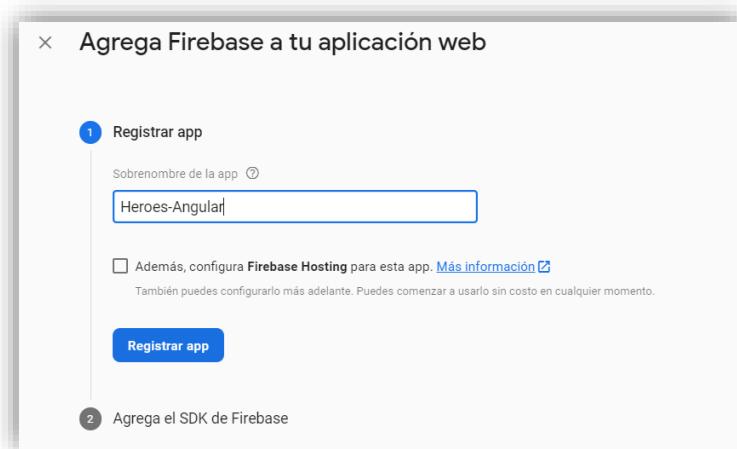
- Document 1:** ID: UVhmtSuQsWWOE18TkkuT
 - campo: nombre, tipo: string, valor: Capitán América
 - campo: imagen, tipo: string, valor: <https://borilio.git/>
 - campo: ataque, tipo: number, valor: 64
 - campo: defensa, tipo: number, valor: 95
 - campo: poder, tipo: number, valor: 135
 - campo: honor, tipo: number, valor: 95
 - campo: vida, tipo: number, valor: 100
- Document 2:** ID: hfZnUXYph7qQ15fLWtkT
 - campo: nombre, tipo: string, valor: Capitán América
 - campo: imagen, tipo: string, valor: <https://borilio.github.io/curso-angular/img/avatars/capitanamerica.svg>
 - campo: ataque, tipo: number, valor: 64
 - campo: defensa, tipo: number, valor: 95
 - campo: poder, tipo: number, valor: 135
 - campo: honor, tipo: number, valor: 95
 - campo: vida, tipo: number, valor: 100
- Document 3:** ID: xly5xTFKFW052t41WKHT
 - campo: nombre, tipo: string, valor: Capitán América
 - campo: imagen, tipo: string, valor: <https://borilio.git/>
 - campo: ataque, tipo: number, valor: 64
 - campo: defensa, tipo: number, valor: 95
 - campo: poder, tipo: number, valor: 135
 - campo: honor, tipo: number, valor: 95
 - campo: vida, tipo: number, valor: 100

Agregar Firebase al proyecto

Una vez creado el proyecto desde la página de Firebase, debemos agregarlo a nuestro proyecto. Por lo que haremos caso a la ventana anterior, y comenzaremos pulsando el ícono de web.



Seleccionamos un nombre para la aplicación dentro del proyecto Firebase y no seleccionamos el servicio de hosting. Pulsamos **Registrar app**.



Nos mostrará la siguiente información para agregarla a nuestro proyecto Angular. Ahora veremos cómo incluirla, ya que no tendremos que pegar todo ese código.

2 Agrega el SDK de Firebase

Copia y pega estas secuencias de comandos en la parte inferior de la etiqueta <body> antes de usar cualquier servicio de Firebase:

```
<!-- The core Firebase JS SDK is always required and must be listed first -->
<script src="https://www.gstatic.com/firebasejs/8.3.1.firebaseio.js"></script>

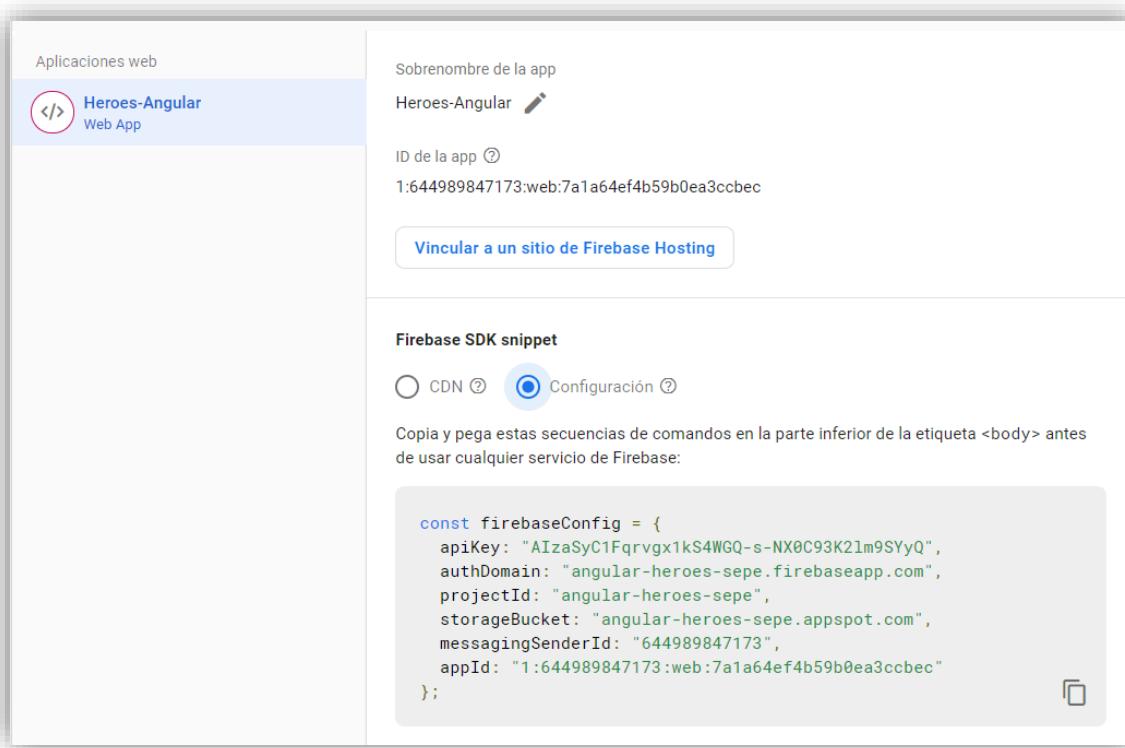
<!-- TODO: Add SDKs for Firebase products that you want to use
      https://firebase.google.com/docs/web/setup#available-libraries -->

<script>
  // Your web app's Firebase configuration
  var firebaseConfig = {
    apiKey: "AIzaSyC1Fqrvgx1kS4WGQ-s-NX0C93K2lm9SYyQ",
    authDomain: "angular-heroes-sepe.firebaseio.com",
    projectId: "angular-heroes-sepe",
    storageBucket: "angular-heroes-sepe.appspot.com",
    messagingSenderId: "644989847173",
    appId: "1:644989847173:web:7a1a64ef4b59b0ea3ccbdc"
  };
  // Initialize Firebase
  firebase.initializeApp(firebaseConfig);
</script>
```

Obtén más información sobre Firebase para la Web con estos recursos: [Primeros pasos](#), [Referencia de API del SDK web](#) y [Muestras](#).

[Ir a la consola](#)

Si volvemos a la consola, ya podremos ver el código de antes como **CDN** (por si queremos incluirlo en un proyecto sin Angular) o como una constante de **Configuración**. Copiamos el código y nos vamos a Visual Studio. A partir de aquí seguiremos los mismos pasos que en la [guía rápida oficial](#) que ofrece Firebase en su página oficial de GitHub.



The screenshot shows the Firebase console interface for a project named 'Heroes-Angular'. On the left, under 'Aplicaciones web', there is a card for 'Heroes-Angular Web App'. The right side shows the app's configuration details. Under 'Sobrenombre de la app', it says 'Heroes-Angular' with a pencil icon. Under 'ID de la app', it shows '1:644989847173:web:7a1a64ef4b59b0ea3ccbec'. Below that is a button labeled 'Vincular a un sitio de Firebase Hosting'. Under 'Firebase SDK snippet', there are two options: 'CDN' (unchecked) and 'Configuración' (checked). A note below says to copy and paste the provided code into the body tag of the index.html file. The code is:

```
const firebaseConfig = {
  apiKey: "AIzaSyC1Fqrvgx1ks4WGQ-s-NX0C93K21m9SYyQ",
  authDomain: "angular-heroes-sepe.firebaseio.com",
  projectId: "angular-heroes-sepe",
  storageBucket: "angular-heroes-sepe.appspot.com",
  messagingSenderId: "644989847173",
  appId: "1:644989847173:web:7a1a64ef4b59b0ea3ccbec"
};
```

Abrimos nuestro archivo ***environment.ts*** y añadimos esa constante a la lista de constantes que ya hay. Le ponemos también **export** para poder importarlo luego. A continuación, te mostramos como queda finalmente:

```
//environment.ts
export const environment = {
  production: false
};
export const firebaseConfig = {
  apiKey: "<INSERTA TU API-KEY AQUÍ>",
  authDomain: "angular-heroes-sepe.firebaseio.com",
  projectId: "angular-heroes-sepe",
  storageBucket: "angular-heroes-sepe.appspot.com",
  messagingSenderId: "644989847173",
  appId: "1:644989847173:web:7a1a64ef4b59b0ea3ccbec"
};
```

Nota: Deberás cambiar todas las credenciales por las tuyas, estas de aquí no te funcionarán.

También puedes agregarle este mismo objeto (**firebaseConfig**) al archivo de producción, llamado ***environment.prod.ts***, y así cuando despleguemos el proyecto funcionará directamente.

Y ya sólo nos queda instalar las librerías de Firebase con AngularCLI. Para hacer este paso, ya debemos tener el proyecto creado en Firebase. Nos vamos a la línea de comandos del proyecto y escribimos:

```
ng add @angular/fire
```

Así incluimos las librerías de Firebase y las asociamos a nuestro proyecto:

```
i Using package manager: npm
✓ Found compatible package version: @angular/fire@6.1.4.
✓ Package information loaded.
✓ Package successfully installed.
UPDATE package.json (1590 bytes)
✓ Packages installed successfully.
? Allow Firebase to collect CLI usage and error reporting information? No
? Paste authorization code here: [Insertar código de autorización aquí]
✓ Preparing the list of your Firebase projects
? Please select a project: angular-heroes-sepe (angular-heroes-sepe)
CREATE firebase.json (651 bytes)
CREATE .firebaserc (146 bytes)
UPDATE angular.json (4028 bytes)
>
```

Durante la instalación, nos abrirá el navegador automáticamente y tendremos que iniciar sesión con nuestra cuenta de Google, y nos **proporcionará un código de autorización**, el cual tendremos que pegarlo dónde nos lo pide. FirebaseCLI así podrá acceder a nuestros proyectos, y tendremos que seleccionar el proyecto asociado de Firebase que queramos usar en nuestro proyecto Angular.

Ahora nos queda configurar el ***app.module.ts***, añadiendo lo siguiente:

```
//app.module.ts
//Módulos
...
import { AngularFireModule } from '@angular/fire';
import { AngularFirestoreModule } from '@angular/fire/firestore';

//Entorno
import { firebaseConfig } from 'src/environments/environment';

@NgModule({
  ...
  imports: [
    ...
    AngularFirestoreModule,
    AngularFireModule.initializeApp(firebaseConfig)
  ],
  providers: [HeroesService],
  ...
})
export class AppModule { }
```

- Se añade los módulos de ***AngularFireModule*** y ***AngularFirestoreModule*** a los imports.

- Llamamos al método `AngularFireModule.initializeApp()`, pasándole el objeto de configuración de Firebase que hemos colocado previamente en `environment.ts`.
- **Nota:** Asegúrate que el import de `environment.ts` sea el correcto, y no el de producción (`environment.prod.ts`). Más adelante veremos cómo configurar el entorno para producción.

Y ya estaría nuestra aplicación configurada para usar Firebase de Google. A continuación, aprenderemos a hacer cada una de las operaciones comunes: **crear, leer, actualizar y borrar** registros de una base de datos, de la forma más simple posible.

Read (Leer registros)

Con todo el proyecto ya configurado, vamos a realizar la primera acción y la más básica, que es leer la información remota de la base de datos. Para no mezclar lo visto anteriormente y tener más limpio el código anterior y el nuevo, vamos a crear un nuevo servicio para Firebase de la misma forma que lo hicimos para Ajax.

1. Creamos un servicio nuevo con “`ng g s services/heroes-fs`”.
2. Le ponemos el siguiente contenido:

```
//heroes-fs.service.ts
import { Injectable } from '@angular/core';
import { AngularFirestore } from '@angular/fire/firestore';
import { Observable } from 'rxjs';

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

  constructor(private firestore: AngularFirestore) {}

  public getHeroes():Observable<any> {
    return this.firestore.collection("heroes").snapshotChanges();
  }
}
```

- ▶ Le añadimos al constructor un atributo de tipo `AngularFirestore`.
 - ▶ Con el método `getHeroes()`, hacemos una consulta a la base de datos firestore, creando un stream de cambios sincronizados sobre la colección “heroes”, manteniendo siempre el `Observable<any>` que devolvemos, actualizado ante cualquier cambio de la bbdd. Así, `CloudFireStore` envía una instantánea inicial (snapshot) de los datos y luego, otra instantánea cada vez que se modifica un documento.
3. En `Listado-crud.component.html`, el único cambio que tendremos que hacer es eliminar el pipe `| async` que teníamos en el `*ngFor`. Nos dará un error, pero lo arreglamos ahora en el siguiente paso. Puedes dejar por ahora la columna `id` para probar que funciona, y ya decides si dejarla o no.

4. Consumimos el nuevo servicio en el componente `Listado-crud.component.ts`. Explicamos poco a poco el proceso y después verás el código final más entendible:

```
//listado-crud.component.ts
imports...
@Component({...})
export class ListadoCrudComponent implements OnInit {
  public heroes: Heroe[];

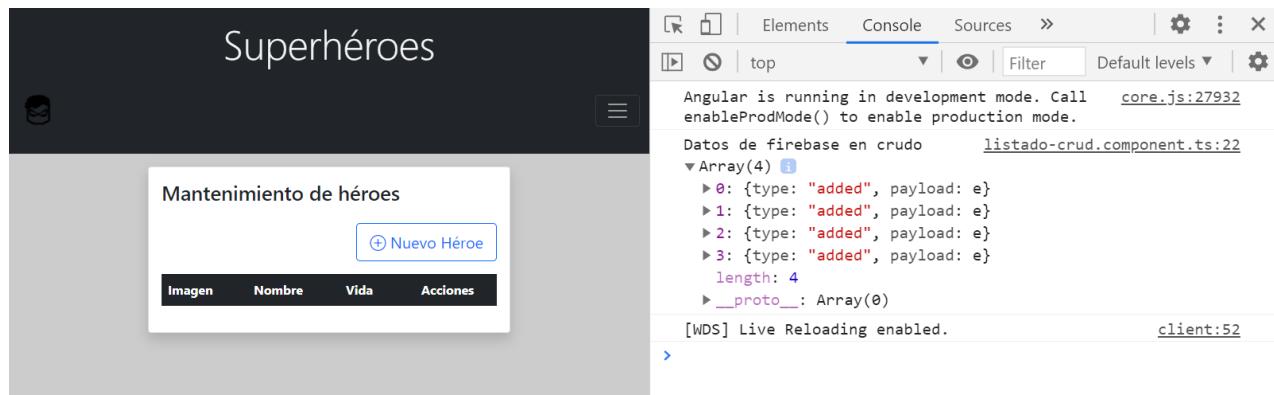
  constructor(private _heroesService: HeroesFSService) {
    this.heroes = [];
  }

  public getHeroes():void {
    this._heroesService.getHeroes().subscribe((datos)=>{}, (error)=>{});
  }

  ngOnInit(): void {
    this.getHeroes();
  }
}
```

- ▶ El atributo `heroes` será donde colocaremos los datos y el que recorrerá la tabla HTML con el `*ngFor`.
 - ▶ Inyectamos el servicio `HeroesFSService` anterior en el constructor. Lo llamamos `_heroesService` ya que es una buena práctica en Angular usar el guion bajo de prefijo en los identificadores de los servicios (y así también lo diferenciamos del anterior que usamos para Ajax).
 - ▶ Nos suscribimos al observable que nos devuelve `getHeroes()` del servicio, de la misma forma que hacíamos en Ajax.
 - ▶ En `ngOnInit()`, llamamos al método `getHeroes()` para que se inicie el proceso.
5. Centrándonos en el método `getHeroes()`, vamos a expandirlo y a llenar los métodos en caso de que se complete o no el observable.

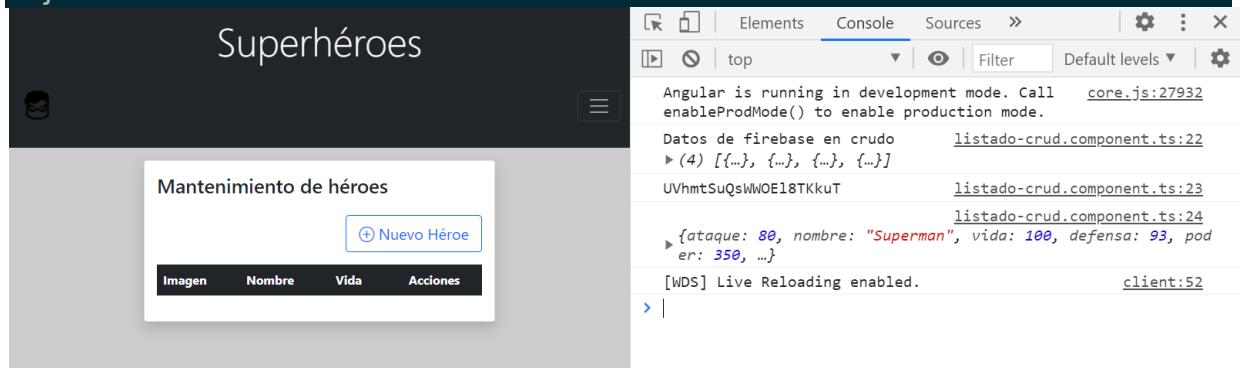
```
//listado-crud.component.ts
public getHeroes():void {
  this._heroesService.getHeroes().subscribe(
    (datos)=>{
      console.log("Datos de firebase en crudo", datos);
    },
    (error)=>{
      console.log("Error inesperado de Firebase:", error);
    }
  );
}
```



```
Angular is running in development mode. Call core.js:27932
enableProdMode() to enable production mode.
Datos de firebase en crudo listado-crud.component.ts:22
▼ Array(4) ⓘ
▶ 0: {type: "added", payload: e}
▶ 1: {type: "added", payload: e}
▶ 2: {type: "added", payload: e}
▶ 3: {type: "added", payload: e}
length: 4
__proto__: Array(0)
[WDS] Live Reloading enabled. client:52
>
```

- ▶ Los datos llegan (tenemos 4 documentos en la base de datos, y ahí salen 4 elementos en un array), pero no de la forma que esperamos. No es un objeto Heroe, si no algo más complejo. Para sacar la información que nos interesa haremos lo siguiente.

```
//listado-crud.component.ts
public getHeroes():void {
    this._heroesService.getHeroes().subscribe(
        (datos)=>{
            console.log("Datos de firebase en crudo", datos);
            console.log(datos[0].payload.doc.id);      //id
            console.log(datos[0].payload.doc.data());  //resto de datos
        },
        (error)=>{
            console.log("Error inesperado de Firebase:", error);
        }
    );
}
```



```
Angular is running in development mode. Call core.js:27932
enableProdMode() to enable production mode.
Datos de firebase en crudo listado-crud.component.ts:22
▶ (4) [{} , {} , {} , {}]
UVhmSuQsIWWOE18TKkuT listado-crud.component.ts:23
listado-crud.component.ts:24
▶ {ataque: 80, nombre: "Superman", vida: 100, defensa: 93, poder: 350, ...}
listado-crud.component.ts:24
[WDS] Live Reloading enabled. client:52
> |
```

- ▶ En `datos[0].payload.doc.id` tenemos la id del primer elemento del array, y en `datos[0].payload.doc.data()` tenemos el objeto completo en formato JSON.
- ▶ Por lo que sólo tenemos que recorrer el array `datos`, y construir los héroes de la forma que necesitemos, y añadirlos al atributo `heroes[]` que tenemos en la clase. Así se mostrarán directamente en la vista HTML.

6. Recorremos el array datos, construimos el héroe, y lo añadimos al atributo heroes.

```
//listado-crud.component.ts
public getHeroes():void {
    this._heroesService.getHeroes().subscribe(
        (datos)=>{
            console.log("Datos de firebase en crudo", datos);
            datos.forEach((elemento:any) => {
                let heroe: Heroe = {
                    "id": elemento.payload.doc.id,
                    ...elemento.payload.doc.data()
                };
                console.log("Héroe:", heroe); //Héroe construido
                this.heroes.push(heroe); //Se añade al array
            });
        },
        (error)=>{
            console.log("Error inesperado de Firebase:", error);
        }
    );
}
```

- ▶ Usamos un `forEach` para recorrer el array `datos`, y creamos un modelo de `Heroe`, pasándole el atributo `id`, y el resto atributos que hay en el objeto `datos`. Para ello usamos el operador `spread (...)`, que lo que hace es “concatenar” el resto de las propiedades al objeto (entre otras muchas posibilidades).
- ▶ Una vez construido el objeto `heroe`, lo añadimos al array de `heroes`, para que la vista los muestre en forma de tabla HTML.

The screenshot shows a web application for managing heroes. The main page title is "Superhéroes". Below it, there's a navigation bar with links: Home, Tabla, Tarjetas, CRUD, Desplegable, and a search bar. A modal window titled "Mantenimiento de héroes" is open, displaying a table of heroes with columns: Imagen, Nombre, Vida, and Acciones. The table contains four rows for Superman, Spiderman, Batman, and Capitán América. Each row has five action icons: magnifying glass, edit, green checkmark, and trash. Above the table is a blue button labeled "+ Nuevo Héroe". The developer console at the bottom right shows the following log output:

```
Angular is running in core.js:27932
development mode. Call enableProdMode() to
enable production mode.

Datos de listado-crud.component.ts:22
firebase en crudo
▶ (4) [{...}, {...}, {...}, {...}]

Héroe: listado-crud.component.ts:28
{id: "UVhmtSuQswNE18TKkUT", poder: 350,
▶ ataque: 80, nombre: "Superman", honor: 9
4, ...}

Héroe: listado-crud.component.ts:28
{id: "hfZnUXYph7qQL5fLWtkT", poder: 150,
▶ defensa: 89, imagen: "https://borilio.git
hub.io/curso-angular/img/avatars/spiderma
n.svg", vida: 100, ...}

Héroe: listado-crud.component.ts:28
{id: "pcgNhiczzbOYESLxKgU", nombre: "Bat
man", vida: 100, ataque: 60, defensa: 7
5, ...}

Héroe: listado-crud.component.ts:28
{id: "xly5xTFKFwB52t41NKHT", honor: 95, v
ida: 100, ataque: 64, poder: 135, ...}

[WDS] Live Reloading enabled. client:52
```

Y por fin estamos mostrando los datos que tenemos en la nube. Cualquier cambio que se produzca en la base de datos, será inmediatamente actualizado en la vista, sin necesidad de refrescar.

Vamos a probarlo, y de camino descubriremos un error y nos ayudará a entender el flujo de la información.

Abre dos ventanas en el navegador, una con el proyecto ejecutándose en el CRUD, y otra con la base de datos de Firebase.

Modifica algún valor que se vea en la tabla, como el nombre, la vida o la imagen de un héroe. Verás que el cambio es instantáneo sin necesidad de tener que recargar la página. Eso es porque tenemos un *stream* directo con la base de datos y cada vez que se realiza cualquier cambio, como estamos suscritos al observable, se lanza el *complete* del **subscribe** (el método que debe ejecutarse cuando se termina de recibir el observable).

Ese método será llamado cada vez que haya un cambio en la base de datos, por lo que, se ejecutará el **forEach** y se VOLVERÁN a añadir al array **heroes**, tantos elementos como documentos existan en la base de datos. Por lo que se van acumulando los héroes. Tiene una fácil solución, volver a vaciar el array cada vez que lo rellenamos.

```
(datos)=>{
    console.log("Datos de firebase en crudo", datos);
    this.heroes = []; //Empezamos de nuevo
    datos.forEach((elemento:any) => {
        let heroe: Heroe = {
            "id": elemento.payload.doc.id,
            ...elemento.payload.doc.data()
        };
        console.log("Héroe:", heroe); //Héroe construido
        this.heroes.push(heroe); //Se añade al array
    });
}
```

El código final lo tienes en el siguiente proyecto de stackblitz:



<https://stackblitz.com/edit/ng-heroes11-1-firebase-read>

Create (Guardar registros)

Formularios reactivos

Para crear nuevos registros (nuevos héroes), vamos a usar los formularios reactivos de Angular. Antes de insertar nuevo registro en Firebase, crearemos la estructura necesaria para pedir al usuario los datos, validarlos y manejarlos de una manera eficiente y simple.

Veamos los pasos a seguir:

1. En nuestro `app.module.ts`, añadimos el módulo de `ReactiveFormsModule`.

```
//app.module.ts
...
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  declarations: [...],
  imports: [
    ...,
    ReactiveFormsModule
  ],
  ...
})
export class AppModule { }
```

2. Vamos a trabajar el componente `nuevo-heroe.component`, que está sin tocar [desde que se creó aquí](#). En el archivo `nuevo-heroe.component.ts`, le añadiremos lo siguiente:

```
export class NuevoHeroeComponent implements OnInit {

  grupoCrearHeroe: FormGroup;
  formEnviado: boolean;

  constructor(
    private fb: FormBuilder
  ){
    // Por defecto, NO hemos enviado el formulario
    this.formEnviado = false;

    // Inicializamos el grupo de formulario "grupoCrearHeroe" con un objeto
    // que contiene los atributos, valor por defecto, y validadores
    this.grupoCrearHeroe = fb.group({
      "nombre" : [null, [Validators.required, Validators.maxLength(20)]],
      "imagen" : ["", ],
      "ataque" : [null, [Validators.required, Validators.min(0), Validators.max(100)]],
      "defensa" : [null, [Validators.required, Validators.min(0), Validators.max(100)]],
      "poder" : [null, [Validators.required, Validators.min(0), Validators.max(400)]],
      "honor" : [null, [Validators.required, Validators.min(-100), Validators.max(100)]],
      "vida" : [100, [Validators.required, Validators.min(0), Validators.max(100)]],
    });
  }

  ngOnInit(): void {
  }
}
```

- 2.1. Un atributo llamado `grupoCrearHeroe: FormGroup`. Este objeto representará al formulario contiendo los atributos, valores, validadores, etc.
- 2.2. Un atributo llamado `formEnviado: boolean`. Inicialmente valdrá false, e indicará si ya hemos pulsado el botón de enviar o no del formulario.
- 2.3. Al constructor le inyectaremos un objeto llamado `fb: FormBuilder`. Con este objeto construiremos el formulario completo de una forma fácil.

2.4. Inicializamos el objeto `grupoCrearHeroe`, con el método `.group()` del `FormBuilder`, pasándole por parámetros un objeto con la estructura siguiente:

```
{
  "atributo1": ["valor por defecto", [Validador1, Validador2...]],
  "atributo2": ...
}
```

- Si se le indica un valor por defecto, los elementos `input` del formulario aparecerán con ese valor, por lo que le pasamos `null` para que no tenga ninguno.
- Los validadores son opcionales, y pueden ser tantos como queramos. Son funciones de la clase `Validators`, y se pueden indicar expresiones regulares, si son requeridos, etc.

3. Enlazamos lo que acabamos de hacer al formulario HTML, por lo que nos vamos al `nuevo-heroe.component.html` y le añadimos la propiedad `[formGroup]="grupoCrearHeroe"`.

```
<!-- nuevo-heroe.component.html -->
...
<form class="d-grid gap-3" [formGroup]="grupoCrearHeroe">
```

4. A cada `input`, lo enlazamos con el atributo `formControlName`, como valor, deberemos poner el nombre que pusimos en el grupo del formulario. Lo repetimos por cada `input` que tenga el formulario.

```
<input
  id="floatingNombre"
  type="text"
  placeholder="Nombre"
  class="form-control"
  formControlName="nombre"
/>
```

El input para la vida, le eliminamos el `name` y `value` que tenía y lo dejamos así:

```
<input type="number" type="hidden" formControlName="vida" />
```

5. Hacemos que al pulsar el botón `submit` del formulario, se lance un método nuestro, así que al formulario le añadimos el evento `(ngSubmit)="agregarHeroe()"`. Nos dará un error porque no está creado todavía. En el siguiente punto lo creamos. También eliminamos el `routerLink` del botón submit (porque no queremos que nos lleve así).

```
<form ... [formGroup]="grupoCrearHeroe" (ngSubmit)="agregarHeroe()">
```

```
<button type="submit" class="btn btn-outline-primary">
  <i class="bi bi-plus-circle"></i> Guardar
</button>
```

6. Y ya nos queda crear el método `agregarHeroe()` en la clase, el cual será llamado cuando hagamos `submit` en el formulario.

```
//nuevo-heroey.component.ts
public agregarHeroe(){
  this.formEnviado = true; //Se ha intentado enviar el formulario
  console.log("Pulsamos enviar: ", this.grupoCrearHeroe);
}
```

Al pulsar en el botón enviar, si miramos por la consola nos deberá salir algo como lo siguiente:

```
Pulsamos enviar:
FormGroup {_hasOwnProperty: false, _parent: null, pristine: false, touched: true, _onCollectionChange: f, ...}
  ▶ controls: {nombre: FormControl, imagen: FormControl, ataque: FormControl, defensa: FormControl, poder: FormControl, honor: FormControl, vida: FormControl}
    ▶ pristine: false
    ▶ status: "VALID"
    ▶ statusChanges: EventEmitter_ {isScalar: true}
    ▶ touched: true
  ▶ value:
    ▶ nombre: "Super López"
    ▶ imagen: "http://nadiaoren.es/blog..."
    ▶ ataque: 34
    ▶ defensa: 28
    ▶ honor: 68
    ▶ poder: 86
    ▶ vida: 100
  ▶ _proto__: Object
  ▶ valueChanges: EventEmitter_ {isScalar: true}
  ▶ _composedAsyncValidatorFn: null
  ▶ _composedValidatorFn: null
```

Podemos ver que en la propiedad `FormGroup.value`, tenemos todos los valores del formulario, y si todos los campos pasan la validación, será válido, `this.grupoCrearHeroe.invalid === false`. Si algún campo no pasa la validación, `this.grupoCrearHeroe.invalid === true`.

7. Vamos a mostrar un mensaje en caso de que no se pase la validación. Será bastante fácil ahora. Colocamos un párrafo, span, div o lo que quieras, en `nuevo-heroey.component.html`. Puedes colocarlo donde prefieras. Un buen sitio sería justo encima del `<form>`.

```
<p *ngIf="formEnviado && grupoCrearHeroe.invalid" class="alert alert-danger">
  Revise los campos y asegúrese de que cumplen los requisitos
</p>
```

Podríamos indicar qué campo pasó la validación y cual no, ya que toda esa información está en `grupoCrearHeroe.controls`.

8. Ya sólo necesitamos extraer la información individual de cada campo en `grupoCrearHeroe`, dentro de `agregarHeroe()` para construir el objeto que queremos guardar. Lo hacemos de la siguiente forma:

```
//nuevo-heroe.component.ts
public agregarHeroe(){
  this.formEnviado = true; //Se ha intentado enviar el formulario
  console.log("Pulsamos enviar: ", this.grupoCrearHeroe);
  if (this.grupoCrearHeroe.invalid) {
    console.log("Formulario no válido");
  } else {
    console.log("Formulario válido");
    let heroe: Heroe = {
      "nombre": this.grupoCrearHeroe.value.nombre,
      "imagen": this.grupoCrearHeroe.value.imagen,
      "ataque": this.grupoCrearHeroe.value.ataque,
      "defensa": this.grupoCrearHeroe.value.defensa,
      "poder": this.grupoCrearHeroe.value.poder,
      "honor": this.grupoCrearHeroe.value.honor,
      "vida": this.grupoCrearHeroe.value.vida
    };
    console.log("Vamos a guardar:", heroe);
    //Nos dará un error, porque no le estamos indicando la id,
    //y como no vamos a indicarla, modificaremos la interfaz haciendo
    //el atributo id como opcional, poniéndole una ?
  }
}
```

9. Deberemos modificar la interfaz `Heroe`, ya que ahora mismo, nos obliga a declarar un atributo `id`. Abrimos `heroе.model.ts`, y le añadimos una interrogación a la `id` para indicarle que es un atributo opcional. Debe ser opcional, ya que cuando vayamos a crear un nuevo héroe, quién decide el valor de `id` es Firebase, y no le pasaremos dicho valor. La cambiamos de `number` a tipo `any`, para permitir que sea `null`.

```
//models/heroе.model.ts
export interface Heroe {
  id?: any;
  ...
}
```

10. Y ya estaría todo preparado. Solo si el formulario es válido, creamos el objeto con los valores obtenidos del formulario. Ya tenemos el objeto validado y listo para pasárselo a Firebase y que lo guarde. Lo veremos en el apartado.

Añadir el registro a Firebase

Una vez obtenido creado el objeto Héroe gracias a los formularios reactivos de Angular, ya podemos insertarlos de una manera bastante simple en nuestra base de datos de Firebase.

1. Agregamos un método nuevo a nuestro servicio `heroes-fs.service.ts`, el cual recibe el héroe que queremos que guarde en la base de datos.

```
//heroes-fs.service.ts
import ...

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

  constructor(
    private firestore: AngularFirestore
  ) {}

  public getHeroes(): Observable<any> {...}

  public agregarHeroe(nuevoHeroe: Heroe): Promise<any> {
    return this.firestore.collection<Heroe>("heroes").add(nuevoHeroe);
  }
}
```

Como vemos, es bastante simple. Usamos el método `.add()` de Firestore, pasándole el objeto que queremos guardar. El método devuelve una `Promesa`, que la tendrá que implementar quién consuma este servicio. Podemos parametrizar la colección, indicando que le vamos a añadir un objeto de la interfaz `Heroe` usando `collection<Heroe>()`.

Una promesa se usa en comunicaciones asíncronas, resumiendo es, una función que tiene en su interior otras dos funciones, una llamada `then()` y otra `catch()`. Se llamarán cuando se complete el trabajo asíncrono o si se produjo un error, respectivamente.

2. Como ya hicimos todo el trabajo de pedir los datos, lo único que tendríamos que hacer es añadir al método `agregarHeroe()` del componente `NuevoHeroeComponent`, una llamada al método del servicio. Presta atención y no te confundas, que vamos a llamar a dos métodos que se llaman iguales (`agregarHeroe()`), pero uno es del componente (el que lo pide), y otro es el del servicio (el que lo hace): Vemos el contenido completo del método, y en resaltado, la línea nueva.

```
//nuevo-heroey.component.ts
public agregarHeroe(){
  this.formEnviado = true; //Se ha intentado enviar el formulario
  console.log("Pulsamos enviar: ", this.grupoCrearHeroe);
  if (this.grupoCrearHeroe.invalid) {
    console.log("Formulario no válido");
  } else{
    console.log("Formulario válido");
    let heroe: Heroe = {
      "nombre": this.grupoCrearHeroe.value.nombre,
      "imagen": this.grupoCrearHeroe.value.imagen,
```

```

    "ataque": this.grupoCrearHeroe.value.ataque,
    "defensa": this.grupoCrearHeroe.value.defensa,
    "poder": this.grupoCrearHeroe.value.poder,
    "honor": this.grupoCrearHeroe.value.honor,
    "vida": this.grupoCrearHeroe.value.vida
  };
  console.log("Vamos a guardar:", heroe);

  //Hacemos la llamada al servicio, pasándole el héroe a guardar
  //Como es una promesa, en el then() hacemos la función a ejecutar,
  //si todo ha ido bien, o el catch() si hubo algún error.
  this._heroesService.agregarHeroe(heroe)
    .then(()=>{
      console.log("Héroe añadido con éxito:", heroe);
    })
    .catch((error)=>{
      console.log("Error al añadir Héroe:", error);
    })
  );
} //endif
} //end-agregarHeroes()

```

Como ves, es bastante fácil. Si es la primera vez que ves las promesas, puede que la sintaxis sea te parezca rara, pero si la construyes poco a poco se entiende mejor que terminada de golpe. Poco a poco sería:

```
this._heroesService.agregarHeroe(heroe).then().catch();
```

```
this._heroesService.agregarHeroe(heroe).then(()=>{}).catch(()=>{});
```

```

this._heroesService.agregarHeroe(heroe)
  .then(()=>{})
  .catch(()=>{})
;

```

```

this._heroesService.agregarHeroe(heroe)
  .then(()=>{
    })
  .catch(()=>{
    })
;

```

3. Ahora mismo ya debería de guardar el héroe correctamente en la base de datos. Probemos a añadir un héroe nuevo y comprobamos por la consola del navegador para ver posibles errores. Nos debería indicar que se guardó correctamente. Vamos a la página de Firestore para asegurar que es cierto y ahí debería aparecer.
4. Por último, lo que deberíamos hacer (además de mostrar algún aviso al usuario), es redireccionar de nuevo al listado. Para redireccionar desde código a una ruta, Angular nos provee de una clase llamada *Router*.

La inyectamos al constructor del servicio para tenerla disponible en los métodos de la misma manera que ya conocemos, y la usamos así:

```
//nuevo-heroe.component.ts
import ...
import { Router } from '@angular/router';

@Component({...})
export class NuevoHeroeComponent implements OnInit {
  ...
  constructor(... , private router: Router){...}

  public agregarHeroe(){
    ...
    this._heroesService.agregarHeroe(heroe).
      then(()=>{
        console.log("Héroe añadido con éxito:", heroe);
        //Y redirigimos a la ruta del listado
        this.router.navigate(['/listado-crud']);
      })
    ...
  }
}
```

La inyectamos en el constructor, se hace la importación automática, y ya podemos redirigir a la ruta que queramos desde código con el método `navigate()`, siempre que hayamos insertado correctamente el registro (por la parte del `then`).

La creación de registros ya la tenemos lista. Deberíamos poder insertar registros en nuestra base de datos de Firestore sin problemas y de una forma bastante rápida y eficiente.

Más adelante, incluiremos mensajes de notificación al usuario e iconos de espera, cada vez que realicemos correctamente o no, modificaciones a los datos. Esto mejorará notablemente la experiencia del usuario.



<https://stackblitz.com/edit/ng-heroes11-2-firebase-create>

Delete (Borrar registros)

Para borrar un héroe, empezaremos como siempre, por un método en el servicio.

1. Crearemos un método llamado `eliminarHeroe()` en el `heroes-fs.service.ts`:

```
//heroes-fs.service.ts
public eliminarHeroe(id: string): Promise<any> {
  return this.firestore.collection<Heroe>("heroes").doc(id).delete();
}
```

El método, recibe la `id` del documento que queremos eliminar, y se la pasamos al método `doc()`, para obtener el documento, al cual le invocamos el método `.delete()` y lo borra.

2. Lo siguiente, será hacer uso del anterior método del servicio, en un método llamado igual, dentro de `Listado-crud.component.ts`.

```
//listado-crud.component.ts
public eliminarHeroe(id: string): Promise<any> {
  console.warn("Queremos borrar el heroe con id=" + id);
  return this._heroesService.eliminarHeroe(id)
    .then(()=>{
      console.log("Héroe eliminado con éxito");
    })
    .catch((error)=>{
      console.log("Error al eliminar el héroe con id:" + id);
      console.error("error = ", error);
    });
}
```

Como el método del servicio nos devuelve un Promise, debemos implementar el `.then()`, que será llamado cuando se complete la promesa, y el `.catch()` que será llamado cuando haya algún error.

Si llamamos al método `eliminarHeroe()` pasándole una id de un héroe, ya funcionaría (eliminaría el héroe que tenga dicha id). Eso es lo que debemos de conseguir en el siguiente paso.

3. En la tabla HTML, `Listado-crud.component.html`, debemos modificar el botón “  Borrar” para que, al hacer clic sobre él, hagamos una llamada al método anterior, pasándole la `id` del héroe.

```
//listado-crud.component.ts
...
<button
  (click)="this.eliminarHeroe(heroe.id)"
  [title]="'Borrar a ' + heroe.nombre"
  class="mx-2 btn btn-outline-danger rounded-pill btn-sm">
  <i class="bi bi-trash"></i>
</button>
```

Puedes comprobar que, al pulsar el botón de borrar, elimina el Héroe de la tabla, y si nos vamos a Firebase, también queda borrado de la base de datos.

Que ganas de ponerle mensajes bonitos al usuario en lugar de mostrarlos por la consola 😊.



<https://stackblitz.com/edit/ng-heroes11-3-firebase-delete>

Update (Actualizar registros)

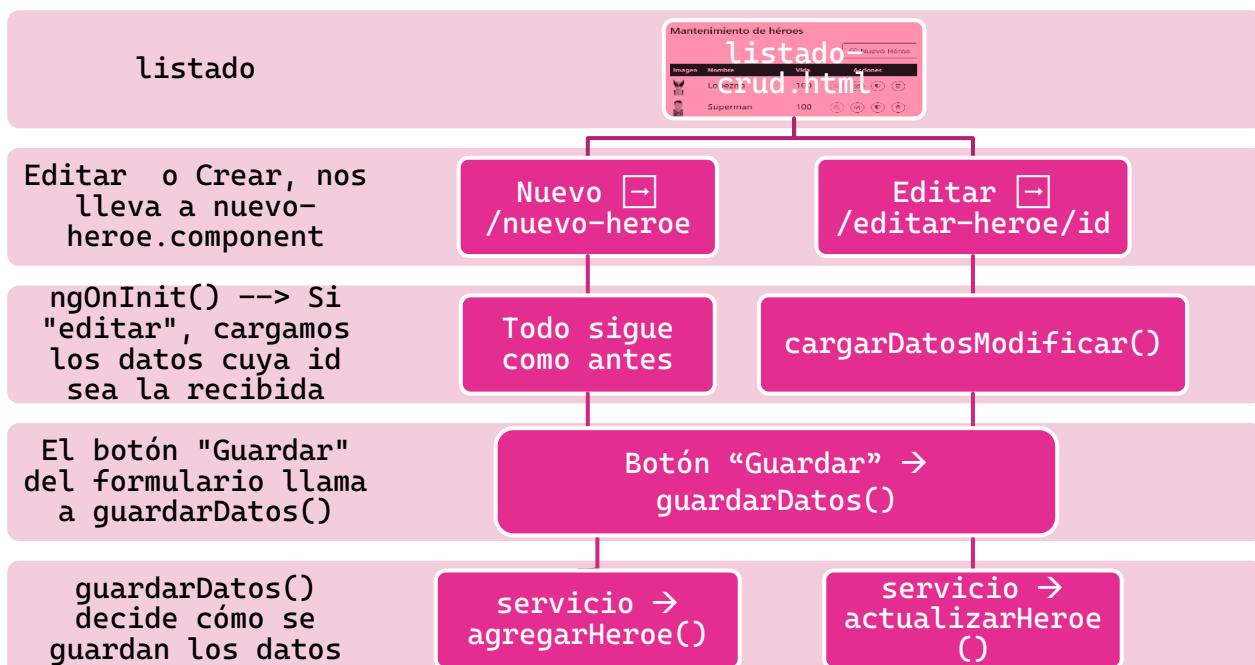
Para actualizar un registro, el proceso es bastante parecido que al de crear, sólo que a Firebase le pasaremos la id del documento que queremos actualizar y el resto de los datos.

Podríamos hacer un componente nuevo y sería bastante sencillo, pero ya que tenemos un formulario realizado, lo más eficiente es reutilizar un componente para que pueda ser capaz de realizar ambas tareas. Por lo que vamos a reestructurar el componente `nuevo-heroe.component`.

Veamos un esquema de cómo se comportaba antes la aplicación:



Y ahora, para poder reutilizar el componente `nuevo-heroe.component`, para la tarea de crear o editar, haremos lo siguiente:



Empecemos con la modificación del código paso a paso:

1. Añadiremos una nueva ruta a nuestro *app-routing.ts*, para ir a ella cuando pulsemos el botón "Editar".

```
const routes: Routes = [
  {...},
  {
    path: "editar-heroe/:id",
    component: NuevoHeroeComponent
  }
]
```

Estamos creando otra ruta, que nos va a dirigir al mismo componente (formulario de héroe), y en la ruta irá una variable (que será la id del héroe a modificar).

2. En el *listado-crud.component.html*, modificaremos el botón de **editar**, para que nos lleve a la ruta anterior:

```
<button
  [routerLink]="['/editar-heroe', heroе.id]"
  [title]="'Editar a ' + heroе.nombre"
  class="mx-2 btn btn-outline-primary rounded-pill btn-sm">
  <i class="bi bi-pencil-square"></i>
</button>
```

El botón nos va a llevar a la ruta /editar-heroe y le pasará por parámetros la **id** del héroe que iba por el forEach. Un ejemplo sería: /editar-heroe/pcgNh1czzbz0YE5LxKgU.

Prueba que todo funcione por ahora. Nos llevará al formulario y se verá por la url la **id** del héroe.

3. El siguiente paso será capturar esa **id**, desde el constructor de *nuevo-heroे.component.ts*, para saber si queremos editar o crear. Necesitaremos una nueva clase llamada *ActivatedRoute* para capturar la variable de la ruta. La inyectaremos en el constructor y la importaremos.

```
// nuevo-heroе.component.ts
private idEditar: string | null;
public titulo: string;
private origen: number;

private static ORIGEN_EDITAR = 1;
private static ORIGEN_CREAR = 2;

constructor( ..., private aRoute: ActivatedRoute){
  this.formEnviado = false;
  this.grupoCrearHeroе = fb.group({...});
  //Extraemos la id de la ruta, si existe...
  this.idEditar = this.aRoute.snapshot.paramMap.get("id");
  console.log("La id que hay en la url es:" + this.idEditar);
  //Inicializamos los atributos, dependiendo de que hagamos
  if (this.idEditar === null) {
    this.titulo = "Nuevo Héroe";
    this.origen = NuevoHeroeComponent.ORIGEN_CREAR;
  } else{
    this.titulo = "Editar Héroe";
    this.origen = NuevoHeroeComponent.ORIGEN_EDITAR;
  }
}
```

Hemos creado 3 atributos nuevos:

- **idEditar** → Aquí colocaremos el parámetro recibido por la url, que indicará la **id** del héroe que queremos modificar, o **null** si es uno nuevo. Lo extraemos con un método de **ActivatedRoute**, pasándole la variable que habíamos indicado en el **app-routing.module.ts**. La declaramos como **string | null**, para que pueda admitir un valor **null**, de lo contrario no lo admitiría. Y el **paramMap** puede devolver un **null** si no existe.
- **título** → Aquí tendremos el título de la tarjeta, que será “Editar Héroe” o “Nuevo Héroe”. Lo cargaremos en la vista mediante interpolación `{} {{ título }}`.
- **origen** → Aquí pondremos un valor que indicará si queremos editar o crear. Podríamos darle **string** directamente, pero mejor usamos los atributos estáticos **ORIGEN_EDITAR** y **ORIGEN_CREAR** para seleccionar una de las dos opciones. Nos da igual el valor que puedan tener, y así la aplicación tendrá una escalabilidad mayor.

Si la variable **idEditar**, vale **null**, es porque venimos a crear uno nuevo, en caso contrario, a editar.

4. Antes de nada, vamos a hacer dos métodos nuevos en nuestro servicio **heroes-fs.service**:

- 4.1. **getHeroe()** → Para obtener UN héroe, en base a una **id** que se le pida. Este lo usaremos ya.
- 4.2. **actualizarHeroe(id:string, heroe:Heroe)** → Recibirá una **id**, y un **héroe**, y guardará ese héroe en un documento que ya exista con esa id. Este lo usaremos al final.

```
//heroes-fs.service.ts
public getHeroe(id: string): Observable<any> {
  return this.firestore.collection<Heroe>("heroes").doc(id).snapshotChanges();
}

public actualizarHeroe(id:string, datos: Heroe): Promise<any> {
  return this.firestore.collection<Heroe>("heroes").doc(id).update(datos);
}
```

5. En el **ngOnInit()**, si venimos a editar, llamamos al método **this.cargarDatosModificar()**, para que se haga una consulta a Firebase con la id que queremos editar.

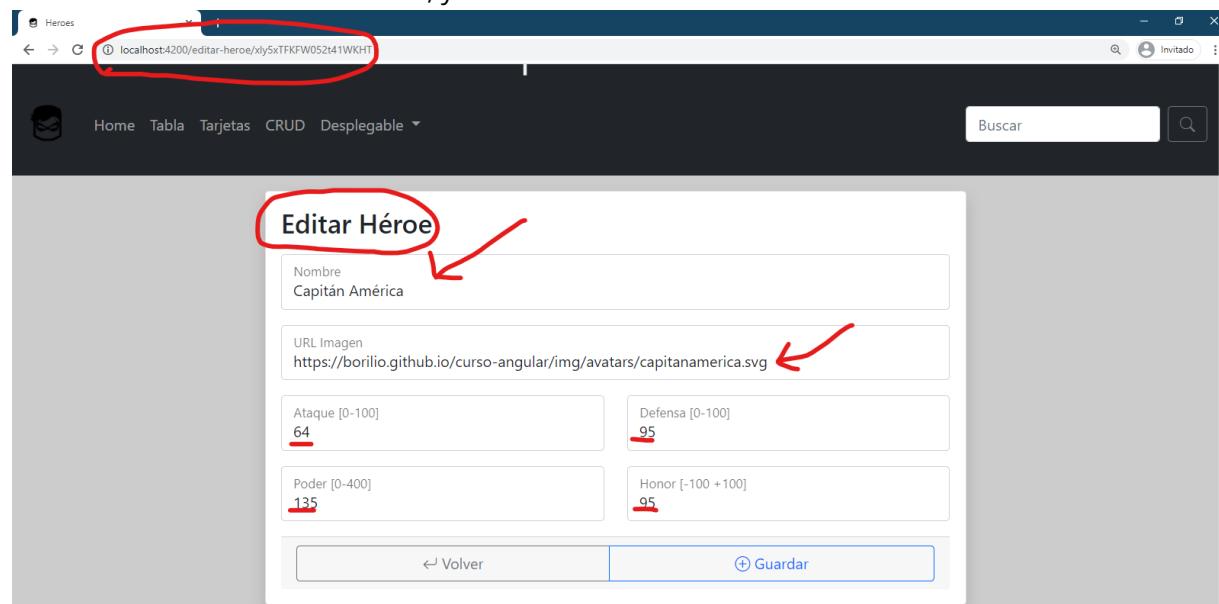
```
//nuevo-hero.component.ts
ngOnInit(): void {
  if (this.origen === NuevoHeroeComponent.ORIGEN_EDITAR) {
    if (this.idEditar!=null) {
      this.cargarDatosModificar(this.idEditar);
    }
  }
}
```

Se podría hacer todo en el constructor, pero no es una buena práctica realizar tareas pesadas en ellos, como llamadas a servicios, por eso lo hacemos en el **ngOnInit()**.

```
//nuevo-heroe.component.ts
/**
 * Método que se encarga de recargar los valores del formulario, con lo que
 * extraigamos de la base de datos, según la id recibida
 */
private cargarDatosModificar(id: string):void {
  this._heroesService.getHeroe(id).subscribe(
    (datos)=> {
      let heroe: Heroe = {
        "nombre" : datos.payload.data()["nombre"],
        "imagen" : datos.payload.data()["imagen"],
        "ataque" : datos.payload.data()["ataque"],
        "defensa" : datos.payload.data()["defensa"],
        "poder" : datos.payload.data()["poder"],
        "honor" : datos.payload.data()["honor"],
        "vida" : datos.payload.data()["vida"],
      };
      this.grupoCrearHeroe.setValue(heroe);
    }
  );
}
```

Hacemos una llamada al servicio que recibe una id y nos suscribimos para extraer los datos de ese héroe. Creamos un objeto `heroe`, y con `setValue()` del `FormGroup`, volcamos esos valores en los input del formulario.

6. Ahora mismo si le damos a editar, ya deberíamos tener los valores de ese héroe en el formulario:



Lo que nos queda, asociar un mismo método que se ejecute al pulsar “Guardar”, y que ya él decida como debe guardar los datos. Ya que si queremos “Nuevo” o “Editar” son distintos métodos del servicio. En el `nuevo-heroe.component.html`, cuando haga `submit` ejecutaremos `guardarDatos()`.

```
<!-- nuevo-heroe.component.html -->
...
<form ... (ngSubmit)="guardarDatos()">
```

```
//nuevo-heroe.component.ts
public guardarDatos():void {
    this.formEnviado = true;
    if (!this.grupoCrearHeroe.invalid) {
        console.log("Formulario válido. Guardamos...");
        //Creamos el objeto a guardar
        let heroe: Heroe = {
            "nombre": this.grupoCrearHeroe.value.nombre,
            "imagen": this.grupoCrearHeroe.value.imagen,
            "ataque": this.grupoCrearHeroe.value.ataque,
            "defensa": this.grupoCrearHeroe.value.defensa,
            "poder": this.grupoCrearHeroe.value.poder,
            "honor": this.grupoCrearHeroe.value.honor,
            "vida": this.grupoCrearHeroe.value.vida
        };
        //Guardamos la id definitiva para actualizar
        //Es un apañío para no tener que hacer un if dentro del siguiente if
        let idGuardar: string = (this.idEditar !== null) ? this.idEditar: "";
        //Y ahora decidimos cual servicio usar actualizar/agregar
        let promesa: Promise<any>;
        if (this.origen === NuevoHeroeComponent.ORIGEN_CREAR) {
            promesa = this._heroesService.agregarHeroe(heroe);
        } else{
            promesa = this._heroesService.actualizarHeroe(idGuardar, heroe);
        }
        //Ahora ya tenemos la promesa, ya sea de editar/crear, la resolvemos
        promesa
            .then( ()=>{
                console.log("Héroe creado/editado con éxito");
                //Y nos vamos al listado
                this.router.navigate(["/listado-crud"]);
            }).catch((error)=>{
                console.error("Error al crear/guardar", error);
            })
        );
    } else{
        console.error("Formulario no válido. No enviamos");
    }
}
```

Creamos el objeto `heroe`, será común para ambos métodos. Y llamamos al método del servicio que corresponda, `agregarHeroe()` o `actualizarHeroe()`. Como ambos nos devuelven una promesa, y en ambos casos hacemos lo mismo (mensaje y redirigir a /listado-crud), pues hacemos el mismo `.then().catch()` para UNA promesa, la que hayamos inicializado en la variable `promesa`.

Ya debería de funcionar nuestro CRUD completo. Es obvio que los métodos y mucha parte del código se pueden hacer de muchas formas distintas, aquí hemos optado por hacer lo más simple posible, dentro de las buenas prácticas, aunque se podrían mejorar muchas cosas.



<https://stackblitz.com/edit/ng-heroes11-4-firebase-update>

Notificaciones con SweetAlert2

Existen muchas librerías de terceros que nos facilitan la tarea de mostrar mensajes emergentes al usuario, como [Sweet Alert](#) y [ToastR](#). Puedes visitar los links para verlas en acción.

Para agregar sweetAlert2 por medio de npm a un proyecto Angular, sigue los pasos:

1. Parar el servidor de desarrollo.
2. Irnos a la consola de comandos, en la raíz del proyecto y escribir lo siguiente:

```
npm install --save sweetalert2 @sweetalert2/ngx-sweetalert2

added 2 packages, and removed 1 package in 7s

102 packages are looking for funding
  run `npm fund` for details
>
```

3. En el `app.module.ts`, añadimos el módulo:

```
import { SweetAlert2Module } from '@sweetalert2/ngx-sweetalert2';

@NgModule({
  declarations: [...],
  imports: [
    SweetAlert2Module.forRoot()
  ],
})
```

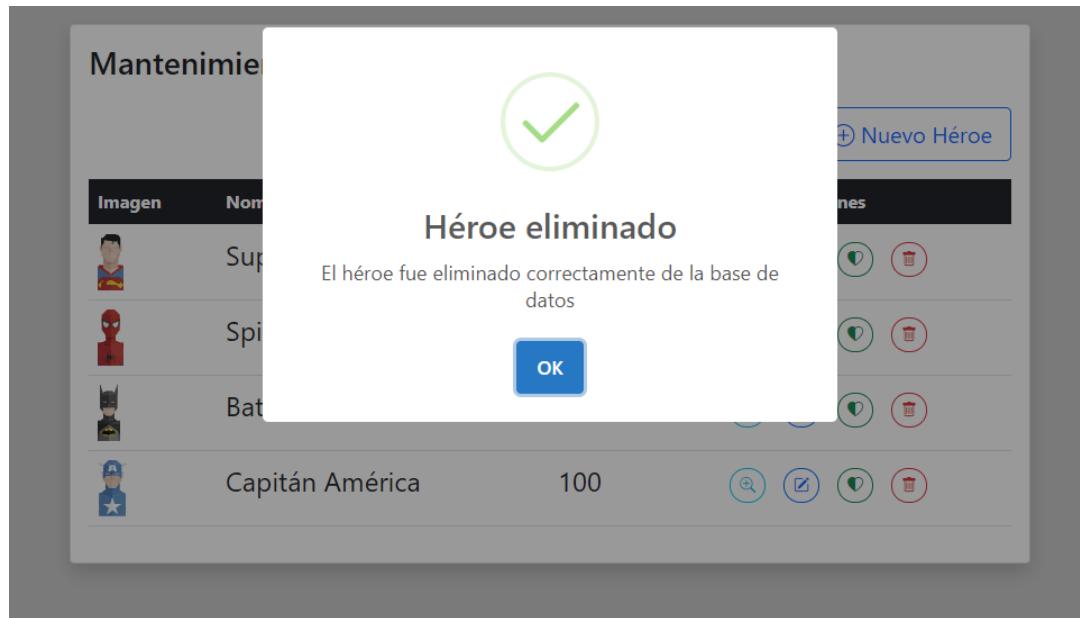
4. En el `archivo.ts` del componente que queramos mostrar la alerta, deberemos hacer el import necesario:

```
import Swal from 'sweetalert2';
```

5. Y ya podemos usar las alertas. En lugar de un `console.log()`, podemos mostrar lo siguiente:

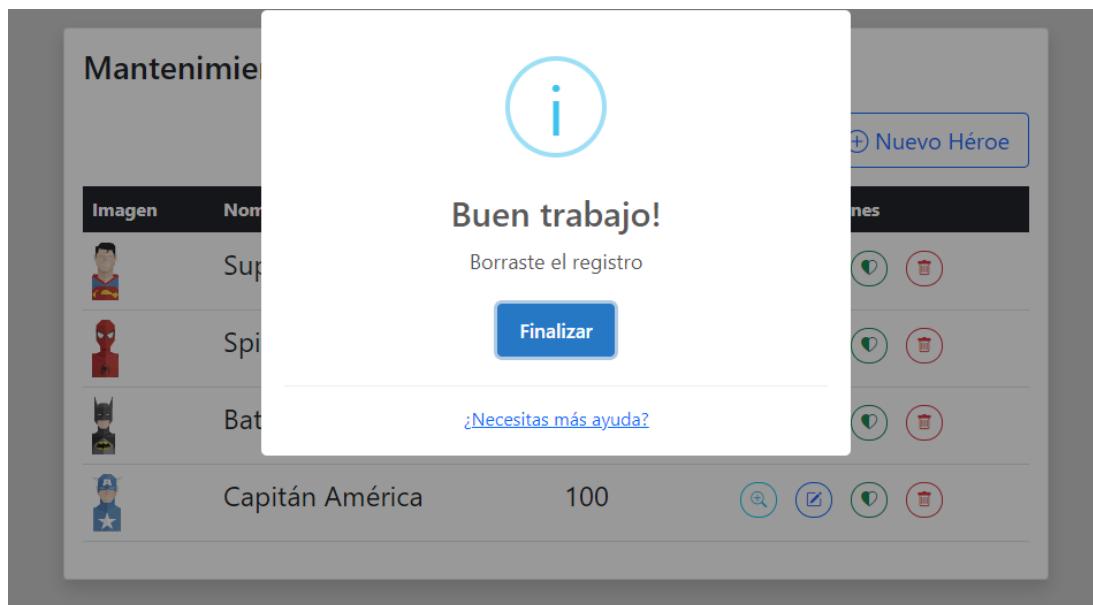
```
console.log("Héroe eliminado correctamente");
Swal.fire(
  'Héroe eliminado', //Encabezado (opcional)
  'El héroe fue eliminado correctamente de la base de datos', //Mensaje
  'success' //Estilo (success, warning, error, info) (opcional)
);
```

De esta forma, el usuario verá algo parecido a lo que se muestra en la siguiente captura:



Hay muchas opciones para personalizarlos. Puedes obtener la documentación en <https://sweetalert2.github.io/#examples>. Probemos otro:

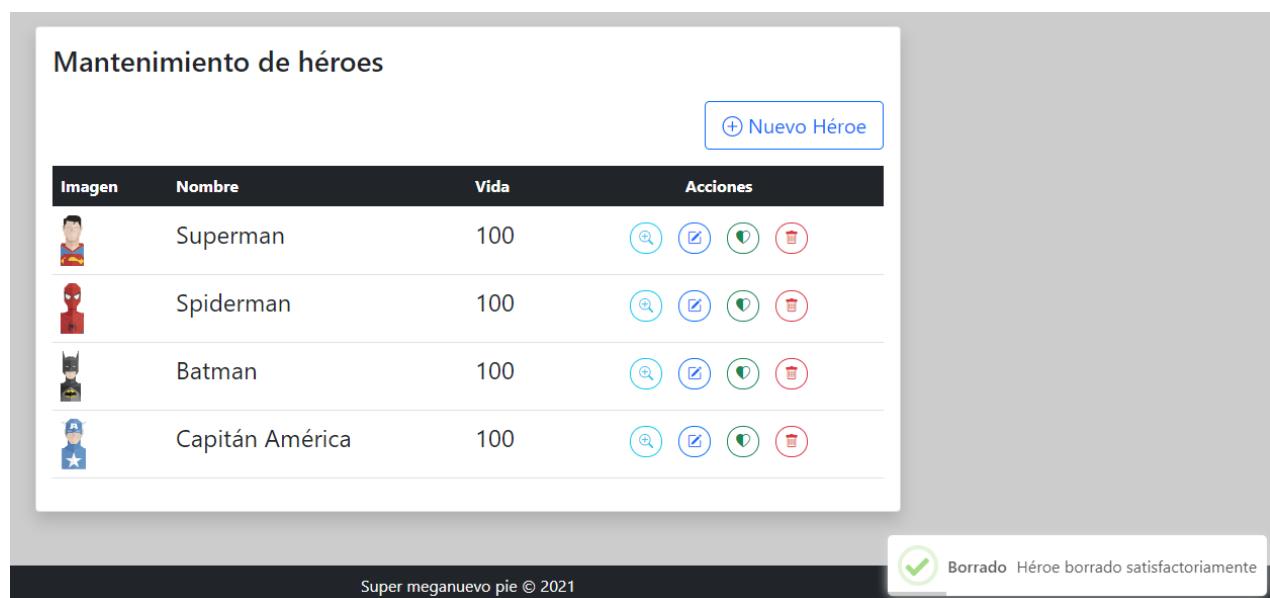
```
Swal.fire({
  title: "Buen trabajo!",
  text: "Borraste el registro",
  icon: "info",
  confirmButtonText: "Finalizar",
  footer: '<a href="#">¿Necesitas más ayuda?</a>'
});
```



¿Muy grande? ¿No quieres que el usuario tenga que darle a ok? Veamos los Toast.

Para las notificaciones **toast** (mensaje tipo notificación), tan sólo hay que añadir unos atributos a la configuración:

```
Swal.fire({
  toast: true,
  position: 'bottom-end', //Pueden ser 'top', 'top-start', 'top-end', 'center', 'center-start', 'center-end', 'bottom', 'bottom-start', or 'bottom-end'.
  title: "Borrado",
  text: "Héroe borrado satisfactoriamente",
  icon: "success",
  showConfirmButton: false, //Quitamos el botón, para sólo mostrar el mensaje
  timer: 4000,
  timerProgressBar: true //Muestra una barra de progreso del temporizador
});
```



The screenshot shows a table titled "Mantenimiento de héroes" (Hero Maintenance) with four rows of superhero data:

Imagen	Nombre	Vida	Acciones
	Superman	100	
	Spiderman	100	
	Batman	100	
	Capitán América	100	

A blue button labeled "+ Nuevo Héroe" is at the top right. At the bottom, there is a footer with the text "Super meganuevo pie © 2021" and a green toast notification on the right that says "Borrado Héroe borrado satisfactoriamente" with a checkmark icon.

También podemos hacer mensajes de confirmación que requieran una respuesta del usuario. Queremos que, al pulsar el botón de borrar, me pregunte si estoy seguro de borrar el héroe, y que sólo lo borre si le digo que sí. Haremos lo siguiente:

Primero, el botón HTML de borrar (*listado-crud.component.ts*) lo tenemos así:

```
<button ... (click)="this.eliminarHeroe(heroe.id)"></button>
```

Pues lo cambiaremos por...

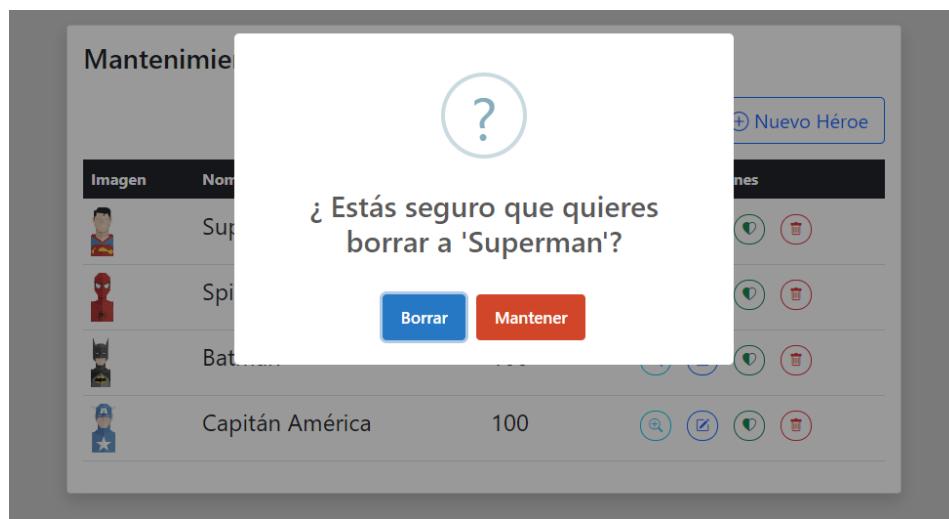
```
<button ... (click)="this.confirmacionBorrarHeroe(heroe)"></button>
```

Simplemente iremos a otra función primero que muestre el alert, **y si pulsamos que sí**, entonces ya **haremos la llamada definitiva al servicio** para que borre el elemento.

En lugar de enviar sólo la id a `confirmaciónBorrarHeroe()` le enviamos el héroe completo, para poder tener disponible toda la información que queramos mostrar.

```
public confirmaciónBorrarHeroe(heroe: Heroe): void {
  Swal.fire({
    title: "¿ Estás seguro que quieres borrar a '" + heroe.nombre + "'?",
    showDenyButton: true,
    showCancelButton: false,
    confirmButtonText: 'Borrar',
    denyButtonText: 'Mantener',
    icon: "question"
  }).then((result) => {
    if (result.isConfirmed) {
      this.eliminarHeroe(heroe.id);
    } else if (result.isDenied) {
      Swal.fire('No se borró a ' + heroe.nombre, '', 'info');
    }
  });
}

public eliminarHeroe(id: string): Promise<any> {
  console.warn("Queremos borrar el heroe con id=" + id);
  return this._heroesService.eliminarHeroe(id)
    .then(()=>{
      Swal.fire('Héroe borrado satisfactoriamente', '', 'success');
    })
    .catch((error)=>{
      Swal.fire('Se produjo un error en firebase', '', 'error');
    });
}
```



<https://stackblitz.com/edit/ng-heroes11-sweetalert>

Testing (Pruebas unitarias con Karma)

Básicamente, el *Testing (Pruebas)*, en cualquier entorno de programación es **asegurarse que la aplicación funciona tal y como se le espera.**

Las **pruebas unitarias** son una forma de comprobar el correcto funcionamiento de una unidad de código. Las unidades de código pueden ser una función, pero también puede ser un componente. Tienen como finalidad asegurarse que el código hace lo que tiene que hacer, recibiendo lo que hemos diseñado que reciba (tanto en número, los nombres y el tipo de los argumentos) como lo que devuelva.

En lugar de hacer pruebas manuales (console.log, alerts, etc.), se realizan una serie de pruebas automatizadas, de forma que se pueden volver a pasar en cualquier momento, especialmente después de actualizaciones o grandes cambios en el código, con el fin de que todo siga funcionando tal y como se esperaba.

AngularCLI ya instala por defecto todo lo necesario para probar una aplicación con el [framework Jasmine test](#). También se pueden usar pruebas unitarias con otras bibliotecas de testing, cada una tendrá su instalación, configuración y sintaxis. **Las pruebas se hacen con Jasmine y se ejecutan con Karma.**

- **Jasmine** es un framework especializado en pruebas unitarias y contiene una serie de herramientas dedicadas.
- **Karma** es un administrador de tareas en el navegador y lo usaremos para ejecutar las pruebas en un entorno realista.

Hasta la versión 11 venía incluido también con [Protractor](#) para las pruebas end-to-end (e2e), pero fue eliminado por defecto en los proyectos creados con AngularCLI a partir de la 12. Siempre se puede añadir esta librería de forma manual según se indica en su [documentación oficial](#). El objetivo de las pruebas e2e (extremo a extremo) es probar todo el software en busca de dependencias, integridad de datos y comunicación con otros sistemas, interfaces y bases de datos para ejercer una producción completa como un escenario.

Es posible que para simplificar nuestros proyectos (por ejemplo, en héroes), hayamos borrado todo lo necesario para el testing. Para hacerlo lo más simple posible, haremos un proyecto nuevo con [ng new demo-testing](#) y haremos las pruebas sobre él.

Antes de hacer nuestras propias pruebas unitarias, vamos a familiarizarnos con el entorno de testing, ejecutar las pruebas de ejemplo e interpretar sus resultados.

Ejecutar las pruebas

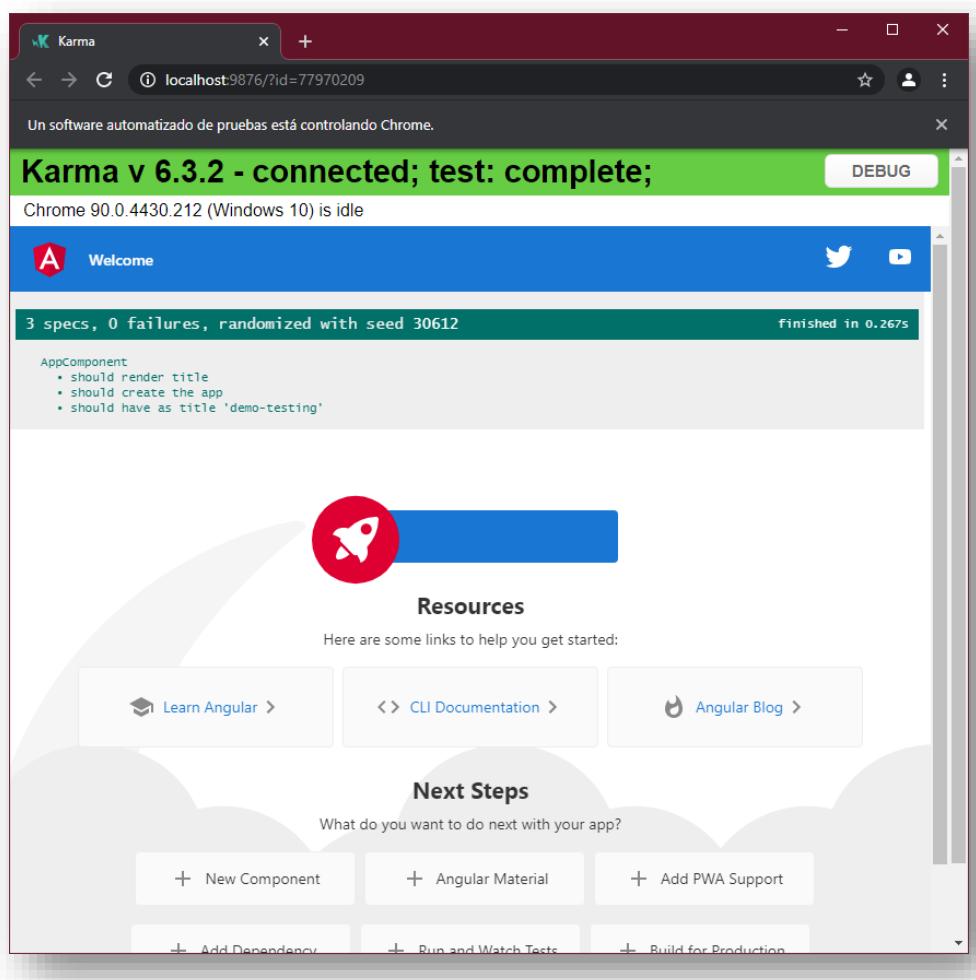
Para ejecutar las pruebas unitarias que tengamos realizadas en nuestra aplicación, deberemos ejecutar el siguiente comando en la carpeta del proyecto:

```
ng test

10% building modules 1/1 modules 0 active
...INFO [karma]: Karma v6.7.1 server started at http://localhost:9876/
...INFO [launcher]: Launching browser Chrome ...
...INFO [launcher]: Starting browser Chrome
...INFO [Chrome ...]: Connected on socket ...
Chrome ...: Executed 3 of 3 SUCCESS (0.135 secs / 0.205 secs)
```

Dicho comando construye la aplicación y ejecuta las pruebas Karma. La última línea de la consola es la más importante. Ahí nos mostrará que Karma ejecutó 3 test, y todos los pasó con éxito.

También se abrirá una ventana con el navegador y mostrará el resultado de la prueba con el “**Jasmine HTML Reporter**” de la siguiente manera:



Normalmente es más fácil de leer e interpretar el resultado del navegador que la salida por consola. Puedes hacer clic en la fila de cada prueba para volver a ejecutar esa prueba en concreto o bien en el grupo para volver a ejecutar todos los test de ese grupo (`AppComponent` en el ejemplo).

Mientras tanto, el comando `ng test` sigue ejecutándose, buscando cambios, por lo que puedes modificar el código para ver que las pruebas pasan con éxito o no.

En el ejemplo de aplicación base, ya tenemos 3 pruebas unitarias de ejemplo. Sirven para asegurarse de que el componente `AppComponent` hace lo siguiente:

- `AppComponent`:
 - Debería mostrar un título.
 - Debería crear la aplicación.
 - Debería tener como título “`demo-testing`”.

Como el componente cumple los requisitos (es decir, hace correctamente las tareas que debería hacer), nos indica que pasó los `3 test con éxito` y hubo `0 fallos`.

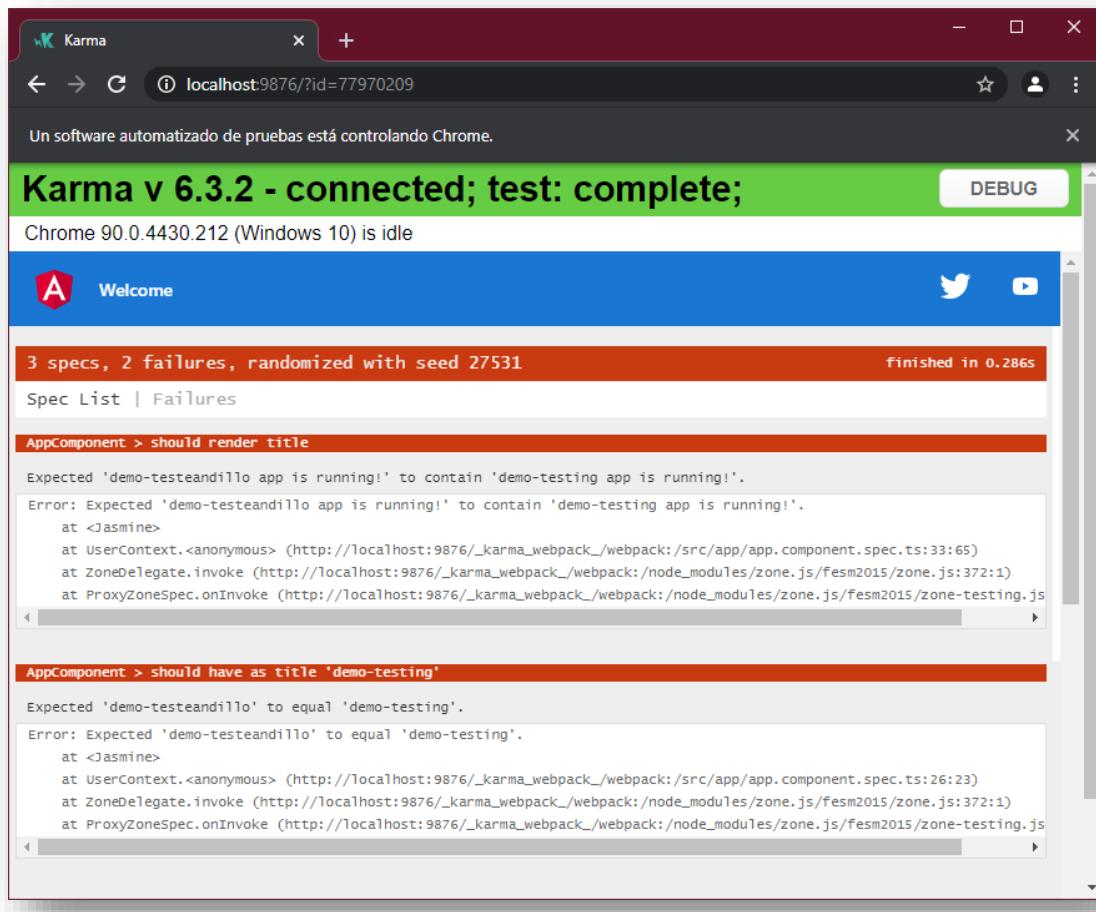
Lo siguiente que haremos será provocar un error (sin querer, por supuesto 😠) en alguno de esos tests, para ver de qué forma nos informaría. El más simple es cambiarle el título de “`demo-testing`” (recuerda que al componente `AppComponent`, siempre tiene un atributo llamado `title` y su valor es el nombre de la aplicación, que es `demo-testing`).

Abrimos el proyecto con VSC y nos iremos a `app.component.ts` para hacer el siguiente cambio:

```
//app.component.ts
import { Component } from '@angular/core';

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

Al guardar los cambios, inmediatamente en la ventana de Karma veremos lo siguiente:



Se han ejecutado las 3 pruebas de nuevo, y dos de ellas han fallado.

- **Should have as title “demo-testing”:** Esta prueba se asegura que el atributo `title` tenga como valor “`demo-testing`”, y se lo hemos cambiado por “`demo-testeandillo`”, por lo que no ha pasado la prueba y así nos lo informa. Esperaba tener un valor, y tenía otro. Este error es obvio y fácilmente detectable de forma manual.
- **Should render title:** Esta prueba esperaba que el elemento `span` del título, tuviese como contenido html el texto “`demo-testing app is running!`”. Y al cambiar el valor del atributo anterior, hemos tenido este error colateral que posiblemente ni nos habríamos dado cuenta de no tener una prueba automatizada que lo comprobara por nosotros.

Sería una buena práctica, tener nuestras pruebas unitarias preparadas, de forma que cada vez que vayamos a hacer un commit importante, o tras añadir una funcionalidad extra, **pasar las pruebas unitarias** que queramos **para asegurarnos que la aplicación sigue funcionando correctamente** y que una modificación no haya afectado colateralmente a otra funcionalidad y no nos hayamos dado cuenta debido a que ni la hemos comprobado, pensando que eso ya estaba más que probado.

💡 Gracias al testing, nos hemos dado cuenta de que, modificando un valor, afectaba a dos pruebas. ¿Sabrías actualizar el código para que una modificación en el atributo `title`, no afectara de forma colateral a la segunda prueba?

Archivos de tests

Los archivos que usa Angular para el testing tienen la extensión `.spec.ts`, de esta forma Karma puede identificar qué archivos pertenecen al testing.

Cada componente debería tener su archivo de testing con el mismo nombre, pero con la extensión `.spec.ts` y deberían estar ambos archivos en la misma carpeta, por las siguientes razones:

- Cada prueba se encontrará más fácil.
- Verás de un vistazo si una parte de la aplicación carece de pruebas.
- Cuando muevas o renombres una parte (componente, servicio), no te olvidarás de actualizar su archivo de prueba asociado.

La realidad es que los archivos de prueba (`*.spec.ts`) no tienen la obligatoriedad de estar junto a ningún otro archivo, y pueden estar en cualquier ubicación dentro del proyecto, Karma los buscará y los ejecutará. Por lo que otra opción es poner todas las pruebas en una carpeta llamada `tests`, en lugar de lo explicado anteriormente, aunque no es lo recomendable.

Creando una calculadora

Ya sabemos ejecutar e interpretar las pruebas unitarias. Nos queda crear nuestras propias pruebas unitarias.

Para verlo de la forma más clara y limpia posible, vamos a crear un nuevo componente llamado `suma`. Borraremos todo el contenido de `app.component.html` y ponemos el componente `app-suma` en el interior de `app-component.html`.

```
<!-- app.component.html -->
<app-suma></app-suma>
```

En `suma.component.*`, lo que vamos a hacer es una simple calculadora. 2 cuadros de texto, un botón para realizar la suma, y un cuadro de texto para poner el resultado.

Para poder acceder a los elementos HTML desde código, en lugar de usar `ngModel` (☞) vamos a ver otra forma. Usando el decorador `ViewChild`.

```
//suma.component.ts
...
export class SumaComponent implements OnInit {
  @ViewChild("num1") public inputNum1?: ElementRef<HTMLInputElement>;
  @ViewChild("num2") public inputNum2?: ElementRef<HTMLInputElement>;
  @ViewChild("btnSumar") public btnSumar?: ElementRef<HTMLButtonElement>;
  @ViewChild("resultado") public inputResultado?: ElementRef<HTMLInputElement>;

  constructor() {}
  ngOnInit(): void {}

  public sumar(n1:any, n2:any):number {
    let resultado = Number(n1) + Number(n2);
    if (this.inputResultado) { //Si existe el input, accedemos a su value
      this.inputResultado.nativeElement.valueAsNumber = resultado;
    }
    return resultado;
  }
}
```

ViewChild lo que hace es asociar un elemento HTML en una variable (objeto) dentro del código, para así poder hacer referencia a sus propiedades desde código y poder modificarlas (por ejemplo, el `value`). Así no tenemos que cargar un módulo entero (`FormsModule`) para poder usar `ngModel` y el doble binding. En el caso del Testing, deberíamos importar ese módulo también más tarde y nos complicaría en exceso el ejemplo, y así de camino, vemos otra forma.

El uso de `ViewChild` es sencillo, veremos su sintaxis con el ejemplo siguiente:

```
@ViewChild("btnSumar") public btnSumar?: ElementRef<HTMLButtonElement>;
```

- `btnSumar` es la ID del elemento HTML. Desde HTML usaremos `#btnSumar` en lugar de `id="btnSumar"`.
- `btnSumar` es el nombre de nuestro atributo dentro del código (podemos usar el mismo y así evitamos confusiones. Le ponemos el signo "?" para indicar que es un elemento opcional y no es necesario inicializarlo en el constructor. Eso nos obligará en cada referencia a preguntar si es nulo o no).
- `ElementRef<Elemento>`. Es el tipo de dato de la variable. Dependiendo del tipo de elemento HTML que sea, usaremos los tipos que define angular para cada uno. usaríamos `HTMLButtonElement` para un botón, `HTMLInputElement` para un cuadro de texto, etc.

El HTML quedaría de la siguiente forma:

```
<!-- suma.component.html -->
<h1>Suma de dos números</h1>
<input type="number" #num1 placeholder="Número 1" />
<span> y </span>
<input type="number" #num2 placeholder="Número 2" />
<span> = </span>
<button #btnSumar (click)="sumar(num1.value,num2.value)">+</button>
<input type="number" #resultado placeholder="Resultado" />
```

Por último, **borraremos las pruebas que vienen por defecto** en el `app.component.spec.ts`, ya que al haber borrado todo el contenido del HTML, las pruebas que ya vimos darán error. Así no las confundiremos con las nuestras. Borramos solo las líneas que tienen `it(...)` dejando el resto.

```
//app.component.spec.ts
describe('AppComponent', () => {
  beforeEach(async () => { ...
  });
  it('should create the app', () => { ...
  });
  ...
});
```

Elaborando pruebas unitarias (it)

Con la calculadora, tendremos un proyecto sobre el cual podremos ir aplicando los fundamentos de testing en Angular. Ahora vamos a crear nuestras propias pruebas unitarias.

Veamos el código por defecto que crea AngularCLI en cualquier archivo de especificaciones (`spec.ts`). Por ahora nos centraremos en lo que está resaltado.

```
//suma.component.spec.ts
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { SumaComponent } from './suma.component';

describe('SumaComponent', () => {
  let component: SumaComponent;
  let fixture: ComponentFixture<SumaComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ SumaComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(SumaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

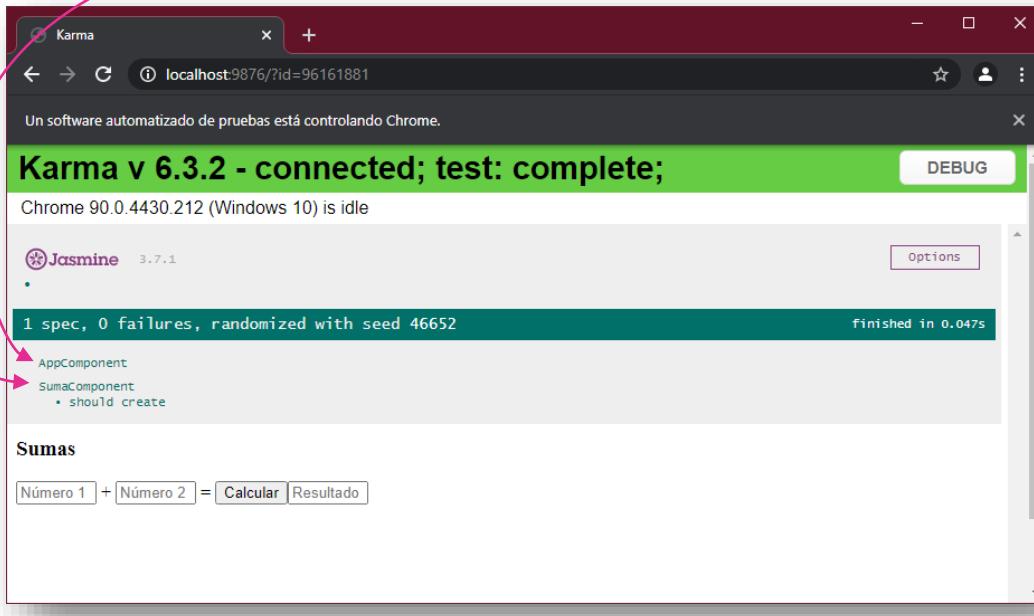
Cada método `it()` describe una prueba unitaria. Posee dos argumentos por defecto, el primero es un string con la descripción de la prueba, y el segundo es la definición de una función (función flecha) que será llamada al ejecutar la prueba.

```
it('descripción de la prueba', () => { contenido de la prueba });
```

Karma se encargará de recorrer el proyecto, buscar todos los archivos `spec.ts`, y hacer una llamada a cada uno de los métodos `it()` que tengamos. Dentro de la función, decidiremos los criterios según los cuales daremos la prueba por válida o no. Ahora mismo damos la prueba por válida si el objeto `component` es `true` (está creado).

Eso por ahora. Más adelante profundizaremos en el resto del código y en otros tipos de `expect`.

Si ahora ejecutamos las pruebas con `ng test`, deberán aparecer las pruebas unitarias para `suma.component` y para `app.component`. Veremos lo siguiente:



`AppComponent` no tienen ninguna prueba unitaria (tenía 3 y las hemos borrado) y `SumaComponent` le hemos dejado la que trae por defecto, que es ‘`should create`’.

Si queremos probar el método `sumar()` que tenemos implementado en `SumaComponent`, lo que siempre hemos hecho es una **prueba manual**, es decir, probar el método con 2 números cualesquiera (2+2) y el resultado *ESPERADO* debería dar su suma (4). En métodos tan simples como este, una prueba manual es más que suficiente, pero el método podría ser contar el número de vocales que tiene un párrafo de 500 letras, y ¿ahora qué? Podríamos echar un ratito entretenido contando para ver si lo hizo bien o no.

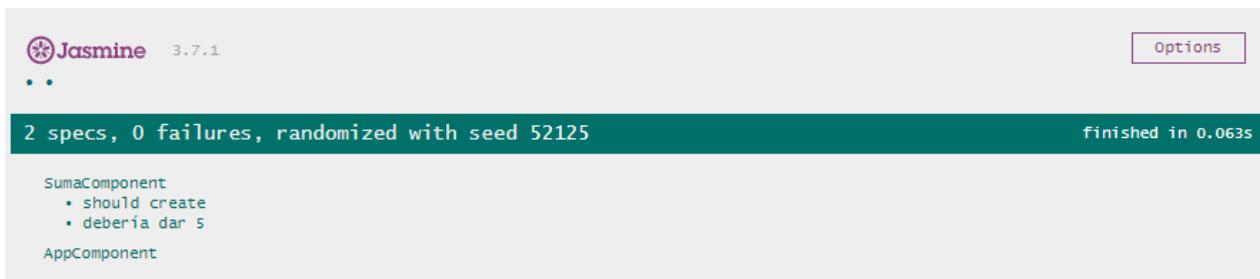
Para hacer de forma automatizada la prueba a `sumar()`, escribiremos el siguiente método `it()` debajo del que ya tenemos en `SumaComponent`.

```
it('debería de dar 5', () => {
  let real = component.sumar(3,2); //Ejecutamos el método
  let esperado = 5; //Guardamos lo esperado
  expect(real).toEqual(esperado); //Los comparamos
});
```

En la variable `component`, tenemos una instancia de `SumaComponent` (con todos sus atributos y métodos), así que hacemos lo siguiente:

1. Guardamos en una variable el resultado *REAL* que me retorna la función si le pasamos 2 números cualesquiera (3 y 2).
2. Guardamos en otra variable el resultado *ESPERADO*. Lo tendremos que calcular a mano. Es 5.
3. Comparamos lo que hemos obtenido (`real`) con lo que esperábamos obtener (`esperado`), y si son iguales, el método superó **con éxito la prueba**, o no pasó la prueba.

Ahora mismo el informe HTML de Jasmine será el siguiente:



Jasmine 3.7.1

Options

2 specs, 0 failures, randomized with seed 52125 finished in 0.063s

SumaComponent

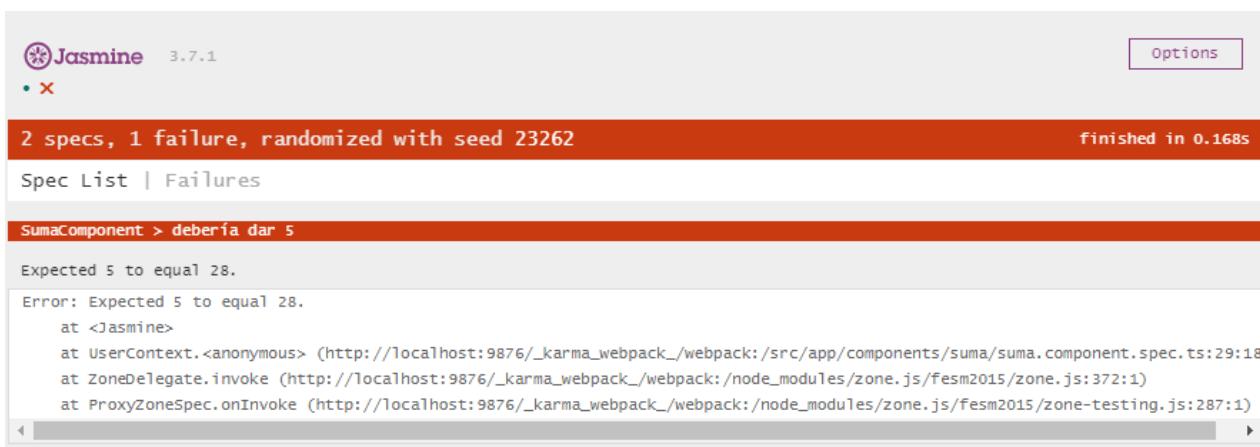
- should create
- debería dar 5

AppComponent

Para provocar un error, podríamos modificar en la prueba unitaria lo siguiente:

```
it('debería de dar 5', () => {
  let real = component.sumar(3,2); //Ejecutamos el método
  let esperado = 28;           //Guardamos lo esperado
  expect(real).toEqual(esperado); //Los comparamos
});
```

Ahora le decimos que el resultado *esperado* de sumar 3 y 2 es 28. Obviamente la función sumará correctamente y dará 5, y como el resultado REAL no es IGUAL al ESPERADO, no pasará la prueba.



Jasmine 3.7.1

Options

2 specs, 1 failure, randomized with seed 23262 finished in 0.168s

Spec List | Failures

SumaComponent > debería dar 5

Expected 5 to equal 28.

Error: Expected 5 to equal 28.
at <Jasmine>
at UserContext.<anonymous> (<http://localhost:9876/_karma_webpack_/webpack:/src/app/components/suma/suma.component.spec.ts:29:18>
at ZoneDelegate.invoke (<http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:372:1>
at ProxyZoneSpec.onInvoke (<http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone-testing.js:287:1>)

Este tipo de pruebas son muy válidas para probar métodos o funciones concretas, y asegurarse de que devuelven lo que se espera de ellas y no tengamos sorpresas pensando que un método funcionaba correctamente y resulta que no era así porque lo probamos con dos valores concretos, y por casualidad cósmica, con esos valores funcionaba.

Por ejemplo, $2+2=4$, y $2*2=4$. Imagina que estamos probando un método llamado `multiplicar(2, 2)`, pero en el método lo que hacemos es sumar los números. Nos devolverá 4 y pensaremos que estaba bien. Si hubiésemos probado otros números nos habríamos dado cuenta del error.

Lo más potente del testing, es que cada vez que ejecutemos los tests, volveremos a probar **TODAS** las pruebas, así comprobando periódicamente que una nueva parte del código no ha provocado errores en otros métodos que funcionaban bien con anterioridad.

 **¿Sabías que...?** Pueden deshabilitarse temporalmente las pruebas, renombrando el método `it()` por `xit()`. De esta forma no se ejecutarán. También funciona con la suite completa, cambiando `describe()` por `xdescribe()`.

Expect

Para las pruebas unitarias, lo más común es probar que un valor que hemos generado sea el mismo que esperamos. No sólo podemos esperar que sea IGUAL. ¿Y si queremos que sea válido si un número está en un rango? O bien queremos que la prueba sea exitosa si un valor no es `null`. O que sea `null`.

No podemos comprobar `==` o con cualquier operador que ya conoczamos. Debemos usar funciones propias de Jasmine. Las más comunes son:

- ▶ `toBe()` → Recibe cualquier valor, y dará la prueba por válida si el valor es `==` al esperado.

```
expect(valorReal:any).toBe(valorEsperado:any);
```

- ▶ `nothing()` → No espera nada, de una forma explícita.

```
expect().nothing();
```

- ▶ `toBeTruthy()` → Recibe cualquier valor, y dará la prueba por válida si es `true`. Se suele usar con valores booleanos primitivos (no para valores `truthy`).

```
expect(valor:boolean).toBeTruthy();
```

//Ejemplo

```
expect(usuario.isAdmin()).toBeTruthy();
```

- ▶ `toBeFalsy()` → Recibe cualquier valor, y dará la prueba por válida si es `false`. Se suele usar con valores booleanos primitivos (no para valores `falsy`).

```
expect(valor:boolean).toBeFalsy();
```

- ▶ `toBeTruthy(),toBeFalsy()` → Lo mismo que con `True` o `False`, pero valores `truthy` o `falsy`. Se usan comprobar que una expresión devuelve un valor booleano porque es `undefined`, `null`, un string vacío, etc. Para testear valores booleanos estrictos (`==`), usar `toBeTruthy` o `toBeFalsy`. [Más información aquí](#).

- ▶ `toBeNull()` → Recibe cualquier valor, y dará la prueba por válida si es `null`.

```
expect(valor:any).toBeNull();
```

- ▶ **Negar con `not`**. Todas las versiones de los métodos se pueden negar anteponiendo `not` de la siguiente forma:

//Ejemplos

```
expect(null).toBeNull();           //Pasará
expect(null).not.toBeNull();      //No pasará
```

- ▶ `toBeUndefined()` → Recibe cualquier valor, y dará la prueba por válida si ES `undefined`. Tiene su versión `toBeDefined()`.

```
expect(valor:any).toBeUndefined();
```

- ▶ `toEqual()` → Recibe cualquier valor, y dará la prueba por válida si igual al valor esperado .

```
expect(valor:any).toEqual(esperado:any);
```

- ▶ ***toBeNaN()*** → Recibe un número y dará la prueba por válida, si es un número.

```
expect(valor:number).toBeNaN(esperado:number);
```

- ▶ ***toBeGreaterThanOrEqual(number)*** → Recibe un número, y dará la prueba por válida si es mayor al valor esperado. Tiene su versión ***toBeGreaterThanOrEqual()***.

```
expect(valor:number).toBeGreaterThanOrEqual(esperado:number);
//Ejemplos
expect(10).toBeGreaterThanOrEqual(25);           //no pasará
expect(50).toBeGreaterThanOrEqual(25);           //pasará
```

- ▶ ***toBeLessThan(number)*** → Recibe un número, y dará la prueba por válida si es menor al valor esperado. Tiene su versión ***toBeLessThanOrEqual()***.

```
expect(valor:number).toBeLessThanOrEqual(esperado:number);
```

- ▶ ***toBeInstanceOf(cLase)*** → Recibe un objeto, y dará la prueba por válida si es una instancia de la clase esperada.

```
expect(valor:instance).toBeInstanceOf(esperado:class);
//Ejemplos
expect('pepino').toBeInstanceOf(String);    //pasará
expect(3).toBeInstanceOf(Number);            //pasará
expect('3').toBeInstanceOf(Number);          //no pasarás
expect(new Audio()).toBeInstanceOf(Audio);  //pasará
```

En una misma prueba unitaria se pueden añadir todos los ***expect()*** que necesitemos, y la prueba la dará por válida si se cumplen todas las especificaciones.

Puedes encontrar la lista completa, actualizada, y con ejemplos en la [documentación oficial de Jasmine](#).

Testing usando el DOM

Para probar métodos y funciones concretas, está bien lo anterior, pero también tenemos otra forma de testear de una forma más natural. Accediendo directamente al DOM para “manejar” la aplicación.

El componente **SumaComponent**, ahora mismo tiene la siguiente salida HTML:

Suma de dos números

Número 1 y Número 2 = + Resultado

En lugar de probar unidades de código independientes (como métodos, servicios, etc.), podríamos decirle al testing que introduzca tal valor en tal cuadro de texto, pulse tal botón, y compruebe que hay en el cuadro de resultado. Sin tener en cuenta que funciones son llamadas durante el proceso, ni los eventos ni nada.

Por ejemplo, desde código, podríamos introducir un **6** en el primer cuadro de texto, un **4** en el siguiente cuadro de texto, pulsar el botón “**+**” y comprobar que en el cuadro de texto “**resultado**” haya un **10**. Si hay un **10** (resultado esperado) damos la prueba por válida. De esta forma no hemos comprobado el método **sumar()**, tan siquiera tenemos que saber que será llamado o lo que hará internamente la aplicación, **si no que hemos probado la aplicación tal y como lo haría una persona**.

Veamos como acceder al DOM desde Jasmine. Creamos un prueba unitaria mediante el método **it()** con el siguiente contenido:

```
//suma.component.spec.ts
...
it('debería de introducir un 6 y 4 y dar un 10 como resultado', ()=>{
  let real;
  if (component.inputNum1 && component.inputNum2 && component.inputResultado
    && component.btnSumar) {
    component.inputNum1.nativeElement.valueAsNumber = 6;
    component.inputNum2.nativeElement.valueAsNumber = 4;
    component.btnSumar.nativeElement.click();
    real = component.inputResultado.nativeElement.valueAsNumber;
  }
  expect(real).toEqual(10);
});
```

Teniendo en cuenta que habíamos definido los elementos HTML mediante atributos con **ViewChild**, ahora podemos acceder a ellos mediante el atributo **component** que Jasmine nos crea por defecto en toda suite de pruebas. Ahí tendremos una instancia de la clase que estemos testeando y podremos acceder a todos sus métodos y atributos, en nuestro ejemplo **SumaComponent**.

Como los definimos en su momento con “?”, ahora tenemos que preguntar si el objeto está definido, porque si no dará un error al no estar inicializado. En el ejemplo, en la sentencia if preguntamos si

todos los objetos que vamos a manipular son `truthly`, si alguno no está definido, la expresión dará `false`, por lo que la prueba no dará valida.

Al ejecutarse, veremos esto en Jasmine:



Un software automatizado de pruebas está controlando Chrome.

Karma v 6.3.2 - connected; test: complete;

Chrome 91.0.4472.124 (Windows 10) is idle

Jasmine 3.7.1

• • • • •

5 specs, 0 failures, 1 pending spec, randomized with seed 07435 finished in 0.055s

AppComponent

SumaComponent

- should create
- en sumar() debería dar 100
- en multiplicar() debería dar 90 y ser un número
- debería recibir 2 números el método sumar() PENDING WITH MESSAGE: Temporarily disabled withxit
- debería de introducir un 6 y 4 y dar un 10 como resultado

Suma de dos números

6 y 4 = + 10

Podemos ver el resultado de manipular el DOM en la vista previa de la app. Hay que tener en cuenta que solo lo veremos si es la última prueba en ejecutarse, y que todas las pruebas, por defecto, se ejecutan en orden aleatorio.

Este tipo de pruebas son útiles porque nuestro método puede funcionar sin problemas usándolo bien, pero quizás en nuestra aplicación hacemos la llamada incorrectamente, o le pasamos los parámetros mal en forma o número, haciendo que el resultado final sea incorrecto. O más simple todavía, que al pulsar el botón “+” no hagamos nada (no tiene el evento click), con lo que el método `sumar()` funciona, pero nunca es llamado. O bien, si en lugar de `sumar()` tenemos `restar()` y en vez de pasarle los parámetros `a` y `b`, le pasamos `b` y `a`, el resultado no sería el esperado, mientras que la función si hace lo que debería. El fallo está en la llamada a la función, no en la función en sí.

BeforeEach y AfterEach

Cada prueba tiene un ciclo de vida, y puede ser necesario ejecutar ciertas tareas ANTES y DESPUÉS de cada una de las pruebas (*beforeEach* y *afterEach*, respectivamente).

Esto es muy interesante sobre todo si hacemos algún tipo de tarea repetitiva entre prueba y prueba, como puede ser reinicio de componentes, preparación de un escenario, etc.

En el código que viene ya por defecto en cualquier archivo `spec` de un componente, tendremos el código siguiente. Además le añadimos un método nuevo llamado `afterEach()`.

```
//suma.component.spec.ts
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { SumaComponent } from './suma.component';

describe('SumaComponent', () => {
  let component: SumaComponent;
  let fixture: ComponentFixture<SumaComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ SumaComponent ]
    })
    .compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(SumaComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  afterEach(()=>{
    //Código a ejecutar tras cada prueba
  });

  it('test1', () => { ... });
  it('test2', () => { ... });
});
```

Expliquemos todos los conceptos que encontramos aquí, a modo de glosario, para entender todo el código que estamos viendo.

1. ***describe()***: Una suite de pruebas comienza con una llamada a la función **describe()**. Recibe dos argumentos, un string y una función. El string es el título de la suite (normalmente el nombre del componente a testear). La función será el bloque de código que implementa la suite. Aquí vemos que se crean dos variables que serán de utilidad en todas las pruebas.
2. ***it()***: Contiene el código necesario para implementar el test. Recibe dos argumentos. Un string y una función. El string es la descripción de la prueba, y la función será el bloque de código que será ejecutado para llevar a cabo el test.
3. ***beforeEach()***: Como su nombre indica, es una función que será llamada **una vez antes** de ejecutar cada prueba unitaria (antes de la llamada a cada **it()**). Aquí por defecto, se inicializan dos objetos, **fixture** y **component**. También usamos **TestBed**.
 - ▶ **fixture**: Nos da acceso a los elementos internos del componente, como pueden ser los párrafos, botones, etc. La detección de cambios no se hace automáticamente, por lo que debemos llamar al método **detectChanges()** para decirle a Angular que detecte los cambios.
 - ▶ **component**: Nos da acceso a los atributos y métodos del componente.
 - ▶ **TestBed**: Es un entorno Mock (falso) de pruebas sobre el cual se crea el componente a probar. Se configura un módulo especial para las pruebas y ahí se ejecuta el componente.
4. ***afterEach()***: Es una función que será llamada una vez después de ejecutar cada prueba unitaria (después de la llamada a cada **it()**).
5. ***beforeEach() (async)***: Antes de ejecutar **beforeEach()**, debemos ejecutar OTRO **beforeEach()**, pero asíncrono. Es decir, deberá terminar la ejecución ESTE para seguir al otro **beforeEach()** (el que no es async). Esto es debido a que primero se ha ejecutar este, ya que aquí se prepara el módulo donde se ejecutará todo (aquí haremos las importaciones necesarias), y es donde se compila las plantillas externas de HTML y CSS con **compileComponents()**. Por todo esto será el primero en ejecutarse y esperarán a que termine.

{{ Pendiente de ampliar contenido }}

Repositorio en GitHub

Podrás encontrar el proyecto completo que hemos realizado durante testing en el siguiente repositorio:



<https://github.com/borilio/curso-angular-testing>

Desplegar en producción

Para que NodeJS compile el proyecto y cree los archivos necesarios para poder ser ejecutado en un servidor web, hay que hacer lo siguiente:

1. Parar el servidor de desarrollo.
2. Irnos a la consola de comandos, en la raíz del proyecto y escribir lo siguiente:

```
ng build
```

```
ng build --base-href=/test/angular
```

Si el proyecto se aloja en la raíz del servidor. Ej: www.miservidor.com

Si va a alojarse en otra ruta. Ej: www.miservidor.com/test-angular

```
expri@DESKTOP-ASGARD MINGW64 ~/Proyectos Angular SSD/heroes (add-firebase)
$ ng build --prod
✓ Browser application bundle generation complete.
✓ Copying assets complete.
✓ Index html generation complete.

Initial Chunk Files           | Names          | Size
main.54031f8c0ec0488de85b.js | main          | 666.97 kB
styles.3241d36ef715dcbfab99.css | styles        | 200.66 kB
scripts.d68dc738b00d7ec22291.js | scripts       | 164.38 kB
polyfills.94daef414b8355106ab.js | polyfills     | 35.98 kB
runtime.0022e3ea26b5a9eaa25a.js | runtime       | 2.22 kB

| Initial Total | 1.05 MB

Lazy Chunk Files             | Names          | Size
4.2a83582e459718001233.js    | -             | 175.22 kB

Build at: 2021-04-04T18:01:20.963Z - Hash: e6a96b44cf1cb9514f1f - Time: 24956ms
expri@DESKTOP-ASGARD MINGW64 ~/Proyectos Angular SSD/heroes (add-firebase)
$
```

Nota: Si obtienes un **warning** o un **error** con el mensaje: initial exceeded maximum budget. Budget 500.00 kB was not met by 570.22 kB with a total of 1.05 MB, [la solución está aquí](#).

3. AngularCLI nos habrá creado en la carpeta de raíz de nuestro proyecto una carpeta llamada **/dist**, con todo el proyecto compilado y minificado. Sólo debemos copiar el contenido de esa carpeta en cualquier alojamiento web que sea capaz de servir HTML, CSS y JS. Es decir, cualquiera 😊. Si no conoces ninguno, te aconsejamos [Netlify.com](#), gratis y fácil. Arrastrar y soltar.

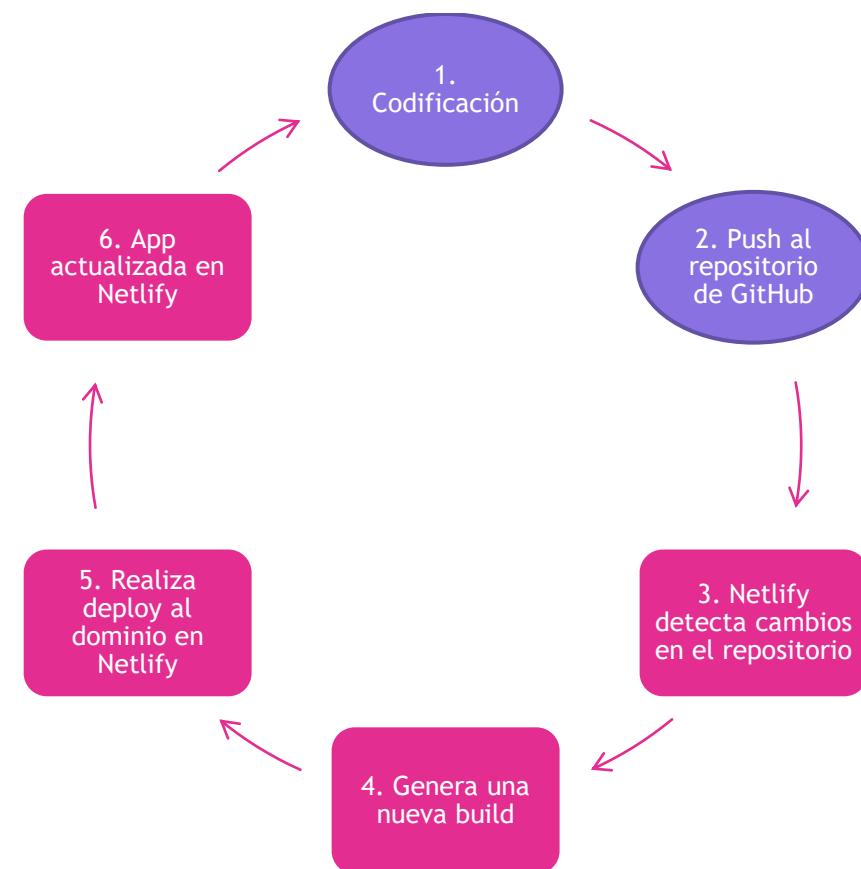
No intentes abrir localmente el proyecto con el navegador, puesto que no te va a funcionar. Aquí te dejamos la aplicación que hemos desarrollado desplegada en Netlify funcionando con Firebase. No destroces mucho la base de datos 😅.



<https://heroes-angular-sepe.netlify.app/>

Desplegando desde GitHub en Netlify.com

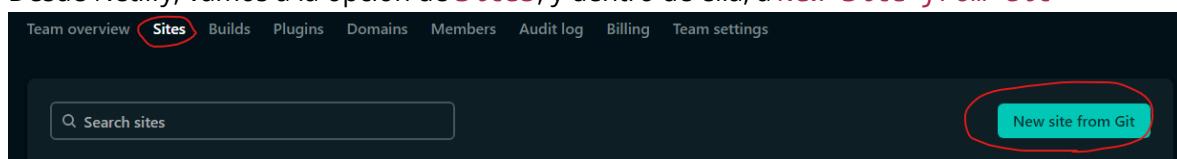
En el punto anterior hemos visto como agregar manualmente la carpeta generada en *dist* a Netlify, tan sólo arrastrando y soltando la carpeta. Netlify posee opciones de integración y desarrollo continuo, subiendo nuestra aplicación a GitHub y dejando a Netlify que haga el trabajo de construcción (*ng build*) y despliegue (*deploy*).



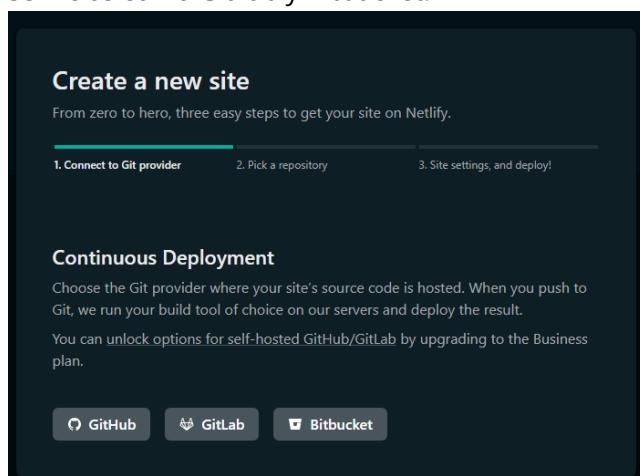
El usuario tan sólo tiene subir los cambios a un repositorio de GitHub (pasos 1 y 2), y Netlify realizará todos los siguientes pasos automáticamente (del 3 al 6).

Para configurarlo todo, realizamos lo siguiente:

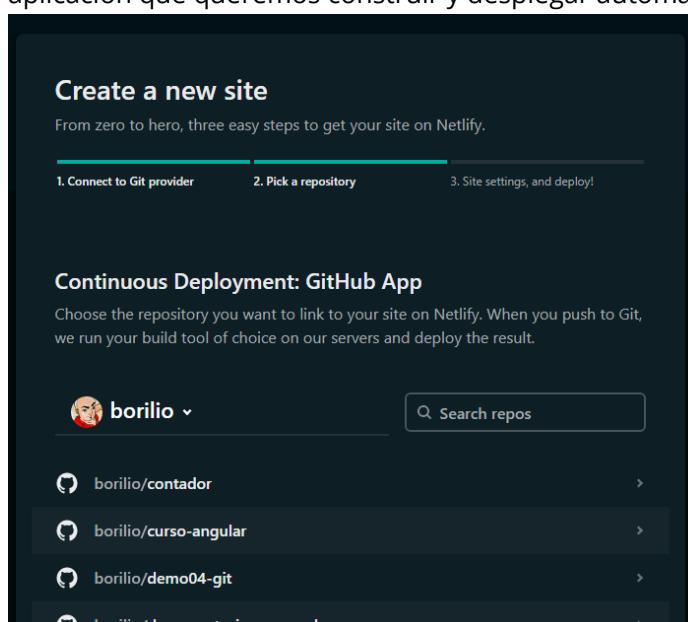
1. Subir nuestra aplicación a GitHub. VSC posee integración directa con GitHub para subir la aplicación directamente desde el IDE.
2. Desde Netlify, vamos a la opción de *Sites*, y dentro de ella, a *New Site from Git*



3. Conectamos con GitHub. Nos pedirá las credenciales de GitHub. También podemos usar otros servicios como GitLab y Bitbucket.



4. Nos mostrará todos los repositorios disponibles en nuestra cuenta. Seleccionamos el de la aplicación que queremos construir y desplegar automáticamente.



5. Seleccionamos la rama (*branch*) que queremos desplegar, normalmente será máster. Por último, para configurar el sitio, debemos decirle el comando que construirá la aplicación, que será ***ng build***, o bien, ***npm run build*** y **muy importante**, en el *publish directory*, tenemos que indicarle también la carpeta que hay en el interior de */dist*. En nuestro ejemplo, *dist/panel-matic*. Haz una vez el ***ng build*** manual para comprobar el nombre de la carpeta y úsalo aquí o bien míralo en el *package.json*.

Site settings for borilio/panel-matic

Get more control over how Netlify builds and deploys your site with these settings.

Owner: borilio's team

Branch to deploy: master

Basic build settings

If you're using a static site generator or build tool, we'll need these settings to build your site.

[Learn more in the docs ↗](#)

Base directory: .

Build command: `ng build`

Publish directory: `dist/panel-matic`

[Show advanced](#)

[Deploy site](#)

6. Pulsamos *Deploy site*. Puede tardar unos minutos en realizar el proceso completo. Puedes pulsar el *Site deploy in progress* para comprobar el estado con más detalle.

mystifying-brahmagupta-e0efaf

- Site deploy in progress

Deploys from GitHub. Created at 2:27 PM.



[Site settings](#) [Domain settings](#)

7. Una vez terminado deberás ver algo así. Ya se indica la url de la aplicación, y que ha sido correctamente desplegada desde la rama máster.

Deploys for mystifying-brahmagupta-e0efaf

- <https://mystifying-brahmagupta-e0efaf.netlify.app>

github.com/borilio/panel-matic, published master@HEAD.

Auto publishing is on. Deploys from master are published automatically.

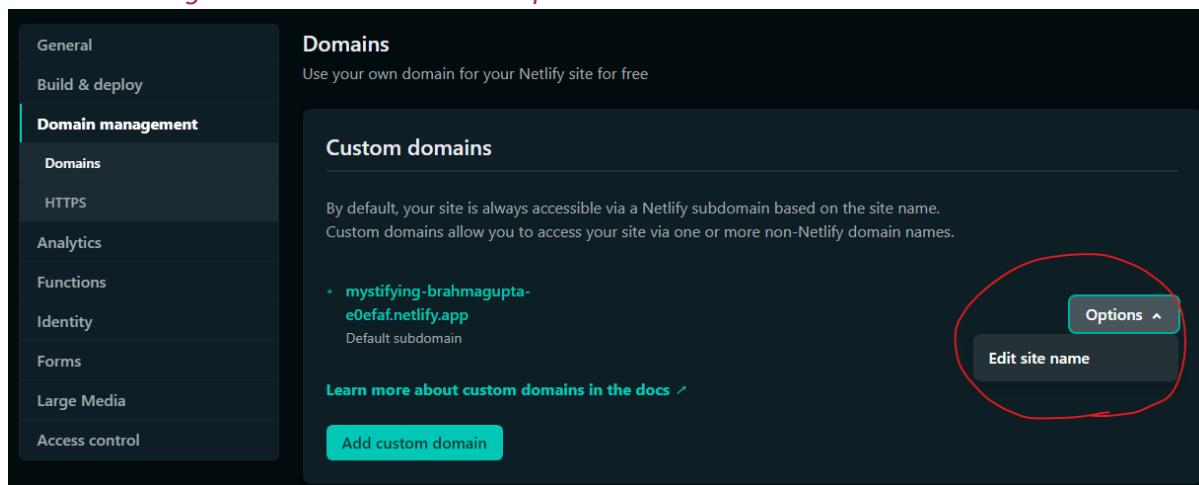
[Deploy settings](#) [Notifications](#) [Stop auto publishing](#)

[Trigger deploy ↗](#)

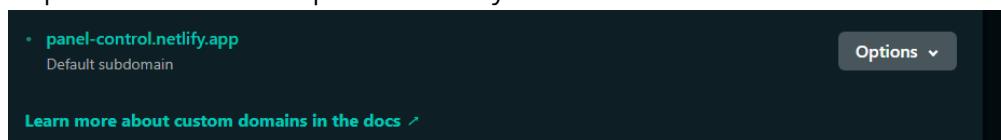
Production: master@HEAD Published Today at 2:27 PM
Deployed in 1m 34s

No deploy message

8. La url la genera Netlify, pero se puede personalizar. Pulsamos en *Deploy settings* → *Domain management* → *Domains* → *Options* → *Edit site name*



9. Le ponemos el nombre que deseemos y listo.

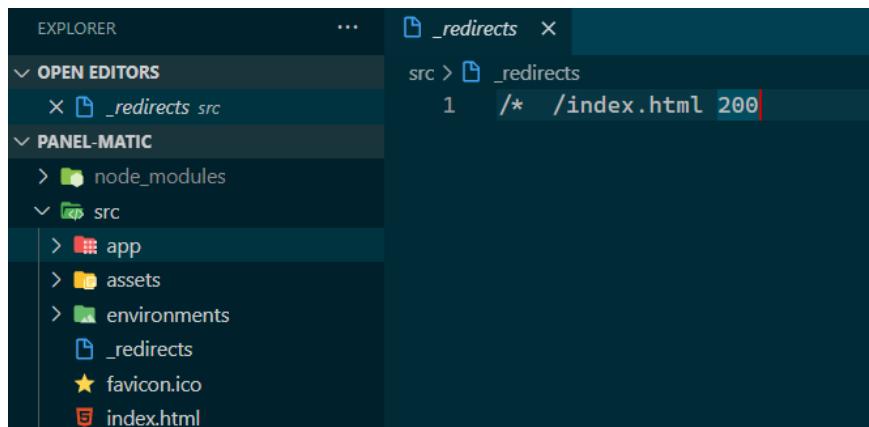


De esta forma, cada vez que hagamos un **push** a nuestro repositorio en la rama **master**, se realizará todo el proceso automáticamente, actualizándose nuestra aplicación en el servidor de producción, sin hacer nada más.

Si nuestra aplicación posee **Routing**, es posible que en Netlify no funcione correctamente, llevándonos a una página **404 Page not found**, cuando no debería. Para habilitar el Routing de Angular en Netlify, tenemos que hacer lo siguiente:

1. Crear un archivo llamado `_redirects` en la carpeta `/src`. Añadirle el contenido siguiente:

```
/* /index.html 200
```



2. Añadirle al `angular.json` el siguiente código, en el atributo `assets`.

```
{
  "glob": "_redirects",
  "input": "src",
  "output": "/"
}
```

Añadimos una captura para entender mejor su ubicación.

```
19      "options": [
20        "outputPath": "dist/panel-matic",
21        "index": "src/index.html",
22        "main": "src/main.ts",
23        "polyfills": "src/polyfills.ts",
24        "tsConfig": "tsconfig.app.json",
25        "assets": [
26          "src/favicon.ico",
27          "src/assets",
28          {
29            "glob": "_redirects",
30            "input": "src",
31            "output": "/"
32          }
33        ],
34        "styles": [
35          "./node_modules/bootstrap/dist/c
```

Créditos y menciones

Todos sois Heroínas y Heroes. Sin ustedes habría costado mucho más trabajo. Muchas gracias.

Parte de los textos con los que se ha elaborado este manual, se han extraído de las siguientes fuentes:

- ✉ [Documentación oficial de Angular](#)
- ✉ [Wikipedia](#)
- ✉ Distintos cursos de Angular en Udemy de [Fernando Herrera](#)
- ✉ Udemy → [Master en Frameworks JavaScript: Aprende Angular, React, Vue](#) de Víctor Robles
- ✉ [medium.com](#) → Observables
- ✉ [Yosoy.dev](#) Sergio Rodríguez Loza → [Valores Truthy y falsy](#)
- ✉ Udemy → [CRUD con Firebase de Tomás Ruiz Díaz](#)
- ✉ [Moldeo Interactive](#) → Testing con Karma, Jasmine
- ✉ [Documentación oficial de Firebase](#)
- ✉ [Developer.mozilla.org](#) → [Promesas](#)
- ✉ Alex García [@alxgcrz](#) → Apuntes de Angular.md
- ✉ [Tutorialsteacher.com](#) → Funciones Flecha (Lambda)
- ✉ [Academia Binaria](#) → Comunicaciones asíncronas. Alberto Basalo.
- ✉ Una pizca de [magia](#) 🧙



Esta obra está bajo una [licencia de Creative Commons Reconocimiento-Compartir Igual 4.0 Internacional](#).