

ANGULAR

ADOLFO SANZ DE DIEGO

PRONOIDE

2 ACERCA DE

2.1 AUTOR

- Adolfo Sanz De Diego
 - Blog: asanzdiego.blogspot.com.es
 - Correo: asanzdiego@gmail.com
 - GitHub: github.com/asanzdiego
 - Twitter: twitter.com/asanzdiego
 - LinkedIn: in/asanzdiego
 - SlideShare: slideshare.net/asanzdiego

2.2 LICENCIA

- Esta obra está bajo una licencia:
 - Creative Commons Reconocimiento-CompartirIgual 3.0

3 INTRODUCCIÓN

3.1 ANGULAR

- Framework de código abierto respaldado por Google, ayuda con la construcción de las **Single Page Applications**.
- El patrón Single Page Applications define que podemos construir o desarrollar aplicaciones web en una única página html.
- Aparte, sigue el patrón **Modelo Vista Controlador (MVC)**.

3.2 CARACTERÍSTICAS

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

3.3 LENGUAJE

- Angular recomienda usar **TypeScript** un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft, considerado como un superconjunto de JavaScript actualizado, que esencialmente añade tipado estático y objetos basados en clases.
- TypeScript no es ni mucho menos obligatorio. Se puede desarrollar en ES5 y ES6 sin problemas. Pero el conjunto de herramientas, la recomendación de la plataforma y la gran ventaja de la programación orientada a objetos, hace de TypeScript la mejor elección.

3.4 LIBRERÍAS Y HERRAMIENTAS

- Para poder empezar con Angular dos vamos a necesitar Node y su manejador de paquetes npm.

<https://nodejs.org/es/>

3.5 EDITORES

- Podemos usar cualquier editor de texto, aunque es recomendable alguno que nos de soporte al lenguaje que utilicemos.
- Está muy bien VSCode instalando varias extensiones para programar con Angular.

3.6 ARQUITECTURA

- Las partes fundamentales de una aplicación en Angular son:
 - Módulos
 - Components
 - Plantillas
 - Servicios
 - Inyección de dependencias
 - Directivas
 - Data binding

3.7 RECURSOS

- Documentación oficial: [angular.io](#)
- Colección curada de recursos: [awesome Angular](#)
- Material Design en Angular: [angular.io](#)

4 HOLA MUNDO A MANO

4.1 INSTALACIÓN DE HERRAMIENTAS

- Como hemos visto, para poder ejecutar Angular necesitamos Node y npm. La manera mas sencilla de empezar es descargarnos e instalar la distribución para nuestro sistema operativo de **Node**, que viene ya con su manejador de paquetes npm.
- Una vez instalado y para comprobar que todo ha ido bien:

```
$ node -v  
$ npm -v
```

4.2 CREACIÓN DE CONTENIDO

- Primero creamos una carpeta dónde añadiremos todos los recursos necesarios para nuestro proyecto. Dentro de ella necesitaremos:
 - **package.json**: Identifica las dependencias de paquetes npm para el proyecto.
 - **tsconfig.json**: Define la configuración de TypeScript y como compilará los archivos a JavaScript.
 - **systemjs.config.js**: Proporcionará la configuración de cómo serán cargados los módulos del proyecto.

4.3 INSTALACIÓN DE PAQUETES

- Para instalar todos los paquetes necesarios, desde consola ejecutamos:

```
$ npm install
```

- Esto descargará todos los paquetes descritos en los archivos de configuración que hemos creado antes. Al terminar, tendremos una carpeta llamada `node_modules` adicional a esta carpeta, crearemos una nueva llamada `app`, nos quedarán las siguientes carpetas.

4.4 COMPONENTES (I)

- Dentro de la carpeta app crearemos el archivo llamado **app.component.ts** con el siguiente contenido.

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

@Component({
  selector: 'my-app', // elemento html consumidor
  template: '<h1>Hello Angular!</h1>' //vista html que se renderizará
})

export class AppComponent { }
```

4.5 COMPONENTES (II)

- Los componentes son los bloques de construcción de Angular que representan regiones de la pantalla.
- Las aplicaciones se definen como árboles de componentes.

4.6 COMPONENTES (III)

- Cada componente a su vez está formado por tres partes:
 - La **vista**: es el código que se renderizará para los usuarios.
Esta plantilla también puede estar en un fichero de extensión .html.
 - La **clase controladora**: En TS usaremos clases para declarar los controladores que exponen datos y funcionalidad a la vista.
 - **Metadata**: Se declara como un decorador, una función especial de TypeScript, que recibe un objeto de configuración. Esto acompaña al controlador en un fichero de extensión .ts

4.7 COMPONENTES (IV)

- `@Component({...})` Lo que hace es asociar al controlador una plantilla HTML y un selector para ser invocado desde otra vista
 -

4.8 APP MODULE.TS (I)

- Todas las aplicaciones en Angular requieren mínimo de un módulo, dentro de la carpeta `app` crearemos un archivo llamado `app.module.ts` con el siguiente contenido.
- Un módulo no es más que una clase contenedora. Cada módulo puede incluir múltiples componentes y servicios. Normalmente un módulo dependerá de otros.

4.9 APP.MODULE.TS (II)

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

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

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ] // componente raíz para el arranque
})

export class AppModule { }
```

4.10 MAIN.TS

- En la carpeta app creamos un nuevo archivo llamado main.ts con el siguiente código.

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

4.11 INDEX.HTML

- Fueras de la carpeta app creamos también el archivo index.html

```
<html>
  <head>
    <title>Angular QuickStart</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script src="node_modules/core-js/client/shim.min.js"></script>
    ...
    <script src="systemjs.config.js"></script>
    <script>
      System.import('app/main.js').catch(function(err){ console.error(err); })
    </script>
  </head>
  <body>
    <my-app>Loading...</my-app>
  </body>
</html>
```

4.12 EJECUCIÓN

- Ahora ejecutaremos nuestro proyecto con el comando `npm start`

```
$ npm start
```

- Al momento de ejecutarlo en nuestra carpeta se crearán archivos JavaScript que fueron compilados de nuestros archivos TypeScript. Después se nos abrirá una ventana en el navegador con la dirección `http://localhost:3000/` y toda nuestra aplicación en Angular cargada.

5 ANGULAR-CLI

5.1 INTRODUCCIÓN

- Angular CLI es el intérprete de línea de comandos de Angular, facilita el inicio de proyectos y la creación del esqueleto, o scaffolding, de la mayoría de los componentes de una aplicación Angular.
- Angular CLI no es una herramienta de terceros, sino que nos la ofrece el propio equipo de Angular.

5.2 CARACTERÍSTICAS

- Nos facilita mucho el proceso de inicio de cualquier aplicación con Angular, ya que en pocos minutos te ofrece el esqueleto de archivos y carpetas que vas a necesitar, junto con una cantidad de herramientas ya configuradas.
- Además, durante la etapa de desarrollo nos ofrecerá muchas ayudas, generando el "scaffolding" de muchos de los componentes de una aplicación.
- Durante la etapa de producción o testing también nos ayudará, permitiendo preparar los archivos que deben ser subidos al servidor, transpilar las fuentes, etc.

5.3 INSTALACIÓN

- Igual que si lo hacemos a mano, necesitamos tener Node y npm instalados. La manera mas sencilla de empezar es descargarlos e instalar la distribución para nuestro sistema operativo de **Node**, que viene ya con su manejador de paquetes npm.
- Para instalar Angular cli, desde la terminal. Una vez instalado, desde la línea de comandos podremos usar el comando ng.
- Para comprobar que todo ha ido bien, ejecutamos **ng -v**.

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

5.4 CREACIÓN DE CONTENIDO

- Creamos un directorio para nuestra aplicación y ejecutamos lo siguiente:

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

- Después de unos segundos tendremos nuestro proyecto creado y listo para ejecutarse.
- Angular cli nos crea un proyecto bastante mas grande que si lo hacemos nosotros a mano, además añade test, pero en general es el mismo proyecto de Angular.

5.5 EJECUCIÓN

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

```
$ ng serve
```

5.6 OTROS COMANDOS (I)

Crear un componente:

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

Crear una directiva:

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

5.7 OTROS COMANDOS (II)

Crear un pipe:

```
$ ng g pipe nombre-pipe
```

Crear un servicio:

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

5.8 OTROS COMANDOS (III)

Crear un interface:

```
$ ng g interface nombre-interface
```

Crear una clase:

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

5.9 OTROS COMANDOS (IV)

Crear un enum:

```
$ ng g enum nombre-enum
```

Crear un módulo:

```
$ ng g module nombre-module
```

5.10 OTROS COMANDOS (V)

A esos comandos les podemos añadir las siguientes opciones:

- **-it**: no genera archivo de plantilla.
- **-is**: no genera archivo de estilos.
- **--flat**: genera los archivos en la carpeta actual en lugar de en una nueva carpeta.

5.11 ESTRUCTURA DEL PROYECTO (I)

- **e2e**: aquí se encuentran los archivos de testing End to End, que se ejecutan con la ayuda de Protractor y Jasmine.
- **node_modules**: aquí se encuentran todas las dependencias de nuestro proyecto.
- **src**: es posiblemente la carpeta más importante del proyecto, aquí se encuentra todo el código de la aplicación, y es en la carpeta que pasaremos la mayor parte del tiempo.

5.12 ESTRUCTURA DEL PROYECTO (II)

- **app**: en esta carpeta vamos a tener todos los componentes, servicios, plantillas y resto de elementos de la aplicación.
- **app.module.ts**: este archivo se encarga de entender que componentes y dependencias tenemos en el proyecto.
- **assets**: aquí guardaremos los assets que no pertenezcan a los componentes de Angular, como imágenes, sonidos...

5.13 ESTRUCTURA DEL PROYECTO (III)

- **environments**: en esta carpeta tendremos los archivos encargados de indicar los entornos de trabajo (producción, desarrollo...).
- **index.html**: este es el archivo donde se inicia la SPA, donde se van a cargar los componentes.
- **main.ts**: es el primer archivo en ejecutarse, y en el se encuentran los parámetros de configuración de la aplicación y del entorno en el que trabajaremos...

5.14 ESTRUCTURA DEL PROYECTO (IV)

- **polyfills.ts**: asegura la compatibilidad con los distintos navegadores.
- **styles.css**: aquí definimos los estilos globales que vamos a darle a la aplicación.
- **test.ts**: es el punto de entrada a los test unitarios, los haremos con Jasmine.

5.15 ESTRUCTURA DEL PROYECTO (V)

- **angular-cli.json**: aquí se encuentra la configuración de Angular-CLI.
- **tsconfig.json**: en este archivo se encuentra la configuración de TypeScript, donde se indica como hacer la compilación a JavaScript.
- **karma.conf.js**: aquí se encuentra la configuración de los test unitarios.

5.16 ESTRUCTURA DEL PROYECTO (VI)

- **package.json**: aquí se indican las dependencias del proyecto, las acciones a ejecutar cuando se lanza el proyecto, y alguna información del proyecto.
- **protractor.conf.js**: aquí se encuentra la configuración de los test End to End.

6 APPMODULE

6.1 INTRODUCCIÓN

- Cuando lanzamos la ejecución de la aplicación, Angular no carga todos los componentes, ni tiene en mente todos los selectores, esto es una perdida increíble de rendimiento.
- El primer archivo que se carga es el `main.ts` que es el que inicia la aplicación.
- Este archivo le indica que cargue lo que hay en `app.module.ts`, donde se define cual es el componente raíz (`app.component`), desde donde van a colgar todos los demás componentes de la aplicación.

6.2 INDEX.HTML

- El componente raíz tiene un selector, que es el que se usa en el archivo `index.html` para cargar el componente.
- En el archivo `index.html` no hay ningún script, por lo que parece que no carga ningún archivo para ejecutar nuestra aplicación, pero si lo inspeccionamos con las *herramientas de desarrollador* de nuestro navegador si que vemos unos cuantos scripts.
- De esos scripts, el más importante es el `main.bundle.js`, es el que tiene todo nuestro código y el código de los paquetes que necesitamos para ejecutar la aplicación.

6.3 EL MÓDULO RAÍZ

- Un módulo en Angular describe cómo encajan las partes de la aplicación. Cada aplicación tiene al menos un módulo, el módulo raíz que es el que inicializa la aplicación, puedes llamarlo como quieras, como convención lo llamaremos **AppModule**.

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

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

6.4 @NgModule

- Después de importar los módulos necesarios, tenemos el decorador/anotación `@NgModule`. Identifica AppModule como un módulo de Angular, coge sus metadatos y le dice a Angular como compilar y ejecutar la aplicación.
 - `imports`: importa clases necesarias para el funcionamiento de la aplicación. `BrowserModule` todas las aplicaciones la necesitan para poder ejecutarse en un navegador.
 - `declarations`: De momento solo añadimos el único componente de la aplicación que tenemos.
 - `bootstrap`: Añadimos el componente raíz, Angular lo creará e insertará dentro de la página `index.html`.

6.5 IMPORTS

- Dentro de los imports añadimos características que van necesitar nuestra aplicación.
- Muchas de estas características se organizan en módulos de Angular, por ejemplo los servicios HTTP están en el módulo `HttpModule`, el routing en el `RouterModule`.
- Añadimos a los *imports* los módulos/características que va a usar nuestra aplicación.

6.6 DECLARATIONS

- Añadimos qué componentes pertenecen al `AppModule`, cada vez que creamos un componente nuevo, deberemos añadirlo a este array. Si usamos un componente sin haberlo declarado antes, el browser nos devolverá un error.
- Solo añadimos en las declaraciones, componentes, directivas y pipes. No debemos añadir, otro módulos, servicios o clases del modelo.

6.7 BOOTSTRAP

- Inicia la aplicación cargando el componente raíz. Este proceso crea todos los componentes y los inserta en el DOM, normalmente desencadena la creación de todo el árbol del DOM.
- Aunque podemos poner más de un componente, no es lo normal, la mayoría de aplicaciones solo tienen un componente raíz desde el que se cargan todos los demás.
- El componente raíz se puede llamar de cualquier manera, pero por convención lo llamamos **AppComponent**.

7 COMPONENTES

7.1 INTRODUCCIÓN

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

7.2 PARTES

- Un componente consta de las siguientes tres partes fundamentales:
 - Un template
 - Una clase
 - Un decorador
- Los componentes a parte de encapsular contenido (como hacen las etiquetas), también encapsulan alguna funcionalidad.

7.3 IMPORTAR

- A la hora de crear un componente, hay que asegurarse de que se añade en el array de **declarations** del **app.module.ts** y de que se importa correctamente.

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

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

7.4 TEMPLATE (I)

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

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

7.5 TEMPLATE (II)

- Se puede definir el template en una linea o definirlo en un archivo externo.

```
@Component({
  selector: 'app-root',
  template: `<h1>Mi componente</h1>`,
  styles: [`color: black;`]
})

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

7.6 CLASS

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

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

7.7 DECORADOR (I)

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

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

7.8 DECORADOR (I)

- En el decorador anterior podemos distinguir las tres propiedades que le está agregando al componente.
 - **selector**: es el nombre de la etiqueta que se crea cuando se procesa el componente. Para mostrar el componente tenemos que llamar a la etiqueta `<app-root></app-root>` en el lugar del HTML donde queremos mostrarlo. El nombre del selector tiene que ser único en la aplicación.
 - **templateUrl**: indica donde se encuentra la plantilla o template del componente.
 - **styleUrls**: indica los archivos de estilos que se le van a aplicar a este componente.

7.9 CICLO DE VIDA DE LOS COMPONENTES

- Los componentes tienen un ciclo de vida que maneja Angular usando los hooks.
- Los hooks del ciclo de vida nos permiten interactuar con los componente en momentos claves del ciclo de vida, como por ejemplo justo cuando se ha inicializado un componente, cuando este sufre cambios en alguna de sus propiedades o cuando va a ser eliminado.
- Estos hooks son interfaces, las cuales tienen un método llamado como la propia interface precedido de ng.

7.10 HOOKS (I)

- **ngOnChanges**: se ejecuta cuando hay un cambio en el valor de alguna propiedad.
- **ngOnInit**: se lanza cuando se han inicializado todas las propiedades del componente, por lo que el componente también se ha inicializado.
- **ngOnDestroy**: se ejecuta justo antes de destruir un componente.

7.11 HOOKS (II)

- **ngDoCheck**: se llama en cada detección de cambios, justo después del **ngOnchanges** y **ngOnInit**.
- **ngAfterContentInit**: se ejecuta cuando se inserta contenido externo al componente en dicho componente (`<ng-content></ng-content>`).
- **ngAfterContentChecked**: cuando hay un cambio en el contenido insertado con `ng-content`.

7.12 HOOKS (III)

- **ngAfterViewInit**: se lanza cuando se ha inicializado la vista del componente y las vistas de sus componentes hijos.
- **ngAfterViewChecked**: se lanza después de checkear las vistas del componente y sus hijos, es decir, justo después del **ngAfterViewInit**.

7.13 EJEMPLO

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

  constructor(private itemService: ItemService) { }

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

8 DATA BINDING

8.1 INTRODUCCIÓN

- El **data binding** consiste en la sincronización entre los datos del componente y la vista.
- En Angular nos encontramos cuatro formas distintas de controlar el flujo en el que se mueven los datos:
 - **String Interpolation**
 - **Property Binding**
 - **Event Binding**
 - **Two-Way Binding**

8.2 STRING INTERPOLATION (I)

- El **string interpolation** se usa para renderizar el valor de una variable en las plantillas de los componentes, como valor de las etiquetas o incluso de alguno de los atributos de estas etiquetas.
- Los datos que se usan son solo de lectura.
- Se usa con `{}{nombreVariable}{}{}`.

8.3 STRING INTERPOLATION (II)

```
nombre: string = 'Sara';
```

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

8.4 PROPERTY BINDING (I)

- Las **property binding** se usan para enviar valores desde el componente a la plantilla.
- Se usa con `[propiedad]="expresión"`.
- La expresión tiene que devolver un valor del tipo que espera obtener la propiedad.
- También es de lectura. Se suele usar con las propiedades del DOM, de los componentes y de las directivas.

8.5 PROPERTY BINDING (II)

```
bordeRojo {  
  border: 1px solid red;  
}  
  
letraAzul {  
  color: blue;  
}
```

```
datosValidos: boolean = false;  
  
setStyles() {  
  return {  
    'font-size': '15px';  
    'text-decoration': 'line-through';  
  };  
}
```

8.6 PROPERTY BINDING (III)

```
<p [ngClass]="{bordeRojo: true, letraAzul: false}">...</p>
<!-- Añade la clase bordeRojo pero no añade la clase letraAzul. -->

<p [ngStyle]="setStyles()">...</p>
<!-- Añade los estilos que devuelve la función setStyles. -->

<button type="submit" [disabled]="!datosValidos">Enviar datos</button>
<!-- Deshabilita el botón si datosValidos es false. -->
```

8.7 PROPERTY BINDING (IV)

- A los componentes se les puede pasar desde fuera del propio componente (por ejemplo el componente padre) algún valor para alguna de sus propiedades.
- Esto es posible por el decorator `@Input`.

8.8 PROPERTY BINDING (V)

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

@Component({
  selector: 'mi-componente',
  template: `{{result}}`,
  styles: []
})

export class PropertyBindingComponent {
  @Input() result: number = 0;
  // valor por defecto si no pasan el Input
}
```

```
<mi-componente [result]="10"></mi-componente>
```

8.9 EVENT BINDING (I)

- Con el **event binding** le asignamos una función de un componente a un evento nativo como lo es el click, el mouseover, submit, ..
- Cuando un elemento detecte un evento sobre el que hay asignada una función, se ejecutará la función.
- En este caso la comunicación va desde la plantilla al componente.
- La sintaxis es `(evento)="expresión"` donde la expresión suele ser una función declarada en el componente.

8.10 EVENT BINDING (II)

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

```
<button (click)="saludar()">Saludame</button>  
<button (click)="console.log('Hola mundo en la consola!!!!')">Saludame en la
```

8.11 EVENT BINDING (III)

- Al igual que se puede crear un property binding, también se puede crear un event binding.

8.12 EVENT BINDING (IV)

- Lo primero de todo es importar EventEmitter que permite disparar o emitir eventos con su método `.emit()`.
- También es necesario usar el decorator `@Output` para permitir que se pueda emitir datos hacia fuera del componente.
- Se crea una instancia de EventEmitter que lleva el `@Output` con un alias para poder captar el evento desde el componente que está escuchándolo.
- Y se crea una función que disparará el evento cuando se realice una acción (por ejemplo un click en un botón).

8.13 EVENT BINDING (V)

- En el siguiente caso, cuando se pincha en el botón, se llama a la función `onClicked()`, la cual emite el evento.

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

@Component({
  selector: 'mi-event-binding',
  template: `<button (click)="onClicked()">Haz click</button>`,
  styles: []
})
export class EventBindingComponent {

  @Output('clickable') clicked = new EventEmitter<string>();

  onClicked() {
    // Emite el evento clickable
    this.clicked.emit('This works!');
  }
}
```

8.14 EVENT BINDING (VI)

- En el componente que está escuchando el evento personalizado se añade un event binding que llamará a una función que se ejecutará cuando ocurra dicho evento.

```
onClicked(value: string) {  
    alert(value);  
}
```

<!-- Llamará a la función onClicked cuando se dispare el evento clickable -->
<mi-event-binding (clickable)="onClicked(\$event)"></mi-event-binding>

- En el \$event se envía lo que va como parámetro en el método emit.

8.15 TWO-WAY BINDING (I)

- El Two-Way binding es la combinación de los dos casos anteriores, el property binding y el event binding y se refleja en la sintaxis.
- Recordad que se usaba [] para leer datos del componente en la plantilla y () para enviar datos desde la plantilla al componente.
- La sintaxis es [(ngModel)]="propiedad", por lo que lee y envía datos. Este caso es el que se suele usar en los formularios.

8.16 TWO-WAY BINDING (II)

```
persona = {  
    nombre: 'Robb'  
    apellidos: 'Stark'  
    edad: 28  
}
```

<!-- Al modificar el nombre en el primer input envía los datos al componente y cambia el valor, y al cambiar el valor en el componente, cambia el valor en el resto de elementos que lo usan. -->

```
<input type="text" [(ngModel)]="persona.nombre" />  
<input type="text" [(ngModel)]="persona.nombre" disabled />  
<label>{{persona.nombre}}</label>
```

9 PIPES

9.1 INTRODUCCIÓN (I)

- Los pipes permiten coger un dato de entrada y transformarlo sin cambiar su valor, solo la forma en que se visualiza.
- Se llaman pipes porque se usan con el símbolo |.

```
valor: string = 'esto es un string';
```

```
<p>{{ valor }}</p> <!-- esto es un string -->
<p>{{ valor | uppercase }}</p> <!-- ESTO ES UN STRING -->
```

9.2 INTRODUCCIÓN (II)

- Otro ejemplo de pipe es el siguiente que se usa con las fechas:

```
fechaNacimiento = new Date(1992, 2, 6);
```

```
<p>{{ fechaNacimiento }}</p> <!-- Thu Feb 06 1992 00:00:00 GMT+0100 -->
<p>{{ fechaNacimiento | date }}</p> <!-- Feb 6, 1992 -->
```

9.3 ARGUMENTOS (I)

- Algunos de los pipes existentes pueden aceptar uno o multiples argumentos.
- Para añadir los argumentos al pipe, hay que añadir al pipe : seguidos y luego el argumento.
- Si le podemos añadir mas de un argumento, sigue el mismo método, : seguidos del argumento.

9.4 ARGUMENTOS (II)

```
precio: number = 10;
```

```
<p>{{ precio }}</p> <!-- 10 -->
<p>{{ precio | currency:'EUR' }}</p> <!-- EUR10.00 -->
<p>{{ fechaNacimiento | date:'dd/MM/yy' }}</p> <!-- 06/02/92 -->
<p>{{ valor | slice:2 }}</p> <!-- to es un string -->
<p>{{ valor | slice:2:14 }}</p> <!-- to es un str -->
```

9.5 ARGUMENTOS (III)

- Es posible encadenar varios pipes poniendo uno detrás de otro separados por |.

```
<p>{{ valor | slice:2:14 | uppercase }}</p> <!-- TO ES UN STR -->
```

9.6 CREAR UN PIPE (I)

- Es posible crear pipes propios con el siguiente comando de angular-cli:

```
$ ng g p miNuevoPipe
```

- Con este comando, Angular-CLI genera dos archivos, uno de testing, y el otro es donde se define el pipe.

9.7 CREAR UN PIPE (II)

- En el archivo donde se define el pipe se encuentra el decorator `@Pipe` el cual añade el nombre con el que vamos a llamar al pipe cuando se quiera usar.

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

@Pipe({
  name: 'mi-nuevo-pipe'
})
export class MiNuevoPipe implements PipeTransform {
  transform(value: any, args?: any): any {
    return (value * 2);
  }
}
```

9.8 CREAR UN PIPE (III)

- La función de `transform` es la que se ejecuta cada vez que usamos el pipe y es la que está definida en el interface que implementa la clase del pipe, el `PipeTransform`.
- Esta función recibe como primer parámetro el valor que vamos a transformar, y como segundo los parámetros que se le pueden pasar al pipe.
- Hay que revisar que el pipe se ha añadido al array de `declarations` que hay en el `app.module.ts`.

9.9 PIPES PUROS E IMPUROS (I)

- Todos los pipes vistos antes son **pure pipes**, es decir que solo se ejecutan si el valor primitivo o la referencia a si mismo cambian.
- También existen los **impure pipes** que se usan cuando se necesita detectar cualquier cambio que se detecte.
- Un ejemplo de impure pipe puede ser cuando añadimos o quitamos elementos de un array.

9.10 PIPES PUROS E IMPUROS (II)

- El pure pipe no se ejecutarían porque no es el valor o la instancia lo que cambia, sino que es el contenido.
- Por defecto, todos los pipes son puros, para indicar que un pipe es impuro hay que poner en el decorator `pure: false`.
- Por supuesto, los pipes impuros conllevan un gasto mayor de recursos porque se ejecutan cada vez que se realiza una acción.

9.11 PIPES PUROS E IMPUROS (II)

```
@Pipe({
  name: 'filter',
  pure: false
})
export class FilterPipe implements PipeTransform {
  transform(value: any, args?: any): any {
    let resultArray = [];
    for (let item of value) {
      if (item.match('^.*' + args + '.*$')) {
        resultArray.push(item);
      }
    }
    return resultArray;
  }
}
```

9.12 PIPES PUROS E IMPUROS (III)

```
<ul>
  <li *ngFor="let item of values | filter:'ea'">{{ item }}</li>
</ul>
<input type="text" value="{{newItem}}">
<button (click)="values.push(newItem.value)">Add Item</button>
```

9.13 ASYNC PIPE (I)

- El **async pipe** es un ejemplo de impure pipe que se subscribe a un **Observable** o a una **Promise** y devuelve el último valor emitido.
- Una vez que se elimina el componente, este pipe cancela la subscripción para evitar perdidas de memoria.

9.14 ASYNC PIPE (II)

```
asyncValue = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Data is here!'), 2000);
});
```

```
<p>{{ asyncValue | async }}</p>
```

10 DIRECTIVAS

10.1 INTRODUCCIÓN

- Las directivas son como componentes (sin template) que le dicen a Angular que hacer en ciertas partes del código.
- Añaden comportamiento dinámico a la aplicación usando etiquetas o selectores.
- Los componentes son directivas con template, de hecho `@Component` es un decorator de `@Directive` con características propias de los templates.

10.2 TIPOS

- Además de los componentes, tenemos otros tipos de directivas:
 - Directivas atributo
 - Directivas estructurales

10.3 DIRECTIVAS ATRIBUTO (I)

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

10.4 DIRECTIVAS ATRIBUTO (II)

- **ngModel**: implementa el mecanismo del two-way binding.

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

10.5 DIRECTIVAS ATRIBUTO (III)

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

```
<p [ngClass]="{bordeRojo: true, letraAzul: false}">...</p>
```

10.6 DIRECTIVAS ATRIBUTO (IV)

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

```
<p [ngStyle]="{'font-size': '15px'}">...</p>
```

10.7 DIRECTIVAS ATRIBUTO (V)

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

10.8 CREAR UNA DIRECTIVA ATRIBUTO (I)

- Para crear tus propias directivas se usa el comando que nos da Angular-CLI para ello:

```
$ ng g d miHighlight
```

10.9 CREAR UNA DIRECTIVA ATRIBUTO (II)

- Este comando nos genera unos archivos, en los que el más importante que es donde se define la directiva es el `highlight.directive.ts`.
- En este archivo se encuentra el decorator `@Directive` en el que se indica el selector con el que se va a llamar.

10.10 CREAR UNA DIRECTIVA ATRIBUTO (III)

- Este decorador no tiene el templateUrl ni el styleUrls porque como hemos dicho antes, las directivas son componentes sin plantilla, y al no haber plantilla no se necesitan estilos.
- Para poder usarla hay que añadirla en el array de declarations del archivo app.module.ts.

10.11 CREAR UNA DIRECTIVA ATRIBUTO (IV)

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

@Directive({
  selector: '[miHighlight]'
})

export class MiHighlightDirective {
  constructor(private elementRef: ElementRef) {
    this.elementRef.nativeElement.style.backgroundColor = 'green';
  }
}
```

10.12 CREAR UNA DIRECTIVA ATRIBUTO (V)

- El ElementRef se inyecta en el constructor de la directiva para dar acceso a los elementos del DOM.

```
<p miHighlight>Esto esta subrayado</p>
```

- Hay que asegurarse de que el selector de la directiva sea único, para evitar conflictos con otras directivas. Una buena práctica es añadirle un prefijo al selector. El prefijo *ng* está reservado para Angular.

10.13 CREAR UNA DIRECTIVA ATRIBUTO (VI)

- La directiva se puede hacer más dinámica, haciendo que reaccione a ciertos eventos.
- Para ello se necesita usar el decorador `@HostListener` que se subscribe a los eventos que recibe el elemento del DOM al que se le ha asignado la directiva.

10.14 CREAR UNA DIRECTIVA ATRIBUTO (VII)

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

export class MiHighlightDirective {
  constructor(private elementRef: ElementRef) { }

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight('yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.elementRef.nativeElement.style.backgroundColor = color;
  }
}
```

10.15 CREAR UNA DIRECTIVA ATRIBUTO (VIII)

- También se le puede pasar como parámetro el color con el que se quiere realizar el marcado del texto.
- Y para eso se ha importado el decorator `@Input` que permite asignar un valor que nos manda el template a la propiedad del componente.

```
@Input() highlightColor: string;
```

```
<p miHighlight highlightColor="yellow">Marcado en amarillo</p>
<p miHighlight [highlightColor]="blue">Marcado en azul</p>
```

10.16 CREAR UNA DIRECTIVA ATRIBUTO (IX)

- De esta forma se le indica el color que se aplica en el elemento que tiene la directiva, y se puede hacer mas simple añadiendole un alias al decorator @Input para aprovechar la llamada a la directiva y asignarle en la misma llamada el color.

```
@Input('miHighlight') highlightColor: string;
```

```
<p [miHighlight]="blue">Marcado en azul</p>
```

10.17 CREAR UNA DIRECTIVA ATRIBUTO (X)

- El `@HostListener` puede recibir un segundo parámetro `[$event]` en el decorador que le indica que evento ha ocurrido y para poder acceder a ese evento, se le pasa como parámetro a la función que se ejecuta.

```
@HostListener('mouseenter', ['$event']) mouseover(event) {  
    // Muestra el evento que ha ocurrido.  
    console.log(event);  
    // Muestra el elemento del DOM sobre el que ha ocurrido el evento.  
    console.log(event.target);  
};
```

10.18 DIRECTIVAS ESTRUCTURALES (I)

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

10.19 DIRECTIVAS ESTRUCTURALES (II)

- ** *ngIf**: esta directiva añade o elimina un elemento del DOM dependiendo de si se cumple o no la condición.

```
mostrar: boolean = false;
```

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

10.20 DIRECTIVAS ESTRUCTURALES (III)

- ** *ngFor**: esta directiva repite el elemento en el DOM una vez por cada item que hay en el iterador que se le pasa.

```
items: [] = [1, 2, 3, 4];
```

```
<li *ngFor="let item of items; let i = index">{{ item }} - {{ i }}</li>
```

- Para poder mostrar el índice del array, usamos **let i = index** donde **index** es una palabra clave que da acceso a dicho índice.

10.21 DIRECTIVAS ESTRUCTURALES (IV)

- Otra directiva estructural es el `ngSwitch` la cual nos permite mostrar algo dependiendo de su valor. Es algo distinta a las anteriores porque ella no va precedida del `**`, *sino que son los elementos `ngSwitchCase`** y ** `ngSwitchDefault` los que llevan el **.*

```
<div [ngSwitch]="value">
  <p *ngSwitchCase="10">10</p>
  <p *ngSwitchCase="100">100</p>
  <p *ngSwitchDefault>Default</p>
</div>
```

10.22 EL PREFIJO * (I)

- Angular usa el *** para usar las directivas estructurales de una forma más sencilla.
- La etiqueta <template> sirve para tener elementos HTML que a priori no se muestran, pero puede que después se necesiten mostrar usando JavaScript.
- Angular podrá usar ese contenido para mostrarlo con alguna de las directivas estructurales.

10.23 EL PREFIJO * (II)

```
<!-- Con el * -->
<div *ngIf="true">Hola mundo</div>

<!-- Sin el * -->
<template [ngIf]="true">
  <div>Hola mundo</div>
</template>
```

- Cuando se cumple la condición del `ngIf`, ejecuta el setter de la directiva para mostrar el `<div>Hola mundo</div>` que obtiene del `<template>`. Esto se ve mas claramente en el siguiente punto.

10.24 CREAR UNA DIRECTIVA ESTRUCTURAL (I)

- Se puede construir una directiva estructural personalizada.
- Hay que lanzar el comando de Angular-CLI para crear una directiva, lo que crea unos archivos.
- En el archivo TypeScript de la directiva se importan los siguientes elementos que son necesarios: `Input`, `TemplateRef` y `ViewContainerRef`.

10.25 CREAR UNA DIRECTIVA ESTRUCTURAL (II)

- El `TemplateRef` te da acceso al contenido que hay dentro del `<template>`, mientras que el `ViewContainerRef` te da acceso a un elemento que tiene la directiva estructural.
- En otras palabras, el `TemplateRef` indica que elemento hay que añadir y el `ViewContainerRef` indica donde hay que añadirlo.
- Para poder usar la directiva con el `***` hay que poner el mismo nombre del selector a la función.

10.26 CREAR UNA DIRECTIVA ESTRUCTURAL (III)

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

@Directive({ selector: '[miUnless]' })
export class MiUnlessDirective {

    constructor(private tempRef: TemplateRef<any>, private viewContainer: ViewContainerRef) {}

    @Input() set miUnless(condicion: boolean) {
        if (!condicion) {
            this.viewContainer.createEmbeddedView(this.tempRef);
        } else {
            this.viewContainer.clear();
        }
    }
}
```

```
<p *miUnless="true">Esto no se muestra</p>
```

11 SERVICIOS E INYECCIÓN DE DEPENDENCIAS

11.1 INTRODUCCIÓN

- Los servicios son una pieza fundamental de Angular, todo valor, función o característica que necesita la aplicación (constantes, lógica de negocio...) se encapsula dentro de los servicios.
- Los servicios son simples clases que se usan para interactuar con la BBDD, acceder a la lógica de negocio entre distintos sitios de la aplicación, para la comunicación entre componentes y clases (por ejemplo emitir un evento y escucharlo en distintos sitios de la aplicación sin tener que construir cadenas de eventos o propiedades).

11.2 CREACIÓN DE SERVICIOS (I)

- Para crear un servicio se usa el comando de Angular-CLI:

```
$ ng g s miServicio
```

11.3 CREACIÓN DE SERVICIOS (II)

- En el archivo mi-servicio.service.ts aparece `@Injectable()`.
- Es un decorador que indica que el servicio espera utilizar otros servicios.
- No es necesario ponerlo si el servicio no usa ningún otro servicio.

```
export class MiServicio {  
  // ...  
}
```

11.4 INYECCIÓN DE DEPENDENCIAS (I)

- La **inyección de dependencias** es un mecanismo que proporciona nuevas instancias de una clase.
- La mayoría de las dependencias son servicios y Angular inyecta estos servicios en los componentes que los necesitan.
- Se pueden injectar las dependencias a través de los constructores de los componentes, las directivas o las clases.

11.5 INYECCIÓN DE DEPENDENCIAS (II)

- Para ello se necesita importar lo que queremos inyectar y pasarlo como argumento en el constructor.

```
import { MiServicio } from './mi-servicio.service';
constructor(miServicio: MiServicio) {}
```

11.6 UNA INSTANCIA VS MÚLTIPLES INSTANCIAS (I)

- Hay que saber si se necesita que solo sea ese componente el que pueda acceder a los datos y funciones que provee ese servicio o se necesita que a parte de ese componente haya otros que puedan acceder a los datos y funciones del servicio.
- Por ejemplo, al implementar una cesta de la compra para un e-commerce, se debería de poder añadir items desde distintos sitios de la aplicación, en este caso tiene sentido que todos los componentes, desde los que se puede añadir items a la cesta, tengan la misma instancia del servicio para que los items que añadas se incluyan en la cesta del usuario.

11.7 UNA INSTANCIA VS MÚLTIPLES INSTANCIAS (II)

- Con el ejemplo anterior, si en lugar de usar una instancia para todos, usamos distintas instancias en cada uno de los componentes desde los que se pueden añadir los items, cada componente añadiría los items en cestas distintas, las cuales no estarían sincronizadas.
- En el caso de querer varias instancias del servicio, hay que indicarlo en el array de providers de cada componente que lo vaya a usar.

11.8 UNA INSTANCIA VS MÚLTIPLES INSTANCIAS (III)

- Por otro lado, si se quiere usar una instancia para varios componentes, basta con añadir el servicio al array de providers de un componente superior que contenga los componentes que lo van a usar. Si se necesita usar en cualquier lugar de la aplicación, la mejor opción es añadirlo en el archivo **app.module.ts**.

```
import { MiServicio } from './mi-servicio.service';

@Component({
  ...
providers: [MiServicio]
})
```

11.9 MÚLTIPLES INSTANCIAS (I)

- Ejemplo de un servicio de Log:

```
$ ng g s log
```

```
export class LogService {  
  writeToLog(logMessage: string) {  
    console.log(logMessage);  
  }  
}
```

11.10 MÚLTIPLES INSTANCIAS (II)

- Para usar este servicio en dos componentes distintos hay que injectarlo en sus constructores.

```
import { Component } from '@angular/core';
import { LogService } from './log.service';

@Component({
  selector: 'app-cmp-a',
  template: `<input type="text"><br />
    <button (click)="onLog()">Log</button>`,
  styles: [],
  providers: [LogService]
})

export class CmpAComponent {
  constructor(private logService: LogService) {}
}
```

11.11 MÚLTIPLES INSTANCIAS (III)

- Para hacer que escriba en la consola lo que se introduce en el input hay que añadir una referencia en el input para poder pasarle el valor como parámetro en la función onLog. Y hay que implementar la función onLog la cual usara el servicio para mostrar el texto en la consola.

```
<input type="text" #input>
<button (click)="onLog(input.value)">Log</button>
```

```
onLog(value: string) {
  this.logService.writeToLog(value);
}
```

11.12 MÚLTIPLES INSTANCIAS (IV)

- Ahora hay que hacer exactamente lo mismo en el segundo componente, el CmpB:

```
import { Component } from '@angular/core';
import { LogService } from './log.service';

@Component({
  selector: 'app-cmp-b',
  template: `<input type="text" #input><br>
    <button (click)="onLog(input.value)">Log</button>`,
  styles: [],
  providers: [LogService]
})

export class CmpBComponent {
  constructor(private logService: LogService) {}
  onLog(value: string) {
    this.logService.writeToLog(value);
  }
}
```

11.13 MÚLTIPLES INSTANCIAS (V)

- Cada componente tiene el LogService en su respectivo providers, por lo que son instancias distintas, pero para el log es indiferente, no es necesario que estén sincronizados.
- Para que los dos componentes usen la misma instancia de un servicio, este habría que añadirlo en el providers del componente padre (en este caso seria el `app.component`), o en el `app.module.ts`.

11.14 UNA INSTANCIA (I)

```
$ ng g s data
```

```
export class DataService {  
  private data: string[] = [];  
  constructor() { }  
  
  addData(input: string) {  
    this.data.push(input);  
  }  
  
  getData() {  
    return this.data;  
  }  
}
```

11.15 UNA INSTANCIA (II)

- Este servicio se añade en el providers del app.component.ts para que los componentes A y B usen la misma instancia. De esta forma, los datos estarán sincronizados.

```
<h1>Services & Dependency Injection</h1>
<app-cmp-a></app-cmp-a>
<app-cmp-b></app-cmp-b>
```

```
import { DataService } from './data.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: [DataService]
})
```

11.16 UNA INSTANCIA (III)

```
import { Component } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-cmp-a',
  template: `<input type="text" #input>
    <button (click)="addData(input.value)">addData</button>
    <ul><li *ngFor="let item of getData()">{{item}}</li></ul>`,
  styles: []
})
export class CmpAComponent {
  constructor(private DataService: DataService) { }
  addData(value: string) { this.dataService.addData(value); }
  getData() {
    this.items = this.dataService.getData();
  }
}
```

11.17 INYECTANDO SERVICIOS EN OTROS SERVICIOS (I)

- Hasta ahora se han injectado servicios en componentes, pero hay veces que se necesita insertar un servicio en otro servicio.
- Siguiendo con el ejemplo anterior, se quiere mostrar por la consola cada item nuevo que se añade en el array del DataService.
- Hay que importar e injectar el LogService en el DataService, y usarlo para mostrar información en la consola de los items que se van añadiendo.

11.18 INYECTANDO SERVICIOS EN OTROS SERVICIOS

(II)

```
import { LogService } from './log.service';

export class DataService {
    private data: string[] = [];
    constructor(private logService: LogService) { }
    addData(input: string) {
        this.data.push(input);
        this.logService.writeToLog('Saved item: ' + input);
    }
    getData() {
        return this.data;
    }
}
```

11.19 INYECTANDO SERVICIOS EN OTROS SERVICIOS

(III)

- Esto no funciona, si se prueba se puede ver que da un error en la consola, y esto se debe a que no se le está indicando que a este servicio se le está inyectando un servicio externo.
- Ahora es el momento de añadir la parte del `@Injectable()`.

```
import { Injectable } from '@angular/core';
import { LogService } from './log.service';

@Injectable()
export class DataService {
  ...
}
```

11.20 INYECTANDO SERVICIOS EN OTROS SERVICIOS (III)

- Ahora se necesita usar el LogService desde el nivel más alto de la aplicación, por lo tanto se añade en el `app.module.ts`.

```
import ... ;

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

11.21 INTERACCIÓN ENTRE COMPONENTES (I)

- Los servicios también se pueden usar para que los componentes interactúen entre ellos sin necesidad de implementar cadenas de eventos a través de los componentes de la aplicación.
- En este caso se va a añadir la funcionalidad para enviar un dato de un componente a otro y que aparezca automáticamente.

11.22 INTERACCIÓN ENTRE COMPONENTES (II)

- En DataService crear un EventEmitter para emitir el valor.

```
import { Injectable, EventEmitter } from '@angular/core';
import { LogService } from './log.service';

@Injectable()
export class DataService {
  pushedData = new EventEmitter<string>();
  private data: string[] = [];
  constructor(private logService: LogService) { }
  addData(input: string) {
    this.data.push(input);
    this.logService.writeToLog('Saved item: ' + input);
  }
  getData() { return this.data; }
  pushData(value: string) {
    this.pushedData.emit(value);
  }
}
```

11.23 INTERACCIÓN ENTRE COMPONENTES (III)

- Añadimos en el componente A un botón para enviar el valor del input.
- Para ello en la función que se ejecuta al pulsar el botón de enviar llamamos a la función que se ha implementado en el DataService y que se encarga de emitir el dato al componente B.

11.24 INTERACCIÓN ENTRE COMPONENTES (IV)

```
import { Component } from '@angular/core';
import { LogService } from './log.service';
import { DataService } from './data.service';

@Component({
  selector: 'app-cmp-a',
  template: `<input type="text" #input>
    <button (click)="onSend(input.value)">Send</button>`,
  styles: []
})
export class CmpAComponent {
  constructor(
    private logService: LogService,
    private dataService: DataService) {
  }
  onSend(value: string) { this.dataService.pushData(value); }
}
```

11.25 INTERACCIÓN ENTRE COMPONENTES (V)

- Y en el componente B hay que suscribirse al evento que se emite en el DataService, es decir que cuando se detecta el emit, se ejecutará la función de callback en la que se va a actualizar el valor de la variable value con el que llega como argumento en el callback.

11.26 INTERACCIÓN ENTRE COMPONENTES (VI)

```
import ...;

@Component({
  selector: 'app-cmp-b',
  template: `<p>{{value}}</p>`,
  styles: []
})
export class CmpBComponent implements OnInit {
  value = '';
  constructor(
    private logService: LogService,
    private DataService: DataService) {
  }
  ngOnInit() {
    this.dataService.pushedData.subscribe( data => this.value = data );
  }
}
```

12 FORMULARIOS

12.1 INTRODUCCIÓN

- Los formularios son una de las partes principales de una aplicación.
- Los formularios se usan para acciones como el log in, la reserva de un vuelo, establecer una reunión...
- Angular permite manejar el uso de formularios de dos formas:
 - **Template-driven approach**
 - **Data-driven approach**
- Al ser un framework, también da soporte a las validaciones y al control de errores.

12.2 FORMULARIO BASADO EN PLANTILLAS (I)

Una vez que se tiene la estructura de un formulario creada, vamos a ir viendo como se va configurando el formulario usando Angular.

12.3 FORMULARIO BASADO EN PLANTILLAS (II)

```
<h1>Template Driven</h1>
<form>
  <label for="nombre">Nombre</label>
  <input type="text" id="nombre" name="nombre">
  <label for="email">E-Mail</label>
  <input type="text" id="email" name="email">
  <label for="password">Password</label>
  <input type="password" id="password" name="password">
  <button type="submit" >Enviar</button>
</form>
```

12.4 FORMULARIO BASADO EN PLANTILLAS (III)

- Para empezar hay que indicarle que cuando detecte el evento de submit con (ngSubmit) ejecute la función correspondiente implementada en el archivo template-driven.component.ts.
- A dicha función habrá que pasarle el formulario para poder acceder a los datos que se van a llenar, por lo que hay que crear una referencia de todo el formulario para lo que usamos la directiva ngForm.

```
<h1>Template Driven</h1>
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
<!-- ... -->
</form>
```

12.5 FORMULARIO BASADO EN PLANTILLAS (IV)

- La directiva `ngForm` hay que importarla para poder usarla.

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

@Component({
  selector: 'app-template-driven',
  templateUrl: './template-driven.component.html',
  styles: []
})
export class TemplateDrivenComponent {
  onSubmit(form: NgForm) {
    // ...
  }
}
```

12.6 FORMULARIO BASADO EN PLANTILLAS (V)

- Se puede realizar algunas validaciones usando las que nos proporciona HTML, como **required** y **pattern**.

```
<h1>Template Driven</h1>
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
  <label for="nombre">Nombre</label>
  <input type="text" id="nombre" name="nombre" required>
  <label for="email">E-Mail</label>
  <input type="text" id="email" name="email" required>
  <label for="password">Password</label>
  <input type="password" id="password" name="password"
    pattern="RegExp" required>
  <button type="submit" >Enviar</button>
</form>
```

12.7 FORMULARIO BASADO EN PLANTILLAS (VI)

- Al inspeccionar cualquier input del form, se puede ver que tiene algunas clases asignadas que no le hemos puesto, sino que se las ha puesto- Angular.
- Estas clases se pueden aprovechar para poner estilos según el estado del input.

12.8 FORMULARIO BASADO EN PLANTILLAS (VII)

- **ng-valid**: El campo es válido.
- **ng-invalid**: El campo no es válido.
- **ng-touched**: Se ha clicado en el campo.
- **ng-untouched**: No se ha clicado en el campo.
- **ng-pristine**: Tiene el valor por defecto.
- **ng-dirty**: No tiene el valor por defecto.

12.9 FORMULARIO BASADO EN PLANTILLAS (VII)

```
@Component({
  selector: 'app-template-driven',
  templateUrl: './template-driven.component.html',
  styles: [`.form-control.ng-invalid {
    border: 1px solid red;
 }`]
})
```

12.10 FORMULARIO BASADO EN PLANTILLAS (VIII)

- Se puede poner valores por defecto en los campos del formulario, para ello en el typescript hay que crearse una estructura de datos con los valores por defecto.
- Y en la plantilla hay que usar [ngModel]="dato" para acceder a esos datos y asignarle el valor al campo.

12.11 FORMULARIO BASADO EN PLANTILLAS (IX)

```
export class TemplateDrivenComponent {

  usuario = {
    nombre: '',
    email: '',
    password: ''
  };

  onSubmit(form: NgForm) {
    // ...
  }
}
```

12.12 FORMULARIO BASADO EN PLANTILLAS (X)

```
<h1>Template Driven</h1>
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
  <label for="nombre">Nombre</label>
  <input type="text" id="nombre" name="nombre" required
    [ngModel]="usuario.nombre">
  <label for="email">E-Mail</label>
  <input type="text" id="email" name="email" required
    [ngModel]="usuario.email">
  <label for="password">Password</label>
  <input type="password" id="password" name="password"
    [ngModel]="usuario.password" pattern="RegExp" required>
  <button type="submit" >Enviar</button>
</form>
```

12.13 FORMULARIO BASADO EN PLANTILLAS (XI)

- Se puede usar el **two-way binding** con `ngModel` para actualizar los datos una vez que se realiza el submit.

```
<h1>Template Driven</h1>
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
  <label for="nombre">Nombre</label>
  <input type="text" id="nombre" name="nombre" required
    [(ngModel)]="usuario.nombre">
  <label for="email">E-Mail</label>
  <input type="text" id="email" name="email" required
    [(ngModel)]="usuario.email">
  <label for="password">Password</label>
  <input type="password" id="password" name="password"
    [(ngModel)]="usuario.password" pattern="RegExp" required>
  <button type="submit" >Enviar</button>
</form>
```

12.14 FORMULARIO BASADO EN PLANTILLAS (XII)

- Angular da la posibilidad de agrupar datos del mismo tipo usando la directiva **ngModelGroup** la cual tiene que ir dentro de un elemento hijo del elemento que tiene la directiva **ngForm** y tiene que envolver los campos que se desean agrupar.

12.15 FORMULARIO BASADO EN PLANTILLAS (XIII)

```
<h1>Template Driven</h1>
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
  <div ngModelGroup="datosUsuario">
    <label for="nombre">Nombre</label>
    <input type="text" id="nombre" name="nombre" required
      [(ngModel)]="usuario.nombre">
    <label for="email">E-Mail</label>
    <input type="text" id="email" name="email" required
      [(ngModel)]="usuario.email">
  </div>
  <label for="password">Password</label>
  <input type="password" id="password" name="password"
    [(ngModel)]="usuario.password" pattern="RegExp" required>
  <button type="submit" >Enviar</button>
</form>
```

12.16 FORMULARIO BASADO EN PLANTILLAS (XIV)

```
export class TemplateDrivenComponent {  
  usuario = {  
    datosUsuario: {  
      nombre: '',  
      email: ''  
    },  
    password: ''  
  };  
}
```

12.17 FORMULARIO BASADO EN PLANTILLAS (XV)

- Se puede usar la directiva *ngIf para mostrar mensajes cuando un campo es invalido, poniendo una referencia que se usa en la condición de la directiva.
- También se pueden usar property bindings para deshabilitar los botones de envio de los formularios.

12.18 FORMULARIO BASADO EN PLANTILLAS (XVI)

```
<h1>Template Driven</h1>
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
  <div ngModelGroup="datosUsuario">
    ...
    <label for="email">E-Mail</label>
    <input type="text" id="email" name="email" required
      [(ngModel)]="usuario.email">
    <div *ngIf="!email.valid">Invalid Email</div>
  </div>
  ...
  <button type="submit" [disabled]="!f.valid">Enviar</button>
</form>
```

12.19 FORMULARIO BASADO EN DATOS (I)

- Los formularios basados en datos o reactivos facilitan la gestión de los datos puesto que se controlan directamente desde la clase del componente.
- De esta forma el componente tiene acceso a los datos y a la estructura del formulario, por lo tanto el componente puede reaccionar a los cambios que observa.
- Las actualizaciones de datos y validaciones son mas rápidas.

12.20 FORMULARIO BASADO EN DATOS (II)

- Con este tipo de formularios hay que crear un árbol de objetos **FormControl** en la clase del componente y asignarlos a los campos correspondientes en la plantilla.
- Para empezar hay que inicializar en el constructor la propiedad que va a contener el formulario con los datos por defecto.
- Hay que importar **FormGroup** que agrupa los datos, en este caso todos los datos del formulario son una agrupación, y cada uno de los campos que hay en esta agrupación son **FormControl**.

12.21 FORMULARIO BASADO EN DATOS (III)

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

@Component({
  selector: 'app-data-driven',
  templateUrl: './data-driven.component.html'
})
export class DataDrivenComponent {
  miFormulario: FormGroup;
  constructor() {
    this.miFormulario = new FormGroup({
      'nombre': new FormControl(),
      'email': new FormControl(),
      'password': new FormControl()
    });
  }
}
```

12.22 FORMULARIO BASADO EN DATOS (IV)

- El nombre de las claves es importante porque es lo que usa Angular para detectar que campo asigna a cada clave del **FormGroup**.
- Al minificar los archivos se les puede cambiar el nombre y en caso de que no coincidan dejaría de funcionar la aplicación.

12.23 FORMULARIO BASADO EN DATOS (V)

- Para sincronizar la plantilla con los datos hay que indicar al formulario con un property binding donde se encuentran los datos.
- Y para indicar que dato corresponde a que campo del formulario se usa la directiva **formControlName**.

12.24 FORMULARIO BASADO EN DATOS (VI)

```
<h1>Data Driven</h1>
<form [formGroup]="miFormulario">
  <label for="nombre">Nombre</label>
  <input type="text" id="nombre" formControlName="nombre">
  <label for="email">E-Mail</label>
  <input type="text" id="email" formControlName="email">
  <label for="password">Password</label>
  <input type="password" id="password" formControlName="password">
  <button type="submit" >Enviar</button>
</form>
```

12.25 FORMULARIO BASADO EN DATOS (VII)

- Las directivas como el `formControlName` solo se pueden usar una vez que se ha importado el modulo que las contiene, **ReactiveFormsModule**.
- Este modulo se importa en el `app.module.ts`.

12.26 FORMULARIO BASADO EN DATOS (VIII)

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

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

12.27 FORMULARIO BASADO EN DATOS (IX)

- Para realizar el submit se usa el ngSubmit como en el template-driven approach, salvo que al tener los datos en el constructor del componente no es necesario crear la referencia al formulario porque ya se pueden acceder a ellos.

```
<h1>Data Driven</h1>
<form [formGroup]="miFormulario" (ngSubmit)="onSubmit()">
<!-- ... -->
</form>
```

12.28 FORMULARIO BASADO EN DATOS (X)

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

@Component({
  selector: 'app-data-driven',
  templateUrl: './data-driven.component.html'
})
export class DataDrivenComponent {
  miFormulario: FormGroup;

  constructor() {
    this.miFormulario = new FormGroup({
      'nombre': new FormControl(),
      'email': new FormControl(),
      'password': new FormControl()
    });
  }
}
```

12.29 FORMULARIO BASADO EN DATOS (XI)

- Las validaciones también son distintas con este método, puesto que el formulario se configura desde el constructor del componente en lugar de hacerlo en la plantilla.
- A los FormControl se le pueden pasar parámetros, donde el primero es el valor por defecto que se le quiera poner y el segundo parámetro son las validaciones para ese campo.
- Las validaciones usan **Validators** que se importa desde **@angular/forms**.

12.30 FORMULARIO BASADO EN DATOS (XII)

```
constructor() {
    this.miFormulario = new FormGroup({
        'nombre': new FormControl('', Validators.required),
        'email': new FormControl('', [Validators.required, Validators.pattern("Re
        'password': new FormControl('', Validators.required)
    });
}
```

12.31 FORMULARIO BASADO EN DATOS (XII)

- En caso de que se necesiten mas de una validación, el segundo parámetro será un array de Validators.
- Podemos encontrar mas sobre validators aquí.
- Ahora se puede deshabilitar el botón de Enviar y mostrar el mensaje de error cuando falla la validación del email de una forma mas sencilla porque no se necesita la referencia al formulario porque se puede acceder a la propiedad del componente.

12.32 FORMULARIO BASADO EN DATOS (XIV)

```
<h1>Data Driven</h1>
<form [formGroup]="miFormulario">

  ...
  <label for="email">E-Mail</label>
  <input type="text" id="email" formControlName="email">
  <div *ngIf="!miFormulario.controls['email'].valid">
    Email invalido</div>
  </div>
  ...
  <button type="submit" [disabled]="!miFormulario.valid">Enviar</button>
</form>
```

12.33 FORMULARIO BASADO EN DATOS (XV)

- Para agrupar los datos hay que crear un nuevo FormGroup que los contenga e indicarlo en la plantilla usando la directiva **formGroupName="nombreGrupoDatos"** en un elemento que envuelva esos datos.

```
constructor() {
  this.miFormulario = new FormGroup({
    'datosUsuario': new FormGroup({
      'nombre': new FormControl('', Validators.required),
      'email': new FormControl('', [Validators.required, Validators.pattern])
    }),
    'password': new FormControl('', Validators.required)
  });
}
```

12.34 FORMULARIO BASADO EN DATOS (XVI)

```
<h1>Data Driven</h1>
<form [formGroup]="miFormulario">
  <div formGroupName="datosUsuario">
    <label for="nombre">Nombre</label>
    <input type="text" id="nombre" formControlName="nombre">
    <label for="email">E-Mail</label>
    <input type="text" id="email" formControlName="email">
  </div>
  <label for="password">Password</label>
  <input type="password" id="password" formControlName="password">
  <button type="submit" >Enviar</button>
</form>
```

12.35 FORMULARIO BASADO EN DATOS (XVII)

- No hay que olvidar cambiar la expresión en la directiva estructural que muestra el mensaje de correo invalido, porque ahora hay que añadir el nuevo nivel (`.controls['datosUsuario']`) para acceder al email.

```
<div *ngIf="!miFormulario.controls['datosUsuario'].controls['email'].valid">
  Email invalido</div>
```

12.36 FORMULARIO BASADO EN DATOS (IXX)

- Para gestionar los arrays se usa el FormArray, el cual representa un array de elementos del formulario.
- En la plantilla hay que asignar la directiva `formArrayName="arrayDeDatos"` a un elemento que envuelva los datos.
- Al ser un array hay que indicar en el `formControlName` el indice del array en lugar de la clave usada.

12.37 FORMULARIO BASADO EN DATOS (XX)

```
<form [FormGroup]="miFormulario">
...
<div formArrayName="hobbies">
  <h3>Hobbies</h3>
  <div class="form-group"
    *ngFor="let hobby of miFormulario.controls['hobbies'].controls;
    let i = index">
    <input type="text" formControlName="{{i}}">
  </div>
  <button type="button" (click)="onAddHobby()">Añadir Hobby</button>
</div>
...
```

12.38 FORMULARIO BASADO EN DATOS (XXI)

```
@Component({
  selector: 'app-data-driven',
  templateUrl: './data-driven.component.html'
})
export class DataDrivenComponent {
  miFormulario: FormGroup;
  constructor() {
    this.miFormulario = new FormGroup({
      'datosUsuario': new FormGroup({
        'nombre': new FormControl('', Validators.required),
        'email': new FormControl('', [Validators.required,
          Validators.pattern(RegExp)])
      }),
      'password': new FormControl('', Validators.required),
      'hobbies': new FormArray([
        new FormControl('Cocinar', Validators.required)
      ])
    });
  }
}
```

12.39 FORMULARIO BASADO EN DATOS (XXII)

- Hay que realizar un cast (`<FormArray>variable`) para que nos permita usar el push del FormArray.
- Y para evitar que no se rellene el hobby, se le añade la validación `required` y se le pone como valor por defecto un "".

12.40 FORMULARIO BASADO EN DATOS (XXIII)

- Hay otra forma de crear los formularios, y es usando el **FormBuilder**, el cual se tiene que injectar en el constructor para poder usarlo.
- De esta forma no hace falta crear los elementos del formulario.

12.41 FORMULARIO BASADO EN DATOS (XXIX)

- Hay que sustituir:
- los `new FormGroup()` por `formBuilder.group()`
- los `new FormArray()` por `formBuilder.array()`
- los `new FormControl()` por `[]`

12.42 FORMULARIO BASADO EN DATOS (XXX)

```
@Component({
  selector: 'app-data-driven',
  templateUrl: './data-driven.component.html'
})
export class DataDrivenComponent {
  miFormulario: FormGroup;
  constructor(private formBuilder: FormBuilder) {
    this.miFormulario = formBuilder.group({
      'datosUsuario': formBuilder.group({
        'nombre': ['', Validators.required],
        'email': ['', [Validators.required, Validators.pattern(RegExp)]]
      }),
      'password': ['', Validators.required],
      'hobbies': formBuilder.array([
        ['Cocinar', Validators.required]
      ])
    });
  }
}
```

12.43 FORMULARIO BASADO EN DATOS (XXXI)

- La aplicación debería de seguir funcionando de esta forma.
- Para resetear un formulario basta con usar `miFormulario.reset()`.

13 ROUTING

13.1 INTRODUCCIÓN (I)

- Las rutas permiten cambiar entre las distintas secciones de la aplicación, y dependiendo de la ruta se cargan unos componentes u otros.
- Angular Router parsea la URL e intenta identificar el componente que tiene que cargar teniendo en cuenta los datos que van en la ruta en caso de que los tenga.
- Para poder usar las rutas hay que configurar el proyecto para poder cambiar de sección desde cualquier parte de la aplicación.

13.2 CONFIGURACIÓN (I)

- Para empezar hay que crear un archivo de rutas `app.routing.ts` donde se van a configurar las rutas.
- Hay que importar `Routes` desde `@angular/router`.

13.3 CONFIGURACIÓN (II)

- Las rutas se guardan en un array (de tipo Routes) de objetos javascript donde cada objeto representa una ruta.
- Cada objeto va a contener como mínimo:
 - **path**: forma de la URL. En el caso de la ruta **home** se deja como un string vacío.
 - **component**: el componente que se va a mostrar cuando estemos en esa URL. Los componentes hay que importarlos.

13.4 CONFIGURACIÓN (III)

- Para que las rutas estén disponibles en toda la aplicación hay que exportarlas para ponerlas en el archivo `app.modules.ts`.
- Para el ejemplo siguiente, necesitamos crear un componente `home` en la carpeta `app`, un componente `usuario` y dentro de la carpeta `usuario`, dos componentes mas, `usuario-info` y `editar-usuario`.

13.5 CONFIGURACIÓN (IV)

```
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';
import { UsuarioComponent } from './usuario/usuario.component';

const APP\_ROUTES: Routes = [
  {path: 'usuario', component: UsuarioComponent},
  {path: '', component: HomeComponent}
];

export const routing = RouterModule.forRoot(APP\_ROUTES);
```

13.6 CONFIGURACIÓN (V)

- Hay que importar las rutas en el `app.module.ts`:

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

import { AppComponent } from './app.component';
import { UsuarioComponent } from './usuario/usuario.component';
import { EditarUsuarioComponent } from './usuario/editar-usuario.component';
import { UsuarioInfoComponent } from './usuario/usuario-info.component';
import { HomeComponent } from './home.component';
import { routing } from './app.routing';

@NgModule({
  declarations: [
    AppComponent,
    UsuarioComponent,
    EditarUsuarioComponent,
    UsuarioInfoComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    routing
  ],
  providers: []
})
```

13.7 CONFIGURACIÓN (VI)

- Ahora mismo debería de fallar porque faltaría decirle donde se tiene que renderizar el componente.
- Para ello hay que usar la etiqueta `<router-outlet></router-outlet>` en el sitio donde se quiere mostrar el componente correspondiente a la ruta.

13.8 CONFIGURACIÓN (VII)

```
<div class="container">
  <div class="row">
    <div class="col-md-10">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

13.9 CONFIGURACIÓN (VIII)

- Ya no tiene que fallar, sino que debería de mostrar el componente.
- Añadimos unos links para poder navegar entre secciones.

13.10 ENLACES (I)

```
<div class="container">
  <div class="row">
    <div class="col-md-10">
      <h1>Hello World!</h1>
      <hr>
      <a href="/">Home</a> |
      <a href="/usuario">Usuario</a>
      <hr>
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

13.11 ENLACES (II)

- Al hacer click sobre cualquiera de los dos enlaces cambia de componente, pero también recarga toda la página, y eso es algo feo y poco usable.
- En lugar de recargar la página debería cambiar de componente.
- Por supuesto, Angular es capaz de hacerlo. En lugar de href hay que usar la directiva routerLink.

13.12 ENLACES (III)

```
<div class="container">
  <div class="row">
    <div class="col-md-10">
      <h1>Hello World!</h1>
      <hr>
      <a [routerLink]="['']">Home</a> |
      <a [routerLink]="['usuario']">Usuario</a>
      <hr>
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

13.13 ENLACES (IV)

- Entre los corchetes se encuentran los segmentos de la URL, donde cada segmento es aquello que va despues de cada /.
- Por ejemplo en una ruta como `localhost:4200/usuario/10` habría dos segmentos, ["usuario", "10"].
- Ahora al navegar por los links debería de cambiar el componente sin recargar la página.

13.14 ENLACES (V)

- Se pueden usar rutas absolutas (con /) o rutas relativas (sin /).

13.15 ENLACES (VI)

- Si estamos en la ruta de inicio (`localhost:4200/home`) y vamos a la ruta del usuario:
 - `/usuario` va a `localhost:4200/usuario`.
 - `usuario` va a `localhost:4200/usuario`.

13.16 ENLACES (VII)

- Pero si estamos en la ruta `localhost:4200/usuario`:
 - `/usuario` va a `localhost:4200/usuario`, es decir, que se queda donde está.
 - `usuario` va a `localhost:4200/usuario/usuario`.

13.17 ENLACES (VIII)

- Cuando ponemos en el segmento / indica que la ruta comienza después del dominio.

13.18 NAVEGACIÓN POR CÓDIGO (I)

- Hay algunos casos que se necesita navegar a otra ruta sin pinchar en un link, sino que se necesita hacer la navegación a través del código, por ejemplo cuando te registras en una página te redirige a la ruta de inicio.
- Para ello hay que usar el método `navigate` del router.
- Este método tiene como argumento un array de segmentos, por lo que actúa de la misma forma que el `routerLink`.

13.19 NAVEGACIÓN POR CÓDIGO (II)

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

@Component({
  selector: 'app-usuario',
  template: `<h3>Usuario</h3>
    <button (click)="onNavigate()">Ir a inicio</button>
    <hr>{{id}}<hr>`,
  styles: []
})
export class UsuarioComponent {
  constructor(private router: Router) { }
  onNavigate() {
    this.router.navigate(['/']);
  }
}
```

13.20 PARÁMETROS (I)

- Las rutas se pueden personalizar un poco mas. De momento al ir a la página de /usuario nos muestra los datos de un solo usuario, pero en todas las aplicaciones hay mas de un usuario por lo que tenemos que indicar en la URL la página del usuario que queremos ver.
- Esto se indica con los parámetros de las rutas que pueden contener un valor. Estos párametros se definen de la siguiente forma { path: 'usuario/:id', component: UsuarioComponent }.

13.21 PARÁMETROS (II)

```
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';
import { UsuarioComponent } from './usuario/usuario.component';

const APP\_ROUTES: Routes = [
  {path: 'usuario/:id', component: UsuarioComponent},
  {path: '', component: HomeComponent}
];

export const routing = RouterModule.forRoot(APP\_ROUTES);
```

13.22 PARÁMETROS (III)

- Y ahora en los links donde se llama al usuario hay que añadir un segundo segmento que es el id de ese usuario.

13.23 PARÁMETROS (IV)

```
<div class="container">
  <div class="row">
    <div class="col-md-10">
      <h1>Hello World!</h1>
      <hr>
      <a [routerLink]="[ '' ]">Home</a> |
      <input type="text" #id (input)="0" />
      <a [routerLink]="['usuario', id.value]">Usuario</a>
      <hr>
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

13.24 PARÁMETROS (V)

- Para obtener los parámetros de una URL, hay que importar `ActivatedRoute` de `@angular/router` e injectarlo en el constructor.
- Y podemos acceder a ellos como viene a continuación:

13.25 PARÁMETROS (VI)

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

@Component({
  selector: 'app-usuario',
  template: `<h3>Usuario</h3>
    <button (click)="onNavigate()">Ir a inicio</button>
    <hr>{{id}}<hr>`,
  styles: []
})
export class UsuarioComponent {
  id: string;
  constructor(private router: Router, private activatedRoute: ActivatedRoute) {
    this.id = activatedRoute.snapshot.params['id'];
  }
  onNavigate() {
    this.router.navigate(['/']);
  }
}
```

13.26 PARÁMETROS (VII)

- El método `snapshot.params['id']` nos da un objeto de clave-valor en el que el `id` es una clave y lo usamos para obtener su valor.
- Pero si le cambiamos el `id` en el `input` para que nos mande al componente correspondiente con ese `id` se puede observar que en la URL se ha cambiado el `id`, pero el componente no se ha cargado.
- Angular es muy eficiente y si detecta que queremos ir al mismo componente, no lo destruye para volverlo a crear, sino que realiza los cambios que detecta.

13.27 PARÁMETROS (VIII)

- Pero como no esta creando el componente otra vez, no detecta el cambio del id y no lanza el constructor otra vez para actualizar el componente.
- La alternativa que hay que usar a snapshot es `params`, que es un observable.
- Un observable envuelve un objeto y escucha los cambios que se producen en dicho objeto para poder reaccionar a ellos.

13.28 PARÁMETROS (IX)

```
import { Component, OnDestroy } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';
import { Subscription } from 'rxjs/Rx';

@Component({
  selector: 'app-usuario',
  template: `<h3>Usuario</h3>
    <button (click)="onNavigate()">Ir a inicio</button>
    <hr>{{id}}<hr>`,
  styles: []
})
export class UsuarioComponent implements OnDestroy {
  private subscription: Subscription;
  id: string;
  constructor(private router: Router, private activatedRoute: ActivatedRoute) {
    this.subscription = activatedRoute.params.subscribe(
      (param: any) => this.id = param['id']
    );
  }
  ngOnDestroy() {
    this.subscription.unsubscribe();
  }
}
```

13.29 PARÁMETROS (X)

- Ahora se subscribe a cualquier cambio en los parámetros, y en el momento en que se produzca, ejecuta el callback pasandole como parámetro el param del que se puede obtener el valor del id.
- La subscripcion la guardamos en una variable de tipo `Subscription` que se importa desde `rxjs/Rx` y de la que vamos a desuscribirnos una vez que se elimine el componente para evitar perdidas de memoria.

13.30 RUTAS HIJAS (I)

- Las rutas hijas son aquellas que dependen de los parámetros de una ruta, hasta ahora teníamos la ruta `usuario/:id`, pero ahora queremos una ruta que permita editar ese usuario por lo que necesitaremos una ruta del estilo `usuario/:id/editar`.
- Esta es una child route porque cuelga de la ruta del usuario y depende del id.

13.31 RUTAS HIJAS (II)

- Para crear estas rutas hay que crearse otro archivo de rutas en este caso en la carpeta de **usuario**.
- Y las rutas definidas en este archivo hay que exportarlas para importarlas y usarlas en el archivo de rutas principal.

13.32 RUTAS HIJAS (III)

```
import { Routes } from '@angular/router';
import { UsuarioInfoComponent } from './usuario-info.component';
import { EditarUsuarioComponent } from './editar-usuario.component';

export const USUARIO_ROUTES: Routes = [
  { path: 'info', component: UsuarioInfoComponent },
  { path: 'editar', component: EditarUsuarioComponent }
];
```

13.33 RUTAS HIJAS (IV)

- En el archivo `app.routing.ts` además de importar las rutas, hay que asignarselas a la ruta de la que van a depender.

13.34 RUTAS HIJAS (V)

```
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';
import { UsuarioComponent } from './usuario/usuario.component';
import { USUARIO\_ROUTES } from './usuario/usuario.routing';

const APP\_ROUTES: Routes = [
  {path: 'usuario/:id', component: UsuarioComponent, children: USUARIO\_ROUTI
    {path: '', component: HomeComponent}
];

export const routing = RouterModule.forRoot(APP\_ROUTES);
```

13.35 RUTAS HIJAS (VI)

- Para que funcione, hay que indicar donde se quiere renderizar el componente hijo con la etiqueta **router-outlet**. Y también tendremos que añadir los enlaces a los componente hijos.

13.36 RUTAS HIJAS (VII)

```
import { Component, OnDestroy } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';
import { Subscription } from 'rxjs/Rx';

@Component({
  selector: 'app-usuario',
  template: `<h3>Usuario</h3>
    <button (click)="onNavigate()">Ir a inicio</button>
    <hr>{{id}}<hr>
    <a [routerLink]=["'editar'"]>Editar usuario</a>
    <a [routerLink]=["'info'"]>Info de usuario</a>
    <router-outlet></router-outlet>`,
  styles: []
})
export class UsuarioComponent implements OnDestroy {
  private subscription: Subscription;
  id: string;
  constructor(private router: Router, private activatedRoute: ActivatedRoute)
```

13.37 RUTAS HIJAS (VIII)

- En el caso de que no se inserte un id, la URL no sería correcta, porque siempre tiene que recibir un id. En este caso, cuando no lo reciba hay que redireccionarla a una URL válida.

13.38 RUTAS HIJAS (IX)

```
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home.component';
import { UsuarioComponent } from './usuario/usuario.component';
import { USUARIO_ROUTES } from './usuario/usuario.routing';

const APP_ROUTES: Routes = [
  {path: 'usuario/', redirectTo: 'usuario/1'},
  {path: 'usuario/:id', component: UsuarioComponent, children: USUARIO_ROUTES},
  {path: '', component: HomeComponent}
];

export const routing = RouterModule.forRoot(APP_ROUTES);
```

13.39 GUARDS

- Las guards se usan para controlar a que partes de la aplicación puede navegar un usuario.
- Por ejemplo, es posible que algunas de las rutas de la aplicación las queramos restringir solo a usuarios logueados, y podemos usar estas guards para controlar las condiciones en las que pueden o no entrar o salir de una ruta.

13.40 CANACTIVATE (I)

- Este guard se ejecuta al entrar en una ruta.
- El guard que se ha creado implementa la interface CanActivate la cual tiene dos parámetros, donde el primero es la ruta que está activada, y el segundo es el estado de esa ruta.

13.41 CANACTIVATE (II)

- Este método devolverá un Observable, una Promise o un boolean.
- Si el valor devuelto es **true** permite pasar a la ruta, mientras que si es **false** no.
- El observable se usa cuando el valor depende de que algo se ejecute asíncronamente y tengamos que esperar a la respuesta.

13.42 CANACTIVATE (III)

- usuario-info.guard.ts.

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

@Injectable()
export class UsuarioInfoGuard implements CanActivate {
  canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    return confirm('Estas seguro??');
  }
}
```

13.43 CANACTIVATE (IV)

- Para asignar el **guard** a una ruta, hay que poner en la ruta la clave **canActivate** cuyo valor será un array con todas las guards que tiene que pasar esa ruta.

13.44 CANACTIVATE (V)

- `usuario.routing.ts`.

```
import { Routes } from '@angular/router';
import { UsuarioInfoComponent } from './usuario-info.component';
import { EditarUsuarioComponent } from './editar-usuario.component';
import { UsuarioInfoGuard } from './usuario-info.guard';

export const USUARIO_ROUTES: Routes = [
  { path: 'info', component: UsuarioInfoComponent, canActivate: [UsuarioInfoGuard] },
  { path: 'editar', component: EditarUsuarioComponent }
];
```

13.45 CANACTIVATE (VI)

- Ahora solo faltaría añadir el guard en el array de providers del app.module.ts para que funcione.

13.46 CANACTIVATE (VII)

- app.module.ts.

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

import { AppComponent } from './app.component';
import { UsuarioComponent } from './usuario/usuario.component';
import { EditarUsuarioComponent } from './usuario/editar-usuario.component';
import { UsuarioInfoComponent } from './usuario/usuario-info.component';
import { HomeComponent } from './home.component';
import { routing } from './app.routing';
import { UsuarioInfoGuard } from './usuario/usuario-info.guard';

@NgModule({
  declarations: [
    AppComponent,
    UsuarioComponent,
    EditarUsuarioComponent
```

13.47 CANDEACTIVATE (I)

- Este guard se lanza cuando te vas a salir de una cierta ruta.
- Es parecido al anterior, pero hay que tener en cuenta algunas cosas.
- Para empezar al crear el guard, hay que importar el CanDeactivate e implementarlo en la clase.

13.48 CANDEACTIVATE (II)

- Este CanDeactivate tiene como parámetro un interface que se va a implementar en los componentes que necesiten usar esta guard.
- Hay que definir el interface en este mismo archivo, el cual no tiene ningún parámetro y devuelve al igual que el guard, un Observable o un boolean.

13.49 CANDEACTIVATE (III)

- editar-usuario.guard.ts.

```
import { CanDeactivate } from '@angular/router';
import { Observable } from 'rxjs/Observable';

export interface ComponentCanDeactivate {
  canDeactivate: () => Observable<boolean> | boolean;
}

export class EditarUsuarioGuard implements CanDeactivate<ComponentCanDeactivate> {
  canDeactivate(component: ComponentCanDeactivate): Observable<boolean> | boolean {
    return component.canDeactivate ? component.canDeactivate() : true;
  }
}
```

13.50 CANDEACTIVATE (IV)

- En el canDeactivate de la clase del guard, se le indica que si el componente tiene implementado el interface de arriba nos deje salir según el valor que devuelva ese método, y si no lo tiene implementado entonces permite salir siempre.
- Una vez definido el interface, hay que implementarlo en el componente.

13.51 CANDEACTIVATE (VI)

- editar-usuario.component.ts.

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { Observable } from 'rxjs/Rx';
import { ComponentCanDeactivate } from './editar-usuario.guard';

@Component({
  selector: 'app-editar-usuario',
  template: `<h3>Editar Usuario</h3>
    <button (click)="hecho=true">Hecho</button>
    <button (click)="onNavigate()">Ir a inicio</button>`,
  styles: []
})
export class EditarUsuarioComponent {
  hecho: boolean = false;
  constructor(private router: Router) { }
  onNavigate() {
    this.router.navigate(['/']);
  }
}
```

13.52 CANDEACTIVATE (VII)

- No hay que olvidar añadir el guard en el array de providers del archivo app.module.ts.

13.53 CANDEACTIVATE (VIII)

- app.module.ts.

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

import { AppComponent } from './app.component';
import { UsuarioComponent } from './usuario/usuario.component';
import { EditarUsuarioComponent } from './usuario/editar-usuario.component';
import { UsuarioInfoComponent } from './usuario/usuario-info.component';
import { HomeComponent } from './home.component';
import { routing } from './app.routing';
import { UsuarioInfoGuard } from './usuario/usuario-info.guard';
import { EditarUsuarioGuard } from './usuario/editar-usuario.guard';

@NgModule({
  declarations: [
    AppComponent,
    UsuarioComponent
```

13.54 CANDEACTIVATE (IX)

- Y por último se añade a la ruta correspondiente con la clave canDeactivate que tiene por valor un array de guards en el que hay que incluir esta que se ha creado.

13.55 CANDEACTIVATE (X)

- `usuario.routing.ts`.

```
import { Routes } from '@angular/router';
import { UsuarioInfoComponent } from './usuario-info.component';
import { EditarUsuarioComponent } from './editar-usuario.component';
import { UsuarioInfoGuard } from './usuario-info.guard';
import { EditarUsuarioGuard } from './editar-usuario.guard';

export const USUARIO_ROUTES: Routes = [
  { path: 'info', component: UsuarioInfoComponent, canActivate: [UsuarioInfoGuard] },
  { path: 'editar', component: EditarUsuarioComponent, canDeactivate: [EditarUsuarioGuard] }
];
```