



ANGULAR 4 desde Cero

La guía práctica y completa para convertirte en
desarrollador Angular *full stack*

Índice

1 TypeScript y Web Components	5
1.1 TypeScript	5
1.2 Instalación de TypeScript	5
1.3 Tipos de datos en TypeScript	6
1.3 Variables en TypeScript.	11
1.5 Clases en TypeScript.	13
1.6 Constructor en TypeScript	15
1.7 Interfaces en TypeScript	18
1.8 Web Components	20
2. Gestión de la configuración: Angular-Cli	21
2.1 Instalación	21
2.2 Requisitos Previos	21
2.3 Instalación de Angular CLI.	22
2.4 Creación de un Proyecto en Angular	22
2.5 Arranque de la Aplicación.	24
2.6 El Archivo index.html	24
2.7 El Archivo styles.css	26
2.8 El Archivo favicon.png	26
2.9 El Directorio Assets	27
3 Módulos y Componentes.	27
3.1 Módulos en Angular.	27
3.2 Componentes en Angular.	30
3.3 Creación de un nuevo componente	36
3.4 Creación de componente con Angular CLI.	40
3.5 Anidado de Componentes	42
4 <i>Data Binding</i>	45
4.1 Interpolación.	46
4.2 Property Binding.	50
4.3 <i>Event Binding</i> .	53
4.4 <i>Two Way Binding</i>	54
5 Directivas.	57
5.1 Directiva ngIf	58
5.2 Directiva ngStyle	62
5.3 Directiva ngClass.	64
5.4 Directiva ngFor	67
5.5 Directiva NgSwitch	71
5.6 Creación de directivas propias.	75

5.7 Utilización de @Input entre componentes	79
5.8 Utilización de @Output entre componentes	84
6 Pipes	89
6.1 Pipe Date	89
6.2 Pipe Uppercase y Lowercase	91
6.3 Pipe Decimal	92
6.4 Pipe Currency	93
6.5 Pipe i18nSelect	94
6.6 Creación de pipes	95
7 Servicios e Inyección de dependencias.	98
7.1 Creación de servicios e inyección en componentes	99
8 Routing	106
8.1 Configuración del routing de una aplicación.	106
8.2 Navegación mediante links	109
9 Formularios	112
9.1 Creación de formularios Template-driven.	112
9.2 Carga de datos en campos select.	120
9.3 Validación de campos mediante HTML	122
9.4 Creación de formularios Reactive.	128
9.5 Validación de campos programática.	136
9.6 valueChanges.	138
10 Conexión con el servidor.	142
10.1. Servicio HTTP: Post.	144
10.2. Servicio HTTP: Get	148
10.3. Servicio HTTP: Delete	164
10.4. Servicio HTTP: Put	153
11 Gestión de Autorizaciones	168
11.1 Registro	168
11.2 Inicio de sesión	179
11.3 Protección de rutas	189
11.4 Link a registro e inicio de sesión.	191
12 Módulos adicionales.	192
12.1 Creación de módulos	192
12.2 Componentes de módulos adicionales.	193
12.3 Componentes CRUD para el módulo adicional.	197
13 Angular <i>Tips & Tricks</i>	198
13.2 Spinner en carga de elementos HTML.	201
13.3 Elementos de respuesta de autenticación.	203
13.4 Consultas a la base de datos y buscador.	205

13.5 Upload de archivos	212
13.5.1 Configuración de Firebase	212
13.5.2 Componente Upload	214
13.5.3 Componente para Archivos	221
13.5.4 Borrado de Archivos.	225
13.6 Menú en pantalla de inicio	227
14 Despliegue a producción	232
14.1 Build	232
14.2 Despliegue a Amazon Web Services S3 Bucket.	233
Anexo Extra I Instalación de Node y NPM	237
Anexo Extra II de Visual Studio Code	237
Anexo Extra III Ciclo de Vida Angular	237

Introducción

¡Hola! Mi nombre es Pedro y he escrito este libro para que aprendas de manera práctica el increíble *framework* Angular, que nos permite crear aplicaciones modernas en la nube, con todos los requisitos que demandan las empresas más exigentes.

No necesitas más que unos ligeros conocimientos de HTML, JavaScript, CSS y un equipo conectado a internet para convertirte en un desarrollador *full stack* con Angular. Si encuentras algún contenido que no conozcas, mándame un email y te remito los mejores recursos gratuitos para aprenderlos.

A lo largo de todo el libro aprenderemos las técnicas Angular para crear los diferentes componentes de un proyecto y para ponerlo en práctica, vamos a crear una aplicación partiendo desde cero para llegar a su despliegue en la nube.

Podrás descargar el código de la aplicación en cada fase del aprendizaje para comparar con el que vayas desarrollando y además, puedes realizarme cuantas consultas del libro necesites en mi mail pjimenez@sapienslearning.com.

¡Mil gracias y bienvenido!

1 TypeScript y Web Components

1.1 TypeScript

Antes de comenzar a aprender Angular, vamos a conocer una serie de pinceladas del lenguaje en el que esta basado, TypeScript.

TypeScript es un lenguaje de programación de distribución y código abierto desarrollado por Microsoft. Su propio fabricante lo define como un *superset* (superconjunto) de JavaScript que, en esencia, mejora JavaScript añadiéndole funciones de tipado y de programación orientada a objetos.

Además, la sintaxis y desarrollo en TypeScript incorpora las mejoras y características de ECMAScript 6, estandarización de JavaScript.

Debido a estas nuevas especificaciones, el *framework* Angular utiliza TypeScript como lenguaje de programación para la lógica de las aplicaciones. Como las características de ECMAScript 6 aún no son soportadas por los navegadores web, el propio *framework* Angular 'transpila' el código TypeScript directamente a JavaScript para que pueda ser interpretado en los clientes web.

1.2 Instalación de TypeScript

Antes de instalar TypeScript y debido a que también es necesario para Angular, debemos instalar Node Js y NPM. Al final del libro dispones de un anexo que explica los pasos para su instalación.

Para instalar TypeScript, podemos emplear el gestor de paquetes NPM. De esta manera, en la consola del equipo completamos:

```
npm install -g typescript
```

Para comprobar la correcta instalación o la versión de TypeScript en el caso de que ya estuviera instalado, completamos en la consola:

```
tsc -v
```

Que nos devolverá la versión instalada y así tendremos nuestro equipo listo para trabajar.

1.3 Tipos de datos en TypeScript

Ya hemos dicho que una de las principales características de TypeScript es que proporciona tipos de datos, siendo la sintaxis de declaración, tipo e inicialización de variables globales, la siguiente:

```
let nombrevariable: tipodedato = valor;
```

De esta manera TypeScript soporta los siguientes tipos básicos de datos:

- Strings

```
let texto: String = 'Cadena de caracteres';
```

Que además, permite multilínea con comillas *backsticks*:

```
let textomulti: String = ` ACME S.A.  
                           Paseo de la Castellana, 100  
                           28.010 Madrid ` ;
```

- Números

```
let decimal: number = 6;  
let hexadecimal: number = 0xf00d;  
let binario: number = 0b1010;  
let octal: number = 0o744;
```

- Arrays.

Que se declaran simplemente con corchetes y el tipo de dato de sus elementos mediante:

```
let lista: number[] = [2,4,8];
```

o bien, de manera más expresiva mediante:

```
let lista: Array<number> = [1, 2, 3];
```

- Any

Permite que la variable tome cualquier tipo de dato:

```
let cualquiera: any = 12;
```

```
let cualquiera = "¡Hola Mundo!";
```

- Booleanos.

Obliga a seleccionar los valores booleanos true o false:

```
let booleano: true = 12;
```

Para más información sobre otros tipos de datos y su empleo, podemos consultar la documentación de TypeScript en:

<https://www.typescriptlang.org/docs/handbook/basic-types.html>.

Vamos a practicar un poco de TypeScript en nuestro editor.

Si quieres utilizar Visual Studio Code, mi editor favorito, al final del libro aprendemos a instalarlo y configurarlo.

En primer lugar, para comprobar los tipos de datos en TypeScript creamos un directorio de nombre `typetest`, por ejemplo, en nuestro equipo. A continuación, vamos a crear dos archivos de ejemplo. El primero, un clásico archivo HTML denominado `index.html` con el siguiente código:

index.html

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <title></title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1"
  </head>
```

```
<body>
  <h1 id="encabezado"></h1> ①
  <script type="text/javascript" src="test.js"></script> ②
</body>
</html>
```

En el cual:

- ① Hemos introducido un simple h1 con el id "encabezado".
- ② Y referenciamos en una etiqueta script a un archivo JavaScript test.js.

Creamos un segundo archivo TypeScript denominado test.ts, con en el que comenzamos escribiendo el siguiente código:

test.ts

```
let nombre: string = "Carlos"; ①

function saludo(nombre){
  return "Hola "+nombre; ②
}
document.getElementById("encabezado").innerHTML = saludo(nombre); ③
```

En el cual:

- ① Declaramos la variable nombre y la inicializamos con el valor "Carlos".
- ② Creamos una función saludo con el parámetro nombre que devuelve un "Hola" más el nombre.
- ③ Y finalmente la inyectamos en el elemento "encabezado" del html con el método JavaScript.

El fichero TypeScript como tal no lo podemos emplear en el html, de hecho en el script hemos indicado un archivo js, ya que los navegadores no lo reconocerían. Por tanto, lo tenemos que 'transpilar' a JavaScript.

Para ello, en la consola de nuestro equipo, dentro de la carpeta `typetest` tecleamos:

```
tsc test.ts
```

Y comprobamos en el editor como se ha creado un archivo JavaScript, con algunos cambios respecto al archivo original, por ejemplo, observamos como la palabra reservada `let` se ha convertido a `var`.

Ahora, cuando carguemos nuestro archivo `index.html` en el navegador verificamos como imprime por pantalla el valor correctamente.

Hola Carlos

Vamos a continuación a realizar una serie de cambios, y para no tener que 'transpilar' manualmente. Añadimos, en la consola del equipo y en el directorio que hemos creado para el ejemplo, `typetest`, el siguiente comando:

```
tsc -w test.ts
```

Podemos cambiar el valor de la variable `nombre` para comprobar como 'transpila' automáticamente.

¡Continuamos! Vamos a modificar ahora el código de `test.ts` de la siguiente manera:

`test.ts`

```
function imprPantalla(a,b){  
    return a+b;  
}  
  
let a: number = 10;  
  
let b: number = 5;  
  
document.getElementById("encabezado").innerHTML = imprPantalla(a,b);
```

Lógicamente este código imprimirá por pantalla el número 15. Pero si ahora modificamos:

```
let a: number = "Jorge";
```

La consola nos devolverá un error pues el valor Jorge no es un número.

```
test.ts(9,1): error TS2322: Type '"Jorge"' is not assignable to type 'number'.  
15:13:29 - Compilation complete. Watching for file changes.
```

Otra de las particularidades de TypeScript, es que se puede establecer el tipo de dato que devolverá una función.

La sintaxis será la siguiente:

```
function nombreFuncion(parametros): tiposdedato {  
    //código de la función  
}
```

Un sencillo ejemplo sería:

```
function devuelveEdad ( edad ): number {  
    let miEdad = edad;  
    return miEdad;  
}
```

1.4 Variables en TypeScript

Ya hemos adelantado en el apartado anterior, que en TypeScript las variables se declaraban con la palabra reservada *let*.

Pero además, TypeScript permite también emplear la palabra reserva *var* como en JavaScript, entonces... ¿cuándo emplear una u otra forma?

Pues dependiendo del alcance de la variable. Si necesitamos declarar la variable de forma local al bloque de código, declaración o expresión donde se ejecuta emplearemos *let*. En cambio si la declaración se necesita realizar de manera global, emplearemos *var*.

Veamos un ejemplo a continuación.

Vamos a editar nuestro archivo TypeScript sustituyendo todo el código por el siguiente:

test.ts

```
let a = 10; ①

if (true){

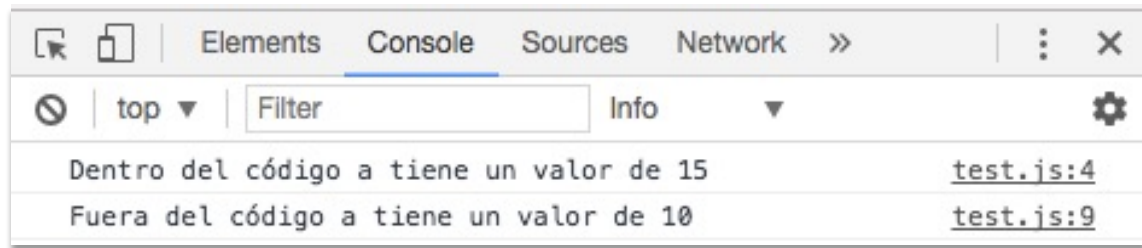
    let a = 15 ②
    console.log("Dentro del if a tiene un valor de "+a);
} else {
    //
}

console.log("Fuera del if a tiene un valor de "+a);
```

En el cual:

- ① Declaramos la variable *a* y la inicializamos con valor 10.
- ② Creamos una estructura if-else donde cambiamos el valor de *a* a 15 y lo imprimimos en la consola dentro del if (el else lo dejamos vacío).
- ③ Y finalmente, imprimimos el valor de *a* fuera del if-else.

Comprobaremos en la consola del navegador, como arroja dos valores diferentes de acuerdo al código que hemos programado:



Si ahora modificamos la declaración de la variable dentro del *if* utilizando la palabra reservada *var*:

test.ts

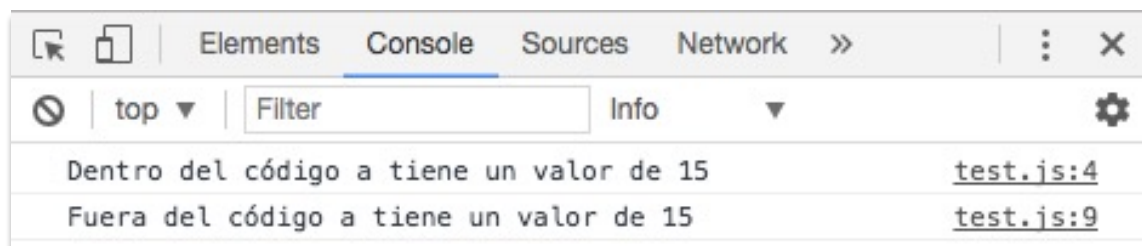
```
var a = 10;

if (true){

    var a = 15
    console.log("Dentro del if a tiene un valor de "+a);
} else {
    //
}

console.log("Fuera del if a tiene un valor de "+a);
```

Como la variable pasa a ser global, su valor dentro del *if* también se utiliza fuera de esta y por tanto ambos *logs* muestran el mismo valor en la consola.



1.5 Clases en TypeScript

En TypeScript disponemos de clases para crear objetos, consiguiendo de esta manera que este lenguaje pueda ser orientado a objetos.

La sintaxis para crear clases será la siguiente:

```
class NombreClase {  
    public/private nombrepropiedad: tipo de dato;  
    ....  
  
    public/private nombremetodo() {  
        //código del método  
    }  
    ...  
}
```

Por ejemplo, en nuestro archivo test.ts sustituimos todo el código para añadir:

test.ts

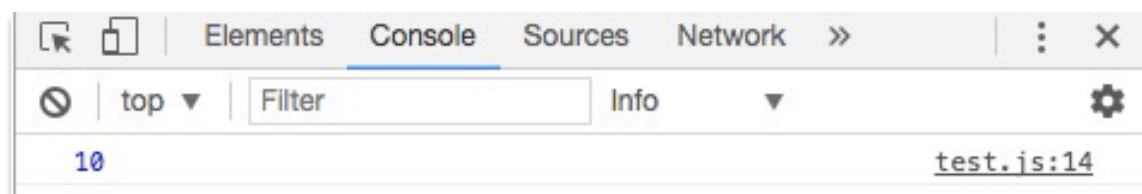
```
class Curso {  
    public titulo: string;           ①  
    public descripcion: string;  
    public horas: number;  
    public inscritos: number;  
  
    public getInscritos() {           ②  
        return this.inscritos;  
    }  
  
    public setInscritos(inscritos: number) {  
        this.inscritos = inscritos;  
    }  
  
    public addInscrito(){  
        this.inscritos++;  
    }  
}
```

```
public remInscrito() {  
    this.inscritos--;  
}  
  
var cursoAngular = new Curso();  
  
cursoAngular.setInscritos(9);  
cursoAngular.addInscrito();  
  
console.log(cursoAngular.getInscritos());
```

En el cual:

- ① Declaramos cuatro propiedades públicas.
- ② Creamos cuatro métodos públicos para devolver el número de inscritos, establecer el número de inscritos, añadir un nuevo inscrito y eliminar un inscrito.
- ③ Creamos un objeto cursoAngular de la clase Curso, establecemos el valor 9 en la propiedad inscritos con el método setInscritos y añadimos uno más con el método addInscrito.
- ④ Imprimimos en la consola la cantidad de inscritos del cursoAngular.

Comprobamos el resultado esperado, en la consola del navegador:



1.6 Constructor en TypeScript

También TypeScript permite el uso de constructores, siendo estos, métodos para inicializar las propiedades o atributos de un clase.

La sintaxis es la siguiente:

```
class NombreClase {  
    public/private nombrepropiedad: tipo de dato;  
    ....  
    constructor () {  
        this.nombrepropiedad = valor;  
        ...  
    }  
  
    public/private nombremetodo() {  
        //código del método  
    }  
    ...  
}
```

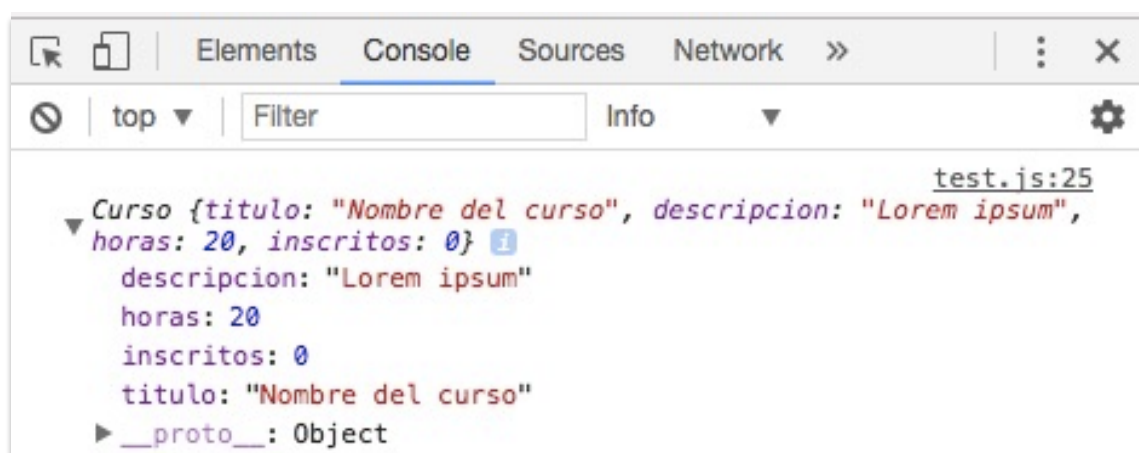
Por ejemplo, podemos sustituir la clase anterior Curso por el siguiente código:

test.ts

```
class Curso {  
    public titulo: string;  
    public descripcion: string;  
    public horas: number;  
    public inscritos: number;  
}  
  
constructor() {  
    this.titulo = "Nombre del curso";  
    this.descripcion = "Lorem ipsum";  
    this.horas = 20;  
    this.inscritos = 0;  
}
```

```
public getInscritos() {  
    return this.inscritos;  
}  
  
public setInscritos(inscritos: number) {  
    this.inscritos = inscritos;  
}  
  
public addInscrito(){  
    this.inscritos++;  
}  
  
public remInscrito() {  
    this.inscritos--;  
}  
  
}  
  
var cursoAngular = new Curso();  
  
console.log(cursoAngular);
```

Comprobaremos en la consola del navegador como se ha creado este objeto de la clase curso y sus variables han sido inicializadas con los valores por defecto del constructor.



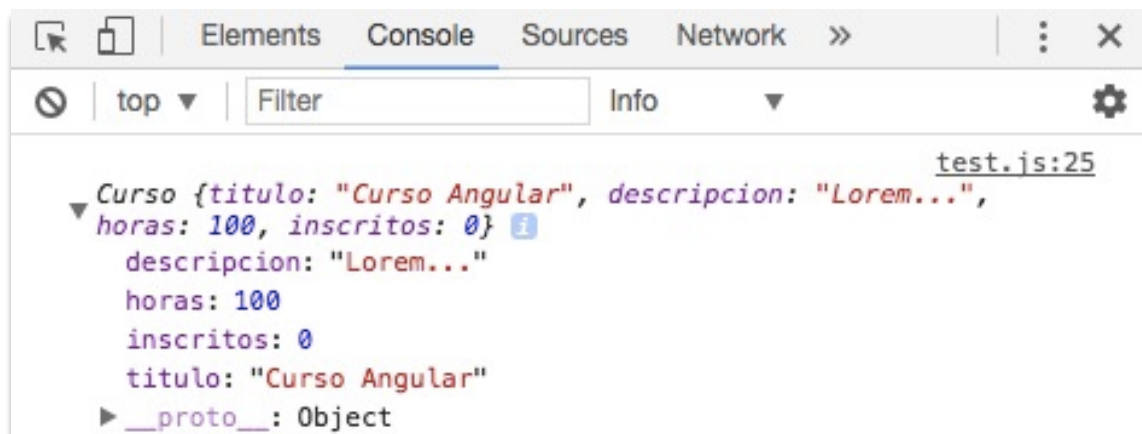
También podemos emplear parámetros en el constructor. Para ello modificamos por ejemplo el constructor de la siguiente forma.

```
...
constructor( titulo, descripcion, horas ) {
  this.titulo = titulo;
  this.descripcion = descripcion;
  this.horas = horas;
  this.inscritos = 0; //en este caso se establecerán con el método
}
...
```

Y sustituimos la línea de creación del objeto cursoAngular:

```
...
var cursoAngular = new Curso("Curso Angular","Lorem...",100);
...
```

De esta forma, el objeto se crea con los parámetros introducidos para cada propiedad.



Si no disponemos de algún parámetro a la hora de crear el objeto, para evitar errores de compilación, se puede emplear como valor *undefined* o *null*, de tal manera que se introduzcan todos los parámetros y en el orden especificado.

1.7 Interfaces en TypeScript

Para concluir con esta introducción a TypeScript, completamos el conocimiento de la sintaxis básica para la creación de objetos con interfaces.

Las interfaces, nos permitirán obligar a definir clases con unas determinadas propiedades o métodos.

Su sintaxis es:

```
interface NombreInterfaz {  
  
    nombrepropiedad: tipodedato;  
    nombreMetodo(): tipodedato;  
  
}
```

Y se implementan en la clase:

```
class Nombredelaclase implements NombreInterfaz {  
    ...  
}
```

En el ejemplo anterior, añadimos antes de la clase en el archivo test.ts, el siguiente código:

test.ts

```
interface DatosMaestros {  
    titulo: string;  
    addInscrito();  
}  
...
```

Modificamos la primera línea de la definición de la clase de la siguiente forma:

```
...  
class Curso implements DatosMaestros {  
    ...  
}
```

Y eliminamos la propiedad titulo en el constructor:

```
...
constructor( descripcion, horas ) {
    this.descripcion = descripcion;
    this.horas = horas;
    this.inscritos = 0; //en este caso se establecerán con el método
}
...
```

Como la interfaz obliga a implementar la clase con todas sus propiedades, la consola nos devolverá un error:

```
test.ts(6,7): error TS2420: Class 'Curso' incorrectly implements interface 'DatosMaestros'.
  Property 'addInscrito' is missing in type 'Curso'.
4:52:07 PM - Compilation complete. Watching for file changes.
```

El resto de características básicas de TypeScript pueden ser consultadas en su documentación:

<https://www.typescriptlang.org/docs/home.html>

1.8 Web Components

Los web components son un estándar del W3C, que permiten componer el desarrollo de aplicaciones web a partir de contenedores dedicados a una cierta funcionalidad. Estarán compuestos por archivos con el lenguaje de marcado HTML5, con código JavaScript para la lógica de negocio y con hojas de estilo CSS para la presentación.

Cada uno de estos componentes se puede implementar en el código de las páginas HTML5 a través de una etiqueta específica, que renderizará el componente y su funcionalidad, con una sintaxis muy simple:

```
<nombre-del-componente></nombre-del-componente>
```

Los web components pueden estar compuestos de 4 elementos:

- Templates. Son plantillas HTML5 con el contenido del componente.
- Custom Elements. Permiten la definición de elementos HTML propios.
- Shadow DOM. Podría definirse como un elemento DOM ajeno al DOM de la página web el cual encapsula el componente.
- HTML Imports. Líneas de código para en HTML, importar los ficheros HTML que contengan los componentes.

Angular, desde su primera versión a la actual, utiliza los web components como tecnología para el desarrollo de aplicaciones web, como veremos a lo largo de todo el libro.

Para más información sobre esta tecnología, su documentación se encuentra en:

https://developer.mozilla.org/es/docs/Web/Web_Components

2. Configuración de proyectos: Angular-Cli

2.1 Instalación

Angular más que un *framework*, es toda una plataforma en la que crear aplicaciones web. Los proyectos en Angular, no se crean como una estructura de archivos estáticos, si no que necesitan una instalación en Node JS para tener un pequeño servidor y también cientos de dependencias para su uso en este servidor, por ejemplo las que traducen los archivos TypeScript que escribiremos, a los archivos Javascript que empleará el navegador.

Realmente Angular no se instala en un equipo local de manera global (aunque podría hacerse) sino que se añade como módulos Node a cada proyecto que creemos con este *framework*.

De esta manera, la forma de instalación hasta la versión 2, fue añadir un archivo `package.json` e instalarlo con el gestor de paquetes NPM.

Aunque es posible seguir implementando Angular a nuestros proyectos a partir de un archivo `package.json`, en la actualidad disponemos una herramienta de línea de comandos, Angular CLI, que realizará ese trabajo por nosotros así como otras funcionalidades más que conoceremos posteriormente.

2.2 Requisitos Previos

Una vez que instalemos Node JS y NPM, comprobamos en la consola sus versiones con los comandos:

```
node -v
```

```
npm -v
```

Ambas tienen que ser superiores a v5.x.x y 3.x.x respectivamente.

2.3 Instalación de Angular CLI

La instalación de Angular CLI se lleva a cabo mediante NPM con el siguiente comando en la consola del equipo:

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

La opción `-g` es para instalarlo de manera global.

Si queremos actualizar una instalación previa de Angular CLI, emplearemos:

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

```
npm cache clean
```

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

2.4 Creación de un Proyecto en Angular

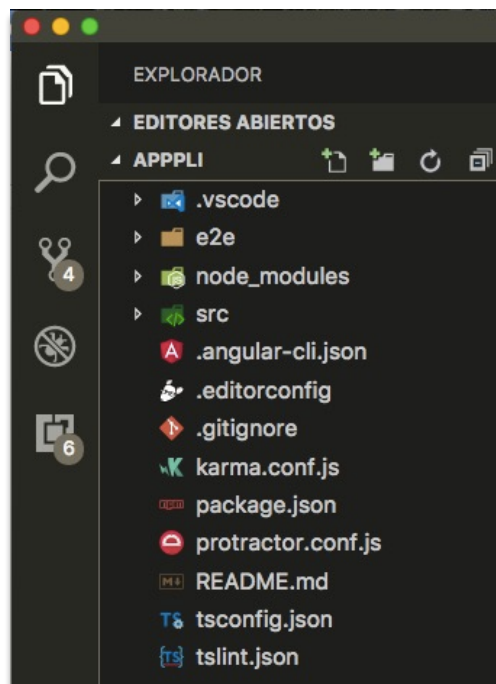
Una vez instalado Angular CLI, para crear un nuevo proyecto, nos situamos en el directorio en el que queremos crear el proyecto en la consola y tecleamos:

```
ng new appCurso
```

El proceso concluirá pasados unos minutos y tras finalizar, podemos acceder al directorio raíz del proyecto mediante:

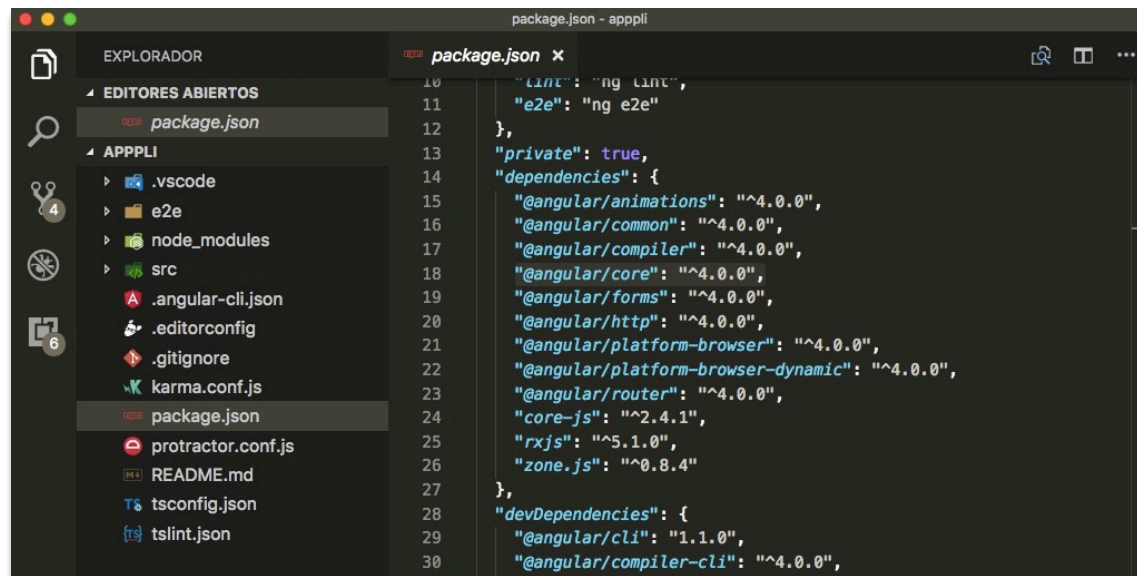
```
cd appCurso
```

También podemos comprobar la estructura del proyecto creada por Angular CLI en el editor que empleemos para el desarrollo de nuestra aplicación.



De momento y sin entrar en más detalles podemos fijarnos en el archivo `package.json` que contendrá las dependencias del proyecto y su número de versión.

En la creación del proyecto, Angular CLI, además de generar este archivo lo instala mediante NPM.



También nos podemos fijar como a partir de este package.json, se ha creado una carpeta de módulos Node JS denominada `node_modules`, que contiene todo el código del *framework* Angular en forma de paquetes, que emplearemos en nuestra aplicación.

Este formato permite que cada módulo y componente llame al paquete de Angular que vaya a utilizar y por tanto, solo cargue el código necesario para la funcionalidad que necesite.

Por último, disponemos de un directorio `src` (por *source*) donde estará el código de nuestra aplicación. El resto de archivos y directorios serán tratados más adelante.

2.5 Arranque de la Aplicación

Para levantar el servidor Node JS de nuestro nuevo proyecto, simplemente desde el directorio raíz del mismo, tecleamos en la consola del equipo:

```
ng serve
```

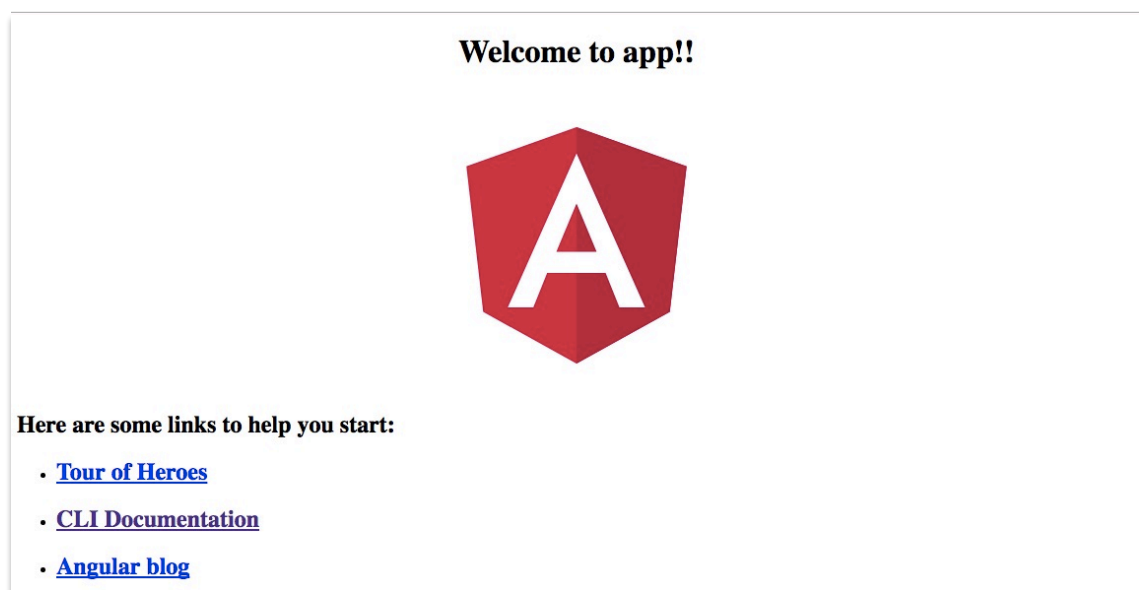
De esta manera, se inicia el servidor en nuestro equipo y podremos acceder a la aplicación en nuestro navegador en la siguiente url:

```
localhost:4200
```

Nota: Para usar un puerto diferente, se puede emplear:

```
ng serve --port <número del puerto>
```

Al acceder a la url, la aplicación nos muestra una plantilla de bienvenida, creada por Angular CLI.



2.6 El Archivo index.html

Como cualquier aplicación o sitio web, los proyectos Angular disponen de un archivo index.html, ubicado en la carpeta src.

Además del código habitual de un archivo html, este archivo se caracteriza por incluir la etiqueta `<app-root></app-root>`, que será la etiqueta del web component donde se 'renderice' todo el código de la aplicación.

Dentro de la etiqueta raíz anterior, podemos añadir un texto que indique que la aplicación está cargando o bien un *spinner*, que se mostrará durante la carga de la aplicación si esta se demora.

Por ejemplo:

src/index.html

```
...  
<app-root><p>Cargando aplicación...</p></app-root>  
...
```

Si la aplicación es muy ligera, cargará tan rápido el componente raíz, que ese texto de carga, directamente no se mostrará.

En la etiqueta `<head></head>` de este archivo, podemos añadir como en cualquier otra aplicación, CDN de librerías de fuentes e iconos así como *frameworks* de estilo como por ejemplo Bootstrap 4.

Por ejemplo, nosotros vamos a emplear las fuentes Google y Bootstrap 4, para lo cual añadimos, dentro del `<head></head>`:

```
...  
<link href="https://fonts.googleapis.com/css?family=Open+Sans"  
rel="stylesheet">  
<link rel="stylesheet"  
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-  
alpha.6/css/bootstrap.min.css" integrity="sha384-
```

```

rwoIResjU2yc3z8GV/NPeZwAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNU
wEyJ" crossorigin="anonymous">
  <script src="https://code.jquery.com/jquery-3.1.1.slim.min.js"
integrity="sha384-
A7FZj7v+d/sdmMqp/nOQwliLvUsJfDHW+k9Omg/a/EheAdgtzNs3hpfag6Ed95
0n" crossorigin="anonymous"></script>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/tether/1.4.0/js/tether.min.js"
integrity="sha384-
DztdAPBWPRXSA/3eYEEUWrWCy7G5KFbe8fFjk5JAlxUYHKkDx6Qin1DkWx51
bBrb" crossorigin="anonymous"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.6/js/bootstrap.min.js" integrity="sha384-
vBWWzlZJ8ea9aCX4pEW3rVHjgt7zpkNpZk+02D9phzyeVKE+jo0ieGizqPLForn
" crossorigin="anonymous"></script>
...

```

Si no queremos usar un CDN, otra forma de añadir Bootstrap a un proyecto Angular es a través de la instalación mediante NPM.

Para ello nos situamos en la consola del equipo, en el directorio raíz de la aplicación y pulsamos:

```
npm install --save bootstrap
```

Una vez instalado, lo importamos en el archivo `.angular-cli.json` mediante la modificación del código:

`.angular-cli.json`

```

...
styles: [
  "../node_modules/bootstrap/dist/css/bootstrap.min.css",
  "styles.css"
]
...

```

2.7 El Archivo styles.css

En el archivo styles.css, ubicado en la carpeta src, se pueden incluir todos los estilos CSS globales a toda la aplicación, sin tener que incluir una llamada al mismo en el index.html, ya que Angular lleva a cabo su carga de manera automática.

2.8 El Archivo favicon.png

También en la carpeta src, se puede sustituir el archivo favicon.png existente, que contiene el logo de Angular, por uno específico de nuestra aplicación en las dimensiones habituales de este elemento.

Un recurso para crear nuestro favicon de manera online es:

<http://www.genfavicon.com/es/>

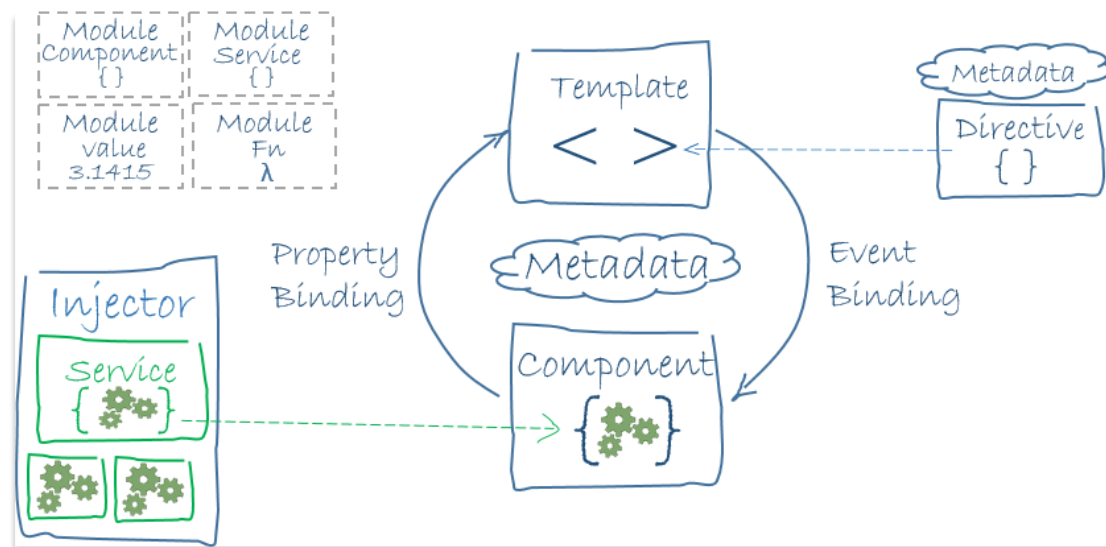
2.9 El Directorio Assets

Dentro del directorio src de la aplicación, en el directorio assets ubicaremos todos los ficheros estáticos de la aplicación, sobre todo las imágenes, logos, videos, etc, que empleemos en la misma.

3 Módulos y Componentes

Las aplicaciones actuales basan su desarrollo en arquitecturas compuestas por módulos que estructuren y organicen el código por funcionalidad y que al ser reutilizables, reduzcan los costes de desarrollo.

Por supuesto, Angular también se caracteriza por emplear esas condiciones de modularidad.



3.1 Módulos en Angular

Un módulo en Angular, es un conjunto de código dedicado a un ámbito concreto de la aplicación, o una funcionalidad específica.

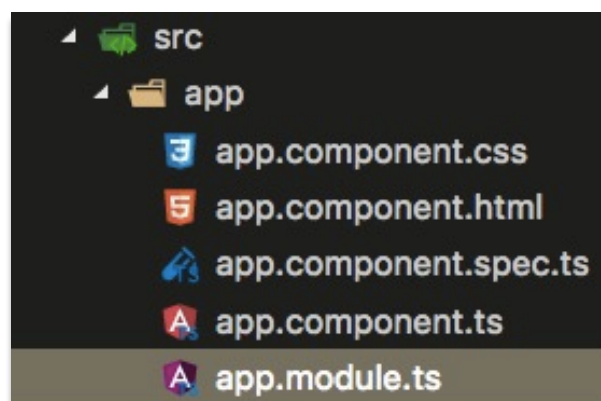
En Angular, los módulos se definen mediante una clase decorada con @NgModule.

Toda aplicación de Angular tendrá al menos un módulo, el llamado módulo principal o raíz (root module), que, para el caso de aplicaciones pequeñas será único.

Los módulos, se definen en archivos TypeScript y podemos decir que están compuestos de tres bloques o apartados de código.

1. En primer lugar una serie de instrucciones de importación de las librerías o paquetes Angular, así como otros elementos externos que emplearemos en el módulo.
2. En segundo lugar, `@NgModule`, un decorador que recibe un objeto de metadatos que definen el módulo. Estos metadatos son los siguientes:
 - *Declarations*. Las declaraciones son las llamadas vistas de un módulo. Hay 3 tipos de vistas o declaraciones, los componentes, las directivas y los *pipes*.
 - *Imports*. En este apartado se indican las dependencias o paquetes que empleará este módulo, cuyo origen se define en las importaciones al inicio del archivo.
 - *Providers*. Son los servicios utilizados por el módulo, disponibles para todos los componentes, y que centralizan la gestión de datos o funciones para inyectarlos en los componentes.
 - *Bootstrap*. Este metadato define la vista raíz de la aplicación y es utilizado solo por el módulo raíz. No confundir con el popular *framework* de estilos del mismo nombre.
3. Y el tercer y último bloque de código, es la instrucción de exportación del módulo como una clase Angular, que será posteriormente introducido en el archivo JavaScript principal de la aplicación.

Insistimos en que en todas las aplicaciones Angular, existe al menos un módulo raíz, que se encuentra ubicado en el archivo `app.module.ts` generado por Angular CLI en el directorio `src/app`.



Si nos fijamos en el código del archivo `app.module.ts` está compuesto de los tres bloques de código detallados en el párrafo anterior.

`src/app/app.module.ts`

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

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

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

export class AppModule {}
```

① Los *import* de las librerías de Angular necesarias para el funcionamiento de este módulo así como el *import* del componente de este módulo.

② El decorador `@NgModule`, que incluye:

- Las declaraciones, con el componente `AppComponent` que a continuación detallaremos.
- Los *imports*, con los módulos a emplear (de momento el de renderización en navegadores y el de formularios ambos de las librerías Angular).
- Los *providers*, que se declaran como un array, y de momento está vacío porque aún no hemos creado ningún servicio.

- Y bootstrap, recordamos exclusivo de este módulo raíz, que define el componente AppComponent para inicializar la aplicación.

③ Para finalizar, el archivo exporta la clase AppModule.

3.2 Componentes en Angular

Podemos definir un componente como una clase Angular que controla una parte de la aplicación, de ahí que sean englobados como un tipo de vista.

Los componentes son definidos con el decorador @Component, y son los archivos en los cuales definiremos y controlaremos la lógica de la aplicación, su vista HTML y el enlace con otros elementos.

Un componente estará también compuesto por tres bloques de código:

1. Imports. Las sentencias de importación de los diferentes elementos que empleará el componente.
2. Decorador @Component. Con al menos, los siguientes metadatos:

Selector. Que define la etiqueta html donde se renderiza el componente.

Template. El archivo html con la vista del componente.

Style. Con el archivo CSS con los de estilos del componente.

3. Export de la Clase. Definición y exportación de la clase con la lógica del componente.

El módulo raíz, como hemos detallado anteriormente, dispone también de un componente obligatorio para el funcionamiento de la aplicación. Se trata del archivo TypeScript app.component.ts y se encuentra en el directorio app.

Podemos observar en su código, generado por Angular CLI al crear el proyecto, los tres bloques de código:


```
import { Component } from '@angular/core'; ①

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',      ②
  styleUrls: ['./app.component.css']
})

export class AppComponent { ③
  title = 'app';
}
```

En el cual:

① Se escriben los *imports del componente*, en este caso de la librería Component de Angular.

② El Decorador @Component, con un objeto de metadatos, que en este caso contiene:

- El selector. Que define en qué etiqueta html se renderizará este componente. En este componente raíz, la etiqueta es app-root que es la que tenemos en el archivo index.html.
- La template o plantilla. Que define el contenido html del componente. En este caso se usa templateUrl y la ruta del archivo template, siendo app.component.html.
- La referencia de estilo. Aquí se definen los estilos CSS particulares de ese componente. En este caso se usa styleUrls y la ruta del archivo de estilo del componente, siendo app.component.css.

③ Exportación de la clase AppComponent que contiene la lógica del componente. Vemos como en este caso, simplemente define una variable 'title' a la que asigna un texto con el valor 'app'.

Veamos a continuación el funcionamiento del componente incluido en el módulo raíz.

El componente define mediante `templateUrl`, cual es el archivo template HTML, en este caso `app.component.html`, que se encuentra en su misma carpeta, y cuyo código es:

src/app/app.component.html

```
<div style="text-align:center">
  <h1>
    Welcome to {{title}}!!
  </h1>
  
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank"
href="https://angular.io/docs/ts/latest/tutorial/">Tour of Heroes</a></h2>
  </li>
  <li>
    ...
  </li>
</ul>
```

Vemos como se trata de un archivo html pero sin la estructura de página web, ya que se inyectará en el archivo `index.html`. Además de las etiquetas html y los textos, el código incorpora la variable `title` mediante la sintaxis *moustache* `{{ title }}`, de tal forma que esta sintaxis lo enlaza componente y mostrará en pantalla el valor definido en la clase de este.

Además de definir la plantilla HTML, el componente también define con el metadato `selector`, en qué etiqueta se renderizará su contenido, en este caso la etiqueta con nombre `app-root`.

Repetimos que el componente se exporta mediante la clase AppComponent, que es importada en el módulo app.module.ts e incluida dentro de su decorador en las declaraciones:

src/app/app.module.ts

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

...
  declarations: [
    AppComponent
  ],
...

```

Por otra parte, este archivo de módulo app.module.ts es el módulo raíz e incluye en el decorador la propiedad Bootstrap igualada al componente, para finalmente exportar todo el módulo con la clase AppModule:

```
bootstrap: [AppComponent]
})
export class AppModule { }
```

¿Dónde empleamos esta clase del módulo que se exporta? En otro archivo situado en la raíz de la aplicación, denominado main.ts.

src/main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module'; ①
import { environment } from './environments/environment';
```

```
if (environment.production) {
  enableProdMode();
}
```

```
platformBrowserDynamic().bootstrapModule(AppModule); ②
```

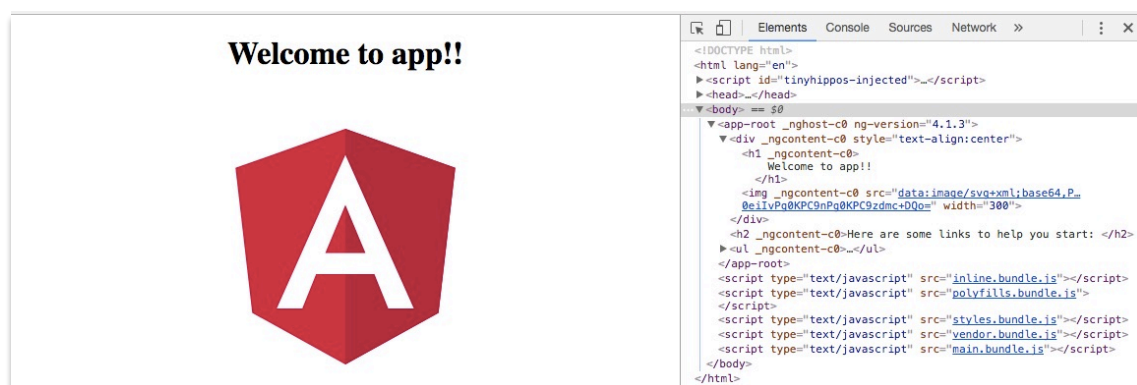
Que ① importa AppModule y en su última línea ② define AppModule como el punto de entrada de la aplicación.

Finalmente, el archivo index.html contiene dentro del body, la etiqueta <app-root>, donde renderizará la plantilla del componente.

src/index.html

```
...
<body>
  <app-root></app-root>
</body>
...
```

Lo podemos comprobar si inspeccionamos el elemento en el navegador con las herramientas de desarrollador:



Otra forma práctica de comprobarlo, es sustituir totalmente el contenido del archivo `app.component.html` por un mítico '¡Hola Mundo!':

`src/app/app.component.html`

```
<div style="text-align:center">
  <h1>¡Hola Mundo!</h1>
</div>
```

Y al acceder ahora a nuestro navegador en `localhost:4200` nos mostrará:

¡Hola Mundo!

También podemos comprobar la lógica de la aplicación sustituyendo en el archivo `app.component.ts` las expresiones en la clase por la siguiente:

`src/app/app.component.ts`

```
...
export class AppComponent {
  destino: string = 'Universo';
}
...
```

Y a continuación, modificamos de nuevo el archivo `app.component.html`, sustituyendo todo el código por el siguiente:

`src/app/app.component.html`

```
<div style="text-align:center">
  <h1>¡Hola {{ destino }}!</h1>
</div>
```

Volvemos al navegador y comprobamos:

¡Hola Universo!

Es importante recordar que la sintaxis de Javascript/Typescript está capitalizada y distingue mayúsculas de minúsculas.

También podemos hacer uso del archivo de estilos CSS del componente, `app.component.ts`, en el que por ejemplo escribir la siguiente clase:

`src/app/app.component.css`

```
.encabezado {  
    text-align: center;  
    color: blueviolet;  
}
```

De nuevo modificamos el archivo de la plantilla, `app.component.html`, sustituyendo todo el código por el siguiente:

`src/app/app.component.html`

```
<div class="encabezado">  
  <h1>¡Hola {{ destino }}!</h1>  
</div>
```

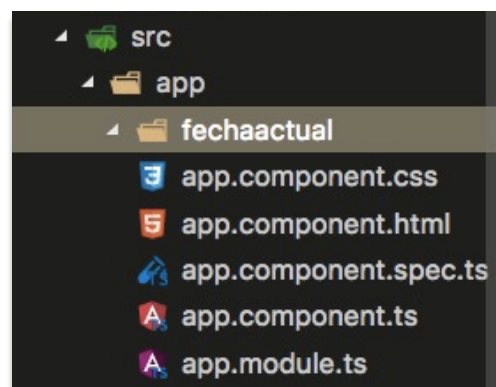
Y comprobamos en el navegador como se han aplicado los estilos:

¡Hola Universo!

3.3 Creación de un nuevo componente

Una vez que conocemos el funcionamiento de un componente, concretamente el raíz, vamos a crear un nuevo componente y veremos cómo, aunque su funcionamiento es idéntico, su integración en la aplicación es diferente a la del componente raíz.

Aunque se pueden crear con Angular CLI, para crear un nuevo componente de manera manual, en primer lugar creamos un directorio con el nombre identificativo del componente, por ejemplo, fechaactual, en el directorio src/app de la aplicación.



Dentro del componente creamos de momento dos archivos, el template con el nombre fechaactual.component.html y el archivo TypeScript fechaactual.component.ts.

La forma de nombrar los archivos es opcional, pero como buena práctica o convención, se usa el nombre del componente seguido de punto, la palabra component y la extensión del archivo, como así ocurre con el componente raíz.

Dentro el archivo fechaactual.component.ts escribimos el siguiente código:

```
src/app/fechaactual/fechaactual.component.ts
```

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

@Component({
  selector: 'app-fechaactual', ①
  templateUrl: './fechaactual.component.html' ②
})
```

```
}}

export class FechaactualComponent {
  hoy: any = new Date();
}
```

Dentro del código, comprobamos como hemos declarado ① `app-fechaactual` en el selector, ② hemos definido el archivo de plantilla en `templateUrl` y ③ hemos creado y exportado la clase `FechaactualComponent` con una sencilla instrucción JavaScript que declara `hoy` como la fecha actual.

Además para las referencias a archivos, es importante observar, cómo se emplean rutas relativas desde el directorio en el que se encuentre nuestro archivo para llamar a otros archivos.

Ahora completamos la plantilla, en el archivo `fechaactual.component.html` con un sencillo código:

```
src/app/fechaactual/fechaactual.component.html
```

```
<p> Sevilla, {{ hoy | date:'d/M/y H:m' }} </p>
```

Además de la variable `hoy`, estamos añadiendo un *pipe* para dar formato de fecha, funcionalidad de Angular que conoceremos más adelante.

El trabajo aún no ha finalizado, ya que necesitamos incorporar al módulo raíz de la aplicación para poder utilizarlo.

Para ello, en primer lugar incluimos nuestro componente en el módulo `app.module.ts` modificando el código de la siguiente manera:

```
src/app/app.module.ts
```

```
...
import { FechaactualComponent } from './fechaactual/fechaactual.component';
...
```



```
...
declarations: [
  AppComponent,
  FechaactualComponent ②
],
...
```

① Importamos la clase FechaactualComponent con la ruta a nuestro archivo (sin la extensión).

② Y en el array de declaraciones la añadimos.

Con esto, nuestro componente queda listo para ser utilizado, y por tanto tenemos que añadir la etiqueta <app-fechaactual> donde queramos que se renderice.

En el componente raíz, su etiqueta era directamente situada en el archivo index.html, pero en el caso de otro componente del módulo que no sea el raíz debemos situarlo en la plantilla de este módulo, para organizar correctamente los componentes.

Por tanto, sustituimos el código de app.component.html de la siguiente manera:

src/app/app.component.html

```
<div class="encabezado">
  <h1>¡Hola {{ destino }}!</h1>
  <app-fechaactual></app-fechaactual>
</div>
```

Y ahora veremos en nuestro navegador, el código del componente raíz e inmediatamente debajo el código del componente fechaactual:

¡Hola Universo!

Sevilla, 10/6/2017 19:56

Si el código html de la vista del componente es pequeño, como es nuestro caso, podemos simplificar la aplicación, eliminando el archivo `fechaactual.component.html`, añadiendo el código de este en el componente mediante el metadado `template`.

Por ejemplo, en nuestro caso podemos sustituir en `fechaactual.component.ts` el decorador por el siguiente:

`src/app/fechaactual/fechaactual.component.ts`

```
@Component({
  selector: 'app-fechaactual',
  template: `
    <p> Sevilla, {{ hoy | date:'d/M/y H:m'}}</p>
  `
})
```

En el que empleamos `template` en lugar de `templateUrl` y las comillas *backstick* para delimitar el código html, permitiendo emplear multilínea.

Ahora eliminamos el archivo `fechaactual.component.html` pues ya no será necesario y el funcionamiento sigue siendo el mismo.

3.4 Creación de componente con Angular CLI

Una de las funcionalidades de Angular CLI es la de generar los archivos de, entre otros, un componente, evitándonos tener que escribir el código común como hicimos en el apartado anterior.

La sintaxis del comando para generar nuevos componentes es:

```
ng generate component <nombredelcomponente>
```

o su versión abreviada:

```
ng g c <nombredelcomponente>
```

Para crear un nuevo componente, nos situamos en la consola del equipo en el directorio raíz del proyecto y completamos por ejemplo:

```
ng generate component copyright
```

Ya en la propia consola vemos como se crean los nuevos archivos y se actualiza el módulo app.module.ts.

Además de los archivos que conocemos, se crea un archivo component.spec.ts solamente para tareas de testing.

```
installing component
create src/app/copyright/copyright.component.css
create src/app/copyright/copyright.component.html
create src/app/copyright/copyright.component.spec.ts
create src/app/copyright/copyright.component.ts
update src/app/app.module.ts
```

Si no queremos que se cree el archivo spec, podemos añadir en el comando anterior de Angular CLI la opción --spec false. En nuestra nueva carpeta copyright accedemos al archivo copyright.component.ts y añadimos el siguiente código a la clase:

```
src/app/copyright/copyright.component.ts
```

```
export class CopyrightComponent implements OnInit {  
  
  copyright: String = '© ACME S.A.';  
  hoy: any = new Date();  
  
  constructor() {}  
  
  ngOnInit() {  
  }  
}
```

Más adelante hablaremos del método `ngOnInit` y del uso del constructor JavaScript en Angular.

El siguiente paso será incluir el código en el template `copyright.component.html`, por ejemplo:

```
src/app/copyright/copyright.component.html
```

```
<p> {{ copyright }} {{ hoy | date:'y' }} </p>
```

Y finalmente añadimos la etiqueta en el template del componente raíz `app.component.html`:

```
src/app/app.component.html
```

```
<div class="encabezado" >  
  <h1>¡Hola {{ destino }}!</h1>  
  <app-fechaactual></app-fechaactual>  
  <app-copyright></app-copyright>  
</div>
```

Podemos comprobar en el navegador la correcta implementación del componente.

¡Hola Universo!

Sevilla, 10/6/2017 20:38

© ACME S.A. 2017

3.5 Anidado de Componentes

Los componentes pueden ser anidados, es decir que la etiqueta donde se renderiza definida por el selector, puede ser empleada en cualquier parte del código.

Por ejemplo dentro del template del componente copyright, podemos sustituir el código para incluir la etiqueta del componente fechaactual:

```
src/app/copyright/copyright.component.html
```

```
<app-fechaactual></app-fechaactual>  
<p> {{ copyright }} {{ hoy | date:'y' }} </p>
```

Y eliminar la etiqueta del código del template raíz app.component.html, para que no se duplique:

```
src/app/app.component.html
```

```
<div class="encabezado" >  
  <h1>¡Hola {{ destino }}!</h1>  
  <app-copyright></app-copyright>  
</div>
```

Siendo el resultado final el mismo.

Otra forma de anidado es incluir la etiqueta de un componente en el decorador de otro. Por ejemplo, modificamos el archivo copyright.component.html para dejarlo como estaba originalmente.

```
src/app/copyright/copyright.component.html
```

```
<p> {{ copyright }} {{ hoy | date:'y' }} </p>
```

Y ahora modificamos el archivo fechaactual.component.ts para incluir en el decorador la etiqueta del componente copyright:

```
src/app/fechaactual/ fechaactual.component.ts
```

```
...  
@Component({  
  selector: 'app-fechaactual',  
  template: `  
    <p> Sevilla, {{ hoy | date:'d/M/y H:m' }}</p>  
    <app-copyright></app-copyright>  
  `,  
})  
...
```

Y dejamos en el componente raíz, en su template app.component.html, solamente la etiqueta del componente fechaactual:

```
src/app/app.component.html
```

```
<div class="encabezado" >  
  <h1>¡Hola {{ destino }}!</h1>  
  <app-fechaactual></app-fechaactual>  
</div>
```

El resultado en el navegador vuelve a ser de nuevo el mismo, por lo que comprobamos que disponemos en Angular de la máxima flexibilidad a la hora de estructurar los diferentes componentes de la aplicación.

4 Data Binding

Podemos definir el *data binding* en Angular como la comunicación o el enlace de datos entre el archivo Typescript del componente y el archivo HTML de su *template*.

De hecho, ya hemos conocido el funcionamiento de la forma más sencilla de este enlace, entre una variable declarada en la clase del componente y la representación de su valor en la plantilla HTML mediante la sintaxis *moustache* con doble llave.

Disponemos de varias formas de llevar a cabo los procesos *data binding*.

En primer lugar tenemos los data binding de tipo *One Way de la fuente de datos* (archivo TypeScript) *a la vista* (archivo HTML). Dentro de este tipo se incluyen:

- Interpolación.
- *Property Binding*.

En segundo lugar se incluyen los de tipo *One Way de la vista a la fuente de datos*:

- *Event Binding*.

Y finalmente, los que enlazan en ambos sentidos y simultáneamente los datos, de la vista a la fuente de datos y viceversa, que se engloban en la siguiente denominación.

- *Two-way Binding*.

4.1 Interpolación

La interpolación, también denominada *String Interpolation*, consiste en incluir dentro del código html y entre dobles paréntesis, una expresión que Angular evalúa para mostrar en la plantilla.

Recordamos que este ha sido el procedimiento que empleamos en el apartado anterior, cuando explicábamos el funcionamiento de los componentes.

Y recordamos su sintaxis, en la que se puede incluir una propiedad, un objeto o una expresión JavaScript entre dobles llaves, conocida como sintaxis *moustache*:

`{{ propiedad/objeto/expresión }}`

En los apartados anteriores, hemos utilizado variables en la clase del componente para después imprimirlas por pantalla en la plantilla HTML.

También podemos emplear la interpolación utilizando propiedades de un objeto del componente como origen de los datos o cualquier expresión JavaScript como tendremos la oportunidad de comprobar.

Vamos a ver un ejemplo de interpolación empleando objetos, pero antes de conocer esta interpolación, vamos a ver como incorporar las clases para crear objetos en Angular.

Una de las formas para crear objetos en Angular es mediante un modelo, es decir, podemos crear un archivo en el que declarar la clase y sus propiedades para posteriormente poder utilizar esta clase en los componentes, de ahí que denomine a estos archivos como modelos.

Vamos a crear una clase que se pueda utilizar en toda la aplicación. En primer lugar creamos un directorio llamado modelos en el directorio app de la aplicación.

Seguidamente, dentro de este directorio, creamos de manera manual un archivo TypeScript denominado alumno.modelo.ts con el siguiente código:

```
export class Alumno {  
  public id: number;  
  public nombre: string;           ①  
  public apellidos: string;  
  public ciudad: string;  
  
  constructor (id: number, nombre: string, apellidos: string, ciudad: string){  
    this.id = id;  
    this.nombre = nombre;  
    this.apellidos = apellidos;    ②  
    this.ciudad = ciudad;  
  
  }  
}
```

En el cual:

- ① Definimos las propiedades de la clase Alumno que se genera y exporta.
- ② Y usamos el método constructor para definir los parámetros que recibirá y el enlace con las propiedades de la clase.

Este archivo no se enlazará en el módulo de la aplicación, app.module.ts, porque de hecho no es ningún tipo de declaración ni tiene ningún decorador, solamente exporta la clase Alumno para poder crear objetos de esa clase en cualquier archivo de la aplicación.

A continuación vamos a ver como utilizamos esa clase Alumno en un nuevo componente. Para ello, en la consola del equipo, en el directorio raíz del proyecto, completamos:

```
ng g c viewmodelo --spec false
```

Dentro de la clase de este nuevo componente en el archivo `viewmodelo.component.ts`, importamos la clase `Alumno` y creamos un objeto de esta clase llamado `alumno1`, de la siguiente manera:

`src/app/viewmodelo/viewmodelo.component.ts`

```
...  
import { Alumno } from '../modelos/alumno.model';  
...  
...  
export class ViewmodeloComponent {  
    alumno1 = new Alumno (1, 'Juan', 'Gutiérrez', 'Madrid'); } ...
```

Ahora podemos añadir en la plantilla, en el archivo `viewmodelo.component.html`, el código necesario para mostrar el objeto de la siguiente forma:

`src/app/viewmodelo/viewmodelo.component.html`

```
<div class="container">  
  <h4>Información del Alumno</h4>  
  <hr>  
  <h5>id: {{ alumno1.id }}</h5>  
  <hr>  
  <h5>Nombre: {{ alumno1.nombre }}</h5>  
  <hr>  
  <h5>Apellidos: {{ alumno1.apellidos }}</h5>  
  <hr>  
  <h5>Ciudad: {{ alumno1.ciudad }}</h5>  
  <hr>  
</div>
```

En el que empleamos, dentro de la interpolación, la sintaxis de la notación del punto JavaScript para mostrar el valor de la propiedad de un objeto:

`nombredelobjeto.propiedaddelobjeto`

Finalmente añadimos la etiqueta del componente en el archivo `app.component.html` para renderizarla en nuestra aplicación:

`src/app/app.component.html`

```
<div class="container" >
  <h3>Ejemplo de Interpolación con Objeto</h3>
  <hr>
  <app-viewmodelo></app-viewmodelo>
</div>
```

Y comprobamos en el navegador el resultado:



Otra opción de uso de la funcionalidad de interpolación es emplear métodos para enlazar datos del componente a la vista.

En un ejemplo muy sencillo, vamos a crear un componente llamado `ejmetodo` en el directorio raíz de la aplicación, tecleando para ello en la consola del equipo:

`ng g c ejmetodo --spec false`

Y una vez creado, añadimos a la clase del componente en `ejmetodo.component.ts` el siguiente código:

`src/app/ejmetodo/ejmetodo.component.ts`

```
...
puntuacion = 9;

getPuntuacion() {
  return this.puntuacion;
}
...
```

Ahora en la template del archivo `ejmetodo.component.html` sustituimos el código por:

`src/app/ejmetodo/ejmetodo.component.html`

```
<p> La puntuación obtenida por la mayoría
  de desarrolladores de Angular es de {{ getPuntuacion() }} sobre 10</p>
```

Y en el archivo `app.component.html` sustituimos el código para añadir la etiqueta `app-ejmetodo`:

`src/app/app.component.html`

```
<div class="container" >
  <h3>Ejemplo de Interpolación con Método</h3>
  <hr>
  <app-ejmetodo></app-ejmetodo>
</div>
```

Comprobamos en el navegador como se carga correctamente:

Ejemplo de Interpolación con Método

La puntuación obtenida por la mayoría de desarrolladores de Angular es de 9 sobre 10

4.2 Property Binding

Property Binding es el segundo tipo de data binding empleados por Angular para enlazar valores de la fuente de datos a la vista.

En este caso, se trata de un enlace que relaciona un atributo con una expresión, con la siguiente sintaxis:

```
[ atributodelelementoHTML ] = " expresión "
```

Vamos a crear un nuevo componente, ejpropertybinding, para lo cual como es habitual, tecleamos desde la consola del equipo en el directorio raíz de la aplicación:

```
ng g c ejpropertybinding --spec false
```

En el template, ejpropertybinding.component.html, sustituimos todo el código por el siguiente:

```
src/app/ejpropertybinding/ejpropertybinding.component.html
```

```
<input type="text" placeholder="Escribe algo...">
```

Y colocamos la etiqueta del componente en el template del componente raíz en app.component.html como en los casos anteriores, de la siguiente manera:

```
src/app/app.component.html
```

```
<div class="container" >
  <h3>Ejemplo de Property Binding</h3>
  <hr>
  <app-ejpropertybinding></app-ejpropertybinding>
</div>
```


Hasta ahora nada de particular, pero vamos a ver cómo modificar el comportamiento del atributo placeholder de manera dinámica mediante *property binding*.

En la clase del componente en `ejpropertybinding.component.ts`, añadimos el siguiente código:

`src/app/ejpropertybinding/ejpropertybinding.component.ts`

```
...  
texto = 'Escribe algo'; ①  
  
constructor() {  
  setTimeout(() => {  
    this.texto = 'por favor'; ②  
  }, 3000);  
}
```

El dinamismo de este código consiste en ① crear una propiedad `texto` inicializada con el valor 'Escribe algo...' y a continuación, ② utilizar dentro del constructor, para no tener que ser llamado, el método JavaScript `setTimeout()` que llevará a cabo el código para cambiar el valor 'Escribe algo...' de la propiedad `texto` a 'por favor' cuando pasen 3000 ms, es decir 3 segundos.

Ahora en el template modificamos el atributo, de manera que lo rodeamos de corchetes y lo igualamos a la propiedad `texto`:

`src/app/ejpropertybinding/ejpropertybinding.component.html`

```
<input type="text" [placeholder]="texto">
```

Y comprobaremos como, al iniciar el navegador, el input tendrá el texto "Escribe algo" en su placeholder, y cuando transcurran 3 segundos, pasará a ser "por favor".



4.3 Event Binding

Los enlaces *event binding* son enlaces de un solo sentido, *one way*, pero en este caso desde la vista a la fuente de datos, ya que como su nombre indica, los desencadena un evento en el cliente web.

En este enlace de datos, se iguala un evento de un elemento HTML de la vista con una expresión, normalmente un método definido en la clase del componente.

La sintaxis tiene la siguiente forma:

```
evento="nombreMetodo()";
```

Vamos a crear otro ejemplo como venimos haciendo, en un nuevo componente. Para ello, tecleamos desde la consola del equipo en el directorio raíz de la aplicación:

```
ng g c ejeventbinding --spec false
```

En la template, archivo `ejeventbinding.component.html`, vamos a crear un botón y un texto con el siguiente código:

```
src/app/ejeventbinding/ejeventbinding.component.html
```

```
<button class="btn btn-success"
  (click)="modTexto()">Modificar Texto</button> ①
<h3> {{ texto }} </h3> ②
```

En el cual:

- ① El botón desencadenará el método `modTexto()` mediante el evento `click` de HTML, cuando lo pulsemos.
- ② El elemento `h3` mostrará el valor de la propiedad `texto`.

Ahora, en el componente, archivo `ejeventbinding.component.ts`, añadimos el siguiente código dentro de la clase:

`src/app/ejeventbinding/ejeventbinding.component.ts`

```
...
texto = 'Originalmente el texto se carga así';

modTexto() {
  this.texto = 'Al pulsar el botón el texto se muestra así';
}

ngOnInit() {
}
...
```

Y en el componente raíz, añadimos la etiqueta del componente de la siguiente forma:

`src/app/app.component.html`

```
<div class="container" >
  <h3>Ejemplo de Event Binding</h3>
  <hr>
  <app-ejeventbinding></app-ejeventbinding>
</div>
```

Si comprobamos la aplicación en el navegador, veremos cómo al pulsar en el botón se modifica el texto.



4.4 Two Way Binding

Para finalizar, disponemos de los enlaces two way binding, que enlazan en ambos sentidos y simultáneamente, los datos de la vista a la fuente de datos y viceversa.

Este enlace emplea una mezcla entre la sintaxis de eventos y la de interpolación conocida como *banana in a box*, y enlaza el valor del elemento HTML de la vista con la propiedad del componente, de la siguiente manera.

`[(directiva)] = "nombredelapropiedad"`

Vamos a crear el último ejemplo de esta serie. Para ello, tecleamos desde la consola del equipo en el directorio raíz de la aplicación:

`ng g c ej2waybinding --spec false`

En la template, archivo `ej2waybinding.component.html`, vamos a crear un campo input y un texto con el siguiente código:

```
src/app/ej2waybinding/ej2waybinding.component.html
```

```
<label>Introduce un valor</label>
<input type="text" class="form-control" [(ngModel)]="texto">
<h3>{{texto}}</h3>
```

En este caso hemos utilizado la directiva `ngModel` que más adelante detallaremos en el apartado correspondiente.

En caso de que no funcione la directiva `ngModel`, puede ser que la versión de Angular CLI no haya implementado `FormsModule`. En ese caso añadimos en el archivo `app.module.ts` la importación:

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

y en el Array de los imports:

```
FormsModule
```

A continuación, en la clase del archivo ej2waybinding.component.ts añadimos simplemente la declaración de la variable, con un valor de ejemplo.

src/app/ej2waybinding/ej2waybinding.component.ts

```
...
texto = 'Texto original al cargar';

constructor() {}

ngOnInit() {
}
...
```

Una vez más, incorporamos la etiqueta del componente al archivo componente raíz, app.component.html:

src/app/app.component.html

```
<div class="container" >
  <h3>Ejemplo de Two Way Binding</h3>
  <hr>
  <app-ej2waybinding></app-ej2waybinding>
</div>
```

Y al refrescar la aplicación, comprobamos como el input y el elemento h3, cargan en primer lugar el valor de la propiedad texto proveniente del componente, pero cuando escribamos algo en el input se actualiza de manera automáticamente en el h3.

Ejemplo de Two Way Binding

Introduce un valor

Texto original al cargar

5 Directivas

Las directivas son clases Angular con instrucciones para crear, formatear e interaccionar con los elementos HTML en el DOM, y son una de las principales características de este *framework*.

Se declaran mediante el empleo del decorador `@Directive`, y Angular incluye un gran número de ellas para la mayor parte de las funcionalidades que se necesitan en una aplicación.

Existen tres tipos de directivas.

- Componentes. Se trata de la aplicación en Angular de la tecnología web component y ya conocemos uno de sus empleos, las etiquetas HTML de cada componente que renderizan su plantilla donde las ubiquemos. Se podría decir que el decorador `@Component` es una aplicación de las directivas.
- De atributos. Son directivas que modifican el comportamiento de un elemento HTML de la misma manera que un atributo HTML, es decir, sin modificar el layout o presentación del elemento en el DOM. Se identifican por tener el prefijo `ng`.
- Estructurales. Estas directivas alteran el layout del elemento en el que se asocian, añadiendo, eliminando o reemplazando elementos en el DOM. Se identifican sobre las directivas de atributo por ir precedidas del símbolo asterisco (*).

5.1 Directiva ngIf

ngIf es una directiva estructural de Angular para implementar estructuras if en nuestra aplicación.

Con esta directiva condicionamos que un elemento html se muestre o no en función de que se cumpla la expresión que define, normalmente una propiedad o método. Su sintaxis básica es.

`*ngIf="expresión/propiedad/metodo"`

Vamos a crear un ejemplo para conocer su implementación. Creamos un nuevo componente denominado ejdirectivangif, en la consola del equipo:

`ng g c ejdirectivangif --spec false`

En el archivo plantilla, ejdirectivangif.component.html escribimos el siguiente código:

src/app/ejdirectivangif/ejdirectivangif.component.html

```
<div class="container">
  <label>Nombre y Apellidos</label>
  <input type="text"
    class="form-control"
    [(ngModel)]="nombre"
    placeholder="Complete su nombre y apellidos">
  <button type="submit" class="btn btn-primary"
    *ngIf="nombre">Enviar</button>
</div>
```

En el cual:

- ① Creamos un input con la directiva ngModel asociada a la propiedad nombre.
- ② Y creamos un botón que asociada la directiva ngIf a la propiedad nombre.

En la clase del componente en el archivo ejdirectivangif.component.ts simplemente declaramos la propiedad nombre:

```
src/app/ejdirectivangif/ejdirectivangif.component.ts
```

```
nombre: string;
```

Y en el archivo app.component.html del componente raíz sustituimos todo el código por el siguiente:

```
src/app/app.component.html
```

```
<div class="container" >
  <h3>Ejemplo de directiva ngIf</h3>
  <hr>
  <app-ejdirectivangif></app-ejdirectivangif>
</div>
```

Al refrescar nuestro navegador mostrará el campo sin que aparezca el botón de enviar, pero cuando comencemos a escribir, comprobamos como se mostrará el botón ya que la directiva localiza la existencia de la propiedad asociada.

Nombre y Apellidos

La directiva *ngIf se completa con el uso de else para resolver qué ocurre cuando no se cumpla la condición. Para ello se añade la etiqueta ng-template, con el contenido que necesitemos y se asocia una id de esa etiqueta a la palabra reservada else en la misma expresión de *ngIf.

Vamos a comprobarlo con nuestro ejemplo. Sustituimos todo el código de nuestro archivo ejdirecivangif.component.html por el siguiente:

src/app/ejdirecivangif/ejdirecivangif.component.html

```
<label>Nombre y Apellidos</label>
  <input type="text" class="form-control" [(ngModel)]="nombre"
placeholder="Complete su nombre y apellidos">
  <button type="submit" class="btn btn-primary" *ngIf="nombre; else
desactivado">Enviar</button> ①
  <ng-template #desactivado> ②
    <button type="submit"
      class="btn btn-primary"
      disabled>Enviar</button>
  </ng-template>
```

En el cual:

- ① Añadimos a la directiva *ngIf la palabra else en la expresión y la id de la etiqueta ng-template.
- ② Y en la etiqueta ng-template introducimos el contenido que se mostrará si no se cumple la condición declarada por *ngIf.

De esta manera, antes de comenzar a escribir en el input, se muestra el botón desactivado, ya que le hemos añadido el atributo disabled.



Pero comenzar a escribir, *ngIf reconoce la propiedad y muestra el botón original sin desactivar.

La directiva `*ngIf` también permite el empleo de métodos para realizar funcionalidades con una lógica definida en el componente.

Por ejemplo, vamos a sustituir el código de `ejdirecivangif.component.html` por el siguiente:

`src/app/ejdirecivangif/ejdirecivangif.component.html`

```
<h3>¿Cuál es la capital de España?</h3> ①
  <input type="text" class="form-control" [(ngModel)]="capital"> ②
  <p *ngIf="!capital; else correcto">Complete la pregunta</p> ③
  <ng-template #correcto>
    <h4 *ngIf="setResultado(); else incorrecto">¡Correcto!</h4>
  </ng-template>
  <ng-template #incorrecto>
    <h4>Lo siento, inténtelo de nuevo</h4>
  </ng-template>
```

En el cual:

- ① Realizamos una pregunta.
- ② Insertamos un input para responder.
- ③ Establecemos un primer texto que se mostrará si `*ngIf` no encuentra la propiedad `capital`. Si encuentra la propiedad `capital` el primer `else` nos remite a la etiqueta con id `correcto` en la que encadenamos otro `*ngIf` que utiliza el método `setResultado()` y si este método devuelve `false`, el siguiente `else` nos remite a la etiqueta con id `incorrecto`.

Ahora en el archivo `ejdirecivangif.component.ts` reemplazamos el siguiente código en la clase:

```
capital: string; ①

constructor() {
}

ngOnInit() {
}

setResultado(){
  return this.capital === "Madrid" ? true : false; ②
}
```

En el cual:

- ① Declaramos la propiedad capital.
- ② Y creamos el método setResultado(), que nos devuelve true si capital es (exactamente) igual a Madrid y false en caso contrario. Este método es el que se emplea en el segundo *ngIf en la plantilla.

Comprobamos el funcionamiento en el navegador:

¿Cual es la capital de España?

¡Correcto!

5.2 Directiva ngStyle

La directiva ngStyle, es una directiva de atributo para establecer de manera dinámica los estilos de un determinado elemento html en función de la expresión asociada, normalmente un método. Al ser de atributo, funciona como estos en las etiquetas html y no alteran el layout del DOM.

Su sintaxis es básicamente:

`[ngStyle]="{ expresión/propiedad/método }"`

De nuevo creamos un ejemplo, completando en la consola del equipo en el directorio raíz del proyecto:

`ng g c ejdirectivangstyle --spec false`

Comenzamos por incluir el siguiente código html en la plantilla ejdirectivangstyle.component.html:

src/app/ejdirectivangstyle/ejdirectivangstyle.component.html

```
<h4>Introduzca la puntuación del Alumno</h4>
<input type="number" class="form-control" [(ngModel)]="puntuacion"> ❶
<hr>
<h4 style="display: inline-block;">Puntuación obtenida:&nbsp; </h4>
<h4 style="display: inline-block;" ❷
  [ngStyle]="{color:setColor()}">{{puntuacion}}</h4>
```

En el cual:

- ❶ Creamos un input para introducir una puntuación.
- ❷ Y a continuación mostramos una línea que, con la directiva ngStyle, mostrará la propiedad puntuación con un estilo de color de la fuente en función del valor del método setColor().

Ahora en la clase del componente en el archivo ejdirectivangstyle.component.ts añadimos:

src/app/ejdirectivangstyle/ejdirectivangstyle.component.ts

```
puntuacion: number;

constructor() {
}

ngOnInit() {
}

setColor() {
  return this.puntuacion >= 5 ? 'green' : 'red'; ①
}
```

En el cual además de declarar la propiedad puntuación de tipo número, añadimos ① el método setColor().

Este método, devolverá 'green' si el valor de puntuación es mayor o igual a 5 o 'red' en caso contrario. Ambos valores son válidos para la propiedad color de CSS, que es la definida en la expresión de [ngStyle].

Finalmente y para comprobar, sustituimos el código de la plantilla del componente raíz app.component.html:

src/app/app.component.html

```
<div class="container" >
  <h3>Ejemplo de directiva ngStyle</h3>
  <hr>
  <app-ejdirectivangstyle></app-ejdirectivangstyle>
</div>
```

De esta manera, ahora se mostrará en el navegador:

Introduzca la puntuación del Alumno

Puntuación obtenida: 3

5.3 Directiva ngClass

ngClass es una directiva similar a ngStyle para establecer de manera dinámica las clases de un elemento HTML.

Veamos un ejemplo. Vamos a crear el componente ejdirectivangclass en la consola del equipo en el directorio del proyecto con:

```
ng g c ejdirectivangclass --spec false
```

En el archivo ejdirectivangclass.component.html de la plantilla añadimos:

```
src/app/ejdirectivangclass/ejdirectivangclass.component.html
```

```
<h4>Introduzca la puntuación del Alumno</h4>
<input type="text" class="form-control" [(ngModel)]="puntuacion">
<hr>
<div *ngIf="puntuacion"> ①

  <h4 *ngIf="puntuacion >= 0 && puntuacion <= 10; else aviso" ②
    [ngClass]="{ aprobado: puntuacion >= 5,
                suspenso: puntuacion < 5 }">
    Puntuación obtenida: {{ puntuacion }}</h4>
  <ng-template #aviso> ③
    <h4 *ngIf="puntuacion > 10" class="advertencia">
      Introduzca una puntuación menor a 10</h4>
    <h4 *ngIf="puntuacion < 0" class="advertencia">
      Introduzca una puntuación mayor o igual a 0</h4>
  </ng-template>
```

El código incorpora de nuevo un campo input para introducir una puntuación, y posteriormente:

① Se crea un div con la directiva ngIf para que se muestre en el momento que se introduzca una puntuación.

② Se establece un div con ngIf-else para que se muestre si la puntuación está entre cero y diez, estableciendo la clase CSS aprobado si es mayor o igual a cinco o la clase CSS suspenso si es menor que cinco.

③ En caso de que la puntuación no estuviera en el rango anterior, el ng-template proporciona dos opciones, si estuviera por encima de diez, avisa para introducir un valor menor y si estuviera por debajo de cero, avisa para introducir un valor mayor, siendo la clase CSS en ambos casos, advertencia.

Como hemos añadido clases CSS en la plantilla, las añadimos en nuestro archivo ejdirectivangclass.component.css de la siguiente manera:

```
src/app/ejdirectivangclass/ejdirectivangclass.component.css
```

```
.advertencia {  
  color: white;  
  background-color: orange;  
  padding: 10px;  
}  
.aprobado {  
  color: white;  
  background-color: green;  
  padding: 10px;  
}  
  
.suspense {  
  color: white;  
  background-color: red;  
  padding: 10px;  
}
```

Para concluir añadimos la propiedad a la clase del componente en ejdirectivangclass.component.ts:

```
src/app/ejdirectivangclass/ejdirectivangclass.component.ts
```

```
puntuacion: number;
```

Y la etiqueta del componente en app.component.html:

```
src/app/app.component.html
```

```
<div class="container" >  
  <h3>Ejemplo de directiva ngClass</h3>  
  <hr>  
  <app-ejdirectivangclass></app-ejdirectivangclass>  
</div>
```

Ahora podemos comprobar en el navegador, las cuatro posibilidades programadas:

Introduzca la puntuación del Alumno

Puntuación obtenida: 10

5.4 Directiva ngFor

Una de las directivas más empleadas en las aplicaciones Angular es la ngFor, dedicada a realizar iteraciones y presentar listados por pantalla. Se trata de una directiva de tipo estructural y su sintaxis es:

`*ngFor="let objeto/propiedad of objetos/propiedades"`

En esta sintaxis, se crea dentro del elemento html una variable local con let que recorrerá el array definido por of y proveniente del componente. Como buena práctica, el nombre de la variable local será singular y el del array plural.

Vamos a crear un ejemplo muy sencillo para esta directiva. Creamos el componente en el directorio del proyecto en la consola del equipo:

`ng g c ejdirectivangfor -spec`

En la clase del componente, dentro del archivo ejdirectivangfor.component.ts declaramos un simple array de cursos:

`src/app/ejdirectivangfor/ejdirectivangfor.component.ts`

```
cursos: string[];

constructor() {
  this.cursos = ['Angular', 'HTML', 'CSS'];
}

ngOnInit() {
}
```

Y en la plantilla, ejdirectivangfor.component.html, implementamos la directiva de la siguiente manera.

```
src/app/ejdirectivangfor/ejdirectivangfor.component.html
```

```
<h3>Cursos Disponibles</h3>
<ul>
  ① <li *ngFor="let curso of cursos"> ② <h4>{{curso}}</h4></li>
</ul>
```

En la cual:

- ① En una lista desordenada incluimos en el item la directiva ngFor para que itere el array cursos y cada iteración la almacene en la variable local.
- ② Dentro del ítem introducimos el valor de curso con la sintaxis de interpolación.

De esta manera, se generarán tantos elementos li como elementos tenga el array.

Sustituimos la etiqueta de app.component.html por la de este nuevo componente:

```
src/app/app.component.html
```

```
<div class="container" >
  <h3>Ejemplo de directiva ngFor</h3>
  <hr>
  <app-ejdirectivangfor></app-ejdirectivangfor>
</div>
```

Y ya podemos comprobar en el navegador como se muestra el listado:

Cursos Disponibles

- Angular
- HTML
- CSS

Otra de las formas habituales de empleo de la directiva ngFor es iterar un array de objetos JSON, la forma estandarizada de objetos JavaScript que utilizan las bases de datos NoSQL basadas en JavaScript.

La sintaxis para declarar un array de objetos JSON es:

`nombredelarray: Array<nombredelaclase>;`

Veamos otro ejemplo, con lo cual vamos a crear un nuevo componente, completando en la terminal en el directorio del proyecto:

`ng g c arrayobjetos`

En primer lugar, dentro del archivo `arrayobjetos.component.ts`, importamos la clase `Alumno`, que recordamos habíamos creado en el archivo `alumno.modelo.ts`:

```
src/app/arrayobjetos/arrayobjetos.component.ts
```

```
...  
import { Alumno } from 'app/modelos/alumno.modelo';  
...
```

A continuación en la clase del componente, declaramos el array de objetos y lo inicializamos con varios objetos de la clase `Alumno`:

```
...  
public alumnos: Array<Alumno> = [  
    {id: 1 , nombre: 'Juan', apellidos: 'Gutierrez', ciudad: 'Madrid'},  
    {id: 2 , nombre: 'Pedro', apellidos: 'Lopez', ciudad: 'Sevilla'}  
];  
...
```

Ahora, en la plantilla del componente, en el archivo arrayobjetos.component.html añadimos el código html de una tabla en la que iteramos en sus filas el array alumnos con la directiva ngFor:

src/app/arrayobjetos/arrayobjetos.component.html

```
<table class="table table-bordered table-striped tabla">
  <thead>
    <tr class="filters">
      <th>id</th>
      <th>Nombre</th>
      <th>Apellidos</th>
      <th>Ciudad</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let alumno of alumnos">
      <td>{{ alumno.id }}</td>
      <td>{{ alumno.nombre }}</td>
      <td>{{ alumno.apellidos }}</td>
      <td>{{ alumno.ciudad }}</td>
    </tr>
  </tbody>
</table>
```

Nos queda finalmente añadir la etiqueta en la plantilla del componente raíz app.component.html:

src/app/app.component.html

```
<div class="container" >
  <h3>Ejemplo de directiva ngFor con objetos</h3>
  <hr>
  <app-arrayobjetos></app-arrayobjetos>
</div>
```

Y comprobamos en el navegador:

id	Nombre	Apellidos	Ciudad
1	Juan	Gutierrez	Madrid
2	Pedro	Lopez	Sevilla

5.5 Directiva NgSwitch

NgSwitch es otra de las directivas ampliamente utilizadas a la hora de crear aplicaciones angular.

Vamos a ver un ejemplo, para lo cual como venimos haciendo, vamos a crear un nuevo componente en la consola del equipo en la raíz del proyecto:

```
ng g c ejdirectivangswitch
```

En primer lugar vamos a crear un array de objetos simples (sin modelo) con formato de documentos JSON en la clase del componente. Para ello, en el archivo ejdirectivangswitch.component.ts incorporamos en la clase:

```
src/app/ejdirectivangswitch/ejdirectivangswitch.component.ts
```

```
jugadores: any[] = [  
  { nombre: 'Earvin Jhonson', equipo: 'L.A. Lakers'},  
  { nombre: 'Larry Bird', equipo: 'Boston Celtics'},  
  { nombre: 'Michael Jordan', equipo: 'Chicago Bulls'}  
]
```

Ahora vamos a crear con la directiva ngFor un listado de los jugadores en el template, por tanto, sustituimos el código del archivo ejdirectivangswitch.component.html por:

```
src/app/ejdirectivangswitch/ejdirectivangswitch.component.html
```

```
<ul *ngFor="let jugador of jugadores">  
  <li>  
    <h4>{{ jugador.nombre }} | {{ jugador.equipo }}</h4>  
  </li>  
</ul>
```


Y sustituimos la etiqueta en el archivo app.component.html por la de este componente:

src/app/app.component.html

```
<div class="container" >
  <h3>Ejemplo de directiva ngSwitch</h3>
  <hr>
  <app-ejdirectivangswitch></app-ejdirectivangswitch></div>
```

Comprobamos en el navegador el listado:

- Kevin Jhonson | Los Ángeles Lakers
- Larry Bird | Boston Celtics
- Michael Jordan | Chicago Bulls

Hasta ahora todo el código responde al ejemplo que vimos en el apartado anterior, por lo que es el momento de mejorarlo con el uso de NgSwitch.

Para ello sustituimos el código de la template en ejdirectivangswitch.component.html:

src/app/ejdirectivangswitch/ejdirectivangswitch.component.ts

```
<div class="text-center">
  <ul *ngFor="let jugador of jugadores" [ngSwitch]="jugador.equipo"> ①
    <li *ngSwitchCase="'L.A. Lakers'"
class="lakers"><h3>{{jugador.nombre}}</h3><h3 class="equipo"> {{ ②
jugador.equipo}} </h3></li>
    <li *ngSwitchCase="'Boston Celtics'"
class="celtics"><h3>{{jugador.nombre}}</h3><h3 class="equipo"> {{
jugador.equipo}} </h3></li>
    <li *ngSwitchCase="'Chicago Bulls'"
class="bulls"><h3>{{jugador.nombre}}</h3><h3 class="equipo"> {{
jugador.equipo}} </h3></li>
  </ul>
</div>
```

En el cual:

- ① En la etiqueta `ul` hemos añadido la directiva `ngSwitch` entre corchetes y la asociamos a la propiedad `equipo` del objeto.
- ② Creamos tres etiquetas `li`, una por cada valor individual de la propiedad `equipo`, y en cada una de ellas igualamos `ngSwitchCase` con cada uno de los valores, para finalmente y también en cada `li` incorporar una clase `css` que se activará en función del valor de la propiedad `equipo`.

Nos queda añadir las clases `CSS`, para lo cual incorporamos al archivo `ejdirectivangswitch.component.css` el siguiente código `CSS`:

`src/app/ejdirectivangswitch/ejdirectivangswitch.component.css`

```
ul {
  text-align: center;
  list-style: none;
}

.lakers {
  max-width: 600px;
  height: 120px;
  background-image: url("../assets/LALakers.png");
  background-position-x: 10px;
  background-position-y: 10px;
  background-repeat: no-repeat;
  border-style: solid;
  border-width: 5px;
  border-color: #552582;
  margin-top: 30px;
  box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba(0, 0, 0, 0.19);
}

.celtics {
  max-width: 600px;
  height: 120px;
  background-image: url("../assets/BostonCeltics.png");
  background-position-x: 10px;
  background-position-y: 10px;
```

```

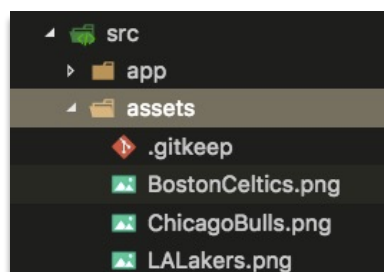
background-repeat: no-repeat;
border-style: solid;
border-width: 5px;
border-color: #008348;
margin-top: 30px;
box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba(0, 0, 0, 0.19);
}

.bulls {
  max-width: 600px;
  height: 120px;
  background-image: url("../assets/ChicagoBulls.png");
  background-position-x: 10px;
  background-position-y: 10px;
  background-repeat: no-repeat;
  border-style: solid;
  border-width: 5px;
  border-color: #CE1141;
  margin-top: 30px;
  box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba(0, 0, 0, 0.19);
}

.equipo {
  color: #707070;
}

```

Como hemos añadido unas imágenes de logo a las clases CSS, las tenemos que ubicar en el directorio assets:



Puedes copiar las imágenes de los archivos de este apartado o bien crear el ejemplo con tus equipos y deporte favorito.

Ahora nos vamos al navegador y comprobamos como cada jugador tiene los colores de su equipo y que ngSwitch establece la clase CSS en función de valor de la propiedad equipo de cada objeto.



Si además, ahora añadimos el siguiente objeto al array en la clase:

```
{ nombre: 'Kareem Abdul-Jabbar ', equipo: 'L.A. Lakers'}
```

Podremos comprobar en el navegador como se añade con los colores y logo de su equipo.

5.6 Creación de directivas propias

Además de las directivas proporcionadas por Angular, el *framework* permite la creación de nuevas directivas para nuestras funcionalidades.

Podemos crear el archivo de manera manual e incorporarlo a la aplicación con sus correspondiente importaciones o bien, de una manera más sencilla, utilizar la herramienta Angular CLI.

Para ello, en el directorio raíz del proyecto, completamos en la consola del equipo por ejemplo:

`ng generate directive menuitem`

De esta manera, Angular CLI, crea un archivo `menuitem.directive.ts` con el siguiente código:

```
src/app/ejdirectivangswitch/ejdirectivangswitch.component.ts
```

```
import { Directive } from '@angular/core'; ①

@Directive({
  selector: '[appMenuitem]' ②
})
export class MenuitemDirective { ③
  constructor() {}
}
```

En el cual se importa ① la clase `Directive` de Angular, se implementa ② el decorador `@Directive` con el metadato `selector` que determinará el nombre de nuestra directiva, y ③ se exporta una clase donde se define la lógica que realizará la directiva.

Dentro de la clase podemos escribir cualquier código de acuerdo a las necesidades de nuestra aplicación, y podemos ayudarnos para ello de una serie de decoradores que proporciona Angular.

En nuestro ejemplo, vamos a usar los decoradores HostListener, HostBinding con lo cual en primer lugar sustituimos la importación por la siguiente:

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

Y a continuación sustituimos el código de la clase por el siguiente:

```
@HostBinding('class.itemOrange') private mostrar: boolean = false; ①

@HostListener('mouseover') onOver() { ②
  this.mostrar = true;
}
@HostListener('mouseout') onOut() { ③
  this.mostrar = false;
}
```

En el cual :

- ① Utilizamos el decorador HostBinding para enlazar la clase CSS itemOrange con una propiedad booleana que denominamos mostrar, inicializada como false.
- ② Utilizamos el decorador HostListener para que cuando se realice un evento mouseover (aproximar el ratón) se ejecute el método onOver que cambia el valor de la propiedad mostrar a true.
- ③ Y de nuevo usamos el decorador HostListener para que con un evento mouseout (quitar el ratón) ejecute el método onOut que cambiará la propiedad mostrar a su estado original, false.

En resumen, estamos realizando una directiva que permitirá aplicar una clase CSS cuando se aproxime el ratón al elemento HTML donde se implemente.

Nuestra nueva directiva estará lista para usar, ya que además de crearla, Angular CLI modifica el archivo app.module.ts para que esté disponible en toda la aplicación.

Lo comprobamos accediendo al archivo:

src/app/app.module.ts

```
...
import { MenuItemDirective } from './menuItem.directive';
...
  MenuItemDirective
]
...
```

Como en nuestro ejemplo de directiva, vamos a implementar una clase CSS, la creamos en el archivo de estilos generales styles.css que se encuentra en el directorio app:

```
.item-orange {
  border-left-style: solid;
  border-width: 5px;
  border-left-color: orange;
  padding-left: 10px;
}
```

Y ahora ya podemos implementar esta clase a través de nuestra directiva con la funcionalidad de activación con el ratón.

Para comprobarlo, creamos un nuevo componente denominado ejmidirectiva en la consola del equipo:

```
ng g c ejmidirectiva --spec false
```

Y en el archivo ejmidirectiva.component.html de la template, vamos a poner un listado de elementos html a los que añadimos nuestra directiva con el nombre que definimos en su selector:

```
src/app/ejmidirectiva/ejmidirectiva.component.html
```

```
<div class="container">  
  <h1 appMenuitem>Título</h1>  
  <h1 appMenuitem>Descripción</h1>  
  <h1 appMenuitem>Precio</h1>  
</div>
```

Implementamos la etiqueta de este componente en el archivo raíz `app.component.html`:

```
src/app/app.component.html
```

```
<div class="container" >  
  <h3>Ejemplo de directiva ngFor con objetos</h3>  
  <hr>  
  <app-ejmidirectiva></app-ejmidirectiva>  
</div>
```

Y comprobamos finalmente en el navegador, el funcionamiento de la directiva:



Como la directiva se puede emplear en cualquier parte de la aplicación, es una buena fórmula para reutilizar el código, y como hemos visto, su desarrollo no es demasiado complicado.

5.7 Utilización de @Input entre componentes

Angular permite compartir a través del decorador @Input, propiedades y valores entre componentes anidados.

Vamos a ver con un ejemplo como se incorpora este decorador. En primer lugar creamos un componente padre desde la consola en el directorio raíz del proyecto con:

```
ng g c padre --spec false
```

Y otro componente hijo:

```
ng g c hijo --spec false
```

Nos vamos a la plantilla del componente padre.component.html y escribimos el siguiente código:

```
src/app/padre/padre.component.html
```

```
<h2>Componente Padre</h2>
<div style=" border-style: solid; border-width: 2px; border-color: orange;
padding: 10px">
</div>
```

E incluimos su etiqueta en la plantilla del componente raíz app.component.html:

```
src/app/app.component.html
```

```
<div class="container" >
  <h3>Ejemplo de uso de @Input</h3>
  <hr>
  <app-padre></app-padre>
</div>
```

Comprobamos en el navegador su correcto funcionamiento:

Componente Padre

Ahora vamos a la clase del componente hijo en el archivo `hijo.component.ts` y creamos la propiedad `aviso` con un *string* de ejemplo:

`src/app/hijo/hijo.component.ts`

```
public aviso: string = 'Este texto proviene del componente hijo';
```

Completamos un sencillo código en el archivo `hijo.component.html` para mostrar el valor de la propiedad `aviso`:

`src/app/hijo/hijo.component.html`

```
<h4>{{aviso}}</h4>
```

Y a continuación, vamos a modificar la plantilla del componente padre en el archivo `padre.component.html` para incluir la etiqueta del componente hijo:

`src/app/padre/padre.component.html`

```
<h2>Componente Padre</h2>
<div style=" border-style: solid; border-width: 2px; border-color: orange;
padding: 10px">
  <app-hijo></app-hijo>
</div>
```

Comprobamos finalmente en el navegador:

Componente Padre

Este texto proviene del componente hijo

Hasta ahora hemos empleado las técnicas de Angular sin mayor novedad, pero podemos incluir valores en las propiedades del hijo a través del padre, es decir compartir las propiedades entre dos componentes.

Para ello emplearemos el decorador Input de la siguiente manera. En primer lugar modificamos el archivo hijo.component.ts para que la propiedad aviso pueda ser compartida.

src/app/hijo/hijo.component.ts

```
import { Component, OnInit, Input } from '@angular/core'; ①

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

  @Input() aviso: string; ②

  constructor() {}

  ngOnInit() {
  }

}
```

En el cual:

- ① Importamos la clase Input del core de Angular,
- ② Y aplicamos el decorador @Input() a la propiedad aviso, sin inicializarla.

Ahora en el componente padre, en el archivo padre.component.ts creamos una nueva propiedad con un valor:

```
src/app/padre/padre.component.ts
```

```
valorPadre: string = 'Este texto se inyecta desde el padre';
```

Y podemos usar la propiedad del hijo para usarla en su plantilla con un valor proveniente del padre. Para ello en el archivo padre.component.html modificamos la etiqueta del hijo para añadir esta sintaxis:

```
src/app/padre/padre.component.html
```

```
<app-hijo [aviso]="valorPadre"></app-hijo>
```

Si ahora regresamos al navegador, podemos comprobar como en la plantilla del componente hijo y la propiedad aviso, hemos inyectado el valor de una propiedad, valorPadre, del componente padre.

Componente Padre

Este texto se inyecta desde el padre

Incluso podemos hacerlo más complejo introduciendo un array de valores desde el padre a la plantilla del hijo. Por ejemplo, en el archivo del componente padre.component.ts, cambiamos la propiedad por este array:

```
src/app/padre/padre.component.ts
```

```
aviso: string[] = ['Aviso 1', 'Aviso 2', 'Aviso 3'];
```

Y ahora en la plantilla en el archivo padre.component.html hacemos uso de ngFor para iterar avisos e introducirlos en la propiedad aviso del component hijo de la siguiente forma:

```
src/app/padre/padre.component.html
```

```
<app-hijo *ngFor="let avisop of avisos" [aviso]="avisop"></app-hijo>
```

Con lo cual en el navegador se mostrará:

Componente Padre

Aviso 1
Aviso 2
Aviso 3

5.8 Utilización de @Output entre componentes

Otro de los decoradores a emplear en Angular entre componentes es Output. Volvamos al ejemplo del apartado anterior y antes de utilizar Output vamos a crear un método para marcar los avisos leídos.

Para ello en el componente hijo, modificamos el archivo hijo.component.html sustituyendo el código por el siguiente:

src/app/hijo/hijo.component.html

```
<div style=" border-style: solid; border-width: 2px; border-color: blue;
padding: 20px; margin: 10px">
  <h4 [ngClass]="{tachado: leído == true}" style="display:      ❶
inline">{{aviso}}</h4>
  <button *ngIf="leído == false" class="btn btn-success float-md-right"
(click)="marcar()">Marcar como Leído</button>
  <button *ngIf="leído == true" class="btn btn-danger float-md-right" ❷
(click)="marcar()">Desmarcar como Leído</button>
</div>
```

En el cual:

❶ Añadimos al aviso una directiva ngClass para que aplique la clase tachado si el valor de leído es true.

❷ Y añadimos dos botones que se mostrarán o no en función de que la propiedad leído sea verdadera o falsa, con un evento clic que desencadena el método marcar() para que modifique el valor de esa propiedad.

A continuación vamos a crear la propiedad leído y el método marcar en el archivo hijo.component.ts:

src/app/hijo/hijo.component.ts

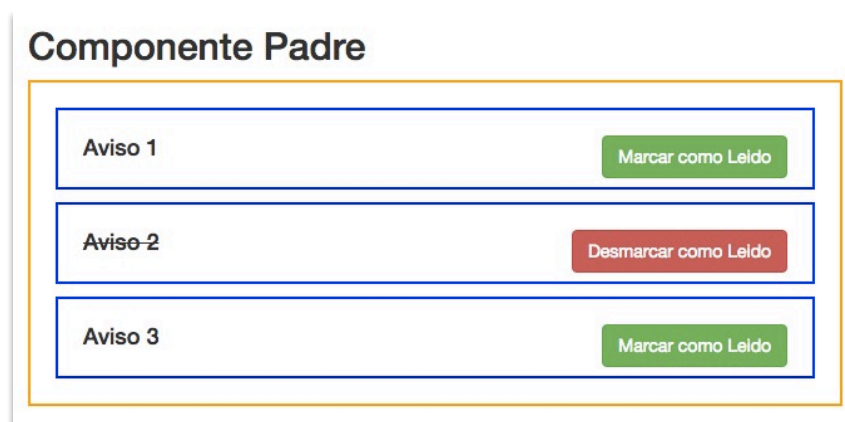
```
...  
@Input() aviso: string;  
  
leido: boolean = false;  
  
marcar(event) {  
  this.leido = !this.leido;  
}  
...
```

Y nos queda añadir la clase tachado en el archivo hijo.component.css:

src/app/hijo/hijo.component.css

```
.tachado {  
  text-decoration: line-through;  
}
```

Ahora podemos en el navegador como al pulsar en cada botón, el aviso se marca y desmarca como tachado:



Pues bien, la funcionalidad Output permite desencadenar un método del componente padre a partir de un método del componente hijo y pasar datos como parámetros.

En primer lugar vamos a modificar el código del componente hijo en su archivo hijo.component.ts, sustituimos la primera línea de importación por:

```
src/app/hijo/hijo.component.ts
```

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

Añadimos el decorador @Output en la clase creando el método mensajeMarcado:

```
...  
@Output() mensajeMarcado = new EventEmitter();  
...
```

Para finalizar en este archivo, creamos en la clase un nuevo método detectar:

```
...  
detectar(event) {  
    this.mensaje = this.aviso;  
    this.mensajeMarcado.emit(this.mensaje);  
}  
...
```

Que detectará qué aviso se ha marcado y los emite como parámetro mensaje del método mensajeMarcado.

A continuación en la template del componente hijo, hijo.component.html, añadimos al botón de marcado de aviso y el método detectar, sustituyendo el código del botón por el siguiente:

src/app/hijo/hijo.component.html

```
...  
<button *ngIf="leido == false" class="btn btn-success pull-right"  
(click)="marcar() ; detectar($event)">Marcar como Leído</button>  
...
```

Pasamos al componente padre ahora. En su plantilla del archivo padre.component.html modificamos el código de la etiqueta del componente hijo para añadir el método que hemos emitido a través de @Output de la siguiente forma:

src/app/padre/padre.component.html

```
...  
<app-hijo *ngFor="let avisop of avisos" [avisop]="avisop"  
(mensajeMarcado)="mostrarMensaje($event)"></app-hijo>  
...
```

En el que vemos como el método del componente hijo se asocia al método mostrarMensaje.

En este archivo padre.component.html, también añadimos una línea para mostrar un mensaje antes del cierre del div:

```
...  
<h3 style="text-align:center"> {{ texto }}</h3>  
</div>
```

Por último nos queda modificar el componente padre en el archivo padre.component.ts. En primer lugar creamos la propiedad texto que se mostrará en el mensaje añadiendo:

```
src/app/padre/padre.component.html
```

```
...  
texto: string;  
...
```

Y a continuación añadimos el método mostrarMensaje que recibe el parámetro mensaje del componente hijo a través del método mensajeMarcado:

```
mostrarMensaje(event) {  
  this.texto = event + ' marcado como leído';  
}
```

Finalizado el código, podemos comprobar en el navegador como al marcar cualquier aviso, se muestra el mensaje con el aviso 'marcado como leído' gracias al enlace de métodos mediante Output entre diferentes componentes.



6 Pipes

Los *pipes* son elementos de código de Angular que permiten modificar la salida por pantalla o vista de los datos que empleamos en la aplicación.

En el apartado de componentes ya vimos un primer ejemplo de *pipe*, con el cual modificábamos el formato de salida de una fecha. También pudimos comprobar cómo se emplea el símbolo *pipe* (|) que da nombre a esta funcionalidad de Angular.

Por tanto la implementación de un *pipe* se lleva a cabo mediante la sintaxis:

```
{{ dato | nombredelpipe:'opciones' }}
```

Vamos a conocer a continuación los principales *pipes* de Angular.

6.1 Pipe Date

Este *pipe* ya lo conocemos de la creación de componentes. Recordemos como en la clase del componente `fechaactual.component.ts` habíamos declarado la siguiente variable:

```
src/app/fechaactual/fechaactual.component.ts
```

```
...  
hoy: number = new Date();  
...
```

y en su plantilla habíamos completado:

```
src/app/fechaactual/fechaactual.component.html
```

```
...  
<p> Sevilla, {{ hoy | date:'d/M/y H:m'}}</p>  
...
```

Lo cual quiere decir que, transformará el formato de objeto de fecha JavaScript (número de milisegundos desde la era epoch) o el formato de fecha ISO, a día/mes/año hora:minutos.

Para poder configurar según nuestras necesidades el formato de fecha, los principales símbolos a emplear en este pipe son los siguientes:

Símbolo	Componente
y	año
M	mes
d	día
h	hora en formato 12 horas
H	hora en formato 24 horas
m	minutos
s	segundos

Si, por ejemplo modificamos en la plantilla nuestro código, por este:

```
src/app/fechaactual/fechaactual.component.html
```

```
...  
<p> Sevilla, {{ hoy | date:'d-M-y'}} a las {{ hoy | date:'H:m Z'}}</p>...
```

Obtendremos este nuevo formato:

Sevilla, 15-6-2017 a las 17:45 GMT+2

6.2 Pipe Uppercase y Lowercase

El *pipe* Uppercase es un sencillo filtro que como su nombre indica, convierte los datos a mayúsculas. Una de las ventajas de los *pipe* en Angular, es que se pueden encadenar en la misma expresión.

Por ejemplo, en el caso anterior, si modificamos esta línea en el template de fechaactual.component.html por la siguiente:

```
src/app/fechaactual/fechaactual.component.html
```

```
...  
<p> {{ ciudad | uppercase}}, {{ hoy | date:'d-M-y' | uppercase}} a las {{ hoy |  
date:'H:m Z'}}</p>  
...
```

Y en la clase añadimos:

```
src/app/fechaactual/fechaactual.component.ts
```

```
...  
ciudad: string = "Sevilla";  
...
```

Comprobamos en el navegador como el string Sevilla pasa a presentarse con todas sus letras en mayúsculas:

```
SEVILLA, 15-6-2017 a las 17:54 GMT+2
```

De manera similar a uppercase, disponemos también del *pipe* lowercase que transformará todos caracteres a minúsculas.

6.3 Pipe Decimal

Con este pipe definimos el formato de salida de un número con decimales.

Por ejemplo, en el componente fechaactual, podemos sustituir el código del template por el siguiente:

```
src/app/fechaactual/fechaactual.component.html
```

```
...  
<p> El resultado es {{ resultado | number:'2.2-2'}}</p>  
...
```

Donde el primer dígito representa el mínimo número de enteros, el segundo dígito, el número mínimo de decimales y el tercer dígito el máximo número de decimales.

Añadimos ahora en la clase:

```
src/app/fechaactual/fechaactual.component.ts
```

```
...  
resultado: number = 1.148;  
...
```

Y comprobamos en el navegador:

El resultado es 01.15

6.4 Pipe Currency

Currency es un *pipe* para añadir el símbolo de moneda a los valores de la aplicación.

En el componente anterior, `fechaactual.component.ts`, podemos sustituir el código del template por el siguiente:

```
src/app/fechaactual/fechaactual.component.html
```

```
...  
<p>La cotización actual del dólar es de {{ dolareuro | currency:'EUR':true}}</p>  
...
```

Donde al pipe `currency` se añade el código ISO de la moneda y `true` para que muestre su símbolo.

Y en la clase añadir la propiedad:

```
src/app/fechaactual/fechaactual.component.ts
```

```
...  
dolareuro: number = 0.94;  
...
```

Lo cual mostrará por pantalla:

La cotización del dolar actual es de €0.94

El problema de este pipe, es que presenta un formato anglosajón en el que el símbolo precede al valor, por lo que para aplicaciones dedicadas a países de la Unión Europea tiene poca utilidad.

Veremos cómo solucionar este inconveniente más adelante.

6.5 Pipe i18nSelect

Este *pipe* nos permite transformar un valor en otro dependiendo de su contenido. Vamos a ver un ejemplo para usar un encabezado en función del sexo de la persona.

Incluimos en el código del componente `fechaactual.component.ts` las siguientes líneas en la clase:

`src/app/fechaactual/fechaactual.component.ts`

```
...
nombre: string = 'Laura';
sexo: string = 'mujer';
encabezamiento: any = { 'hombre': 'Estimado', 'mujer': 'Estimada' }
...
```

Y en su plantilla añadimos el siguiente elemento:

`src/app/fechaactual/fechaactual.component.html`

```
...
<p> {{ sexo | i18nSelect: encabezamiento }} {{ nombre }} </p>
...
```

Ahora podemos comprobar en el navegador, que si el "sexo" es "mujer", transforma el valor "encabezamiento" en "Estimada".

6.6 Creación de *pipes*

Angular permite también crear *pipes* propios para resolver cualquier necesidad de nuestra aplicación.

Vamos a ver como se crean con un ejemplo para resolver el símbolo euro en nuestros valores de moneda.

Podemos crear un *pipe* de nombre euro con Angular CLI, para lo cual en la consola del equipo y en el directorio del proyecto, tecleamos:

```
ng generate pipe euro
```

De esta manera, en el directorio app se ha creado el archivo euro.pipe.ts con la sintaxis necesaria para que solamente tengamos que crear el código de transformación que realizará el *pipe*.

En nuestro caso, añadimos a la clase que exportará el siguiente código:

```
src/app/euro.pipe.ts
```

```
...
transform(value: any, args?: any): any {
  const euro = value + ' €'; ①
  return euro;                ②
}
...
```

En el cual el método transform, utilizado por los *pipes*, recibe el parámetro value y:

① Creamos una constante de nombre euro que será igual a la concatenación de value más el string de espacio y el símbolo de euro, donde value es el valor que recibirá el *pipe*.

② Y devuelve el valor del string euro.

Comprobamos como nuestro *pipe* está importado y añadido al array correspondiente en el archivo `app.module.ts`, labor que ha realizado automáticamente Angular CLI:

`src/app/app.module.ts`

```
...  
import { EuroPipe } from './euro.pipe';  
...  
EuroPipe  
],  
...
```

Por tanto, nuestro pipe ya está creado y listo para poder usar en toda la aplicación Angular.

Regresamos a nuestro componente, `fechaactual.component.ts`, y de nuevo en el código del template añadimos la línea del ejemplo anterior, pero ahora con nuestro pipe euro:

`src/app/fechaactual/fechaactual.component.html`

```
...  
<p>La cotización actual del dólar es de {{ dolareuro | euro }}</p>  
...
```

Añadimos la propiedad `dolareuro` con su valor a la clase del componente:

`src/app/fechaactual/fechaactual.component.ts`

```
...  
dolareuro: number = 0.94;  
...
```

Y ahora comprobamos en el navegador como se aplica el filtro con el formato que buscamos:

La cotización actual del dólar es de 0.94 €

7 Servicios e Inyección de dependencias

Los servicios son una de las características fundamentales de la arquitectura Angular, ya que permiten centralizar el uso de código común a muchos componentes, e incorporarlo a estos mediante la inyección de dependencias.

Normalmente son empleados para el tratamiento de los datos de la aplicación y la comunicación con APIs de servidores de bases de datos, utilizando para ello librerías angular, como por ejemplo, la Http.

Como normalmente abastecen a los componentes de la aplicación de datos, son denominados también providers, y de hecho, es este el nombre del array con el que se implementan en el módulo raíz de la aplicación, recordemos `app.module.ts`.

Una vez que hemos visto las principales características de Angular, llega el momento de crear un nuevo proyecto de aplicación en el que afianzar lo aprendido e implementar de manera práctica el resto de funcionalidades del *framework*.

Por ello, antes de continuar con los servicios en Angular, vamos a crear una nueva aplicación.

Para ello, creamos en nuestro equipo un nuevo directorio para este nuevo proyecto denominado `appcompras` y accedemos a él desde la consola del equipo, en la que tecleamos:

```
ng new appCompras
```

Y tendremos nuestro nuevo proyecto listo para comenzar a desarrollarlo.

7.1 Creación de servicios e inyección en componentes

Comenzamos en nuestro proyecto, añadiendo un primer servicio, para lo cual dentro del directorio app creamos otro directorio llamado servicios para alojar estos archivos.

Además de crearlos de manera manual, introduciendo el código que a continuación veremos, los servicios en Angular se pueden crear de manera más eficiente mediante la herramienta Angular CLI.

Para ello, desde el directorio raíz del proyecto tecleamos en la consola del equipo:

```
ng generate service servicios/proveedores
```

De esta manera se crea el archivo `proveedores.service.ts` en el directorio servicios de nuestra aplicación, con el siguiente código por defecto:

```
src/app/servicios/proveedores.service.ts
```

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

@Injectable()
export class ProveedoresService {

  constructor() {}

}
```

Vemos como el archivo importa `Injectable` del `core` de Angular, lo utiliza en el decorador `@Injectable()` y posteriormente crea una clase `ProveedoresService`.

A diferencia de los componentes, Angular CLI no importa e implementa el nuevo servicio en el archivo `app.module.ts`, así que en este archivo debemos añadir:

```
src/app/app.module.ts
```

```
...  
import { ProveedoresService } from 'app/servicios/proveedores.service';  
...  
  
...  
  providers: [ProveedoresService],  
...  

```

Ahora el servicio ya está listo. Los servicios se inyectan en los componentes a través de métodos así que vamos a comprobarlo creando un sencillo método.

En la clase del archivo `proveedores.service.ts` escribimos el siguiente código:

```
src/app/servicios/proveedores.service.ts
```

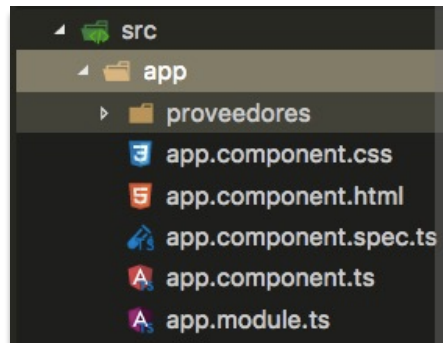
```
getProveedores(){  
  return 'Mensaje desde el servicio';  
}
```

Como vemos en principio, simplemente hemos añadido a la clase un método que devuelve un mensaje.

Usaremos el decorador `@Injectable` para realizar la inyección de dependencias.

Vamos a crear un primer grupo de componentes en nuestra aplicación empresarial de compras para tener el código de la aplicación bien organizado.

Para ello creamos un directorio denominado proveedores en app.



Y creamos un nuevo componente en la consola del equipo en la ruta al directorio recién creado:

```
ng g c proveedores/proveedores - -spec false
```

La forma de implementar el servicio en este componente es mediante su inicialización en el constructor y la llamada a su método.

Así que comenzamos añadiendo en el archivo `proveedores.component.service.ts`, el import de nuestro servicio:

```
src/app/proveedores/proveedores.component.ts
```

```
...  
import { ProveedoresService } from '../servicios/proveedores.service';  
...
```

Y a continuación en la clase añadimos el siguiente código

```
mensaje: string; ①  
  
constructor( private proveedoresService: ProveedoresService) {} ②
```



```
ngOnInit() {  
    this.mensaje = this.proveedoresService.getProveedores();  
}
```

En el cual:

- ① Creamos la propiedad mensaje.
- ② Se implementa en el constructor un parámetro para igualar a la clase del servicio.
- ③ Y dentro de ngOnInit, es decir cuando Angular cargue el componente, se iguala la propiedad mensaje a una propiedad que llama al método getProveedores del servicio.

A continuación, en la plantilla del componente, proveedores.component.html, simplemente añadimos:

```
src/app/proveedores/proveedores.component.html
```

```
<p>{{ mensaje }}</p>
```

Y añadimos la etiqueta al componente raíz app.module.html:

```
src/app/app.module.html
```

```
<div class="container">  
    <app-proveedores></app-proveedores>  
</div>
```

Como estamos en un nuevo proyecto, añadimos en el head del archivo index.html los CDN de Google Fonts, Bootstrap y JQuery (los puedes copiar de los archivos de este apartado):

src/index.html

```
<link href="https://fonts.googleapis.com/css?family=Open+Sans"
rel="stylesheet">
<link ... resto de CDNs...
...
```

Iniciamos el servidor en la consola desde la ruta del proyecto con:

`ng serve`

Y comprobamos el sencillo mensaje en el navegador:

Mensaje desde el servicio

Este sencillo ejemplo, simplemente nos ha servido para comprender como se implementa un servicio pero vemos que no ha aportado nada que no pueda hacer un componente. Lo normal es que el servicio suministre datos a uno o varios componentes.

Vamos a verlo con un ejemplo en el que añadimos un array de objetos en el servicio como fuente de datos para los componentes.

A continuación, modificamos nuestro servicio en el archivo proveedores.service.ts por el siguiente:

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

@Injectable()
export class ProveedoresService {

  proveedores: any = [ ①
    {
      nombre: 'Telefónica',
      cif: 'B12345678',
      direccion: 'Paseo de la Castellana, 100',
      cp: '28.010',
      localidad: 'Madrid',
      provincia: 'Madrid',
      telefono: 911111111,
      email: 'info@telefonica.com',
      contacto: 'Juan Pérez'
    },
    {
      nombre: 'Iberdrola',
      cif: 'B87654321',
      direccion: 'Príncipe de Vergara, 200',
      cp: '28.015',
      localidad: 'Madrid',
      provincia: 'Madrid',
      telefono: 922222222,
      email: 'info@iberdrola.com',
      contacto: 'Laura Martínez'
    }
  ]
  getProveedores(){
    return this.proveedores; ②
  }
}
```

En el cual:

① Creamos un array de objetos proveedores y le introducimos dos objetos JSON a ese array.

② Y los devolvemos en el método `getProveedores`.

Con este código conseguimos tener a disposición de cualquier componente de nuestra aplicación, la colección de documentos JSON Proveedores.

Vamos a modificar ahora en el componente `proveedores.component.ts`, para inyectar el servicio.

En primer lugar, dentro de la clase del componente, creamos una propiedad `proveedores` que recibirá los datos del servicio (eliminamos la anterior propiedad `mensaje`):

```
src/app/proveedores/proveedores.component.ts
```

```
...  
proveedores: any;  
...
```

Y dentro de `ngOnInit`, le decimos a Angular que cuando cargue el componente, iguale estos `proveedores` al método `getProveedores` del servicio, que recordamos devuelve la colección de proveedores

```
ngOnInit() {  
    this.proveedores = this.proveedoresService.getProveedores();  
}  
}
```

Ahora modificamos la plantilla para obtener una tabla con algunos de los campos de cada proveedor, en el archivo `proveedores.component.html`:

```
<h3>Listado de Proveedores</h3>
<table class="table table-bordered table-striped tabla"
  style="margin-top: 40px;">
  <thead>
    <tr class="filters">
      <th>Nombre</th>
      <th>Correo Electrónico</th>
      <th>Telefono</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let proveedor of proveedores">
      <td>{{ proveedor.nombre }}</td>
      <td><a href="mailto:{{ proveedor.email }}">{{ proveedor.email }}</a></td>
      <td>{{ proveedor.telefono }}</td>
    </tr>
  </tbody>
</table>
```

Y obtendremos en el navegador el resultado:

Listado de Proveedores		
Nombre	Correo Electrónico	Telefono
Telefónica	info@telefonica.com	911111111
Iberdrola	info@iberdrola.com	922222222

De esta manera tenemos centralizado los datos de proveedores y los podremos inyectar tanto a este como a otros componentes.

8 Routing

Otra de las características de Angular es la posibilidad de establecer un sistema de rutas de la dirección URL de la aplicación para poder estructurar la navegación entre los elementos de la misma, de forma que se establezca una adecuada experiencia de usuario.

8.1 Configuración del routing de una aplicación

Vamos a conocer, de manera práctica, como implementar las funciones de routing en nuestra aplicación.

Para ello vamos a crear una sencilla estructura de navegación con un componente Inicio, que será el que se muestre en la raíz de la URL, y el componente Proveedores que creamos en el apartado anterior.

Para crear el componente Inicio, como hacemos habitualmente, usamos Angular CLI en la consola en el directorio del proyecto, y completamos:

```
ng g c inicio - -spec false
```

En este nuevo componente, simplemente modificamos la plantilla en el archivo inicio.component.html para añadir:

```
src/app/inicio/inicio.component.html
```

```
<h2>Bienvenido a nuestra aplicación Compras App</h2>
```

Ahora que ya tenemos dos componentes, vamos a configurar el routing para poder navegar entre ellos.

En primer lugar necesitamos modificar el archivo del módulo de la aplicación en app.module.ts al cual añadimos en primer lugar la importación de:

```
src/app/app.module.ts
```

```
...  
import { Routes, RouterModule } from '@angular/router';...
```

Y antes del decorador @NgModule añadimos la siguiente constante:

```
...  
const routes: Routes = [  
  { path: '', component: InicioComponent },  
  { path: 'proveedores', component: ProveedoresComponent }  
];  
...
```

Esta constante de la clase Routes, contiene un array de objetos, con las rutas para cada componente. En nuestro ejemplo la url raíz, es decir /, se asocia al componente Inicio, y la url proveedores, es decir /proveedores, se asocia al componente Proveedores.

Finalmente, dentro de los imports del decorador @NgModule añadimos:

```
...  
imports: [  
  BrowserModule,  
  RouterModule.forRoot(routes)  
],  
...
```

En el siguiente paso, debemos modificar nuestra plantilla del componente raíz en el archivo app.component.html, para añadir la etiqueta router-outlet.

Esta etiqueta, tiene asociada una directiva de Angular para cargar el componente que, de manera dinámica, establece el routing según cada url.

Para ello, sustituimos el código de app.component.html por el siguiente:

src/app/app.component.html

```
<div class="container">
  <router-outlet></router-outlet>
</div>
```

Si ahora vamos al navegador, y tecleamos la url raíz, localhost:4200, la aplicación nos muestra la vista del componente inicio:

Bienvenido a nuestra aplicación de Compras App

Pero si sobrescribimos la url por localhost:4200/proveedores nos mostrará la vista del componente proveedores.

También podemos añadir una máscara para que cuando se teclee una ruta que no exista, redirija al componente Inicio (o a una vista de plantilla 404 en caso necesario).

Para ello, en el archivo app.module.ts modificamos la constante routes de la siguiente forma:

src/app/app.module.ts

```
const routes: Routes = [
  { path: '', component: InicioComponent },
  { path: 'proveedores', component: ProveedoresComponent },
  { path: '**', component: InicioComponent }
];
```

Podemos comprobar ahora como, si introducimos en el navegador cualquier ruta desconocida, por ejemplo localhost:4200/rutainexistente, nos redirigirá al inicio.

8.2 Navegación mediante links

Mediante el routing podemos establecer un sistema de navegación mediante links que proporciona dos ventajas fundamentales:

- Menús. Este sistema, permite establecer links en menús de navegación que permite a los usuarios familiarizarse rápidamente con los componentes de la aplicación.
- SPA. Los links llaman a los componentes sin refrescar la web, lo que permite aumentar la velocidad de ejecución de la aplicación y conservar los estados de la misma, sobre todo en la gestión de datos.

Para comprobar la navegación por links vamos a crear una barra de navegación de tipo header. Para ello creamos un nuevo componente con el mismo nombre, en la consola del equipo:

`ng g c header - - spec false`

En el archivo de la vista, header.component.html, escribimos el siguiente código de una barra de navegación horizontal básica de Bootstrap:

src/app/inicio/header.component.html

```
<nav class="navbar navbar-light bg-faded rounded navbar-togglerable-md">
  <button class="navbar-toggler navbar-toggler-right" type="button" data-
toggle="collapse" data-target="#containerNavbar" aria-
controls="containerNavbar" aria-expanded="false" aria-label="Toggle
navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <a class="navbar-brand" href="#">Compras App</a>

  <div class="collapse navbar-collapse" id="containerNavbar">
    <ul class="navbar-nav mr-auto w-100 justify-content-end">
      <li class="nav-item" routerLinkActive="active">
        <a class="nav-link" routerLink="/">Inicio </a> ①
      </li>
    </ul>
  </div>
</nav>
```

```

    <li class="nav-item">
      <a class="nav-link" routerLink="/proveedores">Proveedores </a> ②
    </li>
  </ul>
</div>
</nav>

```

Donde hemos añadido ① en el link del menú denominado inicio, la referencia `routerLink="/"`, y ② en el link del menú denominado Proveedores la referencia `routerLink="/proveedores"`.

Vamos a continuación a incorporar nuestro nuevo componente a la vista raíz de la aplicación `app.component.html`, para que siempre se muestre justo encima del `div container`:

`src/app/app.component.html`

```

<app-header></app-header>
<div class="container">
  <router-outlet></router-outlet>
</div>

```

Y comprobamos en el navegador, como la barra de navegación funciona de manera fluida entre las dos vistas sin refrescar la página.



Si queremos mejorar aún más los links de navegación, podemos establecer la clase `bootstrap active` de manera dinámica según la vista en la que nos encontremos.

Para ello sustituimos de nuevo los links de la barra en header.component.html por:

src/app/header/header.component.html

```
...
<li routerLinkActive="active"
  [routerLinkActiveOptions]="{exact:true}"><a routerLink="/" >Inicio
</a></li>
<li routerLinkActive="active"><a
routerLink="/proveedores">Proveedores</a></li>
...
```

Donde establecemos la directiva routerLinkActive con la clase CSS que queremos activar y en el caso del link de la url raíz, añadimos routerLinkActiveOptions con el valor exact: true para que solo se active la clase cuando efectivamente estemos en esa ruta.

También añadimos a una clase css al componente header para que establezca un margen inferior que separe la barra de navegación del componente que tenga debajo. Añadimos al archivo header.component.css:

src/app/header/header.component.css

```
nav {
  margin-bottom: 40px;
}
```

y obtendremos en el navegador el resultado deseado.

Compras App

Inicio Proveedores

Listado de Proveedores

Nombre	Correo Electrónico	Telefono
Telefónica	info@telefonica.com	911111111
Iberdrola	info@iberdrola.com	922222222

A medida que avancemos en nuestro proyecto conoceremos más opciones de enrutado en su contexto, especialmente el *routing* programático que emplearemos para redireccionar a diferentes páginas desde la lógica del componente.

9 Formularios

Los formularios son los elementos clave de una aplicación, ya que son la forma que tiene el usuario de introducir datos para realizar las funciones más importantes de la misma, como por ejemplo, crear registros o realizar búsquedas.

Angular permite implementar formularios mediante dos modalidades, Template-Driven y Reactive. En esencia, en la primera modalidad, Angular realiza la lógica de captura de datos y validación del lado del template HTML, mientras que en la segunda opción el formulario es gestionado de manera programática en la clase JavaScript del componente.

9.1 Creación de formularios Template-Driven

Vamos a crear un formulario con esta técnica. Para ello en primer lugar, creamos un nuevo componente para añadir nuevos registros de proveedores a nuestra aplicación, completando en la consola:

```
ng g c proveedores/addprovee - - spec false
```

Comenzamos añadiendo en la plantilla del componente, archivo `addprovee.component.html` el código html de un formulario con los campos que necesitamos:

```
src/app/proveedores/addprovee/addprovee.component.html
```

```
<div class="row">
  <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset 2">
    <h2>Añadir nuevo proveedor</h2>
    <form>
      <div class="form-group">
        <label for="nombre">Nombre</label>
        <input type="text"
          class="form-control"
          id="nombre">
      </div>
      <div class="form-group">
        <label for="cif">C.I.F.</label>
        <input type="text">
      </div>
    </form>
  </div>
</div>
```

```

        class="form-control"
        id="cif">
</div>
<div class="form-group">
  <label for="direccion">Dirección</label>
  <input type="text"
    class="form-control"
    id="direccion">
</div>
<div class="form-group">
  <label for="cp">Código Postal</label>
  <input type="text"
    class="form-control"
    id="cp">
</div>
<div class="form-group">
  <label for="localidad">Localidad</label>
  <input type="text"
    class="form-control"
    id="localidad">
</div>
<div class="form-group">
  <label for="provincia">Provincia</label>
  <input type="text"
    class="form-control"
    id="provincia">
</div>
<div class="form-group">
  <label for="telefono">Teléfono</label>
  <input type="number"
    class="form-control"
    id="telefono">
</div>
<div class="form-group">
  <label for="email">Correo Electrónico</label>
  <input type="text"
    class="form-control"
    id="email">
</div>
<div class="form-group">
  <label for="contacto">Persona de contacto</label>

```

```

<input type="text"
      class="form-control"
      id="contacto">
</div>
<button type="submit"
      class="btn btn-success">
  Añadir Proveedor</button>
</form>
<hr>
</div>
</div>

```

Ahora vamos a añadir el nuevo componente al *routing* en el módulo, añadiendo para ello en el archivo `app.module.ts`:

`src/app/app.module.ts`

```

...
{ path: 'addprovee', component: AddproveeComponent },
...

```

Y también vamos a modificar el listado de proveedores, para añadir un botón de acceso al nuevo componente en el que introducir datos, añadiendo en el archivo `proveedores.component.html` la siguiente línea:

`src/app/proveedores/proveedores/proveedores.component.html`

```

...
<h1>Listado de Proveedores</h1>
<a class="btn btn-primary float-md-right" routerLink="/addprovee">Añadir
nuevo proveedor</a>
<br>
...

```

De esta manera si navegamos al listado de proveedores disponemos de un botón para añadir nuevos proveedores:

Compras App

[Inicio](#) [Proveedores](#)

Listado de Proveedores

Añadir nuevo proveedor

Nombre	Correo Electrónico	Telefono
Telefónica	info@telefonica.com	911111111
Iberdrola	info@iberdrola.com	922222222

Y si lo pulsamos llegamos a la vista del componente addproveedor con el formulario:

Compras App

[Inicio](#) [Proveedores](#)

Añadir nuevo proveedor

Nombre

C.I.F.

Dirección

De momento el formulario es una simple plantilla HTML que no realiza funcionalidad alguna, por lo que llega el momento de llamar a los paquetes de Angular para poder trabajar con él.

Antes de comenzar a añadir la lógica del formulario, debemos modificar el módulo raíz de la aplicación para añadir la funcionalidad de formularios de Angular.

Por tanto, añadimos al código del archivo app.module.ts:

src/app/app.module.ts

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

y en los import de la clase añadimos:

```
imports: [  
  BrowserModule,  
  RouterModule.forRoot(routes),  
  FormsModule  
],
```

El siguiente paso es programar el objeto que recibirá los valores del formulario en nuestro componente, para ello vamos a añadir las siguientes líneas de código en la clase del archivo addprovee.component.ts:

src/app/proveedores/addprovee/addprovee.component.ts

```
...  
proveedor: any;  
  
constructor(){  
  this.proveedor = {  
    nombre: "",  
    cif: "",  
    direccion: "",  
    cp: "",  
    localidad: "",  
    provincia: "",  
    telefono: null,  
    email: "",
```

```

        contacto: ''
    }
}
...

```

Con esto ya tenemos el objeto proveedor preparado para enlazar con el formulario, así que ahora vamos a incorporar al formulario html el código necesario para enlazar con el componente.

En primer lugar modificamos el inicio de la etiqueta form en el archivo `addprovee.component.html`:

```
src/app/proveedores/addprovee/addprovee.component.html
```

```

...
<form (ngSubmit)="onSubmit()" #formpro="ngForm" >
...

```

En la cual hemos añadido una id local llamada `formpro` para identificar nuestro formulario y lo asociamos a la directiva `ngForm` de Angular.

Esta directiva hará que los valores del formulario se asignen a un objeto llamado `formpro` cuando ejecutemos el submit del formulario, que a su vez hará que se ejecute el método `onSubmit()` en el componente.

Además, tenemos que modificar cada campo del formulario, para añadir `ngModel` y el atributo `name` con el nombre del campo:

```

...
    <div class="form-group">
      <label for="nombre">Nombre</label>
      <input type="text"
        class="form-control"
        id="nombre"
        ngModel
        name="nombre">
    </div>
...  <!--repetir en todos los componentes -->

```

A continuación, regresamos al componente para, con los datos de nuestro formulario html, enlazarlos al objeto. De nuevo en `addprovee.component.ts` modificamos los import:

```
src/app/proveedores/addprovee/addprovee.component.ts
```

```
import { Component, OnInit, ViewChild } from '@angular/core';  
import { NgForm } from '@angular/forms';  
...
```

En la clase, añadimos una vista en la que añadir el objeto del formulario:

```
...  
@ViewChild('formpro') formpro: NgForm;  
proveedor: any;  
...
```

Y posteriormente, después del constructor donde inicializamos el objeto proveedor, añadimos el método `onSubmit()`:

```
...  
onSubmit(){  
  
  this.proveedor.nombre = this.formpro.value.nombre;  
  this.proveedor.cif = this.formpro.value.cif;  
  this.proveedor.direccion = this.formpro.value.direccion;  
  this.proveedor.cp = this.formpro.value.cp;  
  this.proveedor.localidad = this.formpro.value.localidad;  
  this.proveedor.provincia = this.formpro.value.provincia;  
  this.proveedor.telefono = this.formpro.value.telefono;  
  this.proveedor.email = this.formpro.value.email;  
  this.proveedor.contacto = this.formpro.value.contacto;  
  
  this.formpro.reset();  
}  
...
```

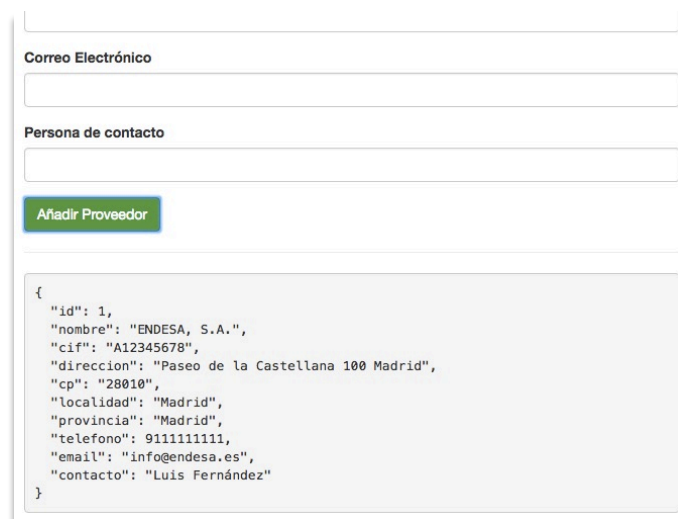
En este método, que se lanza con el botón del formulario, se pasará el valor del objeto formpro y de cada propiedad a su correspondiente en el objeto proveedor del componente.

Finalmente, para comprobar el funcionamiento del formulario, en la vista del archivo addprovee.component.html añadimos después del cierre del formulario, el código para visualizar el objeto que se crea en el formulario, aplicándole el *pipe* json para su presentación "pretty":

```
src/app/proveedores/addprovee/addprovee.component.html
```

```
...  
</form>  
<hr>  
<pre>{{ proveedor | json }}</pre>  
...
```

Cuando accedemos en el navegador comprobamos que al completar los datos de cada campo, y pulsar en el botón añadir, se muestra el objeto:



```
{  
  "id": 1,  
  "nombre": "ENDESA, S.A.",  
  "cif": "A12345678",  
  "direccion": "Paseo de la Castellana 100 Madrid",  
  "cp": "28010",  
  "localidad": "Madrid",  
  "provincia": "Madrid",  
  "telefono": 9111111111,  
  "email": "info@endesa.es",  
  "contacto": "Luis Fernández"  
}
```

9.2 Carga de datos en campos select

Es habitual disponer de campos select en los formularios de una aplicación, así que vamos a comprobar con nuestro ejemplo como implementarlos en Angular.

Supongamos que necesitamos que el campo provincia se complete a través de un campo de tipo select con todas las provincias españolas. Pues gracias a la directiva ngFor podemos crear un array en el componente y recorrerlo en el select para cada opción.

Para llevar a cabo este proceso, en primer lugar añadimos a la clase del componente (antes del constructor) en el archivo addprovee.component.ts el siguiente array:

```
src/app/proveedores/addprovee/addprovee.component.ts
```

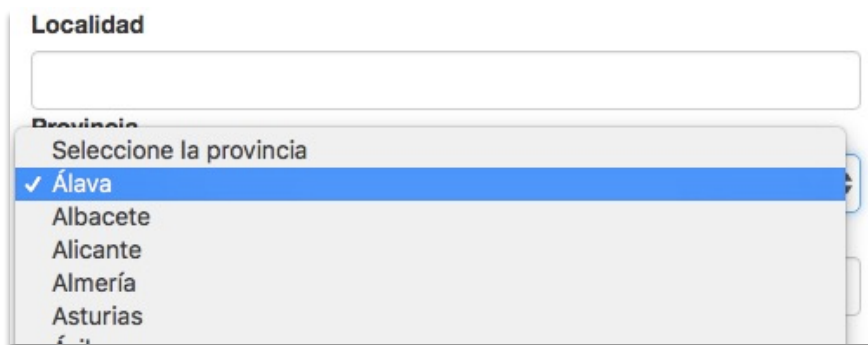
```
...
  provincias: string[] = [
    'Álava', 'Albacete', 'Alicante', 'Almería', 'Asturias', 'Ávila', 'Badajoz', 'Barcelona',
    'Burgos', 'Cáceres', 'Cádiz', 'Cantabria', 'Castellón', 'Ciudad Real', 'Córdoba',
    'La Coruña', 'Cuenca', 'Gerona', 'Granada', 'Guadalajara',
    'Guipúzcoa', 'Huelva', 'Huesca', 'Islas Baleares', 'Jaén', 'León', 'Lérida', 'Lugo',
    'Madrid', 'Málaga', 'Murcia', 'Navarra', 'Orense', 'Palencia', 'Las Palmas',
    'Pontevedra', 'La Rioja', 'Salamanca', 'Segovia', 'Sevilla', 'Soria', 'Tarragona',
    'Santa Cruz de Tenerife', 'Teruel', 'Toledo', 'Valencia', 'Valladolid', 'Vizcaya',
    'Zamora', 'Zaragoza' ]
  ...
```

Puedes cambiar este array de provincias o estados por los de tu país, buscándolos en google como "array javascript de..."

Y modificamos el campo de provincias en la vista, addprovee.component.html de la siguiente forma:

```
...  
<div class="form-group">  
  <label for="provincia">Provincia</label>  
  <select class="form-control"  
    id="provincia"  
    ngModel  
    name="provincia">  
    <option value="">Seleccione la provincia</option>  
    <option *ngFor="let provincia of provincias"  
      [value]="provincia">{{provincia}} </option>  
  </select>  
</div>  
...
```

Y así, ya tendremos disponible todas las provincias como podemos comprobar en el navegador:



9.3 Validación de campos mediante HTML

Angular permite realizar una validación de campos de formularios que en el caso de emplear la técnica anterior, Template Driven, utiliza la validación nativa de HTML.

Gracias a los estados de cada campo, Angular permite implementar clases CSS y elementos HTML dinámicos de ayuda al usuario, para completar los formularios.

Para aprender esta funcionalidad, vamos a comenzar por añadir al campo email los atributos HTML5 required e email en el archivo addprovee.component.html:

src/app/proveedores/addprovee/addprovee.component.html

```
...  
    <div class="form-group">  
      <label for="email">Correo Electrónico</label>  
      <input type="text"  
        class="form-control"  
        id="email"  
        ngModel  
        name="email"  
        required  
        email>  
    </div>  
...
```

De esta manera, ahora el campo de Correo Electrónico será obligatorio y también deberá cumplir la sintaxis de dirección de correo electrónico.

Por defecto, Angular elimina la validación HTML nativa por lo que los anteriores atributos no impedirán que empleemos el botón submit y tampoco lanzarán mensaje alguno.

Pero de momento, nos sirven para identificar los estados por los que pasa un campo en un formulario Angular implementados a través de ngForm.

De estos estados, almacenados en el objeto del formulario, nos vamos a fijar en *pristine* (el usuario aún no ha modificado el campo desde su carga), *dirty* (el usuario ha modificado el campo), *touched* (el usuario ha abandonado el campo) y *valid* e *invalid* (el valor del campo cumple o incumple la regla de validación).

Para ver el estado del campo Email según operemos en el formulario, añadimos a la vista, antes del cierre del form, el siguiente código:

```
src/app/proveedores/addprovee/addprovee.component.html
```

```
...
<hr>
<pre> Estado Dirty: {{ formpro.controls.email.dirty }}</pre>
<pre> Estado Pristine: {{ formpro.controls.email.pristine }}</pre>
<pre> Estado Touched: {{ formpro.controls.email.touched }}</pre>
<pre> Estado Valid: {{ formpro.controls.email.valid }}</pre>
<pre> Estado Invalid: {{ formpro.controls.email.invalid }}</pre>
</form>
...
```

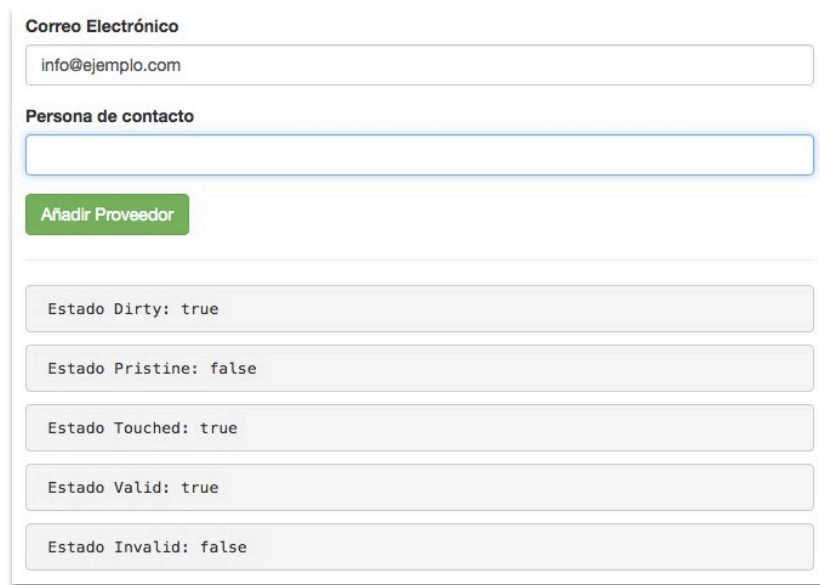
Este código nos muestra el valor de cada estado del campo email. Al iniciar el formulario, el valor del estado *pristine* será *true* (aún no hemos escrito nada) y el de *invalid* y *valid*, *true* y *false* respectivamente, ya que aún no está validado.

Cuando comencemos a escribir, *dirty* pasará de *false* a *true* y al revés *pristine*.

Por otra parte, al abandonar el foco el estado *touched* pasará a *true*.

Finalmente, si introducimos un valor de correo electrónico correcto, el estado *valid* pasará a *true* y el *invalid* a *false*.

Podemos comprobar en el navegador, los cambios a medida que escribimos:



Correo Electrónico

info@ejemplo.com

Persona de contacto

Añadir Proveedor

Estado Dirty: true

Estado Pristine: false

Estado Touched: true

Estado Valid: true

Estado Invalid: false

¿Para qué usar estos estados? Pues por ejemplo, para el uso de clases CSS dinámicas en función del estado que ayuden al usuario a completar el formulario.

Vamos a ponerlo en práctica. En el archivo de estilos, `addprovee.component.css`, añadimos una serie de clases CSS de Angular (las que comienzan por `ng`) relacionadas los estados del campo:

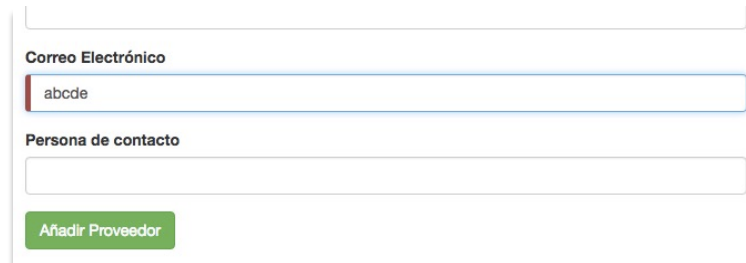
```
src/app/proveedores/addprovee/addprovee.component.html
```

```
input.ng-invalid.ng-touched {  
  border-left: 5px solid #a94442;  
}  
  
input.ng-valid.ng-dirty {  
  border-left: 5px solid #42A948;  
}
```

Que implicará en primer lugar, que cuando el campo tenga los estados inválido y touched, es decir que no sea ha introducido correctamente el valor y se ha pasado al siguiente, se mostrara un borde rojo de 5 px en la izquierda del campo.

En cambio, si el campo se ha comenzado a completar y se valida, se mostrará un borde verde de 5 px.

Lo podemos comprobar en nuestro navegador:



También podemos añadir más ayudas visuales con un icono junto al label del campo y un texto de advertencia empleando la directiva ngIf y los estados.

Por ejemplo, para el campo mail, en el archivo addprovee.component.html, sustituimos su div por el siguiente:

src/app/proveedores/addprovee/addprovee.component.html

```
...
<div class="form-group">
  <label for="email">Correo Electrónico</label>
  <i class="fa fa-check-circle check"
    *ngIf="email.valid"></i> ①
  <i class="fa fa-exclamation-circle uncheck"
    *ngIf="email.invalid && email.touched"></i>
  <input type="text"
    class="form-control"
    id="email"
    ngModel
    name="email"
    required
    email
    #email="ngModel"> ②
  <p class="alert alert-danger" *ngIf="email.invalid && email.touched"> ③
    Por favor introduzca una dirección de correo correcta.
  </p>
</div>
...
```

Donde hemos ① introducido dos iconos que se mostrarán en función de la expresión de su `ngIf`, hemos ② creado la `id` local `email` asociándola a `ngModel` y hemos ③ creado un texto de alerta que se mostrará también de acuerdo a la expresión `ngIf`.

Añadimos las clases CSS de los iconos al archivo `addprovee.component.css`:

`src/app/proveedores/addprovee/addprovee.component.css`

```
...  
.check {  
  color: #42A948;  
}  
  
.unchecked {  
  color: #a94442;  
}  
...
```

Y la referencia al CDN de *Font Awesome Icons* en el `head` del `index.html` para obtener los iconos:

`src/index.html`

```
...  
<script src="https://use.fontawesome.com/bacad173cf.js"></script>  
...
```

Finalmente, comprobamos en el navegador:

Correo Electrónico !

Por favor introduzca una dirección de correo correcta.

Persona de contacto

Añadir Proveedor

Estado Dirty: true
Estado Pristine: false
Estado Touched: true
Estado Valid: false
Estado Invalid: true

Para finalizar la validación de campos de un formulario, podemos añadir la lógica necesaria para que el botón de envío solo se active cuando todos los campos estén en estado valid.

Para ello, simplemente modificamos el botón submit en el formulario en el archivo `addprovee.component.html` de la siguiente forma:

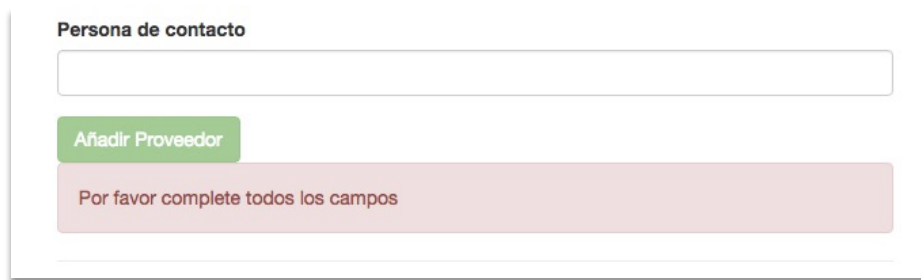
`src/app/proveedores/addprovee/addprovee.component.html`

```
...  
<button type="submit"  
  class="btn btn-success"  
  [disabled]="!formpro.valid" ①  
>Añadir Proveedor</button>  
<p class="alert alert-danger " *ngIf="!formpro.valid"> ②  
  Por favor complete todos los campos  
</p>  
...
```

En el cual:

- ① Añadimos el atributo disabled mientras todo el formulario no sea válido.
- ② Y un texto de advertencia con un ngIf.

También añadiremos los iconos junto al label y el atributo required al resto de campos y podremos comprobar en el navegador el funcionamiento de todo el formulario.



Persona de contacto

Añadir Proveedor

Por favor complete todos los campos

Como tal, nuestro formulario está listo pero los datos quedan almacenados en el objeto proveedor en memoria.

Aprenderemos más adelante, como almacenar de manera persistente los datos mediante la conexión de la aplicación Angular a un servidor de base de datos.

9.4 Creación de formularios Reactive

Otra de las técnicas para crear formularios es Reactive, que lleva la generación y gestión del mismo del lado del archivo TypeScript del componente.

Para implementar esta técnica en nuestra aplicación, en primer lugar vamos a crear un nuevo componente llamado presupuestos. Una vez más, desde el directorio del proyecto en la consola, completamos:

```
ng g c presupuestos/addpres - -spec false
```

A continuación incorporamos el nuevo component a nuestro routing y el paquete de Angular para crear formularios reactivos, para lo cual lo añadimos en el archivo app.module.ts de la siguiente forma:

```
src/app/app.module.ts
```

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

Y:

```
...  
{ path: 'addpres', component: AddpresComponent },  
...  
  
...  
imports: [  
  BrowserModule,  
  RouterModule.forRoot(routes),  
  FormsModule,  
  ReactiveFormsModule  
],  
...
```

Para poder acceder a este nuevo componente en el navegador, añadimos el link al menú de cabecera en el archivo header.component.html, de la siguiente manera:

src/app/header/header.component.html

```
...
<ul class="nav navbar-nav navbar-right">
  <li class="nav-item" routerLinkActive="active"
    [routerLinkActiveOptions]="{exact:true}">
    <a routerLink="/" >Inicio </a>
  </li>
  <li class="nav-item" routerLinkActive="active">
    <a routerLink="/proveedores">Proveedores</a>
  </li>
  <li class="nav-item" routerLinkActive="active">
    <a routerLink="/addpres">Añadir Presupuesto</a>
  </li>
</ul>
...
```

Dejando listo el componente para ser empleado.

Vamos a comenzar a crear nuestro formulario. En primer lugar añadimos el siguiente formulario HTML a nuestro código en el archivo addpres.component.html:

src/app/presupuestos/addpres/addpres.component.html

```
<div class="row">
  <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset 2">
    <h2>Añadir nuevo presupuesto</h2>
    <form>
      <div class="form-group">
        <label for="proveedor">Proveedor</label>
        <input type="text"
          class="form-control"
          id="proveedor">
      </div>
    </form>
  </div>
</div>
```

```

<div class="form-group">
  <label for="fecha">Fecha Presupuesto</label>
  <input type="date"
    class="form-control"
    id="fecha">
</div>
<div class="form-group">
  <label for="concepto">Concepto</label>
  <input type="text"
    class="form-control"
    id="concepto">
</div>
<div class="form-group">
  <label for="base">Base Imponible</label>
  <input type="number"
    class="form-control"
    id="base">
</div>
<div class="form-group">
  <label for="tipo">Tipo de IVA</label>
  <select class="form-control"
    id="tipo">
    <option value="">Seleccione...</option>
    <option value=0> 0 %</option>
    <option value=0.04> 4 %</option>
    <option value=0.10>10 %</option>
    <option value=0.21>21 %</option>
  </select>
</div>
<div class="form-group">
  <label for="iva">Importe IVA</label>
  <input type="number"
    class="form-control"
    id="iva">
</div>
<div class="form-group">
  <label for="total">Total Factura IVA Incluido</label>
  <input type="number"
    class="form-control"
    id="total">
</div>
<button class="btn btn-primary"

```



```
        type="submit">Añadir Presupuesto</button>
    </form>
</div>
</div>
```

Ahora comenzamos a añadir la lógica a nuestro componente. Para ello, en el archivo `addpres.component.ts`, comenzamos modificando las importaciones:

```
src/app/presupuestos/addpres/addpres.component.ts
```

```
import { FormControl, FormGroup, FormBuilder } from '@angular/forms';
...
```

Las cuales, nos permitirán gestionar el formulario desde el componente.

A continuación, dentro de la clase añadimos la propiedad `presupuestoForm` que será del tipo `FormGroup` y el objeto `presupuesto`:

```
...
presupuestoForm: FormGroup;
presupuesto: any;
...
```

Seguidamente en el constructor, para generar el formulario, creamos un objeto de nombre, por ejemplo, `pf` de la clase `FormBuilder` de Angular:

```
...
constructor(private pf: FormBuilder) {}
...
```

Continuamos en la clase, y dentro de `ngOnInit` igualamos `presupuestoForm` al objeto `pf` y inicializamos dentro de este y vacíos, los campos que tendrá el formulario:

```
...
ngOnInit() {
  this.presupuestoForm = this.pf.group({
    proveedor: "",
    fecha: "",
    concepto: "",
    base: "",
    tipo: "",
    iva: "",
    total: ""
  });
}
...
```

Que serán los mismos establecidos en el formulario HTML del archivo `addpres.component.html`. Regresamos a este archivo y añadimos en el inicio de la etiqueta `form`:

```
src/app/presupuestos/addpres/addpres.component.html
```

```
...
<form [formGroup]="presupuestoForm">
...
```

Y en cada campo añadimos el atributo `formControlName` con el nombre del campo, por ejemplo:

```
...
<div class="form-group">
  <label for="proveedor">Proveedor</label>
  <input type="text"
    class="form-control"
    id="proveedor"
```

```

        formControlName="proveedor">
    </div>
... <!--repetir para cada campo-->

```

Para visualizar la toma de datos del formulario, añadimos tras el cierre del form las siguientes dos líneas:

```

...
<pre>Valor del formulario: {{ presupuestoForm.value | json }}</pre>
<pre>Status del formulario: {{ presupuestoForm.status | json }}</pre>
...

```

Ahora vamos al navegador y comprobamos como al completar campos se actualizan automáticamente los valores del formulario en el objeto presupuestoForm:

The screenshot shows a web form with the following fields and values:

- Tipo de IVA:** 21 %
- Importe IVA:** 210
- Total Factura IVA Incluido:** 1210
- Button:** Añadir Presupuesto
- Valor del formulario:**

```

{
  "proveedor": "",
  "fecha": "",
  "concepto": "",
  "base": 1000,
  "tipo": "0.21",
  "iva": 210,
  "total": ""
}

```
- Status del formulario:** "VALID"

A continuación vamos a añadir un método onSubmit, para poder guardar los datos del formulario en el objeto presupuesto del componente.

Para ello, volvemos a modificar el inicio de la etiqueta form en el archivo addpres.component.html de la siguiente forma:

src/app/presupuestos/addpres/addpres.component.html

```
...  
    <form [formGroup]="presupuestoForm" (ngSubmit)="onSubmit()">  
...
```

Y tras el cierre de la etiqueta form, modificamos:

```
...  
    </form>  
    <hr>  
    <pre>Valor del formulario: {{ presupuestoForm.value | json }}</pre>  
    <pre>Status del formulario: {{ presupuestoForm.status | json }}</pre>  
    <hr>  
    <pre>{{ presupuesto | json }}</pre>  
...
```

En el componente, dentro de la clase, en el archivo addpres.component.ts añadimos el siguiente código:

src/app/presupuestos/addpres/addpres.component.ts

```
...  
    onSubmit() {  
        this.presupuesto = this.savePresupuesto();  
    }  
...
```

En el que definimos onSubmit() como el método que iguala el objeto presupuesto del componente con otro método savePresupuesto que definimos a continuación:

```

...
savePresupuesto() {
  const savePresupuesto = {
    proveedor: this.presupuestoForm.get('proveedor').value,
    fecha: this.presupuestoForm.get('fecha').value,
    concepto: this.presupuestoForm.get('concepto').value,
    base: this.presupuestoForm.get('base').value,
    tipo: this.presupuestoForm.get('tipo').value,
    iva: this.presupuestoForm.get('iva').value,
    total: this.presupuestoForm.get('total').value
  };
  return savePresupuesto;
}
...

```

Método que iguala cada valor del formulario con su correspondiente propiedad del objeto del componente y lo devuelve con un return.

Ahora en el navegador, podemos comprobar como los datos introducidos en el formulario son guardados en el objeto presupuesto cuando pulsamos en el botón añadir presupuesto:

2420

Añadir Presupuesto

Valor del formulario: {

```

  "proveedor": "",
  "fecha": "",
  "concepto": "",
  "base": 2000,
  "tipo": "0.21",
  "iva": 420,
  "total": 2420
}
```

Status del formulario: "VALID"

```

{
  "proveedor": "",
  "fecha": "",
  "concepto": "",
  "base": 2000,
  "tipo": "0.21",
  "iva": 420,
  "total": 2420
}
```

9.5 Validación de campos programática

Mediante este procedimiento de formularios reactivos, Angular permite añadir validación programática, es decir, independiente de los atributos de validación HTML5.

Vamos a comprobarlo en nuestro ejemplo, para lo cual en el componente en el archivo `addpres.component.ts` añadimos `Validators`:

```
src/app/presupuestos/addpres/addpres.component.ts
```

```
import { FormControl, FormGroup, FormBuilder, Validators } from
 '@angular/forms';
...
```

A continuación la sintaxis para añadir validaciones en formularios reactivos es, dentro del método `createForm`, crear un array de cada campo con el valor de inicialización, y la propiedad o propiedades de validación mediante la clase de Angular forms `Validators`.

Por ejemplo, añadimos a los campos que necesitamos que sean obligatorios la clase `required`:

```
...
ngOnInit() {
  this.presupuestoForm = this.pf.group({
    proveedor: ['', Validators.required ],
    fecha: ['', Validators.required ],
    cif: ['', Validators.required ],
    concepto: ['', Validators.required ],
    base: ['', Validators.required ],
    tipo: ['', Validators.required ],
    iva: ['', Validators.required ],
    total: ['', Validators.required ]
  });
}
...
```

Ahora en la vista, archivo `addpres.component.html`, modificamos el código del botón para que solo se muestre cuando el formulario sea válido:

`src/app/presupuestos/addpres/addpres.component.html`

```
...
    <button type="submit"
      class="btn btn-primary"
      [disabled]="!presupuestoForm.valid"
    >Añadir Presupuesto</button>
    <p class="alert alert-danger" *ngIf="!presupuestoForm.valid">
      Por favor complete todos los campos
    </p>
...
```

Y en el archivo `addpres.component.css` copiamos las clases CSS para validar, empleadas en el archivo `addprovee.component.css` del apartado anterior.

Comprobamos el funcionamiento en el navegador.

Además de la obligatoriedad de completar el campo, tenemos otras clases de validación, por ejemplo longitud mínima.

Vamos a comprobarlo forzando a que el campo `concepto` tenga un mínimo de 10 caracteres. Para ello modificamos en el archivo `addpres.component.ts` la siguiente línea:

`src/app/presupuestos/addpres/addpres.component.ts`

```
...
concepto: ['', [ Validators.required, Validators.minLength(10)]],
...
```

Que convierte el campo en obligatorio y con un mínimo de 10 caracteres para que pase a estado válido.

El resto de clases de validación de Angular se pueden encontrar en el siguiente enlace de su documentación:

<https://angular.io/api/forms/Validators>

Para finalizar nuestro formulario, podemos implementar para cada campos los iconos y textos de validación del apartado anterior. Pero en este caso el objeto utilizado para obtener el estado del campo tiene una sintaxis diferente, siendo el nombre del formulario seguido de controls y el nombre del campo.

Cambiamos en cada campo del archivo `addpres.component.html` el código por el siguiente:

`src/app/presupuestos/addpres/addpres.component.html`

```
<div class="form-group">
  <label for="proveedor">Proveedor</label>
  <i class="fa fa-check-circle check"
    *ngIf="presupuestoForm.controls.proveedor.valid"></i>
  <i class="fa fa-exclamation-circle uncheck"
    *ngIf="presupuestoForm.controls.proveedor.invalid &&
    presupuestoForm.controls.proveedor.touched"></i>
  <input type="text"
    class="form-control"
    id="proveedor"
    formControlName="proveedor">
  <p class="alert alert-danger"
    *ngIf="presupuestoForm.controls.proveedor.invalid &&
    presupuestoForm.controls.proveedor.touched">
    El campo Proveedor es obligatorio.
  </p>
</div>
```


9.6 valueChanges

Otra de las enormes ventajas que proporciona la gestión de formularios reactive es la posibilidad de observar los cambios que se produzcan en los campos del formulario, lo que permite realizar formularios totalmente dinámicos.

En el ejemplo anterior, tenemos dos campos, importe del IVA y total que en la vida real se obtienen a partir del producto de la base por el tipo de IVA y la suma de la base más el IVA respectivamente.

Vamos a ver como podemos incorporar las funciones aritméticas a nuestro código TypeScript y, lo más importante, como Angular detecta cambios en el valor de un campo para actualizar los valores de otros de manera automática.

Para ello en primer lugar, vamos a modificar la clase del componente en el archivo `addpres.component.ts` de la siguiente manera:

```
src/app/presupuestos/addpres/addpres.component.ts
```

```
...
export class AddpresComponent implements OnInit {

  presupuestoForm: FormGroup;
  presupuesto: any;
  base: any;
  tipo: any;
  iva: any = 0;
  total: any = 0;
  ...
}
```

Donde al comienzo, hemos creado las propiedades `base`, `tipo`, `iva` y `total`, de las cuales las dos últimas se inicializan a cero.

A continuación, modificamos en `ngOnInit` ① los campos `iva` y `total`, para igualarlos a las propiedades anteriores, tambien creamos ② el método `onChanges()`:

```

...
ngOnInit() {
this.presupuestoForm = this.pf.group({
  proveedor: ['', Validators.required ],
  fecha: ['', Validators.required ],
  concepto: ['', [ Validators.required, Validators.minLength(10)] ],
  base: ['', Validators.required ],
  tipo: ['', Validators.required ],
  iva: this.iva ,
  total: this.total
});
this.onChange();
}
...

```

Nos queda finalmente añadir el método onChange:

```

...
onChange(): void {
  this.presupuestoForm.valueChanges.subscribe(valor => {
    this.base = valor.base;
    this.tipo = valor.tipo;
    this.presupuestoForm.value.iva = this.base * this.tipo;
    this.presupuestoForm.value.total = this.base + (this.base * this.tipo);
  });
}
...

```

En cual:

- ① Utilizamos el observable valueChanges y nos suscribimos a el para obtener el objeto valor, que se actualizará cada vez que se produzca un cambio en algún campo del formulario.
- ② Igualamos la propiedad base y tipo a cada campo correspondiente.
- ③ Y establecemos el valor de iva y total en el formulario mediante una fórmula aritmética de los dos valores anteriores.

Una vez modificado el componente, nos queda cambiar la vista en archivo `addpres.component.ts` de los campos iva y total de la siguiente manera:

`src/app/presupuestos/addpres/addpres.component.html`

```
...
<div class="form-group">
  <label for="iva">Importe IVA</label>
  <input type="number"
    class="form-control"
    id="iva"
    formControlName="iva"
    [(ngModel)]="presupuestoForm.value.iva" ❶
    disabled> ❷
</div>
<div class="form-group">
  <label for="total">Total Factura IVA Incluido</label>
  <input type="number"
    class="form-control"
    id="total"
    formControlName="total"
    [(ngModel)]="presupuestoForm.value.total"
    disabled>
</div>
...
```

En el cual:

- ❶ Añadimos la directiva `ngModel` asociada al valor en el formulario de cada campo.
- ❷ Eliminamos los iconos y textos de advertencia de validación y añadimos el atributo `disabled` para que el campo no sea accesible.

Con estos pasos nuestro formulario queda finalizado y podemos comprobar en el navegador como los campos iva y total se actualizan automáticamente cada vez que introducimos los datos necesarios para su cálculo.

Base Imponible ✓

1000

Tipo de IVA ✓

21 %

Importe IVA

210

Total Factura IVA Incluido

1210

Añadir Presupuesto

10 Conexión con el servidor

Angular utiliza los métodos http para realizar conexiones con servidores de bases de datos que nos permitan almacenar de manera persistente los mismos.

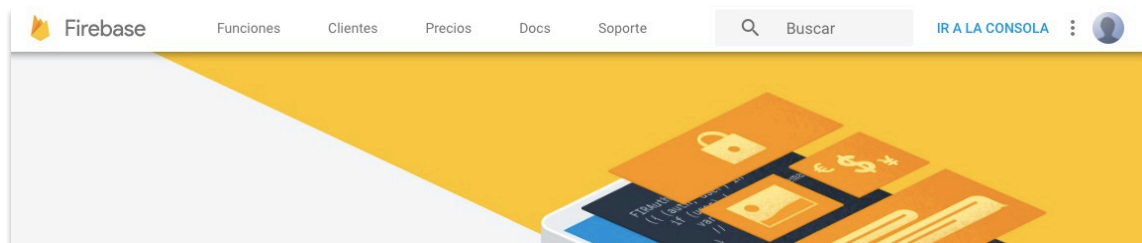
Vamos a ver cómo podemos almacenar los datos de nuestra aplicación realizando estas conexiones en un ejemplo con la plataforma Firebase, para de esta forma conseguir aplicaciones CRUD que nos permitan crear, leer, actualizar y borrar registros en tiempo real.

10.1 Base de datos en Firebase

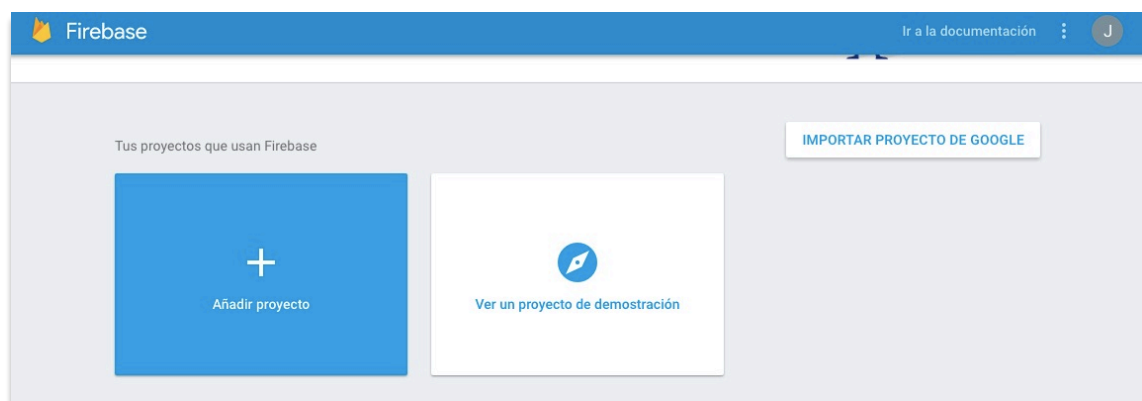
Para emplear Firebase, accedemos a su url:

<https://firebase.google.com>

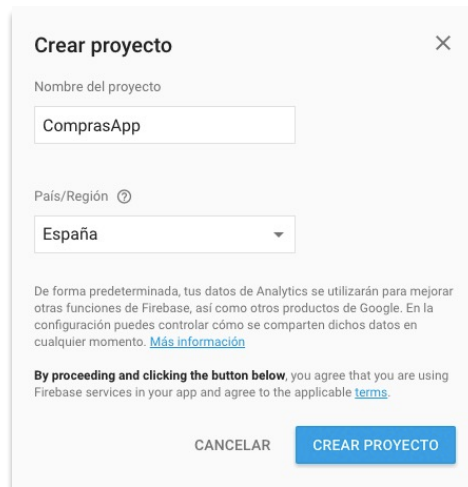
Y si ya estamos 'logueados' en google en nuestro navegador pulsamos directamente en el botón ir a la consola, en caso contrario con cualquier cuenta de google podemos acceder:



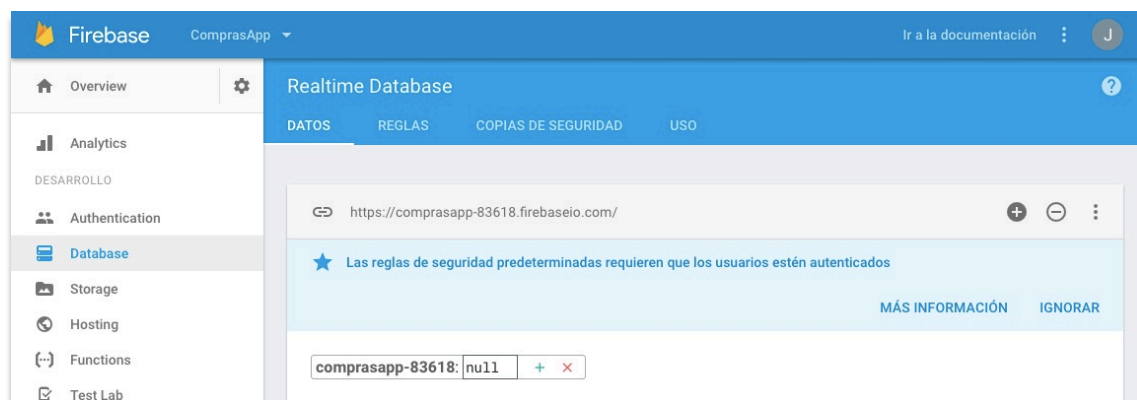
En la consola, pulsamos en crear proyecto:



Le añadimos un nombre identificativo, nuestro país y pulsamos en Crear Proyecto:



Una vez creado nuestro proyecto, pulsamos en la opción Database en el menú lateral izquierdo:



Pulsamos en la pestaña REGLAS, y vamos a modificamos los permisos de escritura y lectura a true:

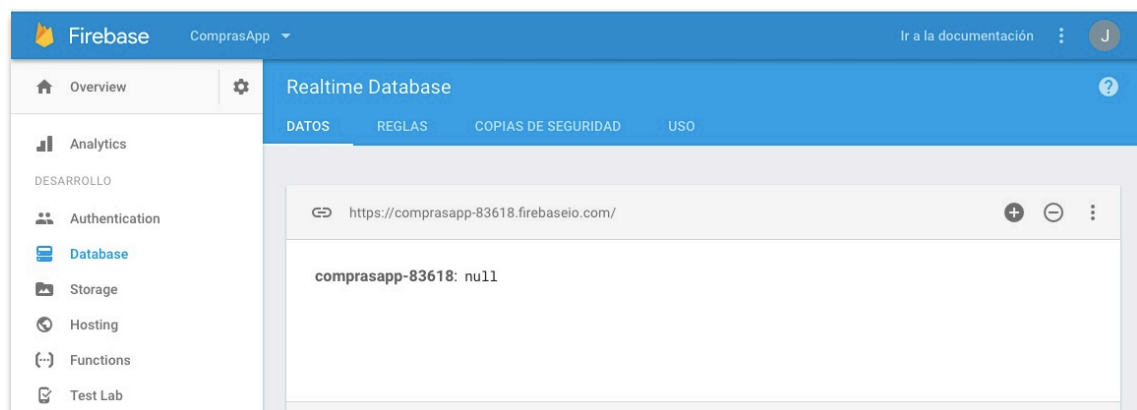


El código es el siguiente:

```
{  
  "rules": {  
    ".read": "true",  
    ".write": "true"  
  }  
}
```

Y pulsamos en publicar. Con esto lo que conseguimos es deshabilitar la protección frente a lectura y escritura de los datos de la base de datos.

Pulsamos de nuevo en la pestaña DATOS y copiamos la url de nuestra base de datos en nuestro portapeles o cualquier documento, ya que la emplearemos en nuestras peticiones HTML:



Con estos pasos nuestra instancia de base de datos en Firebase queda configurada para ser usada en nuestra aplicación.

10.2. Servicio HTTP: Post

En este paso vamos a crear un servicio para implementar las peticiones http de nuestra aplicación al servidor.

Creemos, en la consola dentro del directorio de nuestro proyecto, un nuevo servicio:

```
ng generate service servicios/presupuestos
```

Una vez creado el servicio, lo incorporamos al módulo, conjuntamente con la importación de las librerías http de Angular. Por tanto añadimos al archivo app.module.ts las importaciones:

```
src/app/app.module.ts
```

```
...
import { HttpClientModule } from '@angular/http';
import { PresupuestosService } from 'app/servicios/presupuestos.service';
...
```

Modificamos los imports y providers:

```
...
imports: [
  BrowserModule,
  RouterModule.forRoot(routes),
  FormsModule,
  ReactiveFormsModule,
  HttpClientModule
],
providers: [ProveedoresService, PresupuestosService],
...
```


Ahora vamos a nuestro servicio en el archivo presupuestos.service.ts y comenzamos importando:

```
src/app/servicios/presupuestos.service.ts
```

```
import { Injectable } from '@angular/core';
import { Headers, Http, Response } from '@angular/http';
import 'rxjs/Rx';
...
```

A continuación en la clase, añadimos en primer lugar, ① la propiedad presURL con el valor de la url de nuestro base de datos más el nombre del nodo o colección donde queremos almacenar los datos, en nuestro caso presupuestos, seguido de punto JSON.

También, ② dentro del constructor, realizamos la llamada a la clase Http de Angular:

```
...
presURL = 'https://comprasapp-83618.firebaseio.com/presupuestos.json'; ①
constructor(private http: Http) {} ②
...
```

Posteriormente, añadimos el método postPresupuesto que es el realizará las peticiones http de tipo post al servidor de la base de datos, para crear registros:

```
...
postPresupuesto( presupuesto: any) {
  const newpres = JSON.stringify(presupuesto);
  const headers = new Headers({
    'Content-Type': 'application/json'
  });

  return this.http.post( this.presURL, newpres, {headers})
    .map( res => {
      console.log(res.json());
      return res.json();
    })
}
```

```
}  
...
```

En el cual:

- ① El método `postPresupuesto` recibe del componente el objeto presupuesto que es pasado a string en la constante `newpres`. También se define la cabecera del envío.
- ② Y mediante el método `post` es enviado a la url del servidor devolviendo un objeto `res` con el resultado de la operación.

El servicio queda finalizado y listo para ser usado en el componente.

Ahora vamos a nuestro componente, en el archivo `presupuestos.component.ts` y en primer lugar importamos el servicio:

```
src/app/presupuestos/addpres/addpres.component.ts
```

```
...  
import { PresupuestosService } from '../servicios/presupuestos.service';  
...
```

A continuación, modificamos el constructor para añadir el nuevo servicio de la siguiente manera:

```
...  
constructor(private pf: FormBuilder,  
             private presupuestoService: PresupuestosService) {  
  }  
...
```

Y también modificamos el método `onSubmit` para añadir el método `postPresupuestos` del servicio, que enviará el objeto presupuesto del componente:

```

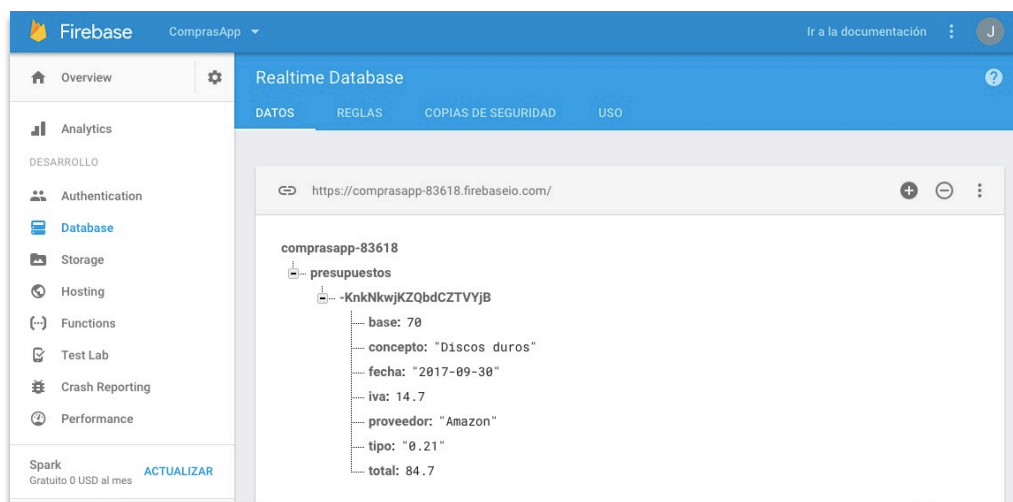
...
onSubmit() {
  this.presupuesto = this.savePresupuesto();
  this.presupuestoService.postPresupuesto( this.presupuesto )
    .subscribe(newpres => {

    })
  }
}
...

```

Si ahora vamos al navegador y añadimos un nuevo registro, comprobamos en la consola del navegador la respuesta del envío HTML:

Y si vamos a la base de datos en Firebase, comprobamos el registro:



Si nos fijamos, Firebase almacena los registros con una estructura de árbol de nodos en el cual el primero sería el equivalente a una tabla en una base de datos SQL y cada registro es a su vez un nodo con una clave id que crea automáticamente Firebase cuyo contenido es el registro en formato JSON, es decir pares clave-valor.

Podemos añadir nuevo presupuestos para comprobar como va creciendo el número de registros en el nodo presupuestos de nuestra base de datos en Firebase.

10.3. Servicio HTTP: Get

También podemos recuperar los datos desde la base de datos a nuestra aplicación con peticiones de tipo get.

Para ello, vamos a modificar el servicio en el archivo `presupuestos.service.ts` en el cual, dentro de la clase, añadimos un nuevo método:

```
src/app/servicios/presupuestos.service.ts
```

```
...
getPresupuestos () {

    return this.http.get( this.presURL )
        .map( res => res.json());
}
...
```

Que realizará una petición get a la base de datos y devolverá un objeto res que es pasado a formato JSON.

Vamos a crear un nuevo componente para crear un listado de presupuestos. Una vez más en la consola del equipo completamos:

```
ng g c presupuestos/presupuestos --spec false
```

Ahora vamos añadir el nuevo componente al routing en el módulo, añadiendo para ello en el archivo `app.module.ts`:

```
src/app/app.module.ts
```

```
...
{ path: 'presupuestos', component: PresupuestosComponent },
...
```

Y en el componente header.component.html modificamos el link de añadir presupuesto por el siguiente:

src/app/header/header.component.html

```
<li class="nav-item" routerLinkActive="active">
  <a class="nav-link" routerLink="/presupuestos">Presupuestos </a>
</li>
```

A continuación, vamos a introducir una tabla para listar los presupuestos, así como un botón de acceso para añadir nuevos presupuestos.

Por tanto, en la plantilla del nuevo componente en el archivo presupuestos.component.html añadimos el siguiente código:

src/app/presupuestos/presupuestos/presupuestos.component.html

```
<h3>Listado de Presupuestos</h3>
<a class="btn btn-primary float-md-right" routerLink="/addpres">Añadir
nuevo presupuesto</a>
<br>
<table class="table table-bordered table-striped tabla" style="margin-top:
40px;">
  <thead>
    <tr class="filters">
      <th>Proveedor</th>
      <th>Fecha</th>
      <th>Concepto</th>
      <th>Base</th>
      <th>IVA</th>
      <th>Total</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let presupuesto of presupuestos">
      <td>{{ presupuesto.proveedor }}</td>
      <td>{{ presupuesto.fecha }}</td>
```

```

        <td>{{ presupuesto.concepto }}</td>
        <td>{{ presupuesto.base }}</td>
        <td>{{ presupuesto.iva }}</td>
        <td>{{ presupuesto.total }}</td>
    </tr>
</tbody>
</table>

```

En la tabla ya hemos incluido la directiva ngFor, para iterar el array presupuestos que definiremos en el componente.

Vamos ahora al componente, en el archivo presupuestos.component.ts y comenzamos añadiendo la importación de nuestro servicio:

```
src/app/presupuestos/presupuestos/presupuestos.component.ts
```

```

...
import { PresupuestosService } from '../servicios/presupuestos.service';
...

```

A continuación, dentro de la clase, creamos un objeto presupuestos como array de cualquier tipo de datos TypeScript:

```

...
presupuestos: any[] = [];
...

```

En el constructor, declaramos el servicio y llamamos a su método getPresupuestos.

Como este método nos devuelve un objeto con todos los presupuestos en el formato de Firebase, lo iteramos con un *for* para convertirlo en objetos individuales con una id. Esos objetos, los añadimos al array presupuestos del componente con el método JavaScript *push*.

```

...
constructor(private presupuestosService: PresupuestosService) {
  this.presupuestosService.getPresupuestos()
    .subscribe(presupuestos => {
      for (const id$ in presupuestos) {
        const p = presupuestos[id$];
        p.id$ = id$;
        this.presupuestos.push(presupuestos[id$]);
      }
    })
}
...

```

Si ahora vamos en el navegador al listado de presupuestos vemos como se ha completado con los registros de la base de datos:

Compras App
Inicio Proveedores Presupuestos

Listado de Presupuestos

Añadir nuevo presupuesto

Proveedor	Fecha	Concepto	Base	IVA	Total
Amazon	2017-09-30	Discos duros	70	14.7	84.7
Endesa	2017-09-25	Suministro Eléctrico	150	31.5	181.5
Telefónica S.L.U.	2017-10-12	Centralita Telefonía	600	126	726

Para finalizar este apartado, podemos completar la navegación del usuario añadiendo un botón de regreso en la vista para crear nuevos componentes, un botón para cancelar y un método para resetear el formulario.

Para ello, en nuestro componente para añadir nuevos presupuestos, comenzamos por añadir un botón de regreso:


```
src/app/presupuestos/addpres/addpres.component.html
```

```
...  
<h2>Añadir nuevo presupuesto</h2>  
  <a class="btn btn-primary float-md-right"  
    routerLink="/presupuestos">Regresar al listado</a>  
  <br>  
...
```

Y otro botón junto al envío, para cancelar:

```
...  
<button type="submit"  
  class="btn btn-primary"  
  [disabled]="!presupuestoForm.valid"  
  >Añadir Presupuesto</button>  
  <a class="btn btn-danger" routerLink="/presupuestos">Cancelar</a>  
...
```

Ahora en el componente añadimos un método en onSubmit para resetear el formulario.

```
src/app/presupuestos/addpres/addpres.component.ts
```

```
onSubmit() {  
  this.presupuesto = this.savePresupuesto();  
  this.presupuestoService.postPresupuesto( this.presupuesto )  
    .subscribe(newpres => {  
    })  
  this.presupuestoForm.reset();  
}
```

10.4. Servicio HTTP: Put

En Angular podemos emplear peticiones put para actualizar los registros de nuestra base de datos. Vamos a ver el ejemplo concreto, para nuestra base de datos en Firebase.

En primer lugar, vamos a añadir el método put a nuestro servicio de presupuestos para lo cual, comenzamos añadiendo en el archivo presupuesto.service.ts dentro de su clase, una nueva url sin la extensión json:

```
src/app/servicios/presupuestos.service.ts
```

```
...  
preURL = 'https://comprasapp-83618.firebaseio.com/presupuestos';  
...
```

También en la clase, añadimos un nuevo método getPresupuesto para recuperar un registro de la base de datos con una determinada id, cargarlo en el componente y poder editarlo:

```
...  
getPresupuesto ( id$: string ) ① {  
  const url = `${ this.preURL }/${ id$ }.json`; ②  
  return this.http.get( url )  
    .map( res => res.json());  
}  
...
```

En el cual:

- ① Se recibe como parámetro la id del registro a recuperar de la base de datos.
- ② Y se crea una url compuesta por la dirección del nodo de firebase más la id del registro a modificar que es utilizada para recuperar con get el registro.

Seguidamente, añadimos el método `putPresupuesto`, para que una vez editado el objeto sea enviado a la base de datos:

```
...
putPresupuesto( presupuesto: any, id$: string) { ①
  const newpre = JSON.stringify(presupuesto);
  const headers = new Headers({
    'Content-Type': 'application/json'
  });

  const url = `${this.preURL}/${id$.json}`; ②

  return this.http.put( url, newpre, {headers}) ③
    .map( res => {
      console.log(res.json());
      return res.json();
    })
  }
...
```

En el cual:

- ① Recibimos dos parámetros del componente, el objeto presupuesto y su id (que corresponde con la emplada por Firebase), de cara a identificar el presupuesto a actualizar en la base de datos. El objeto recibido, se pasa a string para Firebase y se define la cabecera.
- ② Se crea una url compuesta por la dirección del nodo de firebase más la id del registro a modificar.
- ③ Se ejecuta la petición http para actualizar el objeto y se recoge la respuesta.

Ahora vamos a crear un nuevo componente para editar los presupuestos. Para ello, completamos en la consola:

```
ng g c presupuestos/editpres --spec false
```

Como hacemos con todos los nuevos componente, a continuación vamos añadir el nuevo componente al routing en el módulo, añadiendo para ello en el archivo `app.module.ts`:

```
src/app/app.module.ts
```

```
...  
{ path: 'editpres/:id', component: PresupuestosComponent },  
...
```

Vemos como en este caso añadimos a la ruta el símbolo dos puntos seguido de una referencia id para el objeto a editar, de esta manera esta ruta nos permitirá acceder a la edición de un objeto concreto.

A continuación vamos a crear la vista del componente con un formulario similar al empleado para añadir nuevos presupuestos. Para ello, en el archivo editpres.component.html añadimos el siguiente código:

```
src/app/presupuestos/editpres/editpres.component.html
```

```
<div class="row">  
  <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset 2">  
    <h2>Editar Presupuesto</h2> ①  
    <a class="btn btn-primary float-md-right"  
routerLink="/presupuestos">Regresar al listado</a>  
    <br>  
    <form [formGroup]="presupuestoForm" (ngSubmit)="onSubmit()">  
      <div class="form-group">  
        <label for="proveedor">Proveedor</label>  
        <i class="fa fa-check-circle check"  
          *ngIf="presupuestoForm.controls.proveedor.valid"></i>  
        <i class="fa fa-exclamation-circle uncheck"  
          *ngIf="presupuestoForm.controls.proveedor.invalid &&  
presupuestoForm.controls.proveedor.touched"></i>  
        <input type="text"  
          class="form-control"  
          id="proveedor"  
          formControlName="proveedor"  
          [(ngModel)]="presupuesto.proveedor"> ②  
        <p class="alert alert-danger"  
          *ngIf="presupuestoForm.controls.proveedor.invalid &&
```

```

    presupuestoForm.controls.proveedor.touched">
    El campo Proveedor es obligatorio.
  </p>
</div>
<div class="form-group">
  <label for="fecha">Fecha Presupuesto</label>
  <i class="fa fa-check-circle check"
    *ngIf="presupuestoForm.controls.fecha.valid"></i>
  <i class="fa fa-exclamation-circle uncheck"
    *ngIf="presupuestoForm.controls.fecha.invalid &&
    presupuestoForm.controls.fecha.touched"></i>
  <input type="date"
    class="form-control"
    id="fecha"
    formControlName="fecha"
    [(ngModel)]="presupuesto.fecha"> ②
  <p class="alert alert-danger"
    *ngIf="presupuestoForm.controls.fecha.invalid &&
    presupuestoForm.controls.fecha.touched">
    El campo Fecha es obligatorio.
  </p>
</div>
<div class="form-group">
  <label for="concepto">Concepto</label>
  <i class="fa fa-check-circle check"
    *ngIf="presupuestoForm.controls.concepto.valid"></i>
  <i class="fa fa-exclamation-circle uncheck"
    *ngIf="presupuestoForm.controls.concepto.invalid &&
    presupuestoForm.controls.concepto.touched"></i>
  <input type="text"
    class="form-control"
    id="concepto"
    formControlName="concepto"
    [(ngModel)]="presupuesto.concepto"> ②
  <p class="alert alert-danger"
    *ngIf="presupuestoForm.controls.concepto.invalid &&
    presupuestoForm.controls.concepto.touched">
    El campo Concepto es obligatorio y debe tener más de 10 caracteres.
  </p>
</div>
<div class="form-group">
  <label for="base">Base Imponible</label>

```

```

<i class="fa fa-check-circle check"
  *ngIf="presupuestoForm.controls.base.valid"></i>
<i class="fa fa-exclamation-circle uncheck"
  *ngIf="presupuestoForm.controls.base.invalid &&
  presupuestoForm.controls.base.touched"></i>
<input type="number"
  class="form-control"
  id="base"
  formControlName="base"
  [(ngModel)]="presupuesto.base"> ②
<p class="alert alert-danger"
  *ngIf="presupuestoForm.controls.base.invalid &&
  presupuestoForm.controls.base.touched">
  El campo Base Imponible es obligatorio.
</p>
</div>
<div class="form-group">
  <label for="tipo">Tipo de IVA</label>
  <i class="fa fa-check-circle check"
    *ngIf="presupuestoForm.controls.tipo.valid"></i>
  <i class="fa fa-exclamation-circle uncheck"
    *ngIf="presupuestoForm.controls.tipo.invalid &&
    presupuestoForm.controls.tipo.touched"></i>
  <select class="form-control"
    id="tipo"
    formControlName="tipo"
    [(ngModel)]="presupuesto.tipo"> ②
    <option value="">Seleccione...</option>
    <option value=0> 0 %</option>
    <option value=0.04> 4 %</option>
    <option value=0.10>10 %</option>
    <option value=0.21>21 %</option>
  </select>
  <p class="alert alert-danger"
    *ngIf="presupuestoForm.controls.tipo.invalid &&
    presupuestoForm.controls.tipo.touched">
    El campo Tipo de IVA es obligatorio.
  </p>
</div>
<div class="form-group">
  <label for="iva">Importe IVA</label>
  <input type="number"

```

```

        class="form-control"
        id="iva"
        formControlName="iva"
        [(ngModel)]= "presupuestoForm.value.iva"
        disabled>
    </div>
    <div class="form-group">
        <label for="total">Total Factura IVA Incluido</label>
        <input type="number"
            class="form-control"
            id="total"
            formControlName="total"
            [(ngModel)]= "presupuestoForm.value.total"
            disabled>
    </div>
    <button type="submit"
        class="btn btn-primary"
        [disabled]="!presupuestoForm.valid"
        >Guardar Cambios</button>
    <a class="btn btn-danger" routerLink="/presupuestos">Cancelar</a>
    <p class="alert alert-danger" *ngIf="!presupuestoForm.valid">
        Por favor complete todos los campos
    </p>
</form>
<hr>
</div>
</div>

```

Este código HTML es el mismo que tenemos en el componente para añadir los registros de presupuestos con las siguientes modificaciones:

- ① Modificamos el texto previo al formulario.
- ② Añadimos a los campos editables la directiva ngModel asociada a la propiedad del objeto presupuesto correspondiente a cada campo.
- ③ Cambiamos el texto del botón submit.

A continuación, tenemos que editar el código del componente en el archivo editpres.component.ts.

Comenzamos añadiendo las importaciones:

```
src/app/presupuestos/editpres/editpres.component.ts
```

```
...
import { FormControl, FormGroup, FormBuilder, Validators } from
'@angular/forms';
import { PresupuestosService } from '../servicios/presupuestos.service';
import { Router, ActivatedRoute } from '@angular/router';
...
```

Continuamos dentro de la clase declarando las propiedades y objetos que vamos a utilizar en el componente:

```
...
presupuestoForm: FormGroup;
presupuesto: any;
base: any;
tipo: any;
iva: any = 0;
total: any = 0;

id: string;
...
```

A continuación, editamos el constructor para añadir el servicio y las clases de Angular que nos permiten recuperar la id de la url a la que hemos navegado para suscribirnos al método getPresupuesto, obtener el registro del presupuesto de esa id e igualarlo al objeto presupuesto del componente:

```
...
constructor(private pf: FormBuilder,
             private presupuestoService: PresupuestosService,
             private router: Router,
             private activatedRouter: ActivatedRoute) {
  this.activatedRouter.params
```



```

        .subscribe( parametros => {
            this.id = parametros['id'];
            this.presupuestoService.getPresupuesto( this.id)
                .subscribe( presupuesto => this.presupuesto = presupuesto)
        });
    }
    ...

```

Continuamos añadiendo el mismo código que empleamos en el componente para crear el formulario y observar sus cambios:

```

...
ngOnInit() {

    this.presupuestoForm = this.pf.group({
        proveedor: ['', Validators.required ],
        fecha: ['', Validators.required ],
        concepto: ['', [ Validators.required, Validators.minLength(10)] ],
        base: ['', Validators.required ],
        tipo: ['', Validators.required ],
        iva: this.iva ,
        total: this.total
    });

    this.onChange();

}

onChange(): void {
    this.presupuestoForm.valueChanges.subscribe(valor => {
        this.base = valor.base;
        this.tipo = valor.tipo;
        this.presupuestoForm.value.iva = this.base * this.tipo;
        this.presupuestoForm.value.total = this.base + (this.base * this.tipo);
    });
}
...

```

Y finalmente, añadimos el nuevo método `onSubmit` que enviará además del objeto presupuesto la `id` a modificar a través del método `putPresupuesto` del servicio:

```
...
onSubmit() {
  this.presupuesto = this.savePresupuesto();
  this.presupuestoService.putPresupuesto( this.presupuesto, this.id )
    .subscribe(newpre => {
      this.router.navigate(['/presupuestos'])
    })
}

savePresupuesto() {

  const savePresupuesto = {

    proveedor: this.presupuestoForm.get('proveedor').value,
    fecha: this.presupuestoForm.get('fecha').value,
    concepto: this.presupuestoForm.get('concepto').value,
    base: this.presupuestoForm.get('base').value,
    tipo: this.presupuestoForm.get('tipo').value,
    iva: this.presupuestoForm.get('iva').value,
    total: this.presupuestoForm.get('total').value

  };
  return savePresupuesto;
}
...
```

En resumen, la lógica que realiza este componente es la siguiente:

- Utiliza la propiedad `params` de `activatedRouter` para recuperar la `id` de la dirección url a la que accedemos y emplea el método `getPresupuesto` del servicio para recuperar ese registro de la base de datos y cargarlo en el objeto del componente.

- Carga en la vista los datos del objeto para que puedan ser modificados.
- Al pulsar en enviar el método submit modifica el objeto y lo envía junto con su id al método putPresupuesto del servicio para que lo actualice en la base de datos.

Para finalizar el componente añadimos a su archivo editpres.component.css las clases para el formulario:

src/app/presupuestos/editpres/editpres.component.ts

```
input.ng-invalid.ng-touched {
  border-left: 5px solid #a94442;
}

input.ng-valid.ng-dirty {
  border-left: 5px solid #42A948;
}

select.ng-invalid.ng-touched {
  border-left: 5px solid #a94442;
}

select.ng-valid.ng-dirty {
  border-left: 5px solid #42A948;
}

.check {
  color: #42A948;
}

.uncheck {
  color: #a94442;
}
```

El componente ya está operativo, pero para acceder a él, necesitamos introducir en el navegador una url con la id del registro. La mejor forma de acceder a esta url es modificar la tabla de registros para crear un botón o link de acceso.

Para ello, en el componente presupuestos y dentro de su vista en el archivo presupuestos.component.html, añadimos a la tabla:

```
src/app/presupuestos/presupuestos/presupuestos.component.html
```

```
...
    <th class="text-right">Total</th>
    <th>Editar</th>
  </tr>
</thead>
...
...
    <td>
      <button type="button" class="btn btn-success"
        routerLink="/editpres/{{ presupuesto.id$ }}">Editar</button>
    </td>
  </tr>
...
```

En el que introducimos en el link de la ruta, el valor de la id de cada presupuesto que será cargado a través de ngFor.

Ahora en el navegador, en el listado de presupuestos tendremos acceso a la edición de cada uno de ellos:

Compras App Inicio Proveedores Presupuestos

Listado de Presupuestos Añadir nuevo presupuesto

Proveedor	Fecha	Concepto	Suma	Importe IVA	Total	Editar
Amazon	30-09-2017	Discos duros	70.00	7.00	77.00	Editar
Endesa	25-09-2017	Suministro Eléctrico	340.00	71.40	411.40	Editar

Y tras pulsar en el botón editar accederemos a cualquier registro, pudiendo editar y guardar los cambios de cada campo.

Base Imponible

70

Tipo de IVA

10 %

Importe IVA

7

Total Factura IVA Incluido

77

Guardar Cambios

Cancelar

Y mantenemos la funcionalidad reactiva de modificación de los campos de manera inmediata.

10.5. Servicio HTTP: Delete

Para finalizar las operaciones CRUD, vamos a añadir a nuestra aplicación la posibilidad de borrar los registros en la base de datos.

Para ello en primer lugar añadimos a nuestro servicio el método para eliminar registros de la base de datos. En la clase del archivo `presupuestos.service.ts` añadimos:

`src/app/servicios/presupuestos.service.ts`

```
...
delPresupuesto ( id$: string ) {
  const url = `${ this.preURL }/${ id$ }.json`;
  return this.http.delete( url )
    .map( res => res.json());
}
...
```

Que funciona de la misma manera que los anteriores pero empleando la petición `http delete`.

Ahora vamos a añadir a nuestro componente de presupuestos un botón en la tabla para eliminar cada registro. Para ello en el archivo `presupuestos.component.html`, añadimos en la tabla

`src/app/presupuestos/editpres/editpres.component.html`

```
...
<th>Editar</th>
<th>Eliminar</th>
</tr>
...

...
<td>
  <button class="btn btn-danger"
    (click)="eliminarPresupuesto(presupuesto.id$)">Eliminar</button>

```

```

    </td>
  </tr>
</tbody>
...

```

Que al pulsarlo llamará al método eliminar presupuesto que a continuación añadimos en el componente, dentro de la clase del archivo presupuestos.component.ts de la siguiente manera:

src/app/servicios/presupuestos.service.ts

```

...
eliminarPresupuesto(id$) {
  this.presupuestoService.delPresupuesto(id$)
    .subscribe( res => {
      console.log(res);
    })
}
...

```

Este método del componente, a su vez llamará al método delPresupuesto del servicio, pasándole la id del registro para que lo elimine con la petición delete en la base de datos.

De esta manera, pulsando el botón eliminar en la tabla se eliminará el correspondiente registro.

Compras App

Inicio Proveedores Presupuestos

Listado de Presupuestos

Añadir nuevo presupuesto

Proveedor	Fecha	Concepto	Suma	Importe IVA	Total	Editar	Eliminar
Aceros del Norte	10-12-2017	Suministro de piezas	7,000.00	1,470.00	8,470.00	Editar	Eliminar

Con esta solución, la funcionalidad no está finalizada, ya que al borrar un elemento la tabla no se actualiza debido a que el array del componente presupuestos, mantiene los datos que cargó de la base de datos al inicializar el componente.

Para que actualice la tabla en cada borrado, una de las formas es llamar al método `getPresupuestos` del servicio, dentro del método `delPresupuesto`.

Modificamos el archivo `presupuestos.component.ts` de la siguiente manera:

`src/app/servicios/presupuestos.service.ts`

```
...
eliminarPresupuesto(id$) {

  this.presupuestosService.delPresupuesto(id$)
    .subscribe( res => {
      this.presupuestos = [];
      this.presupuestosService.getPresupuestos()
        .subscribe(presupuestos => {
          for ( const id$ in presupuestos ) {
            const p = presupuestos[id$];
            p.id$ = id$;
            this.presupuestos.push(presupuestos[id$]);
          }
        })
    });
}
...
```

Y así conseguimos, que nuestra tabla se actualice tras cada operación de borrado.

11 Gestión de Autorizaciones

Cualquier aplicación de cierta entidad necesita un sistema de accesos para ser utilizado por diversos usuarios.

Vamos a ver a continuación como podemos implementar en Angular un sistema de usuarios con almacenamiento de sus datos en Firebase.

11.1 Registro

En primer lugar vamos a crear un componente para el registro de usuarios. Para ello en la consola de nuestro equipo tecleamos:

```
ng g c autenticacion/registro --spec false
```

Añadimos el componente a las rutas en el archivo app.module.ts de la siguiente manera:

```
src/app/app.module.ts
```

```
...  
  { path: 'registro', component: RegistroComponent },  
...
```

E incorporamos el link al menú en el archivo header.component.html de la siguiente manera:

```
src/app/header/header.component.html
```

```
...  
  <li class="nav-item" routerLinkActive="active">  
    <a class="nav-link" routerLink="/registro">Registro </a>  
  </li>  
...
```

Ahora comenzamos añadiendo el código html de un formulario de tipo reactive de registro, en la vista del archivo registro.component.html:

src/app/autenticación/registro/registro.component.html

```
<div class="row">
  <div class="col-md-6 offset-md-3">
    <form [formGroup]="registroForm" (ngSubmit)="onSubmit()">
      <h4>Introduzca sus datos de registro</h4>
      <label>Correo Electrónico</label>
      <input type="email" id="inputEmail" class="form-control"
        formControlName="email" autofocus>
      <label>Contraseña</label>
      <input type="password" id="inputPassword" class="form-control"
        formControlName="password">
      <hr>
      <button type="submit" class="btn btn-primary">Registrar</button>
    </form>
  </div>
</div>
```

En el que hemos añadido el formulario registroForm y los campos de correo electrónico y contraseñas con la declaración de su atributo formControlName.

A continuación comenzamos a editar el componente en registro.component.ts añadiendo las importaciones para formulario:

src/app/autenticación/registro/registro.component.ts

```
...
import { FormControl, FormGroup, FormBuilder, Validators } from
'@angular/forms';
...
```

y dentro de la clase declaramos los objetos del formulario para recoger los datos de email y contraseña:

```
...
registroForm: FormGroup;
userdata: any;
...
```

El constructor:

```
...
constructor(private formBuilder: FormBuilder) {}
...
```

Y los métodos:

```
...
ngOnInit() {
  this.registroForm = this.formBuilder.group({
    'email': ['', [
      Validators.required,
      Validators.email
    ]
  ],
  'password': ['', [
    Validators.required,
    Validators.pattern('^(?=.*[0-9])(?=.*[a-zA-Z])([a-zA-Z0-9]+)$'), ①
    Validators.minLength(6)
  ]
  ]
  });
  this.registroForm.valueChanges.subscribe(data =>
this.onValueChanged(data));
  this.onValueChanged();

}

onSubmit() {
  this.userdata = this.saveUserdata(); ②
}
```

```

saveUserdata() {

  const saveUserdata = {

    email: this.registroForm.get('email').value,
    password: this.registroForm.get('password').value,
  };
  return saveUserdata;
}

```

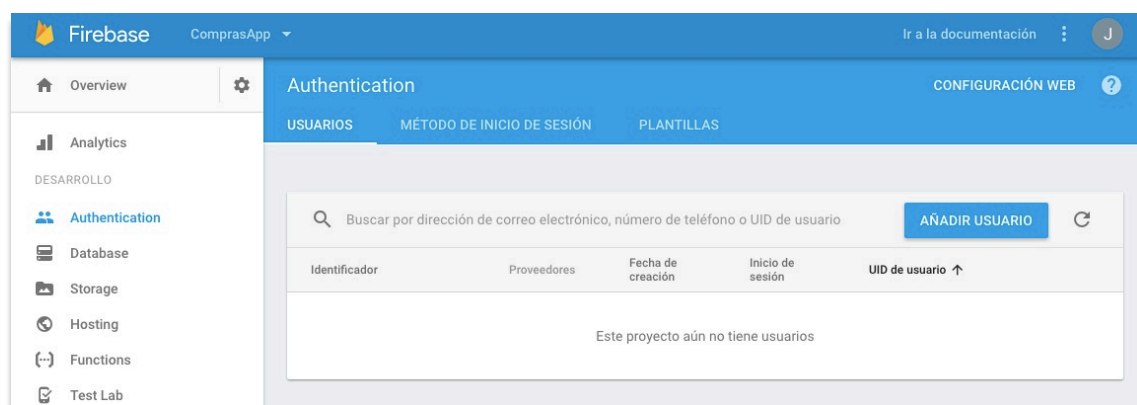
Como novedad, al campo password le hemos ① añadido un patrón de validación programática con una expresión regular, para que la contraseña tenga al menos un número y al menos una letra entre sus caracteres.

De momento, simplemente hemos creado los método del formulario, introduciendo los campos del formulario y ② guardando sus valores en el objeto userdata.

Con lo cual, el siguiente paso e imprescindible es crear el servicio que nos conectará con el proveedor de autenticación, en nuestro ejemplo, también Firebase.

Por tanto, debemos crear nuestro servicio, pero previamente vamos a configurar Firebase para poder emplear su herramienta de autenticación.

En la consola de Firebase, pulsamos en la opción Authentication del menú lateral izquierdo:



Ahora pulsamos en la pestaña método de inicio de sesión, pulsamos en la opción Correo electrónico/contraseña y marcamos la opción Habilitar:



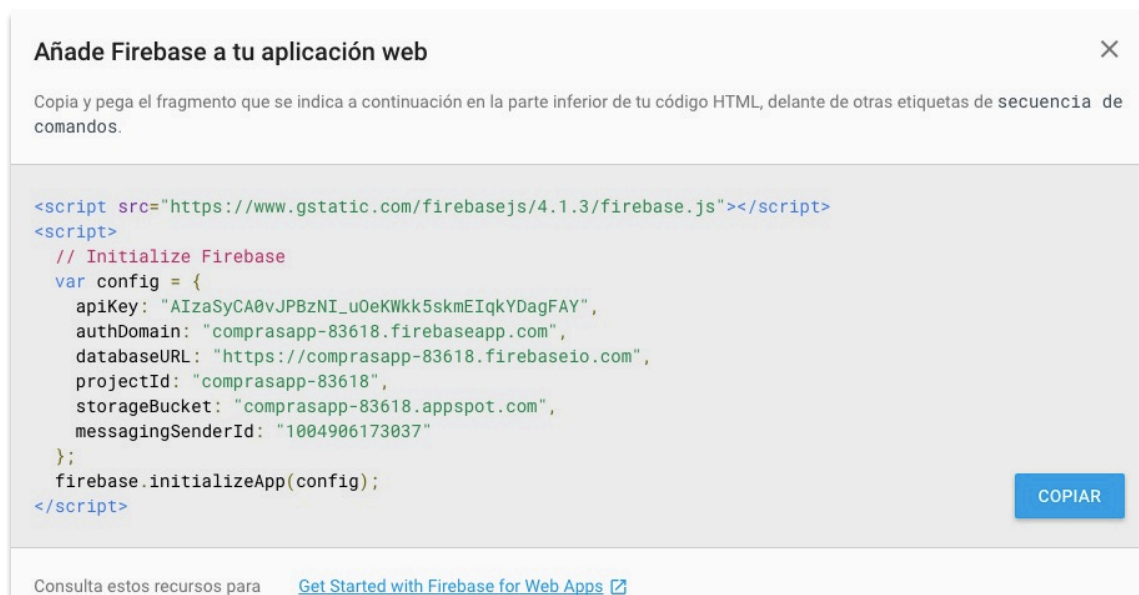
Correo electrónico/contraseña

Habilitar ☒

Permite a los usuarios registrarse con su dirección de correo electrónico y contraseña. Nuestros SDK también proporcionan la verificación de la dirección de correo, la recuperación de contraseñas y las primitivas de cambio de dirección de correo. [Más información](#)

CANCELAR GUARDAR

Y finalmente, pulsamos en la opción configuración web que nos lanza un modo con los datos que necesitaremos posteriormente para conectarnos a este servicio de autenticación:



Añade Firebase a tu aplicación web

Copia y pega el fragmento que se indica a continuación en la parte inferior de tu código HTML, delante de otras etiquetas de secuencia de comandos.

```
<script src="https://www.gstatic.com/firebasejs/4.1.3/firebase.js"></script>
<script>
  // Initialize Firebase
  var config = {
    apiKey: "AIzaSyCA0vJPBzNI_uOeKWk5skmEIqkYDagFAY",
    authDomain: "comprasapp-83618.firebaseio.com",
    databaseURL: "https://comprasapp-83618.firebaseio.com",
    projectId: "comprasapp-83618",
    storageBucket: "comprasapp-83618.appspot.com",
    messagingSenderId: "1004906173037"
  };
  firebase.initializeApp(config);
</script>
```

COPIAR

Consulta estos recursos para [Get Started with Firebase for Web Apps](#)

A continuación vamos a instalar mediante NPM las librerías firebase y angularfire2, mediante el siguiente comando en la consola del equipo:

```
npm install firebase angularfire2 --save
```

El siguiente paso es la creación del servicio Angular en nuestra aplicación, para lo cual también desde la consola completamos:

```
ng g service servicios/autenticacion --spec false
```

Y lo añadimos al archivo app.module.ts de la forma habitual:

```
src/app/app.module.ts
```

```
...
import { AutenticacionService } from './servicios/autenticacion.service';
...

...
providers: [ProveedoresService,
            PresupuestosService,
            AutenticacionService],
...

```

Comenzamos ahora a editar nuestro servicio en el archivo autenticacion.service.ts añadiendo en primer lugar la importación de las clases necesarias del paquete firebase:

```
src/app/servicios/autenticacion.service.ts
```

```
...
import * as firebase from 'firebase';
...

```

Y añadimos el método registroUsuario que recogerá el parámetro userdata del componente y empleará el método createUserWithEmailAndPassword de la librería firebase, para enviar a Firebase su usuario y contraseña.

```
...
registroUsuario (userdata) {
    firebase.auth().createUserWithEmailAndPassword(userdata.email,
    userdata.password)
    .catch(

```

```

        error => {
            console.log(error);
        }
    )
}
...

```

En este caso no empleamos peticiones http para conectarnos a Firebase, si no su propia librería, por lo que necesitaremos que se cargue la conexión al iniciar la aplicación.

Para ello, vamos a modificar nuestro componente raíz en el archivo `app.component.ts`. En primer lugar modificamos las importaciones:

`src/app/app.component.ts`

```

import { Component, OnInit } from '@angular/core';
import * as firebase from 'firebase';
...

```

Y, dentro de la clase, añadimos en `ngOnInit` las claves del servicio de firebase que obtuvimos anteriormente, para que se establezca la inicialización del servicio:

```

...
export class AppComponent implements OnInit {
    ngOnInit () {
        firebase.initializeApp({
            apiKey: 'AlzaSyCA0vJPBzNI_uOeKWkk5skmElqkYDagFAY',
            authDomain: 'comprasapp-83618.firebaseio.com'
        });
    }
}
...

```

Con estos elementos configurados, nos queda llamar al servicio desde el componente de registro de usuarios.

Vamos a modificar nuestro archivo `registro.component.ts`. Comenzamos añadiendo la importación del servicio y del paquete de routing para poder redirigir tras el registro.

```
src/app/autenticación/registro/registro.component.ts
```

```
...
import { AutenticacionService } from '../servicios/autenticacion.service';
import { Router, ActivatedRoute } from '@angular/router';
...
```

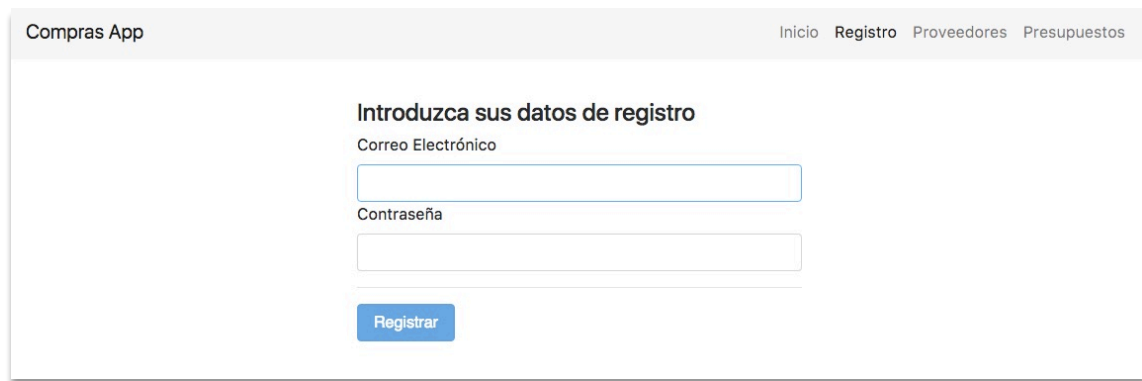
Seguidamente, modificamos el constructor para inyectar el servicio y añadimos los objetos para el enrutado:

```
...
constructor(private FormBuilder: FormBuilder,
             private authService: AutenticacionService,
             private router: Router,
             private activatedRouter: ActivatedRoute
            ) {}
...
```

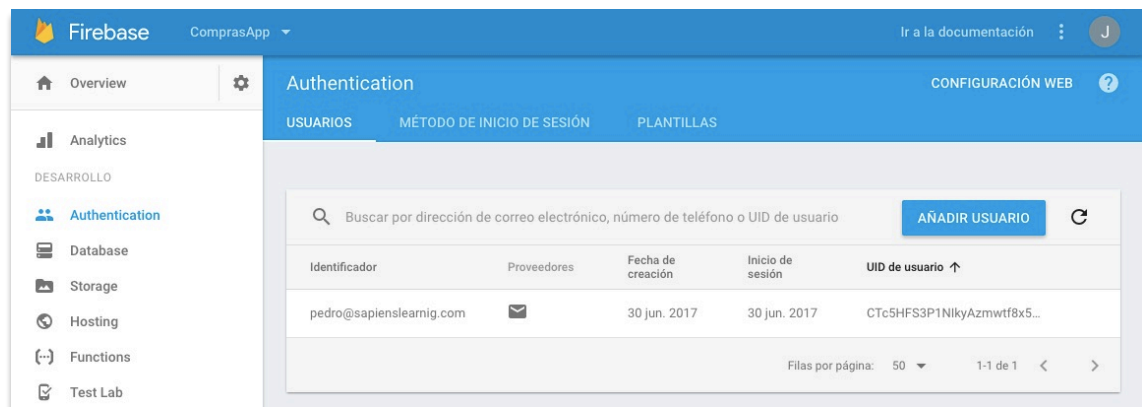
Y añadimos al método `onRegistro`, la llamada al método del servicio `registroUsuario`, y la redirección al inicio de la aplicación tras el registro:

```
...
onSubmit() {
  this.userdata = this.saveUserdata();
  this.authService.registroUsuario(this.userdata);
  this.router.navigate(['/inicio'])
}
...
```


De esta manera, nuestra página de registro quedará lista para crear usuarios:



Y cuando registremos un usuario correctamente, podremos comprobar en la consola de Firebase con se han creado sus credenciales:



Identificador	Proveedores	Fecha de creación	Inicio de sesión	UID de usuario ↑
pedro@sapienslearnig.com		30 jun. 2017	30 jun. 2017	CTc5HFS3P1NlkyAzmwt8x5...

Nuestro componente y servicios están operativos pero carecen de ayudas para mostrar las validaciones, con lo cual debemos añadirlas. Además emplearemos las técnicas de observables, para detectar los cambios en los campos cuando se corrijan.

Comenzamos por editar el componente en el archivo `registro.component.ts`, para añadir dos objetos en la clase antes del constructor:

```
src/app/autenticación/registro/registro.component.ts
```

```
...  
erroresForm = {  
  'email': '',  
  'password': ''  
}
```

```

mensajesValidacion = {
  'email': {
    'required': 'Email obligatorio',
    'email': 'Introduzca una dirección email correcta'
  },
  'password': {
    'required': 'Contraseña obligatoria',
    'pattern': 'La contraseña debe tener al menos una letra un número ',
    'minlength': 'y más de 6 caracteres'
  }
}
...

```

Seguidamente creamos un método para introducir los objetos anteriores según el estado de cada campo:

```

...
onValueChanged(data?: any) {
  if (!this.registroForm) { return; }
  const form = this.registroForm;
  for (const field in this.erroresForm) {

    this.erroresForm[field] = '';
    const control = form.get(field);
    if (control && control.dirty && !control.valid) {
      const messages = this.mensajesValidacion[field];
      for (const key in control.errors) {
        this.erroresForm[field] += messages[key] + ' ';
      }
    }
  }
}
...

```

Dentro de `ngOnInit` añadimos la llamada a la clase `valueChanges` de Angular para que al modificarse los valores del formulario nos suscribamos al método `onValueChanged`:

```

...
ngOnInit() {
...
  this.registroForm.valueChanges.subscribe(data =>this.onValueChanged(data));
  this.onValueChanged();
}
...

```

Nos queda modificar la plantilla para implementar los mensajes de validación y restringir el botón a la validez del formulario.

En el archivo registro.component.html sustituimos:

src/app/autenticación/registro/registro.component.html

```

<div class="row">
  <div class="col-md-6 offset-md-3">
    <form [formGroup]="registroForm" (ngSubmit)="onSubmit()">
      <h4>Introduzca sus datos de registro</h4>
      <label>Correo Electrónico</label>
      <input type="email" id="inputEmail" class="form-control"
        formControlName="email" required autofocus>
      <p class="alert alert-danger" *ngIf="erroresForm.email">
        {{ erroresForm.email }}
      </p>
      <label>Contraseña</label>
      <input type="password" id="inputPassword" class="form-control"
        formControlName="password" required>
      <p class="alert alert-danger" *ngIf="erroresForm.password">
        {{ erroresForm.password }}
      </p>
      <hr>
      <button type="submit"
        class="btn btn-primary"
        [disabled]="!registroForm.valid"
        >Registrar</button>
    </form>
  </div>
</div>

```

Para finalmente comprobar en nuestro navegador, como se muestran los mensajes si los campos no son validados:



The image shows a registration form titled "Introduzca sus datos de registro". It contains two input fields: "Correo Electrónico" and "Contraseña". The "Correo Electrónico" field has a red error message below it: "Introduzca una dirección email correcta". The "Contraseña" field has a red error message below it: "La contraseña debe tener al menos una letra un número y más de 6 caracteres". At the bottom of the form is a blue button labeled "Registrar".

Introduzca sus datos de registro

Correo Electrónico

pl

Introduzca una dirección email correcta

Contraseña

*

La contraseña debe tener al menos una letra un número y más de 6 caracteres

Registrar

11.2 Inicio de sesión

Ahora que ya tenemos nuestro componente para registrar usuarios, vamos a crear un componente para iniciar sesión en nuestra aplicación.

En la consola completamos:

```
ng g c autenticacion/inises --spec false
```

De entrada, introducimos su ruta en el archivo app.module.ts:

```
src/app/app.module.ts
```

```
...  
  { path: 'iniciosesion', component: InisesComponent },  
...
```

Y en la barra de navegación en el archivo header.component.html, eliminamos el link de registro y creamos un botón de inicio de sesión:

```
src/app/header/header.component.html
```

```
...  
<li class="nav-item">  
  <button class="btn btn-primary float-md-right"  
    routerLink="/iniciosesion">Inicie sesión</button>  
</li>  
...
```

A continuación, añadimos en la plantilla un formulario similar al de registro con las modificaciones necesarias para un inicio de sesión.

En el archivo inises.component.html añadimos:

```
src/app/autenticacion/inises/inises.component.html
```

```
<div class="row">
  <div class="col-md-6 offset-md-3">
    <form [formGroup]="loginForm" (ngSubmit)="onSubmit()">
      <h4>Introduzca los siguientes datos:</h4>
      <label>Correo Electrónico</label>
      <input type="email" id="inputEmail" class="form-control"
        formControlName="email" required autofocus>
      <label>Contraseña</label>
      <input type="password" id="inputPassword" class="form-control"
        formControlName="password" required>
      <hr>
      <button type="submit"
        class="btn btn-primary"
        [disabled]="!loginForm.valid"
        >Inicie sesión</button>
    </form>
  </div>
</div>
```

Y de nuevo vamos a añadir la lógica del formulario en el archivo, comenzando por las importaciones:

```
src/app/autenticación/inises/inises.component.ts
```

```
...
import { AutenticacionService } from '../servicios/autenticacion.service';
import { Router, ActivatedRoute } from '@angular/router';
...
```

Igual que en el registro, añadimos los objetos, modificamos el constructor para inyectar el servicio y añadimos los objetos para el enrutado:

```

...
loginForm: FormGroup;
userdata: any;

constructor(private formBuilder: FormBuilder,
             private authService: AutenticacionService,
             private router: Router,
             private activatedRouter: ActivatedRoute
            ) {}
...

```

Iniciamos el formulario en ngOnInit:

```

...
ngOnInit() {
  this.loginForm = this.formBuilder.group({
    'email': ['', [
      Validators.required,
      Validators.email
    ]],
    'password': ['', [
      Validators.required,
      Validators.pattern('^(?=.*[0-9])(?=.*[a-zA-Z])([a-zA-Z0-9]+)$'),
      Validators.minLength(6)
    ]
  ]
});
}
...

```

Y finalmente añadimos los métodos de envío y guardado de los datos del usuario.

```

...
onSubmit() {
  this.userdata = this.saveUserdata();
  this.authService.inicioSesion(this.userdata);
}

```

```

saveUserdata() {

    const saveUserdata = {
        email: this.loginForm.get('email').value,
        password: this.loginForm.get('password').value,
    };
    return saveUserdata;
}
...

```

En este componente acabamos de crear dentro del método onSubmit, una llamada al método inicioSesion de nuestro servicio, por tanto debemos crearlo en este. Vamos a editar el archivo autenticacion.service.ts para añadir en primer lugar la importación de routing Angular:

src/app/servicios/autenticacion.service.ts

```

...
import { Router, ActivatedRoute } from '@angular/router';
...

```

Añadimos Router al constructor:

```

...
constructor(private router: Router,
             private activatedRouter: ActivatedRoute) {}
...

```

y en la clase añadimos el método inicioSesion:

```

...
inicioSesion (userdata) {
    firebase.auth().signInWithEmailAndPassword(userdata.email,
        userdata.password)
        .then(response => {
            console.log(response);
            this.router.navigate(['/inicio']);
        });
}

```



```

    })
    .catch(
      error => {
        console.log(error);
      }
    )
  }
  ...

```

Que empleará el método `signInWithEmailAndPassword` de firebase para enviar los datos de inicio sesión. En caso de respuesta correcta, recibimos un objeto `response` y redirigimos a la página, y en caso de error recibimos el objeto del mismo nombre.

Nuestra página de inicio de sesión ya estará disponible en el navegador y si iniciamos sesión correctamente, obtendremos en la consola un objeto con todos los detalles del inicio de sesión y la aplicación nos llevará al inicio:



Con esta lógica hemos implementado correctamente el sistema de autenticación de Firebase, pero no disponemos de opciones para gestionar que ocurre si se inicia la sesión correctamente o viceversa.

Para poder gestionar dinámicamente las sesiones en nuestra aplicación vamos a añadir en el servicio dos nuevos métodos de la librería firebase.

Añadimos en la clase del archivo `autenticacion.service.ts` los siguientes métodos:

`src/app/servicios/autenticacion.service.ts`

```
...
  isAuthenticated() {
    const user = firebase.auth().currentUser;
    if (user) {
      return true;
    } else {
      return false;
    }
  }
  ...
```

Este primero, `isAuthenticated`, emplea el método `currentUser` del paquete `firebase`, para determinar si el usuario está conectado, por lo que este método devolverá `true` si es así o `false` en caso contrario.

Y también añadimos:

```
...
  logout() {
    firebase.auth().signOut();
  }
  ...
```

Que empleará el método `signOut` de `firebase` para cerrar la sesión.

Ahora que ya tenemos estos dos métodos en nuestro servicio, podemos emplearlos en nuestra barra de navegación para que sea sensible al inicio de sesión.

Sustituimos su código en header.component.html por el siguiente:

src/app/header/header.component.html

```
<nav class="navbar navbar-light bg-faded rounded navbar-togglerable-md">
  <button class="navbar-toggler navbar-toggler-right"
    type="button" data-toggle="collapse" data-target="#containerNavbar"
    aria-controls="containerNavbar" aria-expanded="false"
    aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <a class="navbar-brand" href="#">Compras App</a>
  <div class="collapse navbar-collapse" id="containerNavbar">
    <ul class="navbar-nav mr-auto w-100 justify-content-end">
      <li class="nav-item" routerLinkActive="active"
        [routerLinkActiveOptions]="{exact:true}" *ngIf="isAuth()"> ①
        <a class="nav-link" routerLink="/">Inicio </a>
      </li>
      <li class="nav-item" routerLinkActive="active" *ngIf="isAuth()"> ①
        <a class="nav-link" routerLink="/proveedores">Proveedores </a>
      </li>
      <li class="nav-item" routerLinkActive="active" *ngIf="isAuth()"> ①
        <a class="nav-link" routerLink="/presupuestos">Presupuestos </a>
      </li>
      <li class="nav-item" *ngIf="!isAuth()"> ①
        <button class="btn btn-primary float-md-right"
          routerLink="/iniciosesion">Inicie sesión</button>
      </li>
      <li class="nav-item" *ngIf="isAuth()"> ①
        <button class="btn btn-primary float-md-right"
          (click)="onLogout()">Cerrar sesión</button> ②
      </li>
    </ul>
  </div>
</nav>
```

En el cual:

① Añadimos a cada link del menú un `ngIf` para que solo se muestre si el método `isAuth` es `true`, es decir si el usuario está conectado y al revés en el caso del botón iniciar sesión.

② Añadimos el botón Cerrar Sesión, que solo se mostrará si el usuario está conectado, para que desencadene el método `onLogout` y cierre sesión

Como estamos utilizando nuevos métodos que llamarán a métodos del servicio `autenticacion.service.ts`, debemos añadirlos a este componente, con lo cual en el archivo `header.component.ts` añadimos en primer lugar las importaciones del servicio y de Angular Router:

`src/app/header/header.component.ts`

```
...
import { AutenticacionService } from '../servicios/autenticacion.service';
import { Router, ActivatedRoute } from '@angular/router';
...
```

Seguidamente modificamos el constructor:

```
...
constructor(private authService: AutenticacionService,
             private router: Router,
             private activatedRouter: ActivatedRoute) {}
...
```

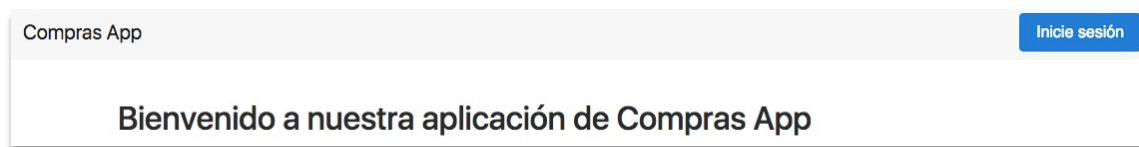
Añadimos el método `isAuth` que devuelve si el usuario está conectado o no:

```
...
isAuth() {
  return this.authService.isAuthenticated();
}
...
```

Y el método onLogout que cerrará la sesión y redireccionará a la página de inicio:

```
...
onLogout() {
  this.authService.logout();
  this.router.navigate(['/inicio'])
}
...
```

De esta manera nuestra barra de menú, mostrará sus elemento y el botón de cierre de sesión solo si el usuario esta conectado, y en caso contrario mostrará el botón de inicio de sesión:



También podemos emplear los métodos de nuestro servicio en el formulario de inicio de sesión para que muestre un mensaje cuando el inicio de sesión no sea correcto.

Para ello, en el archivo inises.component.ts, creamos dentro de la clase una propiedad mensaje booleana, inicializándola como false, y declaramos el método isAuth:

src/app/autenticacion/inises/inises.component.ts

```
...
mensaje = false;
...
isAuth() {
  return this.authService.isAuthenticated();
}
...
```

A continuación, modificamos el método onSubmit para que si una vez que se envíen los datos de inicio de sesión a Firebase, si isAuthenticated es false, es decir, no se haya iniciado sesión correctamente, la propiedad mensaje pase a true:

```
...
onSubmit() {
  this.userdata = this.saveUserdata();
  this.authService.inicioSesion(this.userdata);
  setTimeout(() => {
    if (this.isAuthenticated === false) {
      this.mensaje = true
    }
  }, 2000);
}
...
```

Este if, lo hemos metido dentro de un método JavaScript setTimeout para que no se ejecute hasta que transcurran 2000 milisegundos, de tal forma que no se ejecute antes de recibir la respuesta del servidor de Firebase.

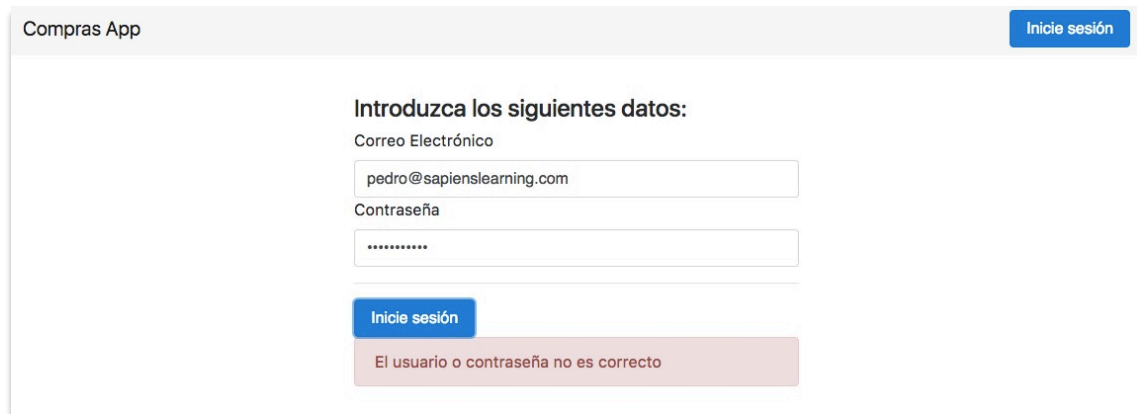
Como tenemos una nueva propiedad mensaje, que será true cuando no se haya producido el inicio de sesión la utilizamos en la plantilla para mostrar un mensaje.

Para ello en el archivo inises.component.html añadimos tras el cierre del form:

src/app/autenticacion/inises/inises.component.html

```
...
</form>
<p class="alert alert-danger" *ngIf="mensaje">
  El usuario o contraseña no es correcto
</p>
</div>
...
```

Ahora, si el inicio de sesión no es correcto, se mostrará el siguiente error:



The screenshot shows the login interface of the 'Compras App'. At the top left is the app name 'Compras App', and at the top right is a blue 'Inicie sesión' button. The main heading is 'Introduzca los siguientes datos:'. Below it are two input fields: 'Correo Electrónico' with the value 'pedro@sapienslearning.com' and 'Contraseña' with masked characters. A blue 'Inicie sesión' button is positioned below the password field. A red error message box at the bottom states 'El usuario o contraseña no es correcto'.

Pero si iniciamos sesión correctamente, accederemos a la aplicación con todas las opciones del menú disponibles y el botón de cierre de sesión:



The screenshot shows the home page of the 'Compras App' after a successful login. The top bar contains the app name 'Compras App' on the left, and a navigation menu with 'Inicio', 'Proveedores', 'Presupuestos', and a blue 'Cerrar sesión' button on the right. The main content area displays the message 'Bienvenido a nuestra aplicación de Compras App'.

11.3 Protección de rutas

Una buena técnica de seguridad para las aplicaciones es proteger las rutas url de acceso a los diferentes componentes de la aplicación.

Para ello Angular dispone de la interfaz CanActivate, que vamos a implementar a través de un nuevo servicio. Por tanto, comenzamos creando un nuevo servicio tecleando en la consola del equipo:

```
ng g service servicios/guard --spec false
```

Y comenzamos a editar el servicio, en el archivo guard.service.ts, añadiendo las importaciones del paquete router de Angular y de nuestro servicio de autenticación:

```
src/app/servicios/inises/guard.service.ts
```

```
...
import { CanActivate,
        ActivatedRouteSnapshot,

        RouterStateSnapshot } from '@angular/router';
import { AutenticacionService } from '../servicios/autenticacion.service';
...
```

Y modificamos toda la clase de la siguiente forma:

```
...
export class GuardService implements CanActivate { ①

    constructor(private autenticacionService: AutenticacionService) {} ②

    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
        return this.autenticacionService.isAuthenticated(); ③
    }
...

```


De tal manera que:

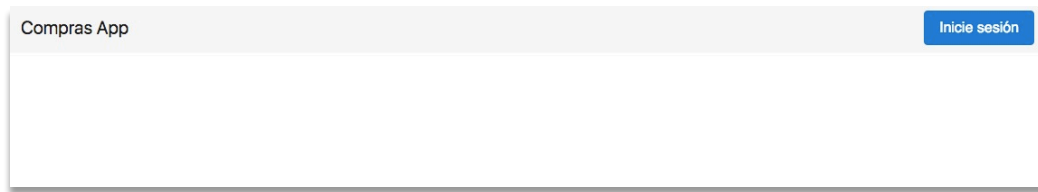
- ① Hemos implementado la interfaz CanActivate.
- ② Introducimos en el constructor nuestro servicio de autenticación.
- ③ En el método canActivate establecemos los parámetros route y state y llamamos al método isAuthenticated de nuestro servicio de autenticación para saber si está iniciada la sesión.

Seguidamente implementamos canActivate las rutas que necesitamos proteger en el array de rutas, que recordamos, tenemos definido en app.module.ts:

src/app/app.module.ts

```
...
const routes: Routes = [
  { path: '', component: InicioComponent },
  { path: 'proveedores', component: ProveedoresComponent, canActivate:
[GuardService]},
  { path: 'addprovee', component: AddproveeComponent, canActivate:
[GuardService]},
  { path: 'addpres', component: AddpresComponent, canActivate:
[GuardService]},
  { path: 'presupuestos', component: PresupuestosComponent, canActivate:
[GuardService]},
  { path: 'editpres/:id', component: EditpresComponent, canActivate:
[GuardService]},
  { path: 'registro', component: RegistroComponent },
  { path: 'iniciosesion', component: InisesComponent },
  { path: '**', component: InicioComponent}
];
...
```

Con estos pasos, nuestra aplicación devolverá una página en blanco si accedemos mediante la url a cualquiera de los componentes protegidos:



11.4 Link a registro e inicio de sesión

Para concluir esta unidad, vamos a añadir al formulario de inicio de sesión un link al formulario de registro y viceversa.

En primer lugar, añadimos en el archivo `inises.component.html` la siguiente línea de código tras el form:

```
src/app/autenticacion/inises/inises.component.html
```

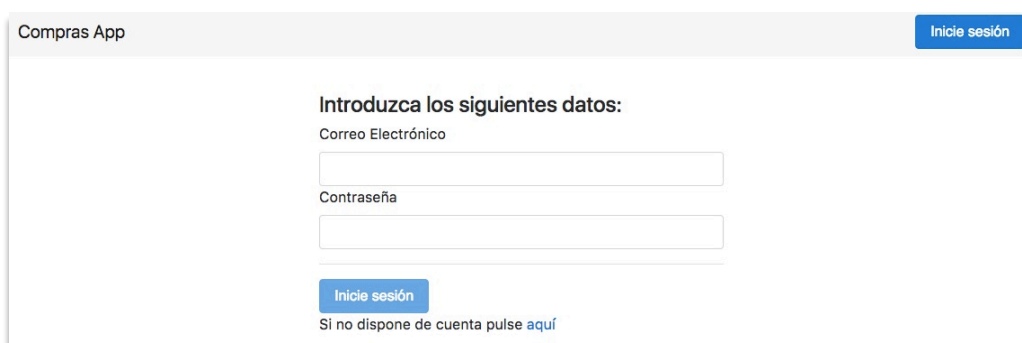
```
...  
<p>Si no dispone de cuenta pulse <a routerLink="/registro">aquí</a></p>  
</div>  
...
```

Y de la misma forma, incluimos el link a inicio de sesión en el archivo `registro.component.html`:

```
src/app/autenticacion/registro/registro.component.html
```

```
...  
<p>Si ya dispone de cuenta pulse <a routerLink="/iniciosesion">aquí</a></p>  
</div>  
...
```

Así conseguimos simultanear entre las vistas de ambos componentes, de una forma rápida y sencilla:



Compras App

Inicie sesión

Introduzca los siguientes datos:

Correo Electrónico

Contraseña

Inicie sesión

Si no dispone de cuenta pulse [aquí](#)

12 Módulos adicionales

En general, salvo que la aplicación comience a tener un número elevado de componentes, con el módulo App es suficiente para atender las necesidades de la misma.

Pero siempre es bueno aprender como disponer de otros módulos en una aplicación para poderlos implementar en el caso de que nuestra aplicación crezca de manera notable.

12.1 Creación de módulos

Vamos a ver como crear un módulo adicional en nuestra aplicación, por ejemplo para gestionar facturas de proveedores.

Para ello comenzamos tecleando en la consola del equipo:

```
ng generate module facturas
```

Con lo cual, Angular CLI crea un nuevo directorio llamado como el módulo, facturas, dentro del directorio del módulo principal, app.

El código del archivo creado, facturas.module.ts dispone de un código diferente al módulo raíz:

```
src/app/facturas/facturas.module.ts
```

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

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class FacturasModule {}
```

En el que se añade el módulo CommonModule de las librerías de Angular que permitirán que los componentes que creamos en nuestro nuevo módulo, por defecto privados, puedan ser empleados en el módulo raíz o en otros módulos.

12.2 Componentes de módulos adicionales

Vamos a continuación a crear un componente en nuestro nuevo módulo para ver como podemos implementarlo en el módulo raíz.

Para crearlo, utilizamos Angular CLI como es habitual, pero debemos asegurarnos que lo creamos en el directorio del módulo adicional, en este caso facturas. Tecleamos en la consola por tanto:

```
ng g c facturas/facturas --spec false
```

Y de esta manera generamos nuestro nuevo componente para gestionar facturas y Angular CLI, como lo hemos creado en el directorio de un módulo, lo importa y añade al mismo.

Ahora podemos fijarnos de nuevo en nuestro módulo, en el archivo facturas.module.ts, y comprobar como se ha importado el nuevo componente:

```
src/app/facturas/facturas.module.ts
```

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FacturasComponent } from './facturas/facturas.component';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [AddfrsComponent]
})
export class FacturasModule { }
```

Pues bien, si necesitamos que nuestro componente addfrs del módulo facturas este disponible en nuestro menú, que recordemos se encuentra en el componente header, necesitamos en primer lugar importar el módulo facturas al módulo raíz de la aplicación.

Para ello, editamos el archivo `app.module.ts` y comenzamos por añadir la importación del nuevo módulo:

`src/app/app.module.ts`

```
...  
import { FacturasModule } from 'app/facturas/facturas.module';  
...
```

y lo añadimos al array de imports:

```
...  
imports: [  
  BrowserModule,  
  RouterModule.forRoot(routes),  
  FormsModule,  
  ReactiveFormsModule,  
  HttpClientModule,  
  FacturasModule  
],  
...
```

Con el módulo `facturas` ya añadido al módulo raíz, continuamos de la forma ya conocida, añadiendo el componente a las rutas. De nuevo en el archivo `app.module.ts` importamos el componente `facturas`:

`src/app/app.module.ts`

```
...  
import { FacturasComponent } from 'app/facturas/facturas/facturas.component';  
...
```

Y lo añadimos al array de rutas:

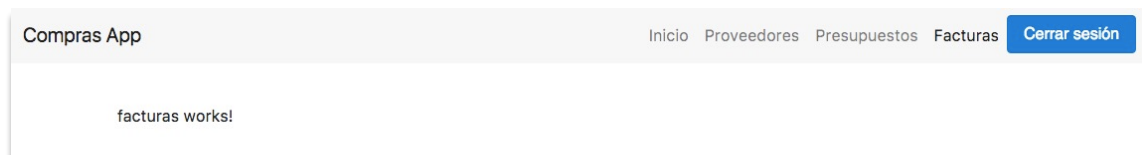
```
...  
{ path: 'facturas', component: FacturasComponent },  
...
```

Así tendremos disponible el componente también el módulo raíz. Finalmente vamos añadirlo a un link en el archivo header.component.html:

src/app/header/header.component.html

```
...  
<li class="nav-item" routerLinkActive="active" *ngIf="isAuth()">  
  <a class="nav-link" routerLink="/facturas">Facturas </a>  
</li>  
...
```

Y dispondremos del componente facturas del módulo facturas en nuestro navegador:



Otro aspecto muy importante en el desarrollo de módulos adicionales en Angular es que para poder usar las principales características de Angular que hemos aprendido, deben ser de nuevo importadas en el módulo además de que estén implementadas en el módulo raíz.

Dicho de otra forma, cada librería que añadimos en el módulo raíz según añadíamos funcionalidad a cada componente, debe ser también añadido en un nuevo módulo si la vamos a usar en los componentes.

Como emplearemos routing, formularios y peticiones Http en los componentes que creamos en el módulo facturas, vamos a añadirlos al módulo en el archivo facturas.module.ts comenzando por las importaciones:

```
src/app/facturas/facturas.module.ts
```

```
...  
import { Routes, RouterModule } from '@angular/router';  
import { ReactiveFormsModule } from '@angular/forms';  
import { HttpClientModule } from '@angular/http';  
...
```

y los agregamos a los imports de la clase:

```
...  
imports: [  
  CommonModule,  
  RouterModule.forRoot(),  
  ReactiveFormsModule,  
  HttpClientModule,  
],  
...
```

Quedando así nuestro módulo operativo para crear más componentes, servicios, etc.

12.3 Componentes CRUD para el módulo adicional.

Podemos añadir a nuestro módulo adicional los componentes para crear, leer, editar y borrar registros de facturas en nuestra aplicación.

Los pasos a realizar son los mismos que los empleados para los componentes encargados de gestionar presupuestos, así que es un buen ejercicio para, por nuestra cuenta, practicar lo aprendido.

No obstante, podemos descargar de los archivos del libro, el código totalmente finalizado de los tres componentes, facturas, addfrs y editfrs y del servicio facturas, necesarios para llevar a cabo estas funcionalidades.

13 Angular *Tips & Tricks*

Hemos aprendido hasta ahora decenas de funcionalidades de Angular y las hemos puesto en práctica en nuestra aplicación.

Llega el momento pues, de mejorar nuestra aplicación a través de una serie de consejos y técnicas antes de desplegar la misma a producción.

13.1 Select con objetos de la base de datos.

Las aplicaciones complejas siempre necesitan cruzar datos entre sus diversas colecciones de objetos de las bases de datos para facilitar las operaciones que se realizan en ellas.

Vamos a ver, mediante un ejemplo en nuestra aplicación, como seleccionar en el componente añadir presupuestos, un objeto proveedor proveniente del componente proveedores.

En los anteriores apartados, hemos comprobado como los servicios en Angular permiten recibir o enviar datos a los servidores de bases datos y mediante la inyección de dependencias, inyectarlos en los componentes.

Precisamente vamos a emplear los servicios como centralización de la gestión de datos para cruzar estos entre componentes.

Nota: Antes de continuar con el ejemplo, rescribe los componentes relacionados con los proveedores de la misma forma que para presupuestos y facturas. Puedes chequear el código o copiarlo de los archivos del libro de este apartado.

De esta manera y en nuestro ejemplo, en primer lugar editamos el componente para añadir presupuestos, en el archivo `addpres.component.ts` para importar el servicio de proveedores:

```
src/app/presupuestos/addpres/addpres.component.ts
```

```
...  
import { ProveedoresService } from '../servicios/proveedores.service';  
...
```

Ahora declaramos dentro del componente un nuevo objeto proveedores de tipo array que almacenará los proveedores que recuperemos del servicio:

```
...  
  proveedores: any[] = [];  
...
```

Modificamos el constructor para introducir el servicio y llamar al método getProveedores:

```
...  
  constructor(private pf: FormBuilder,  
    private presupuestoService: PresupuestosService,  
    private router: Router,  
    private activatedRouter: ActivatedRoute,  
    private proveedoresService: ProveedoresService)  
  {  
    this.proveedoresService.getProveedores()  
      .subscribe(proveedores => {  
        for (const id$ in proveedores) {  
          const p = proveedores[id$];  
          p.id$ = id$;  
          this.proveedores.push(proveedores[id$]);  
        }  
      })  
  }  
...  
}
```

De esta manera ya tenemos los proveedores en nuestro componente, así que vamos a modificar la vista para sustituir el campo proveedores por un select donde se recorra el objeto proveedores.

En el archivo addpres.component.html sustituimos:

```
src/app/presupuestos/addpress/addpres.component.html
```

```
...  
  <div class="form-group">
```

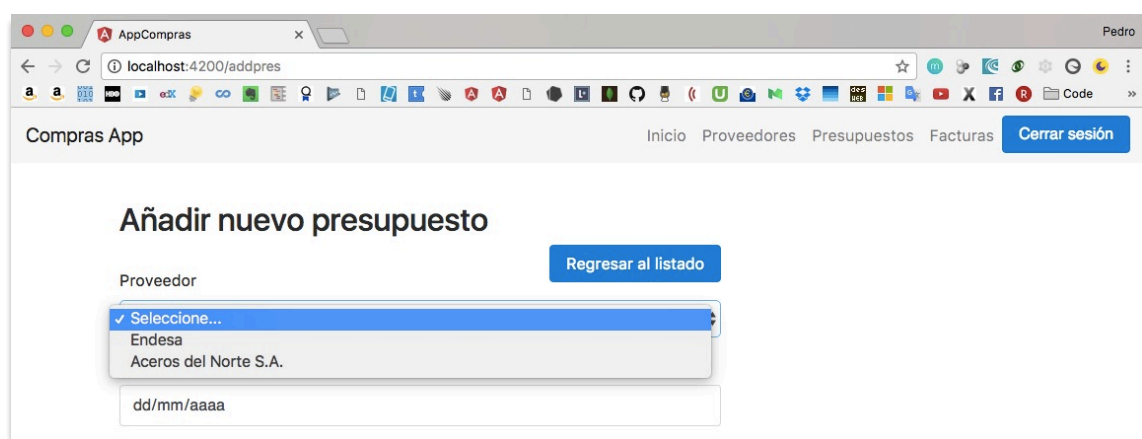
```

<label for="tipo">Proveedor</label>

<i class="fa fa-check-circle check"
  *ngIf="presupuestoForm.controls.tipo.valid"></i>
<i class="fa fa-exclamation-circle uncheck"
  *ngIf="presupuestoForm.controls.proveedor.invalid &&
  presupuestoForm.controls.proveedor.touched"></i>
<select class="form-control"
  id="proveedor"
  formControlName="proveedor">
  <option value="">Selecione...</option>
  <option *ngFor="let proveedor of proveedores" value="{{
proveedor.nombre }}">{{ proveedor.nombre }}</option>
</select>
<p class="alert alert-danger"
  *ngIf="presupuestoForm.controls.proveedor.invalid &&
  presupuestoForm.controls.proveedor.touched">
  El campo Proveedor es obligatorio.
</p>
</div>

```

Y así, gracias a introducir dentro de la etiqueta option de un select la directiva ngFor para iterar el objeto proveedores, conseguimos un select con los proveedores de nuestra aplicación.



Para practicar lo aprendido, podemos realizar la misma operación en el componente de edición de presupuestos e igualmente en el módulo de facturas, chequeando el resultado con el código final de este apartado.

13.2 Spinner en carga de elementos HTML

La carga de elementos HTML en nuestra aplicación puede demorarse debido a las conexiones con la base de datos, por ello es conveniente incorporar a estos elementos, un *spinner* para mejorar la experiencia de usuario.

Vamos a realizar un ejemplo en nuestra tabla de proveedores, así que comenzamos por editar el componente `proveedores.component.ts` para añadir en su clase una propiedad booleana llamada `cargando` que se inicializa como `true`:

```
src/app/proveedores/proveedores/proveedores.component.ts
```

```
...  
  cargando = true;  
...
```

A continuación, dentro del constructor introducimos una línea de código que cambie esta propiedad a `false`, cuando haya finalizado el método `getProveedores`, es decir cuando se hayan cargado al objeto desde el servicio:

```
...  
constructor(private proveedoresService: ProveedoresService) {  
  this.proveedoresService.getProveedores()  
    .subscribe(proveedores => {  
    for (const id$ in proveedores) {  
      const p = proveedores[id$];  
      p.id$ = id$;  
      this.proveedores.push(proveedores[id$]);  
    }  
    this.cargando = false;  
  })  
}  
...  
}
```

Ahora, llega el momento de utilizar esta propiedad en la vista, para ello vamos a crear dos div en los que implementaremos la directiva ngIf condicionada a la propiedad cargando.

Por tanto, editamos el archivo proveedores.component.ts para añadir:

```
src/app/proveedores/proveedores/proveedores.component.html
```

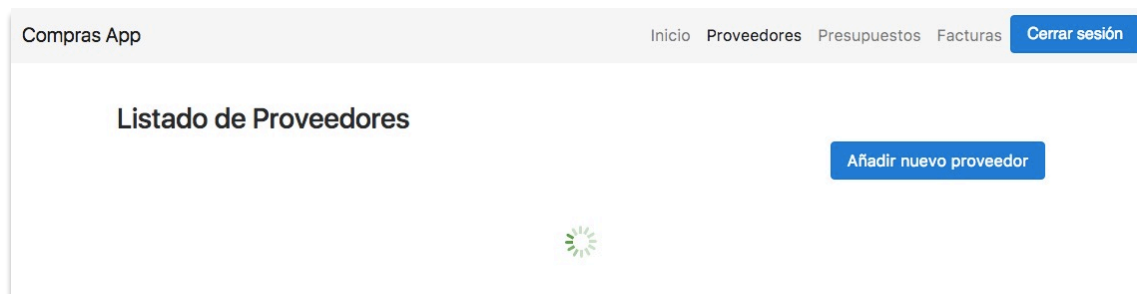
```
...  
<div *ngIf="cargando" class="text-center" style="padding-top: 60px;"> ①  
    
</div>  
<div *ngIf="!cargando"> ②  
  //tabla existente  
</div>  
...
```

En el cual:

① Añadimos un div con ngIf asociado a cargando para que muestre una imagen gif con un spinner mientras cargando sea true.

② Envolvemos la tabla existente con un div que en este caso, también tiene ngIf pero asociado a cargando cuando esté false.

De esta manera, cuando nuestro componente este cargando la tabla se mostrará un spinner:



Hemos simplificado el ejemplo con un spinner incluido en la carpeta de imágenes assets del proyecto, pero, por supuesto se puede añadir cualquier tipo de spinner como los numerosos ejemplos en lenguaje HTML5 y CSS que existen en internet.

Para practicar esta sencilla mejora de nuestra aplicación, es bueno modificar de la misma manera la tabla del componente presupuestos y la del componente facturas y compararlos con el código final de este apartado.

13.3 Elementos de respuesta de autenticación

En un ejemplo similar al del anterior apartado, podemos modificar otros elementos cuando estén consultando al servidor de bases de datos.

Por ejemplo en el formulario de inicio de sesión, es bueno que el botón muestre que se está autenticando al usuario.

Para ello, en el archivo `inises.component.ts` vamos a añadir una propiedad a la clase de nombre `autenticando`, que se inicialice como `false`:

```
src/app/autenticacion/inises/inises.component.ts
```

```
...  
  autenticando = false;  
...
```

Y modificamos el método `onSubmit` para que al pulsar el botón, `autenticando` pase a `true` y tras el retardo del servidor pase de nuevo a `false`:

```
...  
onSubmit() {  
  this.autenticando = true;  
  this.userdata = this.saveUserdata();  
  this.autService.inicioSesion(this.userdata);  
  setTimeout(() => {  
    if (this.isAuth() === false) {  
      this.mensaje = true;  
      this.autenticando = false;  
    }  
  }, 2000);  
}  
...
```

A continuación modificamos el botón del formulario en el archivo inises.component.html:

src/app/autenticacion/inises/inises.component.html

```
...
<button type="submit"
  class="btn btn-primary"
  [disabled]="!loginForm.valid"
  *ngIf="!autenticando"
  >Inicie sesión</button>
<button type="submit"
  class="btn btn-primary"
  *ngIf="autenticando"
  >Autenticando usuario...</button>
...
```

De tal manera que ahora tenemos dos botones, el ya programado que se mostrará cuando autenticando sea false, y un nuevo botón que se mostrará cuando autenticando sea true:

Introduzca los siguientes datos:

Correo Electrónico

pedro@sapienslearning.com

Contraseña

.....

Autenticando usuario...

Si no dispone de cuenta pulse [aquí](#)

13.4 Consultas a la base de datos y buscador

Una manera habitual y eficiente de recuperar datos de un servidor es enviar desde la aplicación una consulta a través del método de conexión, por ejemplo en las peticiones http.

De esta manera, el servidor solo devolverá los datos que necesite el usuario en ese momento, el rendimiento de la aplicación será mayor y las necesidades de la arquitectura de la aplicación se reducen.

En el caso de nuestra aplicación, estamos realizando peticiones a la base de datos para que devuelvan todos los registros de una colección o nodo, con lo cual en el momento en que aumenten notablemente estos, la aplicación tardará mucho tiempo en cargar los listados.

Por tanto, vamos a realizar en el siguiente ejemplo dos mejoras, por una parte las peticiones a la base de datos las realizaremos mediante consultas o *queries* y para determinar esas consultas vamos a implementar un buscador dinámico.

En primer lugar, vamos a añadir un buscador a nuestro componente de listado de proveedores, así que en el archivo `proveedores.component.html` añadimos:

```
src/app/proveedores/proveedores/proveedores.component.html
```

```
...  
  <input type="search"  
    class="form-control"  
    placeholder="Teclee un Proveedor..."  
    [formControl]=" campoBusqueda">  
...
```

El buscador es un simple input al que hemos añadido la directiva `formControl` para poder emplear la funcionalidad reactiva.

A continuación en el componente, comenzamos importando la librería forms de Angular:

```
src/app/proveedores/proveedores/proveedores.component.ts
```

```
...  
import { FormControl, FormGroup, FormBuilder, Validators } from  
'@angular/forms';  
...
```

A continuación añadimos las propiedades en la clase para el formulario y una propiedad búsqueda que almacenará las palabras que introduzcamos en el buscador:

```
...  
campoBusqueda: FormControl;  
busqueda: string;  
...
```

También añadimos otra serie de propiedades booleanas que establecerán la lógica para mostrar los resultados o no, del buscador, similares al comportamiento de la propiedad cargando:

```
...  
cargando = false;  
resultados = false;  
noresultados = false;  
...
```

A continuación eliminamos del constructor el método getProveedores, que recordemos recupera todos los registros de proveedores de la base de datos.

De esta manera, vamos a recuperar solo los datos con coincidencias con nuestro buscador para lo cual, dentro de ngOnInit añadimos el siguiente código:

```
...
ngOnInit() {
  this.campoBusqueda = new FormControl(); ①
  this.campoBusqueda.valueChanges
    .subscribe(term => {
      this.busqueda = term;
      this.cargando = true;
      if (this.busqueda.length !== 0) { ②
        this.proveedoresService.getProveedoresSearch(this.busqueda)
          .subscribe(proveedores => {
            this.proveedores = [];
            for (const id$ in proveedores) {
              const p = proveedores[id$];
              p.id$ = id$;
              this.proveedores.push(proveedores[id$]);
            }
            if (this.proveedores.length < 1 && this.busqueda.length >= 1) {
              this.noresultados = true;
            } else {
              this.noresultados = false;
            }
          })
        this.cargando = false;
        this.resultados = true;
      } else {
        this.proveedores = []; ③
        this.cargando = false;
        this.resultados = false;
      }
    });
}
```

En el cual:

① Inicializamos el formulario y nos suscribimos con ValueChanges a los cambios que se produzcan en tiempo real.

② Establecemos un if-else para que si la longitud del campo búsqueda es distinto de cero nos suscribamos a nuevo método del servicio al que enviamos el valor de búsqueda y el res que recibamos lo iteramos con un for para almacenarlo en el array proveedores

③ Si la longitud del campo búsqueda es cero, entonces vaciamos el array proveedores.

Además del código detallado, en cada paso cambiaremos el estado de las propiedades booleanas para emplearlas en la plantilla mediante directivas ngIf.

Por tanto, como llamamos a un nuevo método del servicio, vamos a crearlo en este. Para ello añadimos al archivo proveedores.service.ts el siguiente código:

src/app/servicios/proveedores.service.ts

```
...
getProveedoresSearch(busqueda: string) { ①
  const url = `${ this.prosURL }?orderBy="nombre"&startAt="${ busqueda
}"&endAt="${ busqueda }\uf8ff"`; ②
  return this.http.get( url )
    .map( res => res.json()); ③
}
...
```

En el cual:

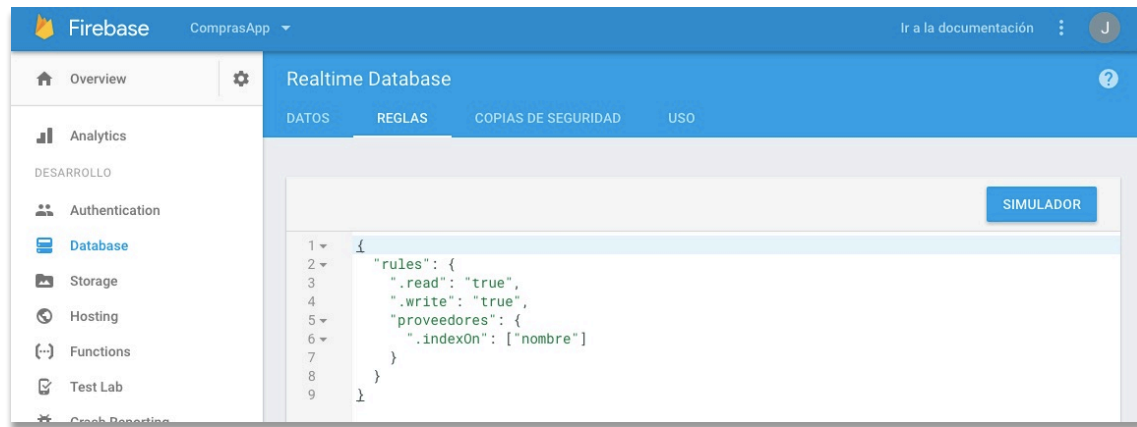
① Recibimos el parámetro búsqueda con el valor introducido en el buscador del componente.

② Establecemos una url con el valor de nombre, el campo por el que vamos a realizar la consulta, y los valores de búsqueda que recibiremos a través del parámetro. Esta url es la que define Firebase en su documentación como consulta para recuperar registros que en un campo comiencen por el valor definido.

③ Recuperamos los registros con map en res y le aplicamos el método .json() para enviarlos al componente.

También la documentación de Firebase nos indica que para emplear en las consultas los valores de un determinado campo como filtro, debemos de establecer un índice en este campo.

Para ello en la consola, modificamos las reglas para añadir:



Cuyo código es:

```
...  
  "proveedores": {  
    ".indexOn": ["nombre"]  
  }  
...
```

Que indica que en el nodo proveedores, se cree un índice sobre el campo nombre.

Con estos pasos nuestro buscador está listo, pero queda añadir los ngIf en la plantilla para emplear las propiedades booleanas.

De nuevo en el archivo proveedores.component.html tendremos finalmente el siguiente código:

```
src/app/proveedores/proveedores/proveedores.component.html
```

```
<h3>Listado de Proveedores</h3>  
<a class="btn btn-primary float-md-right" routerLink="/addprovee">Añadir  
nuevo proveedor</a>  
<br>  
<br>
```

```

<input type="search"
      class="form-control"
      placeholder="Teclee un Proveedor..."
      [formControl]="campoBusqueda">
<div *ngIf="cargando" class="text-center" style="padding-top: 60px;">
  
</div>
<div *ngIf="resultados"> ①
<table class="table table-bordered table-striped tabla" style="margin-top:
40px;">
  <thead>
    <tr class="filters">
      <th>Nombre</th>
      <th>Correo Electrónico</th>
      <th>Teléfono</th>
      <th>Editar</th>
      <th>Eliminar</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let proveedor of proveedores">
      <td>{{ proveedor.nombre }}</td>
      <td><a href="mailto:{{ proveedor.email }}">{{ proveedor.email }}</a></td>
      <td>{{ proveedor.telefono }}</td>
      <td>
        <button type="button" class="btn btn-success"
          routerLink="/editprovee/{{ proveedor.id$ }}">Editar</button>
      </td>
      <td>
        <button class="btn btn-danger"
          (click)="eliminarProveedor(proveedor.id$)">Eliminar</button>
      </td>
    </tr>
  </tbody>
</table>
</div>
<div *ngIf="noresultados"> ②
  <p>No existe ninguna coincidencia</p>
</div>

```


En el cual:

- ① Mostraremos la tabla con los proveedores solo cuando resultados sea true, es decir cuando se halla iniciado el método `getProveedoresSearch` y haya devuelto resultados.
- ② Cuando `noresultados` sea true, es decir se haya ejecutado el método `getProveedoresSearch` pero el array `proveedores` tenga cero elementos, se mostrará un párrafo indicando que no hay resultados.

A partir de este momento, como el buscador tiene implementado la funcionalidad de observable en sus valores, el componente filtrará los resultados en tiempo real, a partir de la consulta en la base de datos con el valor que tecleemos:

Compras App

[Inicio](#) [Proveedores](#) [Presupuestos](#) [Facturas](#) [Cerrar sesión](#)

Listado de Proveedores

Añadir nuevo proveedor

Te|

Nombre	Correo Electrónico	Teléfono	Editar	Eliminar
Tech Consulting Inc	info@tech.com	9199999999	<button>Editar</button>	<button>Eliminar</button>
Telefónica S.L.U.	info@telefonica.net	918701701	<button>Editar</button>	<button>Eliminar</button>

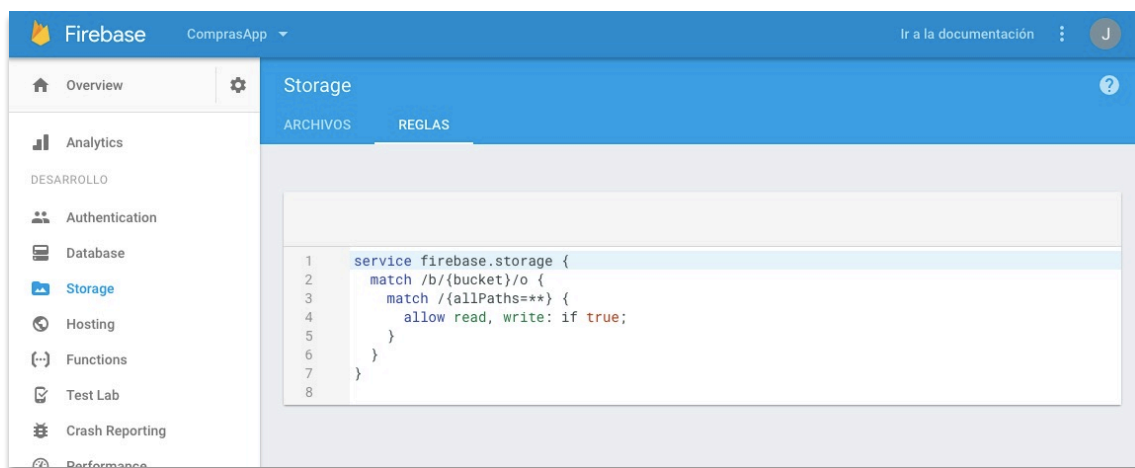
13.5 Upload de archivos

Una funcionalidad muy apreciada en las aplicaciones es la posibilidad de subir archivos a la misma y almacenarlos en una base de datos.

Vamos a ver en este apartado como subir ficheros a la aplicación y almacenarlos en el servicio de almacenamiento de Firebase.

13.5.1 Configuración de Firebase

Previamente, en nuestro proyecto en Firebase, accedemos a la consola y vamos a habilitar el almacenamiento de archivos. Pulsamos en Storage en el menú lateral y en la pestaña reglas:



Modificamos el código para registrar el siguiente:

```
service firebase.storage {  
  match /b/{bucket}/o {  
    match /{allPaths=**} {  
      allow read, write: if true;  
    }  
  }  
}
```

A continuación vamos a emplear todos los parámetros de Firebase en nuestra aplicación.

En primer lugar creamos directorio denominado config en el directorio app y en este añadimos el siguiente archivo:

src/app/config/firebase.config.ts

```
...
export const environment = {
  production: false,
  firebase: {
    apiKey: "AlzaSyCA0vJPBzNI_uOeKWkk5skmElqkYDagFAY",
    authDomain: "comprasapp-83618.firebaseio.com",
    databaseURL: "https://comprasapp-83618.firebaseio.com",
    projectId: "comprasapp-83618",
    storageBucket: "comprasapp-83618.appspot.com",
    messagingSenderId: "1004906173037"
  }
}
...
```

Recuerda que en el apartado 11.1 vimos como obtener los datos de nuestro proyecto Firebase.

Ahora vamos a añadir estos parámetros a nuestro módulo, para lo cual en el archivo app.module.ts añadimos en primer lugar las importaciones:

src/app/app.module.ts

```
...
import { AngularFireModule } from 'angularfire2';
import { AngularFireDatabaseModule } from 'angularfire2/database';
import { AngularFireAuthModule } from 'angularfire2/auth';
import { environment } from '../config/firebase.config';
...
```

Y añadimos en los import:

```
...
imports: [
  BrowserModule,
  RouterModule.forRoot(routes),
  FormsModule,
  ReactiveFormsModule,
  HttpClientModule,
  FacturasModule,
  AngularFireModule.initializeApp(environment.firebase),
  AngularFireDatabaseModule,
  AngularFireAuthModule,
],
...
```

Con esto tenemos nuestra aplicación lista para comenzar a escribir los componentes para subir archivos.

13.5.2 Componente Upload

Comenzamos creando un componente upload para realizar la carga de archivo. En la consola de nuestro equipo, tecleamos:

```
ng g c upload --spec false
```

A continuación, dentro de la carpeta upload vamos a crear una clase modelo para nuestros archivos que emplearemos tanto en el componente, como en el servicio de subida de archivos.

Creamos el archivo .model.ts:

```
src/app/upload/file.modal.ts
```

```
export class Archivo {
  $key: string;
  file: File;
  name: string;
  url: string;
  progress: number;
```

```

    createdAt: Date = new Date();
    constructor(file: File) {
        this.file = file;
    }
}

```

Ya tenemos nuestra clase Archivo, así que a continuamos creando un servicio que será el que enlace a Firebase para enviar y recibir los archivos.

De nuevo en la consola del equipo tecleamos:

```
ng g service servicios/loadfile --spec false
```

Y vamos a editar el servicio, comenzando por añadir las importaciones de los paquetes Angular para Firebase y la clase Archivo:

```
src/app/servicios/loadfile.service.ts
```

```

...
import { AngularFireDatabase, FirebaseListObservable,
        FirebaseObjectObservable } from 'angularfire2/database';
import * as firebase from 'firebase';
import { Archivo } from '../upload/file.model';
...

```

Ahora vamos a definir dos propiedades, basePath que define la carpeta en Firebase donde se almacenarán los archivos, y uploads, un array de la clase Archivos de tipo FirebaseListObservable:

```

...
private basePath:string = '/uploads';
uploads: FirebaseListObservable<Archivo[]>;
...

```

En el constructor, declaramos `angularFireDatabase` un objeto que permite la comunicación en tiempo real con la base de datos de una manera específica de Firebase, y que por tanto no emplea las peticiones `http` como empleamos en otros componentes:

```
...  
  constructor( public angularFireDatabase: AngularFireDatabase) {}  
...
```

Finalmente establecemos los métodos `pushUpload` y `pushSaveFileData` que almacenarán los archivos en el servicio `Storage` de Firebase y los metadatos de ese archivo en un nodo:

```
...  
  pushUpload(upload: Archivo) {  
    const storageRef = firebase.storage().ref();  
    const uploadTask =  
storageRef.child(`${this.basePath}/${upload.file.name}`).put(upload.file);  
  
    uploadTask.on(firebase.storage.TaskEvent.STATE_CHANGED,  
      (snapshot) => {  
        // upload in progress  
        upload.progress = (snapshot.bytesTransferred / snapshot.totalBytes) * 100  
      },  
      (error) => {  
        // upload failed  
        console.log(error)  
      },  
      () => {  
        // upload success  
        upload.url = uploadTask.snapshot.downloadURL  
        upload.name = upload.file.name  
        this.saveFileData(upload)  
      }  
    );  
  }  
}
```

```
private saveFileData(upload: Archivo) {  
    this.angularFireDatabase.list(`${this.basePath}/`).push(upload);  
}  
...
```

Recordemos que para que el servicio este disponible, debemos importarlo y añadirlo al array provider del archivo app.module.ts de la forma que ya conocemos.

El siguiente paso es editar el componente upload. Comenzamos añadiendo las importaciones en upload.component.ts:

src/app/upload/upload.component.ts

```
...  
import { Archivo } from './file.model';  
import { LoadfileService } from '../servicios/loadfile.service';  
import * as _ from 'lodash';  
...
```

A continuación dentro de la clase creamos tres propiedades, selectedFiles de clase FileList para seleccionar los archivos que subamos, currentUpload de la clase Archivo y un booleano denominado loading inicializado false:

```
...  
selectedFiles: FileList;  
currentUpload: Archivo;  
  
loading = false;  
...
```

Seguidamente, llamamos al servicio en el constructor y creamos el métodos uploadSingle:

```
...
constructor (public loadfileService: LoadfileService) {}

uploadSingle() {
  const file = this.selectedFiles.item(0);
  this.currentUpload = new Archivo(file);
  this.loading = true;
  this.loadfileService.pushUpload(this.currentUpload);
}
...
```

Ahora, vamos a añadir en la plantilla del componente, archivo upload.component.ts un div para subir los archivos, un botón para subir, otro para cancelar y la barra de progreso:

src/app/upload/upload.component.ts

```
<div class="row">
  <h3>Contratos</h3>
</div>
<hr>
<div class="row">
  <div class="col-md-2">
  </div>
  <div class="col-md-8 col-md-offset-2">
    <div class="uploader">
      <p style="font-size: 40px">
        <i class="fa fa-file-pdf-o"></i>
        <i class="fa fa-file-word-o" style="padding-left: 20px;"></i>
        <i class="fa fa-file-excel-o" style="padding-left: 20px;"></i>
        <i class="fa fa-file-picture-o" style="padding-left: 20px;"></i>
      </p>
      <input type="file" (change)="detectFiles($event)">
    </div>
    <div class="text-center">
      <button type="button"
        class="btn btn-success">
```



```

        [disabled]="!selectedFiles"
        (click)="uploadSingle()">
        Subir archivo</button>

        <a class="btn btn-primary"
        routerLink="/">
        Cancelar</a>
    </div>
    <br>
    <div *ngIf="loading">
        <div class="progress">
            <div class="progress-bar progress-bar-animated" [ngStyle]="{
'width': currentUpload?.progress + '%' }"></div>
        </div>
        Subiendo {{currentUpload?.name}} | {{currentUpload?.progress |
number:'1.2-2' }}% Completado
    </div>
</div>

```

Para finalizar añadimos al archivo upload.component.css los estilos CSS de los anteriores elementos:

src/app/upload/upload.component.ts

```

.uploader {
    background-color: #ea6112;
    color: white;
    width: 100 %;
    height: 160px;
    margin: 40px;
    padding-top: 40px;
    text-align: center;
}

```

De esta manera nuestro componente ya está listo así que vamos a añadir su ruta al archivo app.module.ts:

src/app/app.module.ts

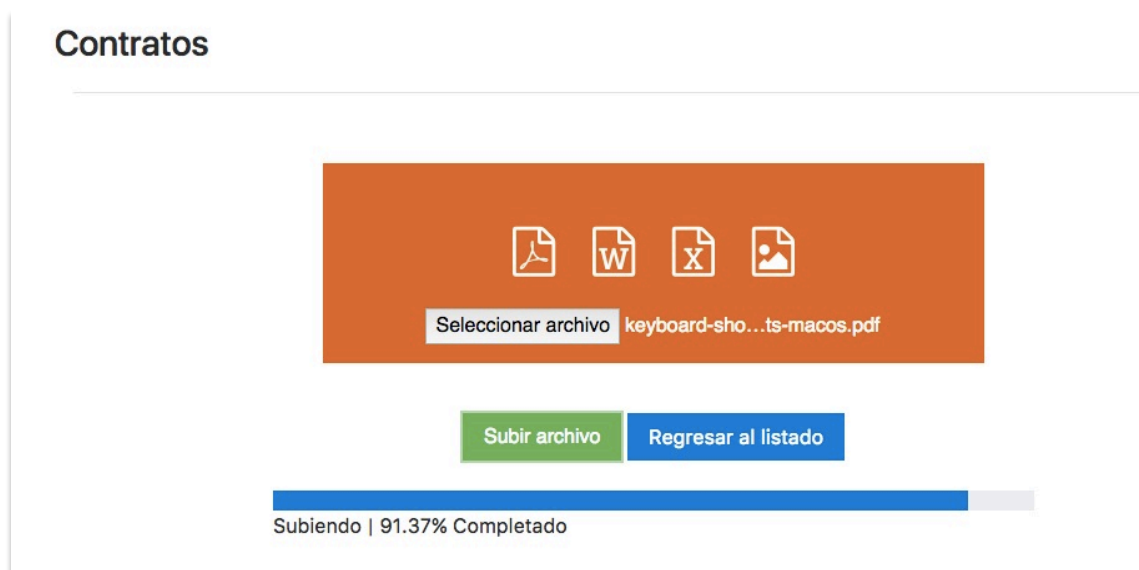
```
...  
{ path: 'uploads', component: UploadComponent },  
...
```

Y un link en el componente header:

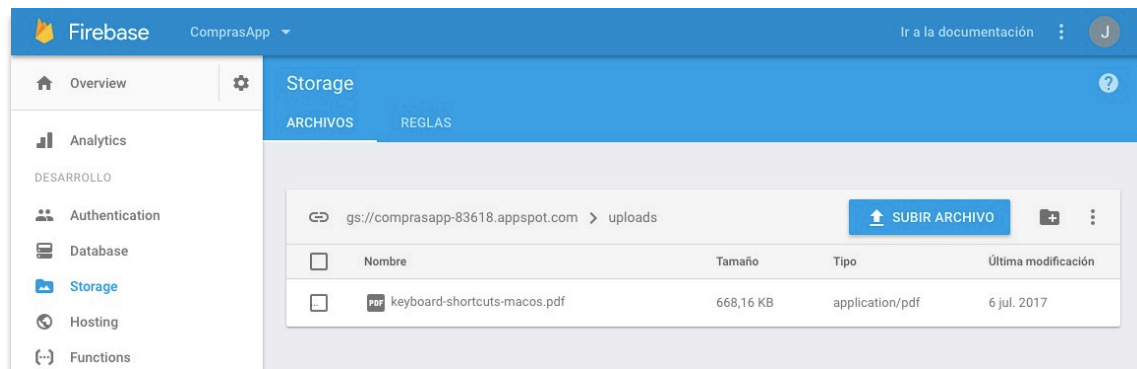
src/app/header/header.component.html

```
...  
<li class="nav-item" routerLinkActive="active" *ngIf="isAuth()">  
  <a class="nav-link" routerLink="/uploads">Uploads </a>  
</li>  
...
```

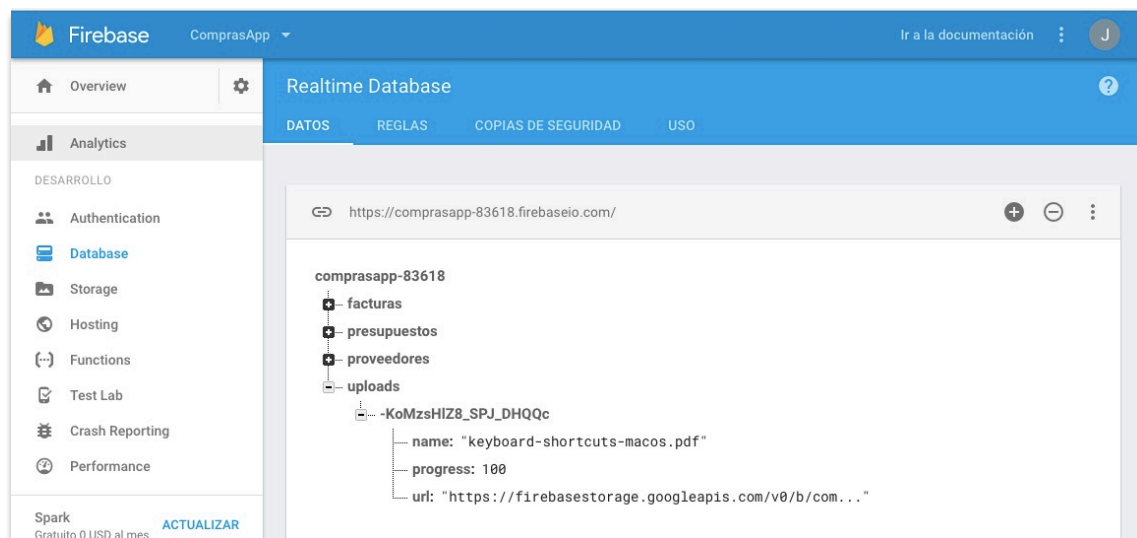
Ahora en el navegador comprobamos como podemos subir cualquier archivo:



Que quedará almacenado en Firebase en la opción Storage, dentro del directorio uploads:



Y sus metadatos en la base de datos:



13.5.3 Componente para Archivos

Lógicamente vamos a necesitar un componente para mostrar los enlaces a nuestros archivos y que así los usuarios de la aplicación puedan visualizarlos o descargarlos.

Así que, de nuevo, nos vamos a la consola del equipo y completamos:

```
ng g c contratos --spec false
```

Ya tenemos nuestro nuevo componente así que en primer lugar, vamos a editar el archivo `contratos.component.ts` y comenzamos a añadir las importaciones:

```
src/app/contratos/contratos.component.ts
```

```
...
import { FirebaseListObservable } from 'angularfire2/database';
import { LoadfileService } from '../servicios/loadfile.service';
import { Archivo } from '../upload/file.model';
...
```

Dentro de la clase creamos el array:

```
...
uploads: FirebaseListObservable<Archivo[]>;
...
```

Y llamamos al servicio en el constructor para a continuación igualar, dentro de `ngOnInit()`, el array al método del servicio `getUploads`.

```
...
constructor(private loadfileService: LoadfileService) { }

ngOnInit() {
  this.uploads = this.loadfileService.getUploads();
}
...
```

Hemos llamado a un método del servicio para que nos devuelva los datos de los archivos que tenemos alojados en Firebase, así que tenemos que editar el mismo para añadir este nuevo método.

Por tanto, en el archivo `loadfile.service.ts` añadimos en la clase un método que emplea la clase `list` para recuperar los objetos de archivo y devolverlos en `uploads`, array que ya creamos en el paso anterior:

`src/app/contratos/contratos.component.ts`

```
...
getUploads() {
  this.uploads = this.angularFireDatabase.list(this.basePath);
  return this.uploads
}
...
```

Como ya tenemos la lógica del componente resuelta, vamos a editar la plantilla, añadiendo el siguiente código al archivo `contratos.component.html`:

`src/app/contratos/contratos.component.html`

```
<h3>Listado de Contratos</h3>
<a class="btn btn-primary float-md-right"
  routerLink="/uploads">Subir nuevo contrato</a> ①
<br>
<div *ngFor="let upload of uploads | async"> ②
  <app-detalles [upload]='upload'></app-detalles>
</div>
```

En el cual:

- ① Añadimos un botón para acceder al componente para subir nuevos archivos.
- ② Creamos un `div` con `ngFor` pero en este caso, con el *pipe* `async`, para iterar el array `uploads`. Para poder posteriormente utilizar el objeto para borrarlo, introducimos un componente hijo que crearemos posteriormente con el nombre `detalles`.

Antes de crear el componente detalles en el que cargaremos cada objeto upload, vamos a finalizar este paso, añadiendo la ruta del nuevo componente, al archivo app.module.ts:

```
src/app/app.module.ts
```

```
...  
  { path: 'contratos', component: ContratosComponent },  
...
```

Y modificando en el archivo header.component.html el link de subidas por este nuevo componente:

```
src/app/header/header.component.html
```

```
...  
  <li class="nav-item" routerLinkActive="active" *ngIf="isAuth()">  
    <a class="nav-link" routerLink="/contratos">Contratos </a>  
  </li>  
...
```

Ahora ya si, vamos a crear el componente hijo denominado detalles, tecleando como siempre en la consola del equipo:

```
ng g c contratos/detalles --spec false
```

Y procedemos a editar el archivo detalles.component.ts para en primer lugar añadir:

```
src/app/contratos/detalles/detalles.component.ts
```

```
import { Component, OnInit, Input } from '@angular/core';  
import { Archivo } from '../upload/file.model';  
...
```

Y en la clase añadimos @Input con el objeto upload de la clase Archivo que nos vendrá del componente padre.

```
...  
@Input() upload: Archivo;  
...
```

A continuación vamos a añadir el código de la vista en el archivo detalles.component.html para añadir una tarjeta bootstrap:

src/app/contratos/detalles/detalles.component.html

```
<div class="card" style="width: 20rem;">  
  <div class="card-block">  
    <h4 class="card-title">Contrato</h4>  
    <p class="card-text">{{ upload.name }}</p>  
    <a href="{{upload.url}}"  
      class="btn btn-primary float-md-left">Visualizar</a>  
    <button class="btn btn-danger float-md-right">Eliminar</button>  
  </div>  
</div>
```

En la cual añadimos el nombre del archivo, un botón la url del archivo para visualizarlo o descargarlo y un botón para eliminarlo que programaremos en el siguiente apartado.

Ya podemos comprobar en el navegador, como se carga el archivo que subimos a Firebase y podemos visualizarlo o descargarlo:



13.5.4 Borrado de Archivos

Para finalizar esta funcionalidad solo nos queda poder borrar los archivos, con lo cual comenzamos editando el servicio, en el archivo `loadfiles.service.ts` en el cual añadimos:

`src/app/servicios/loadifile.service.ts`

```
...
deleteUpload(upload: Archivo) {
  this.deleteFileData(upload.$key)
  .then( () => {
    this.deleteFileStorage(upload.name)
  })
  .catch(error => console.log(error))
}

private deleteFileData(key: string) {
  return this.angularFireDatabase.list(`${this.basePath}/`).remove(key);
}

private deleteFileStorage(name:string) {
  const storageRef = firebase.storage().ref();
  storageRef.child(`${this.basePath}/${name}`).delete()
}
...
```


Hemos añadido 3res métodos, deleteUpload que recibirá del componente el objeto upload a eliminar, deleteFileData que eliminará por su identificador el objeto de la base de datos Firebase, y deleteFileStorage que eliminará por su propiedad name el archivo del servicio de almacenamiento de Firebase.

Una vez modificado el servicio, vamos a llamar al método deleteUpload desde el componente, para lo cual editamos detalles.component.ts para añadir:

```
src/app/contratos/detalles/detalles.component.ts
```

```
...  
import { LoadfileService } from '../servicios/loadfile.service';  
...
```

Y en la clase:

```
...  
constructor(private loadfileService: LoadfileService) {}  
...  
deleteUpload(upload) {  
    this.loadfileService.deleteUpload(this.upload);  
}  
...
```

Ya simplemente, tenemos que editar la vista en detalles.component.html para añadir el método al botón de eliminar:

```
src/app/contratos/detalles/detalles.component.html
```

```
...  
<button class="btn btn-danger float-md-right"  
    (click)='deleteUpload(upload)'>Eliminar</button>  
...
```

Y nuestra aplicación tendrá totalmente operativo la eliminación de los archivos subidos a Firebase y sus referencias.

13.6 Menú en pantalla de inicio

Para concluir nuestras mejoras a la aplicación vamos a añadir un menú visual a nuestra pantalla de inicio, que se mostrará o no en función de si el usuario ha iniciado sesión.

Editamos el componente de inicio en el archivo inicio.component.ts para añadir la llamada al servicio de autenticación y el mismo método para saber si esta iniciada la sesión que en el componente header:

src/app/inicio/inicio.component.ts

```
...
import { AutenticacionService } from '../servicios/autenticacion.service';
...
...
constructor(private authService: AutenticacionService) {}
...
isAuth() {
  return this.authService.isAuthenticated();
}
...
```

Y en la plantilla, inicio.component.html añadimos un mensaje de inicio de sesión condicionado a que no isAuth sea false y un menú visual con cuatro botones a cada funcionalidad en caso contrario:

src/app/inicio/inicio.component.html

```
<div class="text-center" style="margin: 40px;">
  <h3>Bienvenido a nuestra aplicación de Compras App</h3>
</div>
<div class="text-center" style="margin: 20px;" *ngIf="!isAuth()">
  <h4>Inicie sesión para acceder</h4>
</div>
<div class="row" *ngIf="isAuth()">
  <div class="col-md-3">
    <a routerLink="/proveedores" style="text-decoration: none">
      <div class="rectorange">
```

```

        <i class="fa fa-address-book-o" style="font-size: 60px"></i>
        <p style="font-size: 24px; padding-top: 15px">Proveedores</p>
    </div>
</a>
</div>
<div class="col-md-3">
    <a routerLink="/presupuestos" style="text-decoration: none">
        <div class="rectgreen">
            <i class="fa fa-address-book-o" style="font-size: 60px"></i>
            <p style="font-size: 24px; padding-top: 15px">Presupuestos</p>
        </div>
    </a>
</div>
<div class="col-md-3">
    <a routerLink="/facturas" style="text-decoration: none">
        <div class="rectviolet">
            <i class="fa fa-address-book-o" style="font-size: 60px"></i>
            <p style="font-size: 24px; padding-top: 15px">Facturas</p>
        </div>
    </a>
</div>
<div class="col-md-3">
    <a routerLink="/contratos" style="text-decoration: none">
        <div class="rectpink">
            <i class="fa fa-address-book-o" style="font-size: 60px"></i>
            <p style="font-size: 24px; padding-top: 15px">Contratos</p>
        </div>
    </a>
</div>
</div>

```

Finalmente añadimos las clases CSS al archivo inicio.component.css:

src/app/inicio/inicio.component.css

```

.rectorange {
  background-color: #ea6112;
  color: white;
  text-align: center;
  padding: 20px;
}

```

```
    margin: 20px;
}

.rectorange:hover {
    box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.4), 0 6px 20px 0 rgba(0, 0, 0, 0.4);
}

.rectgreen {
    background-color: #12ea6f;
    color: white;
    text-align: center;
    padding: 20px;
    margin: 20px;
}

.rectgreen:hover {
    box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.4), 0 6px 20px 0 rgba(0, 0, 0, 0.4);
}

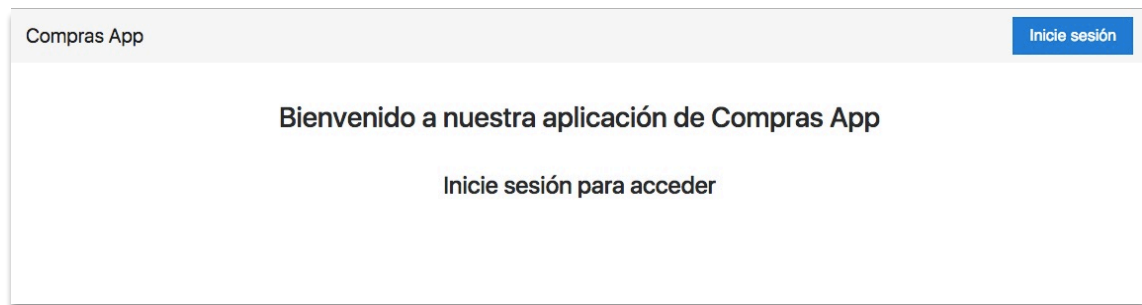
.rectviolet {
    background-color: #b412ea;
    color: white;
    text-align: center;
    padding: 20px;
    margin: 20px;
}

.rectviolet:hover {
    box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.4), 0 6px 20px 0 rgba(0, 0, 0, 0.4);
}

.rectpink {
    background-color: #ea1261;
    color: white;
    text-align: center;
    padding: 20px;
    margin: 20px;
}

.rectpink:hover {
    box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.4), 0 6px 20px 0 rgba(0, 0, 0, 0.4);
}
```

De esta manera, cuando no esté iniciada sesión, la aplicación mostrará:



Y al iniciar sesión:



14 Despliegue a producción

Ha llegado el momento de llevar a producción nuestra aplicación, por lo que en esta unidad vamos a aprender a realizar el build de la aplicación y posteriormente subirla a un bucket de Amazon Web Services, para hacerla accesible desde cualquier navegador web.

14.1 Build

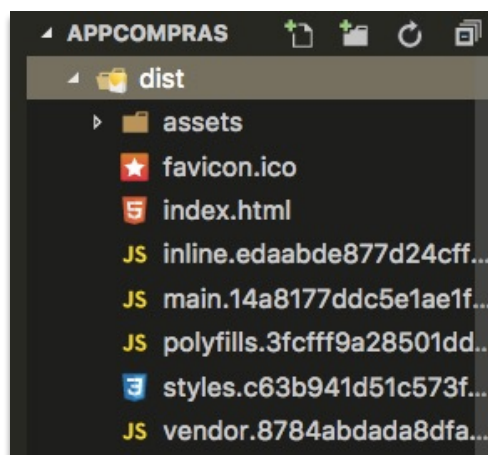
Angular permite a través de su herramienta Angular CLI realizar un build de la aplicación para ser desplegada en producción.

Este proceso crea un directorio dist (de distribución) en nuestro proyecto que compila, minifica y comprime los archivos de tal manera que conseguimos un bundle de muy pequeño tamaño para subir al servidor, cuyo punto de entrada seguirá siendo el archivo index.html.

El proceso es tan sencillo como, en la consola del equipo y en el directorio del proyecto escribir:

```
ng build --prod
```

El proceso puede durar varios minutos en función del tamaño de la aplicación, y una vez finalizado tendremos la versión de producción de nuestra aplicación en el directorio dist:



14.2 Despliegue a Amazon Web Services S3 Bucket

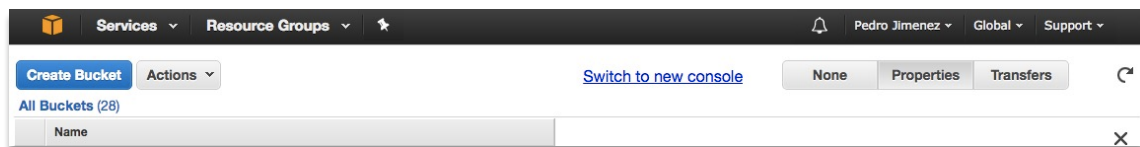
Una vez que tenemos la versión de aplicación de nuestra aplicación existen cientos de opciones para desplegarla en un servidor web.

Una sencilla opción es emplear un Bucket del servicio S3 de Amazon Web Service, para lo cual necesitaremos una cuenta en esta plataforma.

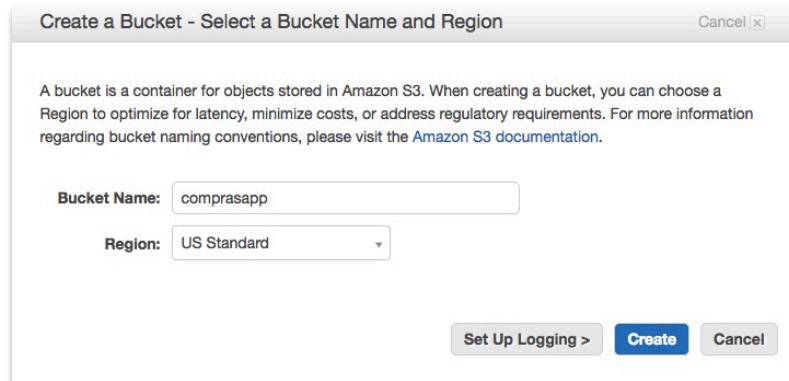
Si no dispones de cuenta, este enlace del proveedor documenta los pasos a seguir para conseguir una con recursos gratuitos durante un año:

<https://aws.amazon.com/es/free/>

Una vez dentro de Amazon Web Services, accedemos al servicio S3 y pulsamos en el botón Create Bucket:

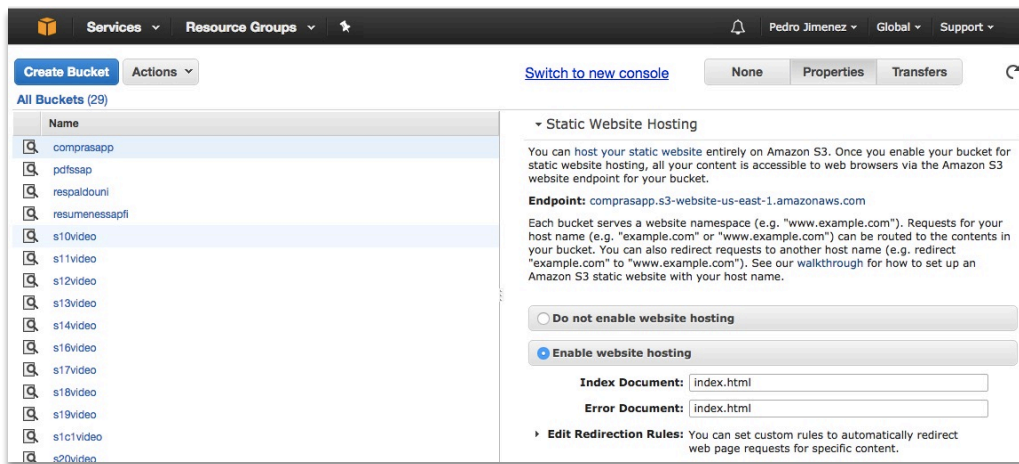


En el cuadro completamos un nombre, por ejemplo comprasapp, una zona, por ejemplo US Standar y pulsamos en create:

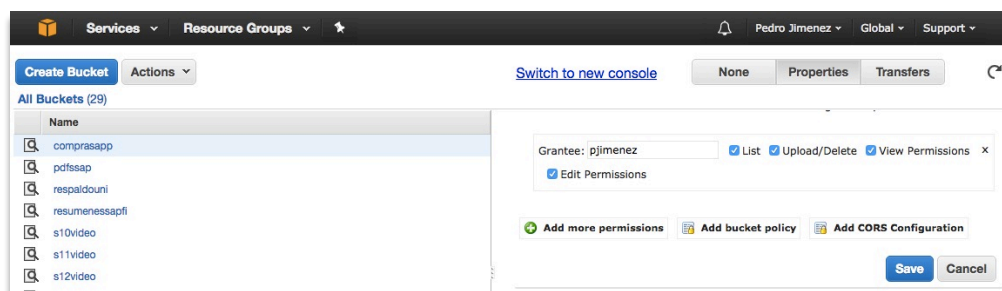


En nuestro nuevo Bucket pulsamos en la opción Static Website Hosting y dentro de ella en Enable Website Hosting.

En esta opción, añadimos index.html tanto en el campo Index Document como en el Error Document, y guardamos los cambios:



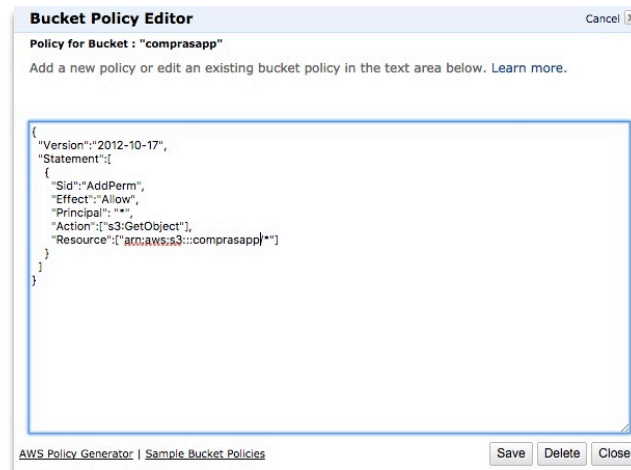
A continuación vamos a cambiar los permisos del Bucket, para lo cual pulsamos en la opción Permissions y dentro de esta pulsamos en Add bucket policy:



En el cuadro vamos a pegar la siguiente política, que como vemos contiene el nombre de nuestro bucket:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AddPerm",
      "Effect": "Allow",
      "Principal": "*",
      "Action": ["s3:GetObject"],
      "Resource": ["arn:aws:s3:::comprasapp/*"]
    }
  ]
}
```

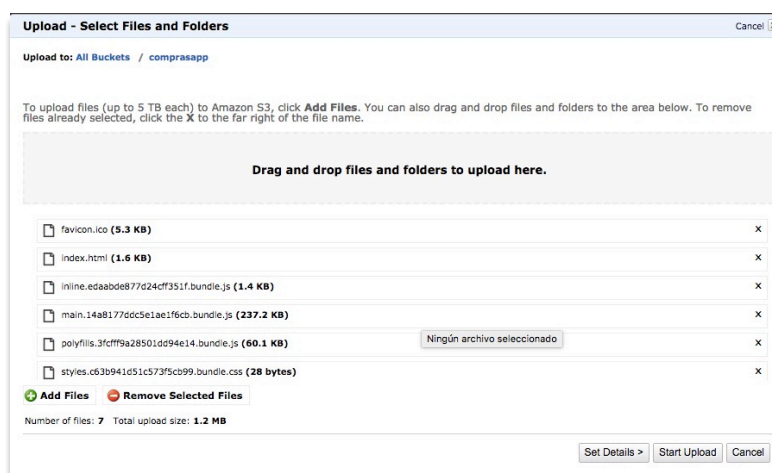

Y guardamos los cambios:



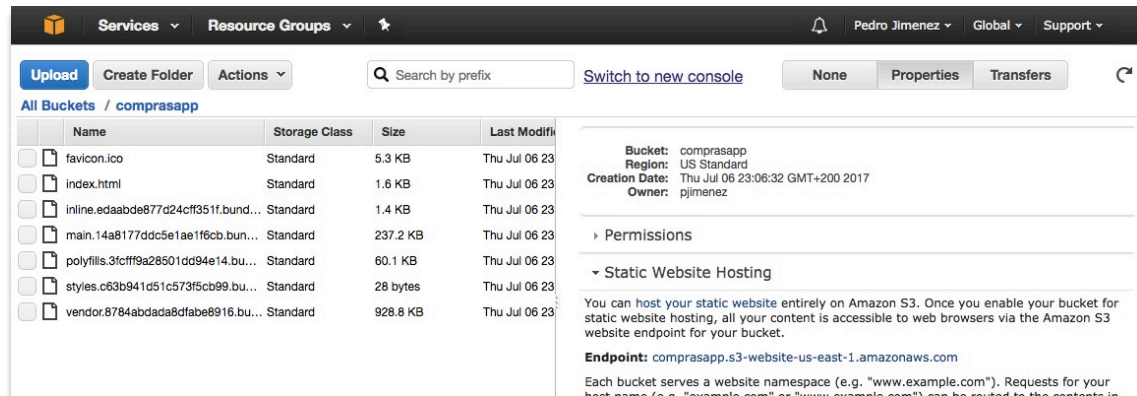
Nuestro bucket ya está listo así que hacemos doble clic sobre él y pulsamos en la opción Upload :



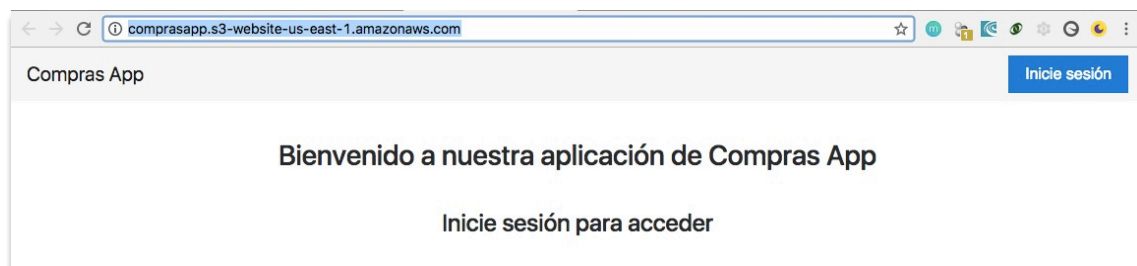
Y seleccionamos nuestros archivos de producción que recordemos se encuentran en la carpeta dist del proyecto, para finalmente pulsar en Start Upload:



Una vez que finalice la carga, accedemos de nuevo a la opción Static Website hosting y en la opción Endpoint se encuentra la url de nuestro bucket:



Que será la url de nuestra aplicación en producción:



Por supuesto se le puede añadir un nombre de dominio de AWS o cualquier otro proveedor, así como cualquier sistema de protección recomendable.

Anexo Extra I Instalación de Node y NPM

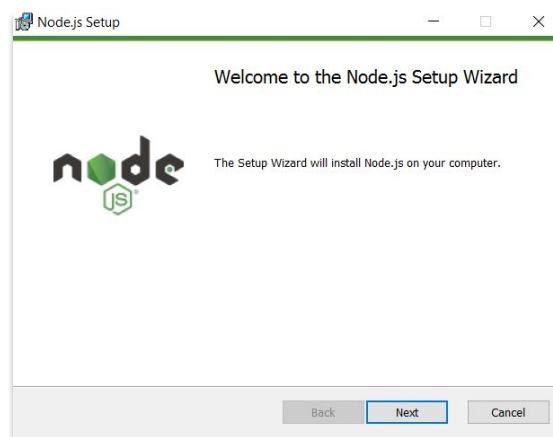
Para instalar Node y NPM en nuestro equipo, accedemos a la web de Node JS en la url:

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

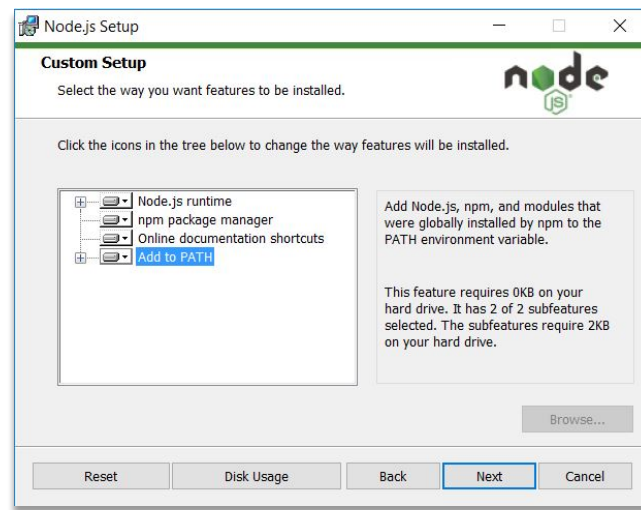
Directamente en la web pulsamos en Descargar para Windows (la web detectará nuestro sistema operativo si empleamos otro) para descargar el paquete LTS de instalación:



Una vez descargado el archivo .msi lo ejecutamos con doble clic desde nuestra carpeta de descargas para iniciar el asistente:



Realizamos los pasos que nos vaya solicitando el asistente por defecto, y si nos fijamos, el asistente instalará tanto Node Js como NPM, y añadirá sus rutas al path de windows:



Una vez concluida la instalación es necesario reiniciar el equipo para que las rutas se carguen y podamos emplear node o nmp desde la consola.

Una vez que se reinicie el equipo, comprobamos la correcta instalación tecleando en la consola:

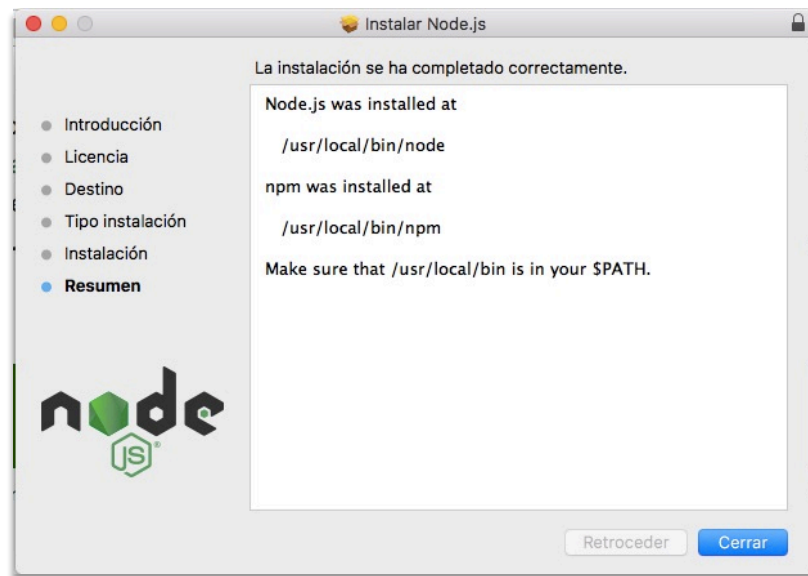
`node -v`

Y

`npm -v`

Para obtener las versiones y la certeza de que ambos están instalados.

En el caso de instalar Node y NPM en un equipo Mac el proceso es exactamente igual, siendo la última pantalla del asistente la siguiente:



En el cual se nos especifica donde han sido instalados Node JS y NPM y como debemos tener establecido en el path la ruta /usr/local/bin/

Anexo Extra II de Visual Studio Code

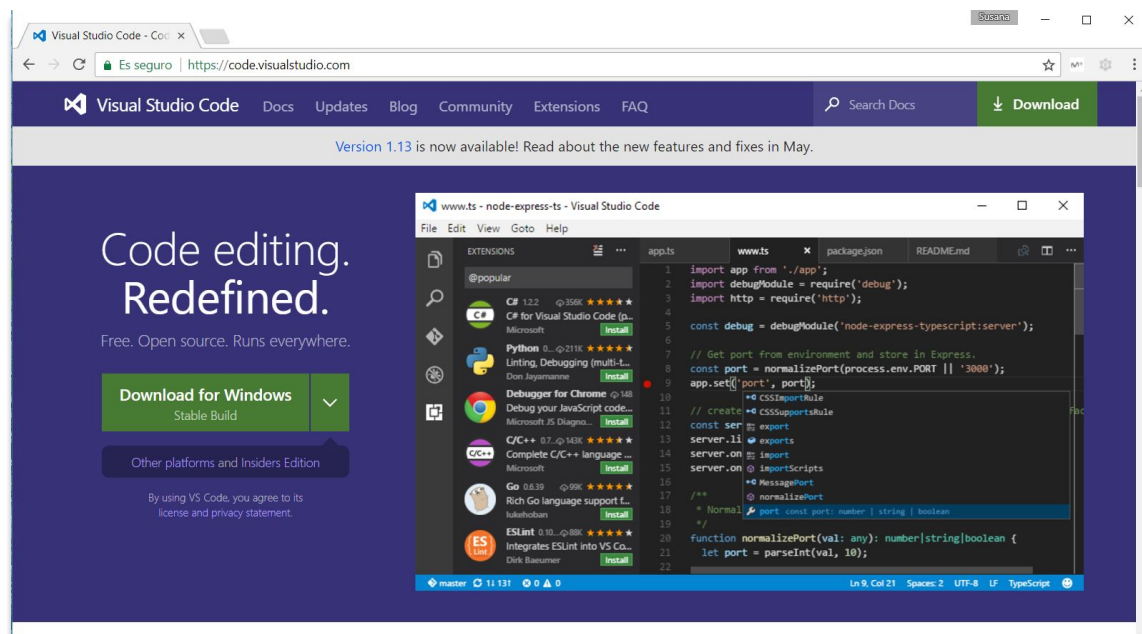
Existen multitud de editores de código e IDEs para desarrollar aplicaciones en Angular.

Nuestro editor favorito es Visual Studio Code, que además distribuye libremente Microsoft.

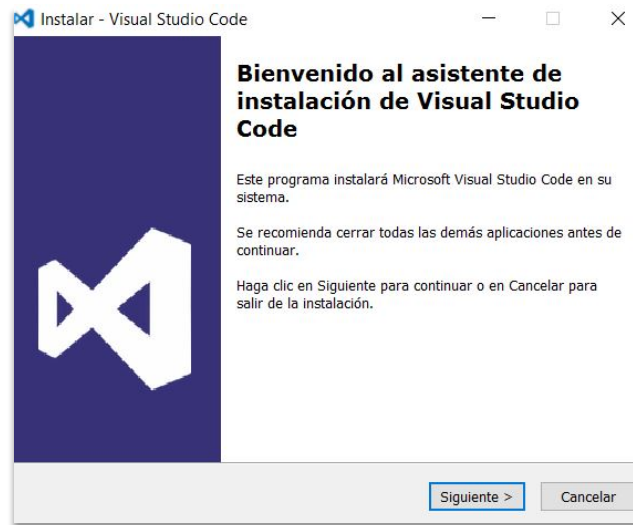
Para instalarlo accedemos a la url:

<https://code.visualstudio.com/>

Y directamente pulsamos en Download for Windows (la web detectará nuestro sistema operativo si empleamos otro):

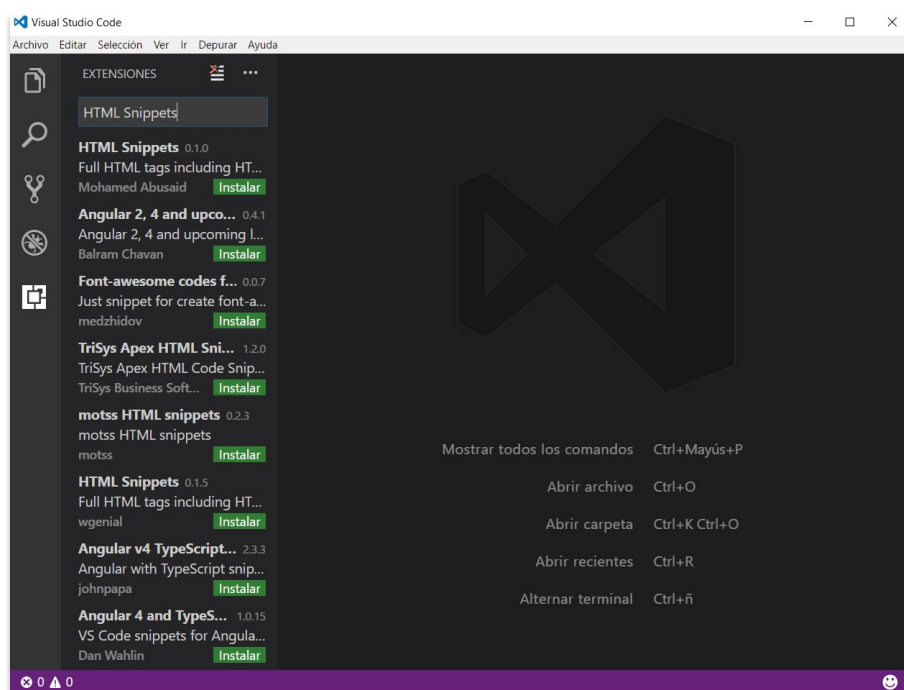


Una vez descargado el paquete, lo ejecutamos y realizamos los pasos que nos vaya solicitando:

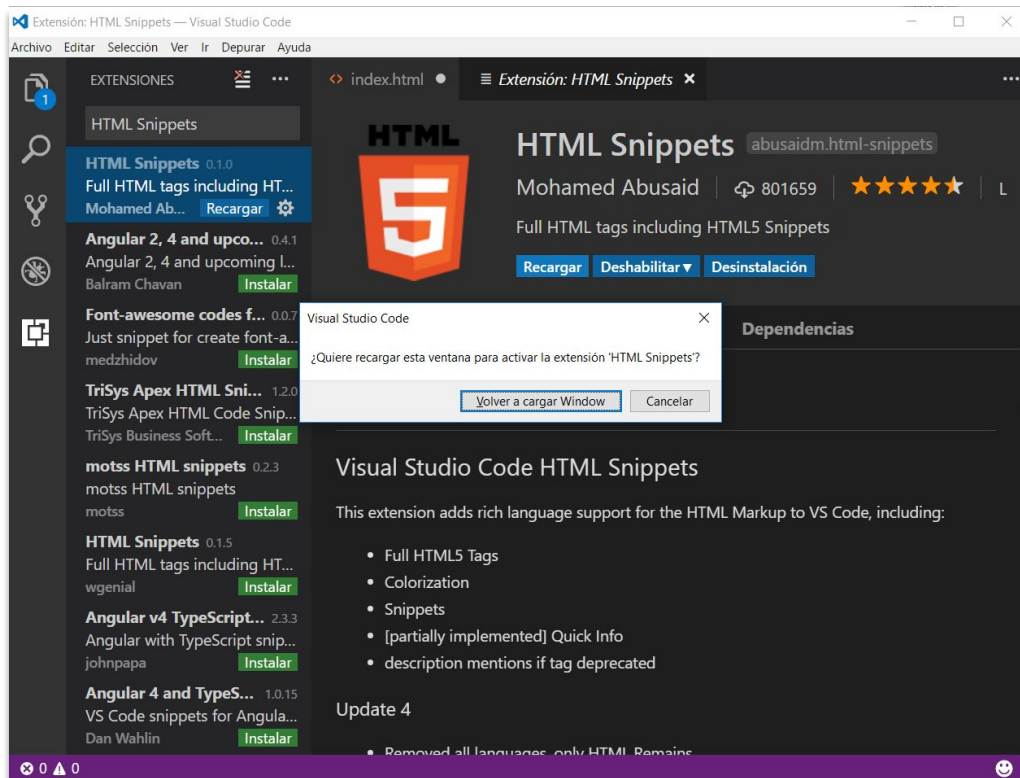


Reiniciamos el equipo y dispondremos de Visual Studio Code. La mayoría de funcionalidades interesantes de Visual Studio Code, se implementan a través de *plugins*.

Para instalar un plugin, pulsamos en el último botón del menú lateral izquierdo y en el buscador, completamos el nombre de un *plugin*, por ejemplo HTML Snippets y pulsamos en el botón verde Instalar:



Para que funcione, en la pantalla de detalle del plugin pulsamos en el botón azul Recargar y en el cuadro de diálogo pulsamos en Volver a cargar Windows.



Con esto, tendremos el plugin implementado y funcionando.

Para desarrollar en Angular, además de este plugin, son también muy interesantes los siguientes:

Angular Essentials del desarrollador johnpapa

Autorename Tag del desarrollador Jun Han

Bracket Pair Colorizer del desarrollador CoenraadS

Material Icon Theme del desarrollador Phillip Kief

Path Intellisense del desarrollador Christian Kohler

TSlint del desarrollador egamma

Anexo Extra III Ciclo de Vida Angular

Angular tiene un ciclo de vida compuesto por las diferentes fases en las que genera la aplicación. Al tratarse de una arquitectura de componentes, las fases del ciclo de vida van instanciando las clases de los componentes y Angular dispone de una serie de métodos hooks que se llaman a medida que se genera la aplicación.

Cuando se instancia un componente en el navegador a partir de la etiqueta html definida en su selector, Angular pasa por una serie de fases en ese proceso de creación del componente, llamando a varios métodos hook.

El primer hook es `ngOnChanges`, llamado justo al inicio de la creación del componente y posteriormente llamado cada vez que se produzca un cambio en una de las propiedades.

El segundo método es `ngOnInit`, llamado justo cuando el componente ha sido inicializado. Este ciclo permite inicializar objetos o propiedades que vayamos a necesitar en el componente de una forma similar al constructor, y Angular CLI lo implementa como interfaz.

La principal diferencia entre el constructor y `ngOnInit` es que mientras el primero es un elemento propio de JavaScript y podríamos decir, ajeno a Angular, el segundo se trata de una funcionalidad propia de Angular y por tanto sometida al control por parte del framework, aunque en esencia, realizan prácticamente la misma función.

El siguiente método del ciclo de vida `ngDoCheck` se ejecuta cada vez que se produce, en general, un cambio en la template del componente, tanto cambios en propiedades, como eventos, modificaciones del DOM, etc.

`ngAfterContentInit` es el siguiente de los métodos, ejecutado cuando se proyecta el contenido de `ng-content` sobre la vista, seguido de `ngAfterContentChecked`, que chequea el contenido proyectado.

`ngAfterViewInit` se ejecutará una vez que las vistas del componente (y sus vistas hojas) se han inicializado, seguido de `ngAfterViewChecked`.

Para concluir, el ciclo de vida del componente finaliza con el método `ngOnDestroy` cuando el componente es eliminado del DOM.