El gran libro de Angular

Miquel Boada Oriols
Juan Antonio Gómez Gutiérrez

· 100 ejercicios prácticos

Alfaomega

Marcombo

El gran libro de Angular

Miquel Boada Oriols y Juan Antonio Gómez Gutiérrez

El gran libro de Angular

Miquel Boada Oriols y Juan Antonio Gómez Gutiérrez

Alfaomega

Marcombo

Datos catalográficos
Boada, Miquel y Gómez, Juan Antonio
El gran libro de Angular
Primera Edición

Alfaomega Grupo Editor, S.A. de C.V., México

ISBN: 978-607-538-378-1

Formato: 17 x 23 cm Páginas: 456

Diseño de la cubierta: ENEDENÚ DISEÑO GRÀFICO
Revisor técnico: Pablo Martínez Izurzu
Maquetación: ArteMio

El gran libro de Angular
Miquel Boada Oriols y Juan Antonio Gómez Gutiérrez
ISBN: 978-84-267-2604-9, de la edición publicada por MARCOMBO, S.A., Barcelona, España

Primera edición: Alfaomega Grupo Editor, México, enero 2019

Pitágoras 1139, Col. Del Valle, 03100, México D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **atencionalcliente@alfaomega.com.mx**

ISBN: 978-607-538-378-1

Nota importante:
La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro y en el material de apoyo en la web, ni por la utilización indebida que pudiera dársele. d e s c a r g a do en: e y b o oks. c o m

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, Ciudad de México – C.P. 06720. Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490.
Sin costo: 01-800-020-4396 – E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia, Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile
Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Av. Córdoba 1215, piso 10, CP: 1055, Buenos Aires, Argentina, – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaeditor.com.ar

DEDICATORIA DE MIQUEL BOADA

A Irela, compañera de vida, gracias por estar a mi lado y apoyarme en los retos que he afrontado. Te quiero.

A mis padres, Lluís y Concepció, gracias por todo vuestro amor, esfuerzo y dedicación. Sin vosotros dificilmente hubiera llegado a escribir este libro.

También dedico el libro a mis hermanas Montse e Imma, y a todas aquellas personas que forman parte de mi vida.

DEDICATORIA DE JUANTO

Dedico este libro a mi mujer Victoria y a mi hijo Pablo ya que son la principal fuente de amor y ánimo que me motiva a emprender cualquier reto.

También quisiera dedicarlo a mis padres y a mi hermana Sebi (especialmente) ya que son los que más me han animado a emprender, entre otros, el apasionante camino de las publicaciones.

Quiero agradecer a mis amigos Víctor G. y Jordi P., el cariño y reconocimiento (por otra parte, mutuo) que siempre me ha ayudado a seguir esforzándome.

Por último, quisiera agradecer a mis buenos compañeros Jordi B. y Xavi T., todo el cariño mostrado durante una buena parte de mi carrera profesional.

A vosotros…

CONTENIDO

HTTP CLIENT

FORMS

MEAN STACK

CONCEPTOS SOBRE LENGUAJES COMPLEMENTARIOS

HERRAMIENTAS INDIRECTAS

BOOTSTRAP

PLATAFORMA DE CONTENIDOS INTERACTIVOS

Para tener acceso al material de la plataforma de contenidos interactivos del libro, siga los siguientes pasos:

1. Ir a la página: **http://libroweb.alfaomega.com.mx**
2. Ir a la sección Catálogo y seleccionar la imagen de la portada del libro, al dar doble clic sobre ella, tendrá acceso al material descargable, complemento imprescindible de este libro, el cual podrá descomprimir con la clave **ANGULAR1**

Conocer Angular

001 Introducción

Angular es una plataforma que permite desarrollar aplicaciones web en la sección cliente utilizando **HTML** y **JavaScript** para que el cliente asuma la mayor parte de la lógica y descargue al servidor con la finalidad de que las aplicaciones ejecutadas a través de Internet sean más rápidas. El hecho de estar mantenido por **Google**, así como una serie de innumerables razones técnicas, ha favorecido su rápida adopción por parte de la comunidad de desarrolladores.

Importante

Angular es una plataforma que permite desarrollar aplicaciones web utilizando **HTML** y **JavaScript** en la parte cliente descargando al servidor de buena parte del trabajo, con lo que se consigue una mayor velocidad en la ejecución y, por tanto, un mayor rendimiento.

Permite la creación de aplicaciones web de una sola página (**SPA: *single-page application***) realizando la carga de datos de forma asíncrona.

Además de mejorar el rendimiento de las aplicaciones web, su utilización en dispositivos móviles está optimizada ya que, en ellos, los ciclos de CPU y memoria son críticos para su óptimo funcionamiento. Ello permite el desarrollo de aplicaciones móviles híbridas con **Ionic 2**.

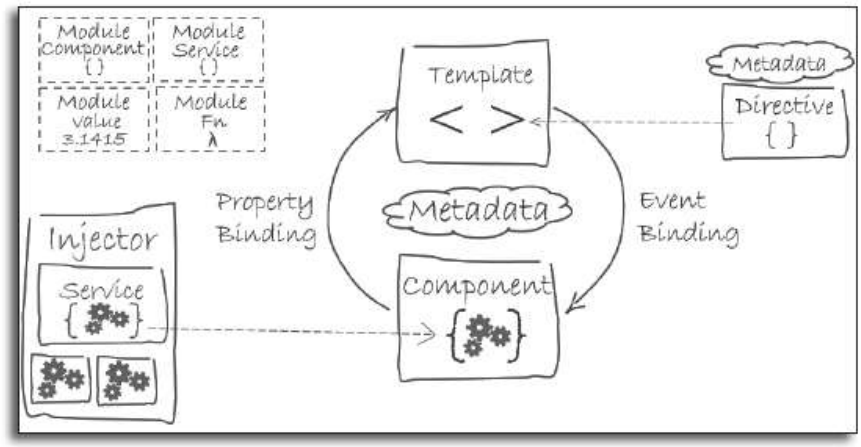
Gracias al uso de componentes, se puede encapsular mejor la funcionalidad facilitando el mantenimiento de las aplicaciones.

Parecería ser la continuación de **AngularJS,** pero, en realidad, más que una nueva versión es realmente un framework o plataforma diferente. El hecho de utilizar componentes como concepto único en lugar de controladores, directivas y servicios de forma específica, como sucedía en **AngularJS**, simplifica mucho las cosas.

Angular está orientado a objetos, trabaja con clases y favorece el uso del patrón **MVC** (**Modelo-Vista-Controlador**).

Permite el uso de **TypeScript** (lenguaje desarrollado por **Microsoft**) con las ventajas que supone poder disponer de un tipado estático y objetos basados en clases. Todo ello, gracias a la especificación **ECMAScript 6**, que es la base sobre la que se apoya **TypeScript**. Gracias a un compilador **(transpilador**) de **TypeScript**, el código escrito en este lenguaje se traducirá a **JavaScript** original.

En la web de **Angular** podemos observar el diagrama de arquitectura que muestra cómo se relacionan los elementos principales de una app los cuales son los siguientes:



- Modules.
- Components.
- Templates.
- Metadata.
- Data binding.
- Directives.
- Services.
- Dependency injection.

Una visión más sencilla de la arquitectura **MVC** nos permitiría enumerar los siguientes elementos:

- **Vistas:** Componentes.
- **Capa de control:** Router.
- **Backend:** Servicios.

A lo largo del libro, analizaremos cada uno de estos elementos y veremos, entre otras muchas cosas, que los componentes son la suma de los templates, las clases y los metadatos.

Veremos también que la inyección de dependencias favorece el **testing** y reduce el acoplamiento entre clases. Con **Lazy SPA** no es necesario tener en memoria todos los elementos que componen una aplicación, de forma que podemos cargarla por partes a medida que los necesitemos.

Podemos ejecutar **Angular** en el servidor gracias a **Angular Universal**, con lo que podremos reducir el tiempo de espera que se produce en la primera visita ya que, en lugar de cargar todo lo necesario para ejecutar la aplicación, se pueden enviar vistas "prefabricadas" directamente al cliente.

002 Introducción a las aplicaciones SPA

Angular es un framework principalmente enfocado a la creación de aplicaciones web de tipo single-page application (SPA). En este capítulo haremos un breve repaso a los distintos tipos de aplicación web y, a continuación, analizaremos con más detalle en qué consisten las aplicaciones single-page application (SPA).

Importante

Angular es un framework que ofrece todas las herramientas necesarias para crear aplicaciones de tipo single-page application (SPA).

Una aplicación web la podemos definir como toda aquella aplicación proporcionada por un servidor web y utilizada por los usuarios a través de un cliente web (browsers o navegadores). En otras palabras, son aquellas aplicaciones codificadas en un lenguaje soportado por los navegadores web para que puedan ejecutarse desde allí.

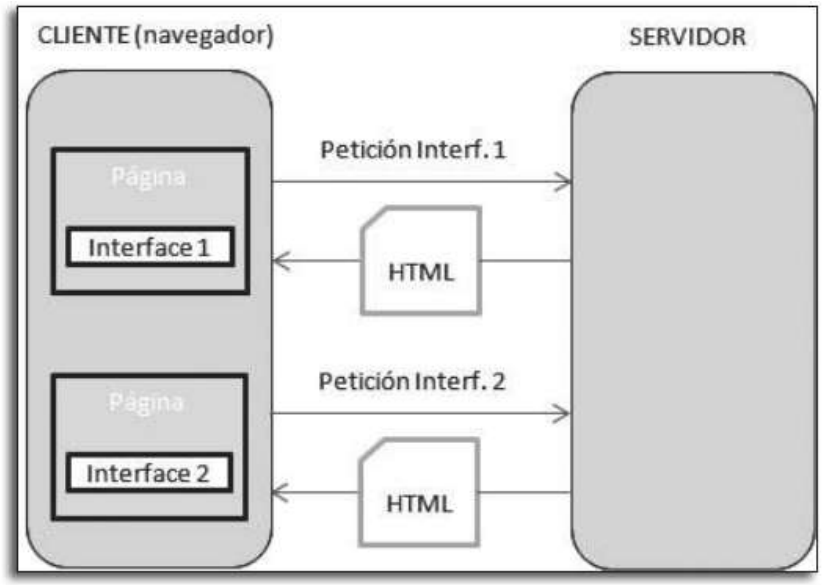
Toda aplicación web tiene una arquitectura básica de tipo cliente-servidor. Esto es así porque toda aplicación web sigue un modelo de aplicación distribuida en la que, por una parte, está el servidor que provee de recursos y servicios y, por la otra, está el cliente que los demanda. A partir de aquí, según lo estática o dinámica que sea cada parte, tenemos distintos tipos de aplicaciones web:

- Cliente y servidor estáticos: no hay ningún tipo de dinamismo y el servidor siempre devuelve los mismos recursos sin ningún tipo de cambio. Más que de aplicaciones web, aquí hablaríamos de páginas web.
- Cliente estático y servidor dinámico: el servidor devuelve recursos dinámicos, por ejemplo, páginas web con el resultado de consultas a base de datos, etc.
- Cliente/servidor dinámicos: el cliente es dinámico porque las páginas web recibidas del servidor incluyen JavaScript, que se ejecuta en el propio navegador dando todo tipo de funcionalidades diversas.

Hoy en día, la mayoría de aplicaciones web disponen de una parte cliente dinámica con el objetivo de disminuir la comunicación con el servidor y mejorar la fluidez y experiencia del usuario. Sin embargo, según hasta donde se lleve esta técnica, podemos hablar de grandes patrones de diseño:

- Multi page web applications (MPA).
- Single-page application (SPA).

Las aplicaciones multi page web applications (MPA) corresponderían al tipo tradicional de aplicación web. Su característica principal es que la mayoría de acciones de usuario se transforman en solicitudes al servidor de páginas completas (incluyendo diseño, css [hojas de estilo], JavaScript y contenido).



Este tipo de diseño es perfectamente funcional para cualquier tipo de aplicación; sin embargo, para aplicaciones complejas pueden existir problemas de lentitud en la navegación. Por ejemplo, una aplicación con una rica interfaz de usuario seguramente tendrá páginas muy complejas y, en este contexto, generar estas páginas en el servidor, transferirlas al cliente y visualizarlas en el navegador, puede requerir un tiempo considerable que afecte negativamente a la experiencia de usuario.

Sin embargo, para mejorar este posible problema de fluidez, las aplicaciones MPA suelen hacer uso de AJAX. AJAX no es una tecnología en sí misma, sino que es un conjunto de tecnologías independientes (JavaScript, XML, etc.) usadas con el fin de permitir refrescar de forma asíncrona partes de una página sin necesidad de recargarla toda de nuevo.

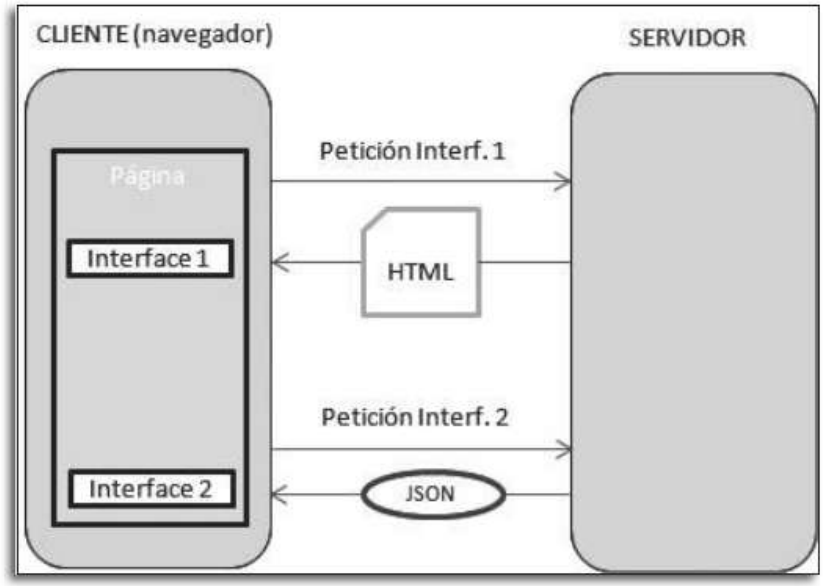
Por otra parte, el patrón de diseño single-page application (SPA) es una evolución del patrón de diseño MPA + AJAX, pero llevando al extremo el uso de AJAX. Hasta el punto de que en el cliente se carga una única página que se modifica desde el propio cliente (navegador) según las acciones de usuario. Por tanto, toda la navegación por las distintas pantallas o interfaces de la aplicación se realizará sin salir de esa única página.

Una de las principales ventajas de las aplicaciones SPA respecto las MPA es la mejora de experiencia de usuario debido a la reducción en el tiempo de respuesta ante las acciones del usuario. Esto se consigue gracias a que:

- Ya no se crean páginas completas por cada acción del usuario.
- Solo se intercambia la información necesaria con el servidor.

En las aplicaciones SPA, la responsabilidad del aspecto de la aplicación recae principalmente en la parte cliente, mientras que el servidor tiene la función de ofrecer al cliente una API de servicios para dar acceso a la base de datos de la cual se alimenta la aplicación. Los datos intercambiados entre cliente y servidor suelen estar en formato JSON, que es un formato más óptimo que el tradicional XML.

En las aplicaciones Web, la programación del lado cliente se realiza con html5 + JavaScript. Sin embargo, no suele usarse estos lenguajes de forma directa, sino que suele hacerse mediante librerías (jQuery) y/o frameworks (Angular, Backbone.js, Ember.js, etc.). Las librerías sirven para facilitar el uso de JavaScript aportando todo tipo de herramientas, pero los frameworks, aparte de aportar herramientas, aportan patrones de diseño para el desarrollo de aplicaciones complejas. Angular es uno de estos framework, quizás uno de los más usados y, como veremos a lo largo del libro, aporta todo lo necesario para crear y mantener aplicaciones single-page application (SPA).



003 Breve historia de Angular

El concepto de las single-page applications (SPA) empezó a debatirse en 2003, pero no fue hasta 2005 que se utilizó el termino por primera vez. Como vemos, se trata de un tipo de aplicación relativamente nueva, sin embargo, su uso se está extendiendo rápidamente y a día de hoy ya detectamos su presencia en aplicaciones tan conocidas como Gmail, Outlook, Facebook y Twitter.

El auge de este tipo de aplicaciones se debe en parte a las ventajas que aportan respecto a otras soluciones, pero, sobre todo, a la gran inversión que están haciendo grandes empresas tecnológicas para desarrollar frameworks que permitan su desarrollo. Este es el caso de Google con el framework Angular, o Facebook con el framework React.js.

Importante

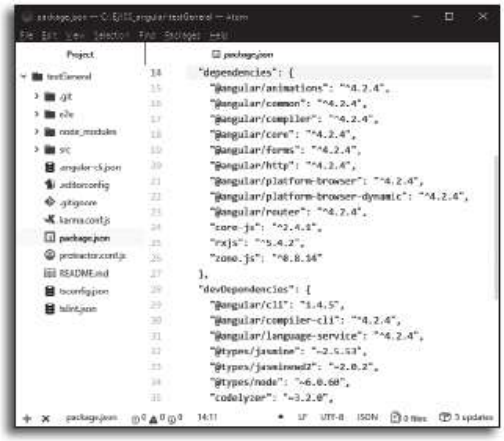
El framework Angular se rediseñó por completo en la versión 2.0, y pasó de llamarse AngularJS a Angular. Para no dejar colgados a los desarrolladores de AngularJS, Google sigue dando soporte y mantenimiento a AngularJS. En la actualidad, AngularJS se encuentra en la versión 1.6.4, y Angular en la versión 4.0.

AngularJS fue desarrollado en 2009 por Miško Hevery y Adams Abrons. Originalmente era el software detrás de un servicio de almacenamiento online de archivos JSON, pero, poco tiempo después, se abandonó el proyecto y se liberó AngularJS como una biblioteca de código abierto. Adams Abrons dejó el proyecto, pero Miško Hevery, como empleado de Google, continuó el desarrollo y mantenimiento del framework.

Desde entonces se han ido lanzando muchas versiones de AngularJS, pero, sobre todas ellas, cabe destacar la salida de la **versión 2.0** por los siguientes motivos:

- **Se rediseñó por completo todo el framework** (nueva arquitectura basada en componentes, etc.).
- Se introdujo el uso de **TypeScript** de Microsoft (superconjunto de JavaScript) como lenguaje de programación.
- Se cambió el nombre del framework. Pasó de llamarse **AngularJS** a **Angular.**
- Los desarrolladores anunciaron que darían soporte y mantenimiento tanto para AngularJS (versiones 1.X.Y) como para Angular (versiones superiores), pero que una y otra seguirían ciclos de vida independientes. Actualmente, **AngularJS se encuentra en la versión 1.6.4**, mientras que **Angular se halla en la versión 4.0.**

En un primer momento, la salida de Angular tuvo muchas críticas por parte de los desarrolladores que usaban AngularJS, ya que veían como la actualización de sus desarrollos al nuevo framework requería un trabajo muy laborioso de migración. Por otra parte, también veían que su experiencia y conocimiento no les servía de mucho con los nuevos conceptos de Angular.



Sin embargo, a medida que los desarrolladores apreciaron las ventajas de la nueva versión, la situación mejoró. También ayudaron las distintas iniciativas de Google, como anunciar que seguirían dando soporte y mantenimiento a las antiguas versiones del framework (AngularJS) o la de documentar un protocolo de migración de los desarrollos al nuevo Angular (https://angular.io/guide/upgrade).

A continuación, sin entrar en demasiado detalle, podremos analizar algunas de las diferencias entre AngularJS y Angular mediante un sencillo ejemplo. El ejemplo consiste en una aplicación web que muestra un "Hola mundo". Primero lo veremos implementado en AngularJS y luego en Angular.

Ejemplo AngularJS

- index.html:

```
<!DOCTYPE html>
<html ng-app="testAngularJS">
  <head>
    <link rel="stylesheet" type="text/css" href="bootstrap.min.css" />
    <script type="text/javascript" src="angular.min.js"></script>
    <script type="text/javascript" src="app.js"></script>
  </head>
  <body>
    <entity-directive1></entity-directive1>
  </body>
</html>
```

- app.js: En este archivo tenemos definido el módulo y la directiva de tipo "entidad" a los que se hacía referencia en el index.html.

```
(function() {
    var app = angular.module('testAngularJS', []);
    app.directive('entityDirective1', function() {
      return {
        restrict: 'E',
        template: '<h1>{{title}}</h1>',
        scope: {},
        link: function(scope, element){
          scope.title='AngularJS: Hola mundo!!';
        }
      };
    });
  })();
```

Ejemplo Angular. Aparte de los archivos indicados a continuación, el proyecto incluye muchos otros archivos relacionados con la carga de bibliotecas necesarias y el uso de TypeScript. En la imagen anterior puede ver algunas de estas dependencias referenciadas en el archivo **package.json** de un proyecto Angular.

- index.html:

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

- *app.component.ts:* Definición del componente usado en el index.html.

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

@Component({
  selector: 'app-root',
  template: '<h1>{{title}}</h1>',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Angular: Hola mundo!!!!';
}
```

004 Instalación

Para trabajar con **Angular** necesitamos realizar diversas instalaciones, algunas de las cuales son imprescindibles, como por ejemplo Node.js, y otras son opcionales, aunque recomendadas, ya que nos facilitarán enormemente el trabajo. Por ejemplo, el usuario puede utilizar el editor que más le guste, pero recomendamos emplear algunos editores que aportan un sinfín de utilidades a la hora de comprobar la sintaxis de nuestras sentencias o de aportar bloques de código a modo de plantilla.

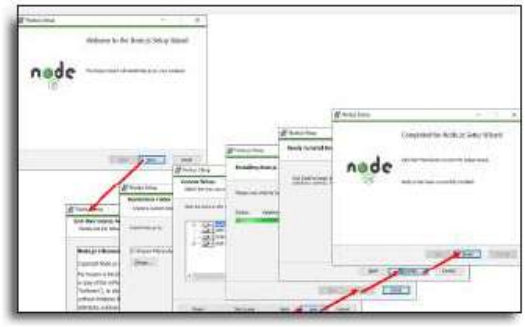
Importante

Procure instalar la última versión de los productos indicados para mantenerse actualizado y disfrutar de las últimas actualizaciones en todo momento.

1. En primer lugar instalaremos **Node.js**. Para ello podemos descargarlo desde la siguiente página:

   https://nodejs.org/es/download/

   En esta página, seleccionaremos la opción que más nos interese según nuestro SO y el número de bits (**32** o **64**). En nuestro caso, seleccionamos la opción **Windows Installer (.msi) 64-bit** y descargamos el instalador en una carpeta desde la que realizaremos posteriormente la instalación.

   Una vez descargado, ejecutaremos el instalador y podemos aceptar todos los valores que nos propone por defecto hasta que lleguemos a la pantalla final y pulsemos **Finish**.

2. Seguidamente, instalaremos **TypeScript**. Para ello, podemos acudir al siguiente link:

   https://www.typescriptlang.org/

Al pulsar sobre el botón **download** aparece una pantalla en la que se muestra la sentencia a ejecutar para instalar **TypeScript**

```
npm install -g typescript
```

A continuación, copiaremos dicha sentencia y abriremos una ventana de **CMD** (**Ejecutar->CMD**) para pegarla y ejecutarla.

3. A continuación, instalaremos **Angular-Cli**. Para ello, acudiremos a la siguiente página:

https://cli.angular.io/

En dicha página, ya podemos ver diversas instrucciones entre las que hallamos la siguiente:

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

De nuevo, abrimos una ventana **CMD** (**Ejecutar->CMD**) y ejecutaremos la sentencia anterior.

Para obtener más información sobre los detalles de la instalación y prerrequisitos, puede consultar la siguiente página:
https://github.com/angular/angular-cli

4. Podemos realizar nuestras pruebas en muchos navegadores, pero nosotros realizaremos todas las explicaciones basándonos en **Google Chrome** y en sus herramientas para desarrolladores. Por tanto, sugerimos que utilice este navegador, el cual puede descargar de la siguiente página:

https://www.google.es/chrome/browser/desktop/index.html

5. Por último, tal y como hemos indicado en la introducción, necesitará un editor que le ayude a desarrollar sus componentes con **Angular** y puede utilizar alguno de los siguientes:

Atom	https://atom.io/
Visual Studio Code	https://code.visualstudio.com/download
Sublime Text	https://www.sublimetext.com/3

Nosotros utilizaremos **Atom** en nuestras explicaciones y, en este caso, una vez instalado **Atom**, recomendamos añadir una serie de funcionalidades que nos facilitarán el desarrollo con Angular. Para añadir funcionalidades, accederemos a **Help->Welcome Guide** y, seguidamente, accederemos a **Install a Package->Open Installer**. Cuando aparezca la pantalla de **Install Packages**, teclearemos el nombre del paquete que queramos añadir y, una vez que nos muestre el package deseado, pulsaremos sobre **Install**.

Algunas de las funcionalidades recomendadas se hallan en los siguientes packages:

- Angular-2-TypeScript-Snippets,
- Angular2-Snippets,
- Atom-Bootstrap 3,
- Atom-TypeScript,
- file-icons,
- Linter,
- Linter-UI-Default,
- PlatformIO-IDE-Terminal.

005 TypeScript. Introducción (variables, clases, transpilación, etc.)

TypeScript es un lenguaje de programación orientado a objetos fuertemente tipado que se traduce a **JavaScript** añadiéndole características que no posee. La operación de traducir TypeScript a JavaScript se conoce como **transpilación**.

Gracias al uso de TypeScript, es posible localizar errores de sintaxis antes incluso de su ejecución. De ahí que haya ganado mucha aceptación entre los desarrolladores del mundo web.

Importante

TypeScript permite desarrollar un código con menos errores y aprovechar toda la potencia de una programación orientada a objetos.

Podríamos dedicar un libro completo a TypeScript, pero, en este ejercicio, simplemente ofreceremos una pincelada sobre el lenguaje y elaboraremos el clásico programa **Hola Mundo** para que podamos ver cómo se compila y ejecuta.

Los programas escritos con TypeScript se suelen escribir en ficheros con la extensión **ts**.

Una vez escritos, dichos programas se transpilan (compilan) o, lo que es igual, se traducen a JavaScript para que puedan ejecutarse posteriormente. Cualquier instrucción escrita en JavaScript "puro" se copia igual (sin traducción) en el fichero resultante de la transpilación.

Para compilar usaremos el comando **TSC** y para ejecutar emplearemos el comando **node**, como veremos más adelante y el cual ya hemos instalado en capítulos anteriores.

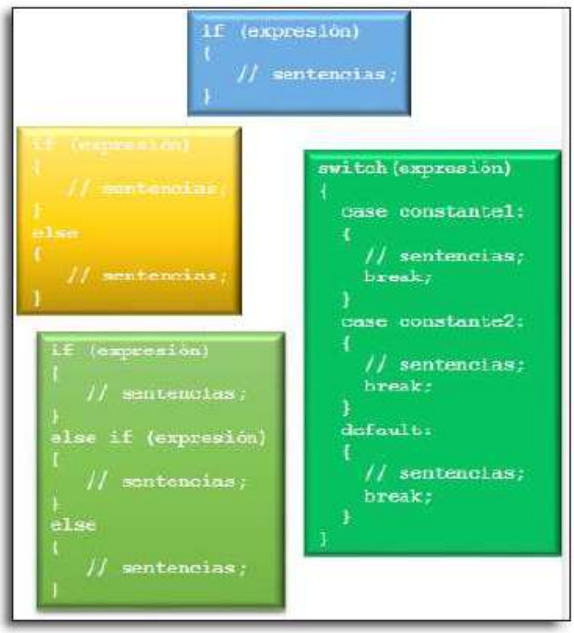
La sintaxis que hallaremos en TypeScript está compuesta por módulos, funciones, variables, sentencias, expresiones y comentarios.

Respecto de las funciones, TypeScript permite definir funciones con nombre, anónimas, tipo Lambda o de flecha, además de permitir la sobrecarga de funciones utilizando diferentes parámetros y/o tipos en sus llamadas.

Asimismo, disponemos de un gran surtido de **operadores**, **sentencias condicionales**, **loops**, **funciones** y **métodos** para **numéricos** y **strings**.

También es posible definir interfaces y clases, incluyendo en las mismas campos, constructores y funciones utilizando la herencia y encapsulación propias de **POO**.

Tipo operador	Operadores
Aritméticos	+, -, *, /, %, ++, --
Relacionales	>, <, >=, <=, ==, !=
Lógicos	&&, ||, !
Asignaciones	=, +=, -=, *=, /=
Condicional	test ? expre1 : expre2
Typeof	Devuelve tipo de datos del operando




```
function nombre_funcion():tipo_retorno
{
   //sentencias
return valor;
```

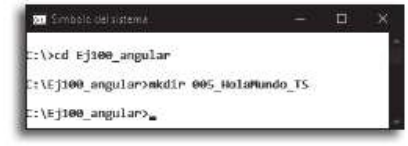
Numéricos	String		
• toExponential()	• charAt()	• charCodeAt()	• concat()
• toFixed()	• indexOf()	• lastIndexOf()	• localeCompare()
• toLocaleString()	• match()	• replace()	• search()
• toPrecision()	• slice()	• split()	• substr()
• toString()	• substring()	• toLocaleLowerCase()	• toLocaleUpperCase()
• valueOf()	• toLowerCase()	• toString()	• toUpperCase()
	• valueOf()		

Para acabar con la breve mención al lenguaje, es importante resaltar que el concepto de **namespace** permite organizar mejor el código y sustituye a los antiguos **Internal Modules**.

A continuación, vamos a crear un programa con TypeScript que simplemente muestre el texto **Hola Mundo**:

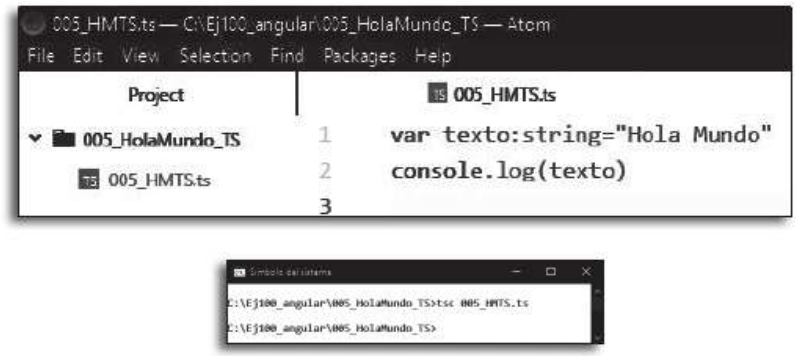
1. Para ello, nos ubicaremos en nuestro directorio de ejercicios (**Ej100_angular**) y crearemos el directorio **005_HolaMundo_TS**.

2. Seguidamente, abriremos nuestro editor favorito y, dentro del directorio recién creado, crearemos el fichero **005_HMTS.ts** con el siguiente contenido:



```
var texto:string=”Hola Mundo”
console.log(texto)
```

3. Una vez creado y guardado el fichero, accederemos a la ventana de **CMD** y nos ubicaremos en nuestro nuevo directorio (**005_HolaMundo_TS**) para teclear lo siguiente:




```
tsc 005_HMTS.ts
```

4. Observaremos que, si no se ha producido ningún error, la compilación se realiza de forma silenciosa devolviendo el control al prompt de la ventana de **CMD**.

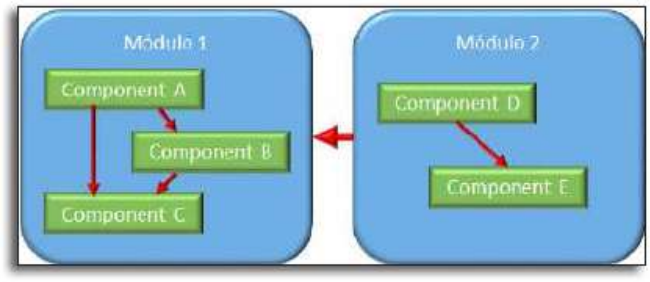
5. En este punto, procederemos a la ejecución de nuestro programa tecleando lo siguiente:

```
node 005_HMTS
```

6. Observaremos como, efectivamente, se muestra el texto **Hola Mundo** en pantalla.

006 Definición de elementos en una aplicación

En el capítulo de introducción, enumeramos los elementos que había que tener en cuenta en Angular. A continuación, los definiremos brevemente.

Elemento	Comentario
Módulo	Conjunto de códigos dedicados a resolver un objetivo que generalmente exporta una clase. Cada aplicación posee al menos una clase de módulo que denominamos "**módulo raíz**" y que, por defecto, posee el nombre de **AppModule**. En el mismo podemos apreciar una clase con un decorador (**@NgModule**) y una serie de propiedades entre las que destacamos las siguientes: • **Declarations:** define las vistas que pertenecen a este módulo. • **Exports:** declaraciones que han ser visibles para componentes de otros módulos. • **Imports:** clases que otros módulos exportan para poderlas utilizar en el módulo actual. • **Providers:** servicios usados de forma global y accesibles desde cualquier parte de la aplicación. • **Bootstrap:** la vista principal o componente raíz, que aloja el resto de vistas de la aplicación. Propiedad establecida solo por el módulo raíz. Un módulo puede agrupar diversos componentes relacionados entre sí. 

Componente	Elemento que controla una zona de espacio de la pantalla que representa la "vista". Define las propiedades y métodos que usa el propio template y contiene la lógica y la funcionalidad que utiliza la vista. Pueden tener atributos tanto de entrada (decorador **@Input**) como de salida (decorador **@Output**). También podemos definir al componente como: **Componente = template + metadatos + clase**
Template	Define la vista de un **Componente** mediante un fragmento de **HTML**.
Metadatos	Permiten decorar una clase y configurar así su comportamiento. Extiende una función mediante otra sin tocar la original.
Data binding	Permite intercambiar datos entre el template y la clase del componente que soporta la lógica del mismo. Existen 4 tipos de **Data binding** (que analizaremos en profundidad más adelante) y que son: • **Interpolación:** muestra el valor de una propiedad en el lugar donde se incluya **{{ propiedad }}**. • **Property binding:** traspaso del objeto del componente padre al componente hijo (**[objHijo]="objPadre"**). • **Event binding:** permite invocar un método del componente pasándole un argumento al producirse un evento (p. ej., **(click)="hazAlgo(obj)"**). • **Two-way binding:** binding bidireccional que permite combinar **event** y **Property binding** (p. ej., **input [(ngModel)]= "obj"**).
Directiva	Permite añadir comportamiento dinámico a HTML mediante una etiqueta o selector. Existen directivas estructurales (***ngFor** o ***ngIf**) o de atributos que modifican el aspecto o comportamiento de un elemento **DOM** (**NgSwitch**, **NgStyle** o **NgClass**).
Servicio	Son clases que permiten realizar acciones concretas que ponen a disposición de cualquier componente. Por ejemplo, permiten obtener datos, resolver lógicas especiales o realizar peticiones a un servidor.
Dependency injection	Permite suministrar funcionalidades, servicios, etc., a un componente sin que sea el propio componente el encargado de crearlos. Utiliza el decorador **@Injectable()**. Generalmente proporciona servicios y facilita los test y la modularidad del código.

Al crear una aplicación, por defecto se crea el módulo **app.module.ts**, como veremos en ejercicios posteriores.

En el mismo, podemos apreciar diferentes partes correspondientes a:

- Los "**imports**":
  - **BrowserModule:** usado para renderizar la aplicación en el navegador.
  - **NgModule:** decorador necesario para el módulo.
  - **AppComponent:** componente principal del módulo.

Importante

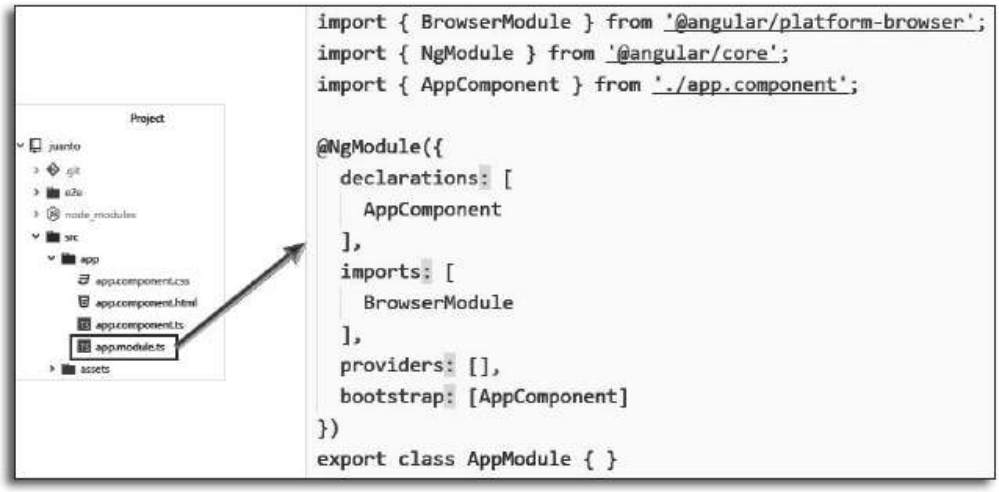
Cree módulos para separar los grupos de componentes que dan soporte a una lógica determinada y que pueden reutilizarse en diversas aplicaciones.

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

- Al decorador **@NgModule** con su metadata:
  - **Declarations:** AppComponent (vista del módulo por defecto).
  - **Imports:** BrowserModule (clase necesaria para el navegador).
  - **Providers:** en este caso, no disponemos de ninguno.
  - **Bootstrap:** AppComponent (componente principal).
- A su clase: permite cargar la aplicación **(AppModule)**.

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

El archivo total agrupa todas las partes descritas que constituyen el módulo en sí mismo.



007 Definición de un componente

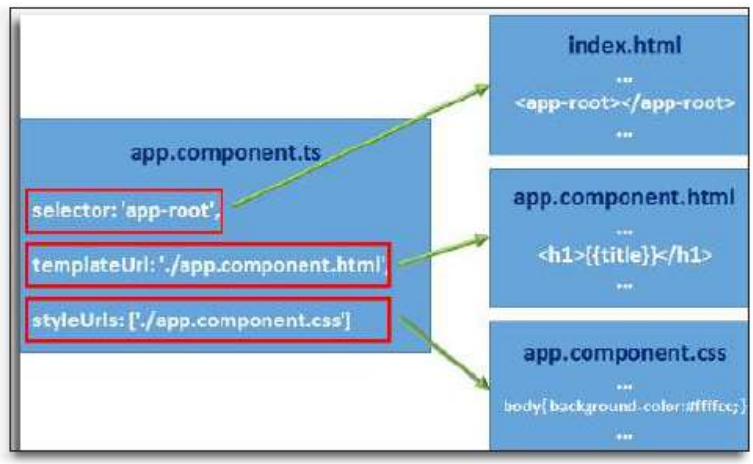
Un componente es una clase a la que se le añade un decorador **(@Component)** y que permite crear nuevas etiquetas **HTML** mediante las cuales podemos añadir funcionalidad a nuestras páginas controlando determinadas zonas de pantalla. Básicamente, los componentes se organizan en forma de árbol teniendo un componente principal donde, por defecto, su propiedad "selector" posee el valor **app-root**, gracias al cual el componente puede incluirse en nuestras páginas **HTML** (p. ej., en **index.html**) mediante el tag **<app-root></app-root>** como veremos más adelante.

index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>007 Componentes</title>
  <base href="/">
  <meta name="viewport"
        content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon"
        href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Los componentes se componen de tres partes:

- **Anotaciones:** metadatos que agregamos a nuestro código y que permite definir ciertas propiedades como son:



- **Selector:** nombre a utilizar como tag en HTML para indicar a Angular que cree una instancia del componente cuando se encuentre con dicho tag.
- **Template:** plantilla a utilizar para el componente y que se describe en el propio fichero **ts** del mismo.
- **TemplateUrl:** indica el URL en el que se halla la plantilla en caso de definirse externamente.
- **Style:** definición de estilos para la plantilla.
- **StyleUrl:** array que contiene los diferentes archivos de estilos que pueden utilizarse en el componente en curso.
- **Host:** permite encapsular ciertas partes del componente.
- **Directive:** array con las directivas que pueden utilizarse en el componente.
- **Input:** define variables que recogen información del padre.
- **Output:** define variables que pasan información al padre.

- **Vista:** contiene el template que usaremos para elaborar la pantalla asociada al componente. Esta pantalla podemos definirla en un archivo auxiliar o bien dentro del propio archivo donde definimos la clase controlador encerrando el código HTML entre apóstrofos (`**<h1>{{title}}</h1>**`).

- **Controlador:** clase que contendrá la lógica del componente (definiciones, funciones, etc.)

app.component.html

```
<div class="container">
  <h1>{{title}}</h1>
</div>
```

```
export class AppComponent {
  title = '007 Componentes';
}
```

Generalmente, un componente suele llevar asociados diversos archivos con el siguiente cometido:

- **app.component.html:** contiene el código HTML mediante el que fabricamos la pantalla (o **Vista**).

- **app.component.css:** define los estilos que usaremos en la vista del componente.

- **app.component.ts:** archivo que contiene la lógica del componente (**Controlador**) y que se escribe en **TypeScript** para ser traducido a **JavaScript** posteriormente.

- **app.component.spec.ts:** archivo usado para la realización de test unitarios.

Importante

Cree tantos componentes como pantallas necesite y compártalos entre módulos para reutilizarlos.

En ejercicios posteriores, veremos que, al generar una aplicación con determinadas utilidades (p. ej., **Angular-CLI**), se crea un componente por defecto con una serie de bloques como los que se han descrito, cuyo código es similar al siguiente:

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


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


export class AppComponent {
  title = '007 Componentes';
}
```

Veremos también cómo se crea cada uno de los archivos mencionados anteriormente con un contenido por defecto.

Recuerde que el objetivo del componente es gestionar la vista. Por tanto, si necesita escribir mucha lógica, es mejor plantearse elaborar un servicio donde ubicarla.

Cuando se analice el ciclo de vida de un componente, veremos que es posible realizar ciertas inicializaciones al crear el componente en el método **ngOninit**.

Un componente puede tener atributos de entrada y/o de salida para intercambiar información con otros componentes padres e hijos. Dichos atributos requerirán del uso de las directivas **@Input()** y **@Output().**

008 Metadata - definición

Importante

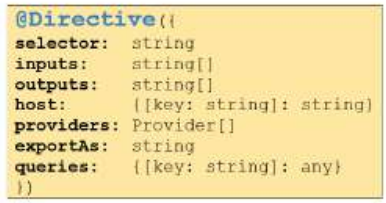
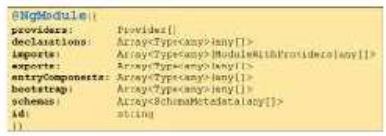
Cada decorador posee su metadata para definir el comportamiento del elemento asociado (módulo, componente, etc.).

Los metadatos permiten configurar el comportamiento de una clase a la que se le ha asociado un decorador. En función del decorador asociado, los metadatos varían y definen características diferentes. Por ejemplo, una clase no se trata como un componente hasta que no se le asocia el decorador **@Component** con su metadata necesaria. Los decoradores más comunes son **@Component** y **@Injectable** Como recordará, hablamos de metadatos cuando presentamos los módulos y componentes en los primeros ejercicios de este libro.

A continuación, se muestra una relación de propiedades asociadas a una serie de decoradores, así como el módulo que hay que importar para su utilización:

Decorador	Propiedades metadata	Módulo a importar
`@Input`	`bindingPropertyName: string`	`Input`
`@Output`	`bindingPropertyName: string`	`Output`
`@Attribute`	`attributeName: string`	`Attribute`
`@Hostlistener`	`args: string[]` `eventName: string`	`HostListener`
`@HostBinding`	`hostPropertyName: string`	`HostBinding`

`@Component`
`@Component({`
`changeDetection: ChangeDetectionStrategy`
`viewProviders: Provider[]`
`moduleId: string`
`templateUrl: string`
`template: string`
`styleUrls: string[]`
`styles: string[]`
`animations: any[]`
`encapsulation: ViewEncapsulation`
`interpolation: [string, string]`
`entryComponents: Array<Type<any>|any[]>`
`preserveWhitespaces: boolean`
`selector: string`
`inputs: string[]`
`outputs: string[]`
`host: {[key: string]: string}`
`providers: Provider[]`
`exportAs: string`
`queries: {[key: string]: any}`
`})`
	`animations: any[]`
	`changeDetection:`
	`ChangeDetectionStrategy`
	`encapsulation: ViewEncapsulation`
	`entryComponents:`
	`Array<Type<any>|any[]>`
	`interpolation: [string, string]`
	`moduleId: string`
	`styles: string[]`
	`styleUrls: string[]`
	`template: string`
	`templateUrl: string`
	`viewProviders: Provider[]`
		`Component`

@Directive 	exportAs: string host: {[key: string]: string} inputs: string[] outputs: string[] providers: Provider[] queries: {[key: string]: any} selector: string	Directive
@Host	-	Host
@Inject	token: any	Inject
@Injectable	-	Injectable
@NgModule 	bootstrap: Array<Type<any>|any[]> declarations: Array<Type<any>|any[]> entryComponents: Array<Type<any>|any[]> exports: Array<Type<any>|any[]> id: string imports: Array<Type<any> |ModuleWithProviders|any[]> providers: Provider[] schemas: Array<SchemaMetadata|any[]>	NgModule
@Optional	-	Optional
@Pipe	name: string pure: boolean	Pipe
@Self	-	Self
@SkipSelf	-	SkipSelf

En los próximos ejercicios veremos muchos ejemplos de metadata asociados a módulos, componentes y directivas. A continuación, describimos las propiedades más importantes:

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

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

```
@Directive({
  selector: '[destacar]'
})
```

- **Animations:** lista de animaciones del componente.
- **Bootstrap:** define los componentes que deberían tenerse en cuenta en el arranque cuando este módulo es Bootstrap.
- **ChangeDetection:** estrategia de detección de cambios utilizada por el componente.
- **Declarations:** lista de directivas/pipes pertenecientes al módulo.
- **Encapsulation**: estrategia de encapsulación de estilo usada por el componente.
- **EntryComponents:** lista de componentes que deben compilarse cuando se define este módulo.
- **ExportAs:** nombre usado para exportar la instancia del componente en una plantilla.
- **Exports:** lista de directivas, pipes y módulos que pueden usarse en otros componentes que tengan importado el módulo en curso.
- **Host:** mapa de propiedad de clase para enlaces de elementos de host para eventos, propiedades y atributos.
- **Id:** usado para identificar módulos en getModuleFactory.
- **Imports:** lista de módulos cuyas directivas/pipes exportadas pueden utilizarse en el módulo.
- **Inputs:** lista de nombres de propiedades de clase para enlazar datos como entradas de componentes.
- **Interpolation:** marcadores de interpolación personalizados utilizados en la plantilla del componente.
- **ModuleID:** ID del módulo ES/CommonJS del archivo donde se define el componente.
- **Outputs:** lista de nombres de propiedades de clase que exponen eventos de salida a los que otros pueden suscribirse.
- **PreserveWhitespaces:** permite mantener o eliminar espacios en blanco.
- **Providers:** define el conjunto de objetos inyectables disponibles en el módulo para este componente y sus hijos.

- **Queries:** consultas que pueden inyectarse en el componente.
- **Schemas:** usado para declarar elementos y propiedades que no son componentes ni directivas Angular.
- **Selector:** tag con el que se identifica el componente en una plantilla.
- **Styles:** estilos "en línea" que hay que aplicar a la vista del componente.
- **StyleUrls:** lista de URL con hojas de estilo que pueden aplicarse a la vista del componente.
- **Template:** plantilla "en línea" aplicada a la vista del componente.
- **TemplateUrl:** URL que apunta al archivo que contiene la vista del componente.
- **ViewProviders:** lista de proveedores disponibles para el componente e hijos.

009 Hola Mundo (Manual)

A continuación, mostraremos cómo fabricar una aplicación "mínima" que nos permita mostrar una página sencilla en la que se muestre el texto '**Hola Mundo (by TwoBy2)**'.

En sucesivos ejercicios veremos una forma más sencilla y eficiente de hacer lo mismo (usando Angular-Cli), pero consideramos interesante mostrar cómo hacer manualmente lo mismo para que podamos valorar mejor otras utilidades.

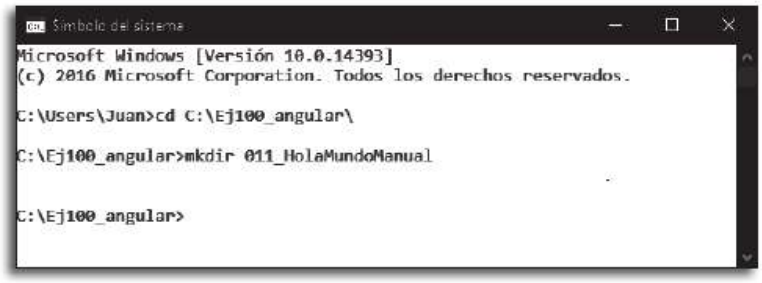
Importante

Instale Node.js y, una vez creados los ficheros mínimos descritos, instale las dependencias.

1. En primer lugar, instalaremos **Node.js** (si aún no lo hemos hecho). Por favor, revise el ejercicio **004** dedicado a la **instalación**.

2. Seguidamente, crearemos el directorio **009_HolaMundoManual**. Para ello, abriremos una ventana de **CMD** y teclearemos:

```
>cd C:\Ej100_angular

>mkdir 009_HolaMundoManual
```



3. A continuación, dentro de este directorio, crearemos los siguientes ficheros utilizando el editor que deseemos (p. ej., **Atom**):

- **tsconfig.json**: compilación utilizada por el compilador de **TypeScript**.

```
tsconfig.json
{
  "compilerOptions": {
    "target": "es5",
    "module": "system",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false
  },
  "exclude": [
    "node_modules",
    "typings/main",
    "typings/main.d.ts"
  ]
}
```

- **typings.json**: identifica definiciones de **TypeScript** que hagan falta.

```
typings.json
{
  "ambientDependencies": {
    "es6-shim": "github:DefinitelyTyped/DefinitelyTyped/es6-shim/es6-shim.d.ts#7de6c3dd94feaeb21f20054b9f30d5dabc5efabd",
    "jasmine": "github:DefinitelyTyped/DefinitelyTyped/jasmine/jasmine.d.ts#7de6c3dd94feaeb21f20054b9f30d5dabc5efabd"
  }
}
```

- **package.json**: permite definir las dependencias del proyecto.

```
package.json
{
  "name": "011_HolaMundo_Manual",
  "version": "1.0.0",
  "scripts": {
    "start": "tsc && concurrently \"npm run tsc:w\" \"npm run lite\" ",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "lite": "lite-server",
    "typings": "typings",
    "postinstall": "typings install"
  },
  "license": "ISC",
  "dependencies": {
    "angular2": "2.0.0-beta.14",
    "systemjs": "0.19.25",
    "es6-shim": "^0.35.0",
    "reflect-metadata": "0.1.2",
    "rxjs": "5.0.0-beta.2",
    "zone.js": "0.6.6"
  },
  "devDependencies": {
    "concurrently": "^2.0.0",
    "lite-server": "^2.2.0",
    "typescript": "^1.8.9",
    "typings":"^0.7.12"
  }
}
```

4. Seguidamente, instalaremos las dependencias del proyecto. Para ello, nos ubicaremos en el directorio **C:\Ej100_angular\009_HolaMundoManual** y ejecutaremos **npm install**:

```
>cd C:\Ej100_angular\009_HolaMundoManual

>npm install
```

5. Crearemos el directorio **app** (**C:\Ej100_angular\009_HolaMundoManual\app**):

```
>cd C:\Ej100_angular\009_HolaMundoManual

>mkdir app
```

6. Dentro del directorio app crearemos los siguientes ficheros:

- **app.component.ts**: permite definir un componente.
- **main.ts**: permite cargar el componente raíz lanzando el framework.

app.component.ts

```
1  import {Component} from 'angular2/core';
2
3  @Component({
4      selector: 'my-app',
5      template: '<h1>Hola Mundo (by TwoBy2)</h1>'
6  })
7  export class AppComponent { }
8
```

main.ts

```
1  import {bootstrap}    from 'angular2/platform/browser';
2  import {AppComponent} from '../app/app.component';
3
4  bootstrap(AppComponent);
5
```

7. El último fichero que se creará será el fichero **index.html** en el directorio raíz (**C:\Ej100_angular\009_HolaMundoManual\index.html**):

8. Finalmente, ejecutaremos **npm start** y observaremos el resultado en nuestro navegador.

010 Comandos básicos. Hola Mundo (Angular CLI)

Angular CLI es una herramienta que permite crear aplicaciones Angular desde el terminal apoyándose en plantillas de código que, además de normalizar los desarrollos, agilizan la fabricación de todo tipo de elementos como puede ser los componentes, los servicios y los pipes que veremos más adelante.

CLI significa ***command line interface*** (interfaz de línea de comando) y, como su nombre indica, permite lanzar comandos desde un terminal para crear elementos de Angular.

Entre las funcionalidades que obtendremos al usar Angular CLI destacan las siguientes:

- Herramientas para **testing**.
- Herramientas para **despliegue**.
- **Servidor HTTP** para lanzar el proyecto.
- Sistema **live-reload** que permite replicar de forma inmediata en el navegador cualquier cambio detectado en el proyecto.

Importante

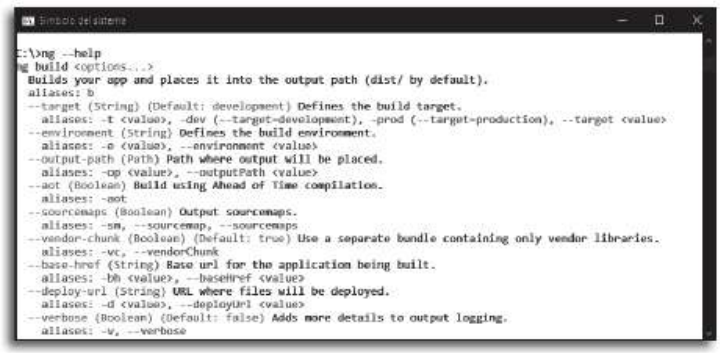
Utilice **Angular CLI** siempre que le sea posible porque apreciará el trabajo que la utilidad realiza por usted no solo creando elementos en base a templates, sino también incorporando referencias e **imports** en los ficheros de configuración que lo requieren.

En el capítulo de instalación, ya mencionamos cómo instalar Angular CLI, pero, a modo de recordatorio, adjuntamos la sentencia necesaria para la misma:

```
npm install -g angular-cli
```

La opción **-g** nos permite realizar la instalación de forma global, lo que significa que podemos usarla desde cualquier lugar. Al final de la instalación, observaremos que el cursor se coloca de nuevo en el prompt de la ventana **CMD**, esperando alguna acción.

A partir de este momento, podemos realizar una serie de acciones que se ejecutarán mediante el comando **ng** seguido de alguna palabra clave u opción.



Por ejemplo, podemos empezar a probar la ayuda con **ng -- help**.

Los comandos más importantes son los siguientes:

Comando	Descripción
help	Invoca a la ayuda.
new	Permite crear una nueva aplicación.
serve	Construye la aplicación y arranca el servidor web.
generate	Permite generar los siguientes elementos: class, component, directive, enum, guard, interface, module, pipe, service, etc.
lint	Permite usar tslint (herramienta que permite revisar el código durante su edición).
Test	Permite generar un directorio de salida para pruebas.
e2e	Sirve la aplicación y ejecuta pruebas.
build	Compila la aplicación en un directorio de salida.
get/set	Obtiene/establece un valor de la configuración.
doc	Abre la documentación oficial Angular para buscar información sobre la palabra indicada.

eject	Permite extraer el archivo de configuración de webpack.
xi18n	Extrae los mensajes i18n de las plantillas.

A continuación, vamos a crear un proyecto con Angular CLI para fabricar el clásico **Hola Mundo**.

1. Para ello, abriremos una ventana de **CMD** y nos situaremos en el directorio **C:\Ej100_angular** para teclear el siguiente comando:

```
C:\Ej100_angular>ng new HMAC
```

- Observaremos que, al final de la creación, aparece un mensaje que indica que el proyecto se ha creado satisfactoriamente.
- Seguidamente, abriremos el proyecto con nuestro editor (p. ej., en **Atom** bastará con arrastrar la carpeta del proyecto sobre el propio **Atom** y se abrirá el proyecto completo).
- Una vez abierto el proyecto, modificaremos el archivo **app.component.ts** ubicado en **HMAC/src/app** y a la variable **title** le asignaremos el valor '**Hola Mundo Angular CLI**'.
- Podemos modificar también el fichero **html** que se crea por defecto ubicado en **HMAC/src/app/app.component.html** para dejarlo simplemente con el siguiente contenido:

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

2. Una vez salvado el archivo, pasaremos a ejecutar la aplicación (utilizando el comando **serve**) accediendo al directorio del proyecto y tecleando lo siguiente:

```
C:\Ej100_angular>cd HMAC

C:\Ej100_angular\HMAC>ng serve
```

3. Observaremos que, tras una serie de mensajes, la aplicación arranca y recibimos un mensaje indicando que la compilación se ha realizado satisfactoriamente.

4. Por último, abriremos nuestro navegador y comprobaremos el resultado tras acceder al siguiente URL:

http://localhost:4200/

011 Elementos que se pueden crear con Angular CLI (Component, Directive, etc.)

Tal y como comentamos en el ejercicio anterior, una de las acciones que podemos realizar con Angular CLI es crear diferentes tipos de elementos e integrarlos con nuestra aplicación de forma automática. Para ello, usaremos el subcomando **generate** (ng generate o ng g en modo abreviado) seguido del tipo de elemento que deseamos crear y una serie de opciones.

La sintaxis general es:

```
ng generate tipo_elemento nombre_elemento
                    opciones
```

Es recomendable utilizar nombres sencillos, ya que los elementos que se crean automáticamente utilizan dicho nombre para añadirles alguna partícula o extensión según cada caso.

Importante

Siempre que pueda utilice **ng generate** para crear elementos ya que, además de crearlos con una plantilla, se insertan las referencias de forma automática, lo que facilita mucho el trabajo.

Una de las ventajas de utilizar ng es que la utilidad se encarga de incluir las referencias a los nuevos elementos donde sea necesario como, por ejemplo, incluir el import y las declarations del elemento en el fichero **app.module.ts** si procede.

Los distintos elementos que podemos crear son los siguientes:

Elemento	Abreviación	Comentario
Component	`ng g c`	`ng g component nomComponente` Es el principal elemento mediante el cual construimos los elementos y la lógica de la página.
Directive	`ng g d`	`ng g directive nomDirectiva` Permite añadir comportamiento dinámico a HTML (estructurales y de atributos).
Pipe	`ng g p`	`ng g pipe nomPipe` Genera una salida transformando un dato obtenido desde la entrada.

Service	`ng g s`	`ng g service nomServicio` Son clases complementarias a los componentes que permiten realizar cierta lógica o acciones como puede ser proporcionar datos a los componentes o realizar peticiones al servidor, etc.
Class	`ng g cl`	`ng g class nomClase` Añade una clase a la aplicación.
Guard		`ng g guard nomGuard` Añade funciones que permiten controlar o activar ciertas rutas.
Interface		`ng g interface nominterface` Añade interfaces a la aplicación (contratos que otras clases han de cumplir para utilizarlos).
Enum	`ng g e`	`ng g enum nomEnum` Genera una enumeración.
Module	`ng g m`	`ng module nomModulo` Elemento que permite agrupar componentes, servicios, directivas, etc., para crear una aplicación.

A continuación, mostramos algunas opciones:

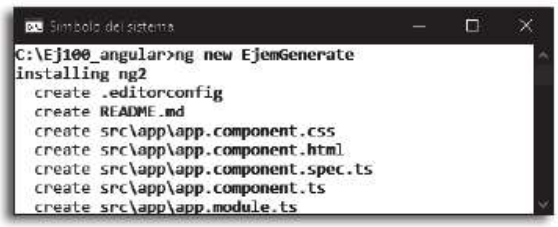
--flat	Permite crear el elemento sin crear un nuevo directorio.
--route=<route>	Indica la ruta principal. Solo se utiliza para generar componentes y rutas.
--skip-router-generation	No crea la configuración de la ruta. Solo se emplea para generar rutas.
--default	Indica que la ruta debe ser la predeterminada.
--lazy	Indica que la ruta es "lazy" (carga los componentes cuando se necesitan).
-is	**inline css.** Evita la creación de un fichero **css.**
-it	**inline html.** Evita la creación de un fichero **html.**

Para comprobar el funcionamiento de **generate**, simplemente crearemos una aplicación a la que le añadiremos un componente y al que no daremos ningún uso, pero que nos permitirá observar qué ficheros se crean durante el proceso.

1. En primer lugar, abriremos una ventana de **CMD** y nos situaremos en la carpeta **Ej100_angular**.

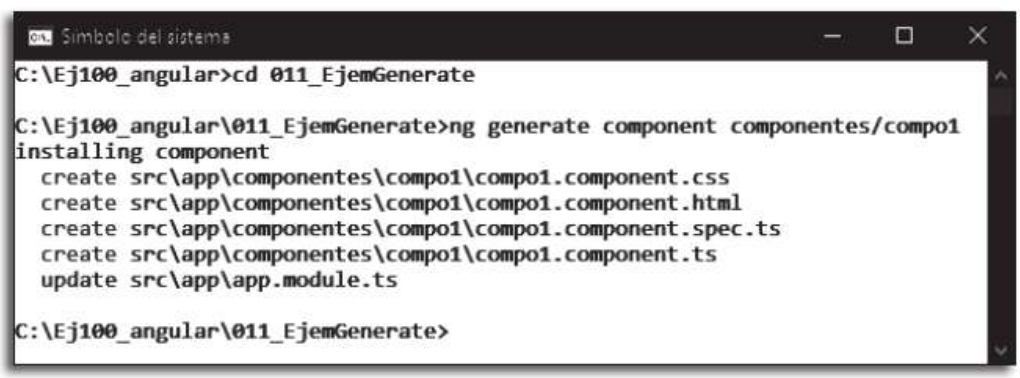
2. Seguidamente, crearemos una nueva aplicación con el comando **ng new** y la renombraremos para añadirle el número de ejercicio:

```
ng new EjemGenerate

rename EjemGenerate 011_EjemGenerate
```



3. A continuación, nos ubicaremos en el directorio recién creado para la nueva aplicación y crearemos un componente con la siguiente sintaxis:

```
ng generate component componentes/compo1
```

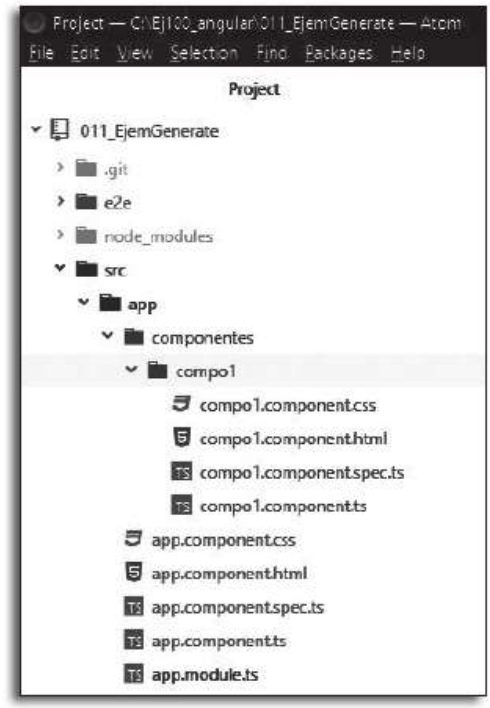


4. Podíamos haber utilizado la sintaxis abreviada:

```
ng g c componentes/compo1
```

5. Fíjese que al haber indicado **componentes/compo1**, nos ha creado de paso la carpeta componentes bajo el directorio **src/app** para que podamos almacenar todos los componentes que creemos en la misma carpeta, de forma que quede más organizado.

6. Si abre el proyecto en su editor (p. ej., Atom), observará los ficheros que se han creado por defecto.



7. Si analizamos el fichero **app.module.ts**, apreciaremos que se han añadido de forma automática el **import** del componente y su referencia en el apartado **declarations**.

app.module.ts

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { FormsModule } from '@angular/forms';
4  import { HttpModule } from '@angular/http';
5
6  import { AppComponent } from './app.component';
7  import { Compo1Component } from './componentes/compo1/compo1.component';
8
9  @NgModule({
10   declarations: [
11     AppComponent,
12     Compo1Component
13   ],
14   imports: [
15     BrowserModule,
16     FormsModule,
17     HttpModule
18   ],
19   providers: [],
20   bootstrap: [AppComponent]
21 })
22 export class AppModule { }
```

012 Descripción de un proyecto

Al crear un proyecto, por defecto se crean una serie de carpetas y ficheros que son la base de nuestra aplicación y que iremos completando según la funcionalidad que queramos dar a la misma.

Para analizar la estructura de una aplicación podemos utilizar alguna de las creadas en ejercicios anteriores como por ejemplo la que creamos para mostrar cómo utilizar **ng generate** para crear un componente (**011_EjemGenerate**).

Así pues, abrimos dicha aplicación con nuestro editor (p.e. **Atom**) y observamos la siguiente estructura:

Importante

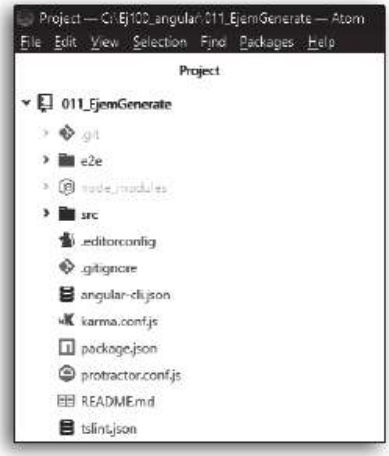
Mantenga organizada la estructura creando carpetas para almacenar los diferentes tipos de elementos que necesite. Por ejemplo, cree una carpeta de componentes para incluir **components**, otra para servicios y así sucesivamente.

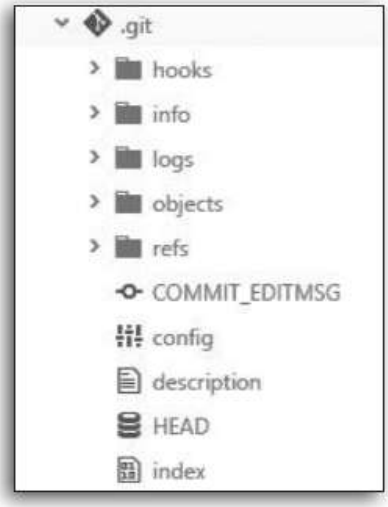
Elemento	Comentario
.git	Contiene los archivos necesarios para la gestión con GIT.
e2e	Carpeta que contiene los archivos de prueba, denominados end to end (de extremo a extremo) que se ejecutan con Jasmine.
node_modules	Contiene todos los paquetes y dependencias de Node.js de nuestro proyecto.
src	Carpeta que contiene los archivos fuentes raíz de la aplicación.
.editorconfig	Contiene la configuración de nuestro editor.
.gitignore	Permite indicar archivos que queremos ignorar de cara al control de versiones git.
angular-cli.json	Contiene la configuración de CLI.
karma.conf.js	Contiene la configuración para los test.
package.json	Describe las dependencias necesarias para ejecutar la aplicación.
protractor.conf.js	Configuración de test con Jasmine.

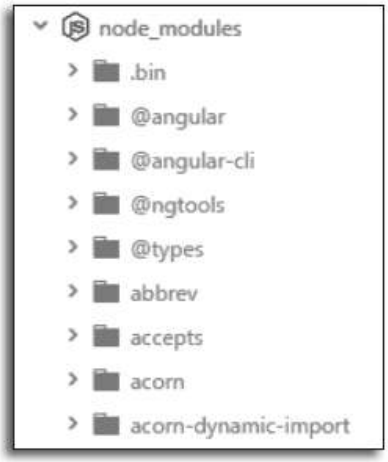
README.md	'Readme' clásico de cualquier proyecto compartido en GitHub.
tslint.json	Contiene la configuración del Linter para Typescript.

La carpeta **src** es una de las más importantes puesto que es donde iremos incluyendo nuestros componentes, servicios y resto de elementos de nuestra aplicación. Si la expandimos, podemos observar lo siguiente:

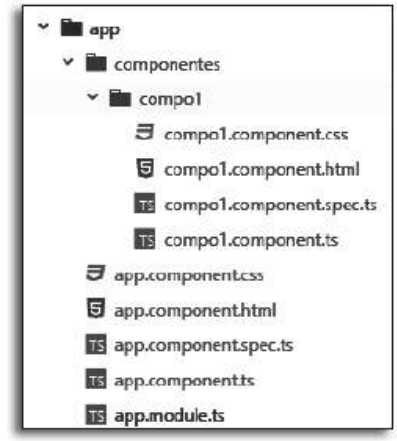
Elemento	Comentario
app	Carpeta raíz que contendrá los componentes, servicios y resto de elementos que constituyen nuestra aplicación.
assets	Imágenes, vídeos y archivos en general que no son propiedad de ningún componente.
environments	Contiene características relativas al entorno de trabajo.
favicon.icon	Imagen que permite identificar nuestra aplicación.
index.html	Archivo raíz donde se inicia la aplicación
main.ts	Es el primer archivo que se ejecuta y contiene todos los parámetros de la aplicación.
polyfills.ts	Asegura la compatibilidad con los navegadores.
styles.css	Estilos del proyecto.
test.ts	Puede contener test unitarios.
tsconfig.json	Configuración de TypeScript.






node_modules
.bin
@angular
@angular-cli
@ngtools
@types
abbrev
accepts
acorn
acorn-dynamic-import


Project — C:\Ej100_angular\011_EjemGenerate — Atom
File Edit View Selection Find Packages Help
Project
011_EjemGenerate
.git
e2e
node_modules
src
app
assets
environments
favicon.ico
index.html
main.ts
polyfills.ts
styles.css
test.ts
tsconfig.json


app
componentes
compo1
compo1.component.css
compo1.component.html
compo1.component.spec.ts
compo1.component.ts
app.component.css
app.component.html
app.component.spec.ts
app.component.ts
app.module.ts


assets
.gitkeep


environments
environment.prod.ts
environment.ts

013 Módulos: Creación

Los módulos son clases que nos permiten organizar nuestra aplicación encapsulando funcionalidades concretas dentro de los mismos. Son agrupadores de componentes, pipes, directivas, etc., relativos a una funcionalidad que permiten crear bloques reutilizables.

Utilizan el decorador **@NgModule** que permite añadir **metadata**.

Los metadatos contienen:

- **Declarations:** declaración de componentes, directivas, pipes, etc.
- **Exports:** exportación de clases para poderlas compartir con otros componentes.
- **Imports:** importación de otros módulos que ofrecen clases usadas en nuestro módulo.
- **Providers:** carga servicios para toda la aplicación y permite que estos se pasen al resto de componentes por inyección de dependencias.
- **Bootstrap:** define el componente principal para el arranque y se utiliza en el módulo principal **(root module)**.

Importante

Cuando la aplicación se amplía es mejor crear diferentes módulos para organizar mejor las funcionalidades y permitir que las cargas se realicen cuando realmente sean necesarias (lazy loading).

Como mínimo, siempre existe un módulo ya que, al crear una aplicación, se crea el módulo **root**, el cual permite que la aplicación se lance, pero este módulo lo analizaremos en detalle en el próximo ejercicio.

Existen módulos que, a su vez, son librerías de módulos, como ocurre con el propio Angular, que ofrece librerías asociadas a paquetes **npm** y que permiten realizar importaciones selectivas según nuestras necesidades. Por ejemplo, Angular posee librerías que son módulos como **FormsModule**, **HttpModule** o **RouterModule**.

Al igual que las clases, los módulos pueden exportar u ocultar componentes, servicios, etc.

Podemos crear módulos a mano, pero, por facilidad, recomendamos hacerlo con **Angular CLI** y, en este caso, los módulos se crean usando el comando **generate** de la siguiente manera:

```
ng generate module mimodulo
```

Al crear el módulo, por defecto suceden dos cosas:

- Se crea el directorio **src/app/mimodulo**.
- Se crea el archivo **src/app/mimodulo/mimodulo.module.ts**, con la definición del propio módulo **MimoduloModule** dentro.

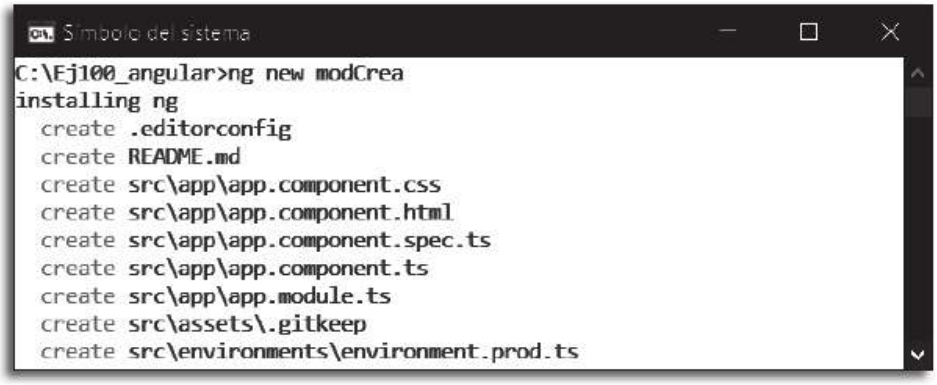
Las opciones disponibles son:

app	`--app (aliases: -a) default value: 1st app` Especifica el nombre o índice de la aplicación a utilizar. El índice hace referencia al orden de la app dentro del array asociado a "apps" en angular-cli.json. Por defecto, usa la primera app.
flat	`--flat` Si se indica –flat, no se crea un directorio.
module	`--module (aliases: -m)` Especifica dónde se debe importar el módulo. Por ejemplo, si estamos creando **mod3** y añadimos la opción **--module mod2**, se importaría el módulo recién creado en **mod2**.
spec	`--spec` Si se indica **–spec**, se crea el archivo **mimodulo.module.spec.ts**.
routing	`--routing` Si se indica **–routing**, se crea el archivo **mimodulo-routing.module.ts**. (archivo de módulo de enrutamiento).

A continuación, crearemos una aplicación para añadirle posteriormente un par de módulos y analizar la estructura asociada a los mismos.

1. En primer lugar, nos ubicaremos en **ej100_angular** y crearemos el proyecto **modCrea** mediante el comando **ng new**:

```
C:\Ej100_angular>ng new modCrea
```



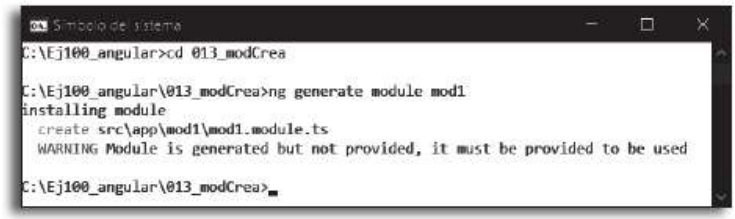
2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename modCrea 013_modCrea
```

3. Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso, usamos **Atom**) y observamos la estructura del proyecto poniendo el foco en la carpeta **app**.

4. A continuación, ubicados en **013_modCrea**, crearemos el módulo **mod1** tecleando lo siguiente :

```
C:\Ej100_angular>cd 013_modCrea

C:\Ej100_angular\013_modCrea>ng generate module mod1
```

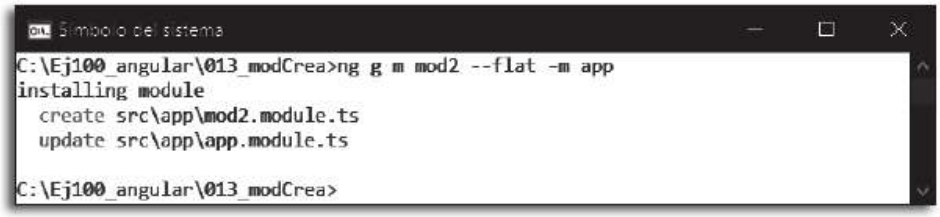


5. Si analizamos la estructura, vemos que dentro de **app** ha creado una carpeta denominada **mod1** y dentro de la misma, un archivo denominado **mod1.module.ts** con la clase **Mod1Module** y el decorador **@NgModule**.


```
mod1.module.ts
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class Mod1Module { }
```

6. Ahora, crearemos un segundo módulo (**mod2**) pero esta vez le diremos que no cree ningún directorio para el mismo y que, además, que una vez creado realice la importación en nuestro archivo **app.module.ts**. La sentencia que habrá que teclear será:

```
ng g m mod2 --flat -m app
```

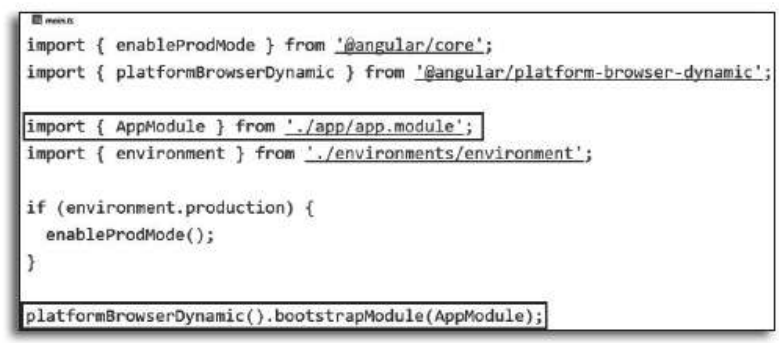


7. Tal y como habíamos dicho, con **–flat** indicamos que no se cree carpeta para el módulo y con **--m app** indicamos que realice la importación en el módulo **app**. Efectivamente, analizando la estructura de la app, vemos que no se ha creado un directorio para **mod2** y que se ha realizado el **import** de **Mod2Module** en **app.module.ts**.

014 Módulos: RootModule

Tal y como comentamos en el ejercicio anterior, al crear una aplicación con **Angular CLI**, se crea un módulo que por convenio se denomina **AppModule** y se almacena en el archivo **app.module.ts**, el cual denominamos módulo raíz y que es el que usa la aplicación en el arranque de la misma.

La aplicación se inicializa pasando el módulo **AppModule** como parámetro a **Bootstrap** en **main.ts**.



Existen 2 formas de hacer **Bootstrap**:

- **Dinámico:** se usa el compilador **JIT** (**Just-in-Time**) para compilar la aplicación en el navegador y luego inicializar la aplicación. Veamos cómo queda **main.ts** con esta opción:

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

import { AppModule } from './app.module';

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

- **Estático:** precompila nuestra aplicación antes de enviarla al navegador, con lo que se obtiene una aplicación más pequeña que se carga con rapidez. El tamaño es mucho menor y la velocidad de ejecución más rápida. En este caso, **main.ts** queda así:

```
import { platformBrowser } from '@angular/platform-
   browser';
import { AppModuleNgFactory } from './app.module.
   ngfactory';
platformBrowser().
   bootstrapModuleFactory(AppModuleNgFactory);
```

Ambas formas de compilación generan clases de tipo **AppModuleNgFactory**. Como vemos en nuestro caso, por defecto se usa el siguiente código:

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

Es decir, se usa la compilación dinámica (**JIT**) y arranca al **AppModule** descrito anteriormente.

Para el análisis de este módulo, utilizaremos el existente en la aplicación creada en el ejercicio anterior **(013_Modulos_Creacion)**.

1. Abrimos el proyecto **013_Modulos_Creacion** con nuestro editor (en nuestro caso, **Atom**) y localizamos el archivo **app.module.ts**.

2. Al analizar su contenido vemos que el archivo consta de una serie de **imports** como son **BrowserModule** y **NgModule** procedentes de las librerías de Angular y, **AppComponent** de nuestra propia aplicación. En nuestro ejemplo, importamos **Mod2Module** porque expresamente habíamos creado el módulo **mod2** con la opción **–m**. Vemos también

el decorador **@NgModule** seguido de su metadata y, por último, de la clase **AppModule**, la cual exporta para que sea visible por otros elementos de la aplicación. Al entrar un poco más en detalle, obtenemos:

Elemento	Comentario
Imports	Importa los elementos que se necesitan en el módulo.
NgModule	Decorador usado con los módulos en Angular y que define los metadatos que se van a utilizar en el módulo.
BrowserModule	Este módulo se importa solo en el root **AppModule** y proporciona servicios esenciales para lanzar y ejecutar la aplicación en un navegador.
Declarations	Define listas de clases **view** que se van a utilizar (en este caso, **AppComponent**). Angular utiliza 3 tipos de clases **view**: componentes, directivas y pipes.
Providers	Servicios accesibles desde todas las partes de la aplicación.
Exports	Define el subconjunto de declaraciones que estarán disponibles en las plantillas de componentes de otros módulos.
Bootstrap	Define el componente a llamar al inicializar la aplicación (p. ej., **AppComponent**). Cuando se lanza la aplicación, expande el template HTML de **AppComponent** en el **DOM**, dentro de las etiquetas del elemento <**app-root**> existentes en **index.html**. 

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

3. **Metadata**, en definitiva, indica a Angular cómo se debe compilar y ejecutar un módulo, además de relacionar los componentes, directivas y pipes incluidos en el mismo.

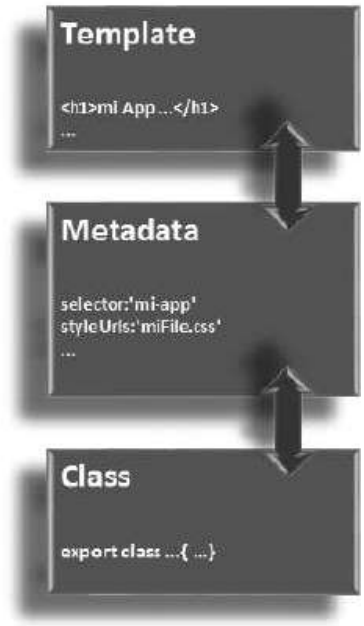
Importante

El arranque de la aplicación se realiza a través del **root module** que, por convenio, se denomina **AppModule** y se almacena en el archivo **app.module.ts**.

015 Componentes: Creación

Importante

Cree tantos componentes como necesite para que cada uno de ellos muestre la parte de información que se requiera en cada caso. Crearlos con **Angular CLI** simplifica mucho el trabajo.



El componente es el elemento básico del desarrollo en Angular. Una aplicación suele constar de una especie de árbol de componentes de varios niveles donde un componente puede llamar a sus componentes hijos y, a su vez, estos llamar a sus propios hijos (o nietos del primero), y así sucesivamente. Un componente, básicamente, es una clase acompañada del decorador **@Component** y se encarga de controlar lo que podríamos llamar **Vista** o zona de pantalla. A grosso modo, un componente está formado por un **template**, una **metadata** y una **clase**.

Según vimos en ejercicios anteriores, al crear una aplicación con **Angular CLI**, se crea un módulo por defecto y, dentro del mismo, se crea también un componente denominado **app.component**. El contenido por defecto del mismo es el siguiente:

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

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

La composición de un componente es la siguiente:

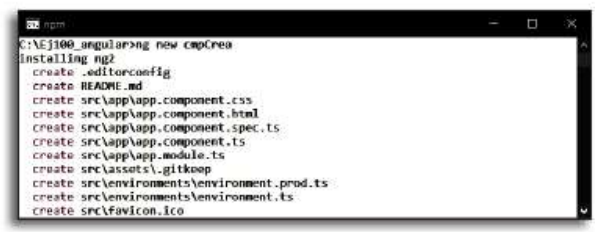
- **Import:** permite importar otros componentes a utilizarse dentro del componente que se está definiendo. Como mínimo, importamos **Component** de **@angular/core**.
- **Decorador:** función de tipo decorador que posee un objeto con los metadatos que indican a Angular cómo se crea, compila y ejecuta el componente.
- **Export class:** exporta la clase asociada al componente para hacerla visible a otros componentes, de forma que puedan usar sus propiedades y métodos dependiendo a su vez de la visibilidad establecida a los mismos. Hace las veces de controlador en la clásica arquitectura **MVC (Modelo-Vista-Controlador)**.

Los campos de metadata pueden ser los siguientes:

Campo	Comentario
Selector	Nombre que se utiliza para hacer referencia a nuestro componente (p. ej., <**mi-aplicacion**>)
Template	Contenido HTML de nuestro componente.
TemplateURl	Cuando el contenido es voluminoso, usamos un fichero externo para almacenar el template.
StyleUrls	Array que contiene las hojas de estilo que se van a utilizar.
Directives	Enumeración de las directivas que se van a emplear en el componente.
Providers	Enumeración de los **providers** que se van a utilizar.

En los siguientes ejercicios detallaremos los diferentes campos de tipo template y veremos cuándo es mejor usar un tipo u otro según cada caso. En el siguiente ejercicio, además de crear un proyecto y ver cómo crea el componente por defecto, crearemos otro componente adicional para ver cómo se hace con **Angular CLI** y cómo podemos utilizarlo fácilmente en nuestra aplicación.

1. En primer lugar, nos ubicaremos en **ej100_angular** y crearemos un proyecto denominado **cmpCrea** mediante `ng new`.



2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, simplemente escribiremos lo siguiente:

```
C:\Ej100_angular>rename cmpCrea 015_cmpCrea
```

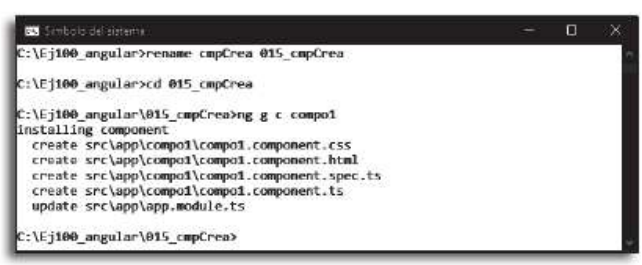
3. Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso, usamos **Atom**) y observamos cómo ha quedado nuestro componente por defecto.

4. A continuación, nos ubicaremos de nuevo en nuestro directorio **015_cmpCrea** y crearemos nuestro nuevo componente **compo1** tecleando lo siguiente (**`ng generate component compo1`** o de forma abreviada):

```
ng g c compo1
```

5. Observaremos el resultado de la creación en la ventana de **CMD** y también la estructura que ha creado en el proyecto. Seguidamente, modificamos **app.component.html** para añadir el tag **<app-compo1>** y arrancamos nuestra app para ver cómo aparece en el navegador.
Para ello, en la ventana de **CMD** teclearemos:

```
C:\Ej100_angular\015_cmpCrea>ng serve
```

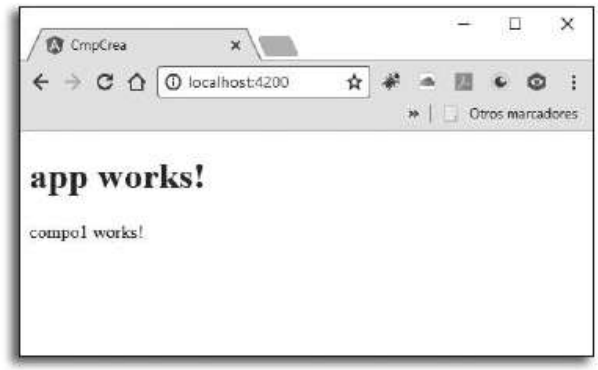


app.component.html

```
1  <h1>
2    {{title}}
3  </h1>
4  <app-compo1>
5
```

```
C:\Ej100_angular\015_cmpCrea>ng serve
** NG Live Development Server is running on http://localhost:4200. **
Hash: 8d5074d156663df9cba8
Time: 12762ms
chunk    {0} polyfills.bundle.js, polyfills.bundle.map (polyfills) 232 kB {4} [initial] [rendered]
chunk    {1} main.bundle.js, main.bundle.map (main) 5.56 kB {3} [initial] [rendered]
chunk    {2} styles.bundle.js, styles.bundle.map (styles) 9.71 kB {4} [initial] [rendered]
chunk    {3} vendor.bundle.js, vendor.bundle.map (vendor) 2.63 MB [initial] [rendered]
chunk    {4} inline.bundle.js, inline.bundle.map (inline) 0 bytes [entry] [rendered]
webpack: Compiled successfully.
```

6. En el navegador, veremos cómo nuestra aplicación, además de mostrar el contenido de app.component.html (**app Works!**), muestra también el contenido por defecto de **compo1.component.html** (compo1 works!).



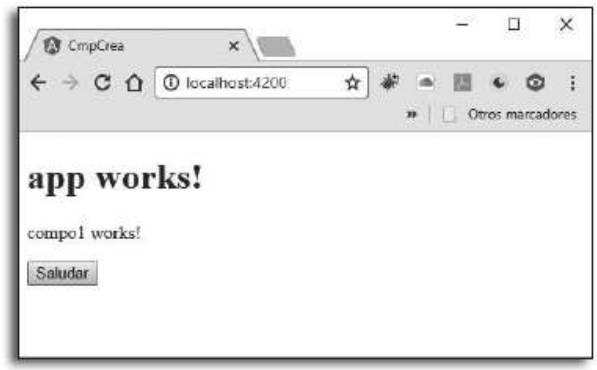
Vamos a hacer una pequeña modificación en nuestro componente para darle un poco de funcionalidad y, para ello, modificaremos el fichero **compo1.component.html** para que tenga el siguiente contenido:

```
<p>   compo1 works!</p>

<div >

  <button (click)="saludar()">Saludar</button><br>

  <h1>{{ texto }}</h1>

</div>
```

7. Observamos que simplemente hemos incluido un botón al que le hemos asociado un evento (**click**), el cual invocará a la función **saludar()** cuando pulsemos sobre el mismo. Estos conceptos se explicarán con más detalle en ejercicios posteriores. Modificaremos también al archivo **compo1.component.ts** para que la clase que se ha creado por defecto **Compo1Component** tenga lo siguiente:

```
export class Compo1Component {

  texto: string;

  saludar() { this.texto = "Hola Mundo en compo1";  }

}
```

8. Vemos que en la clase simplemente se ha añadido la variable texto de tipo **string** y una función llamada **saludar()** que se invocará cuando pulsemos el botón que hemos añadido en nuestra vista. Mientras que la aplicación esté en marcha, cada modificación que realicemos se actualizará sobre la marcha en el propio navegador así que, en este punto, deberíamos ver nuestro botón **Saludar**.



9. Si pulsamos sobre el mismo, observaremos cómo nos saluda.

016 Componentes: Template inline

El Template es la parte del componente que permite diseñar la vista utilizando HTML y otros elementos como por ejemplo las directivas que veremos en ejercicios posteriores.

Dicho template puede ser definido dentro del propio componente mediante el campo **template** situado dentro de la metadata del componente o bien, en el campo **templateUrl** mediante el cual definiríamos un archivo externo al que se haríamos referencia mediante su **URL**.

Importante

Utilice este método cuando el **HTML** a definir sea realmente pequeño. En general, es mejor usar **templateUrl** para tener mejor organizado cada bloque de código.

En el siguiente ejercicio, crearemos un componente al que definiremos su **template inline** usando la opción **--inline-template**.

1. En primer lugar, nos ubicaremos en **ej100_angular** y crearemos un proyecto denominado **cmpInline** mediante **ng new**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, simplemente escribiremos lo siguiente:

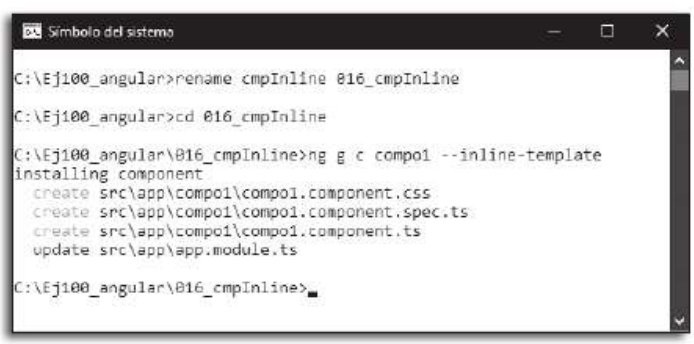
```
C:\Ej100_angular>rename cmpInline 016_cmpInline
```

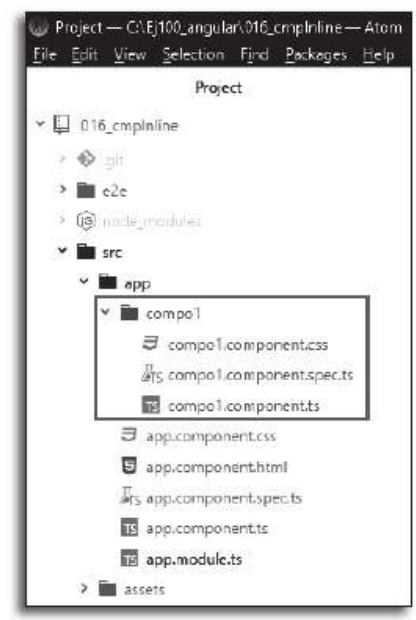
Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso usamos **Atom**) y observamos como ha quedado nuestro componente (por defecto) **app/app.component**.

3. A continuación, nos ubicaremos de nuevo en nuestro directorio **016_cmpInline** y crearemos nuestro nuevo componente **compo1** tecleando lo siguiente (**ng generate component compo1 --inline-template** o de forma abreviada):

```
C:\Ej100_angular\016_cmpInline>ng g c compo1 --inline-template
```

Observaremos el resultado de la creación en la ventana de **CMD** y también, la estructura que ha creado en el proyecto. En ambas observaciones, podemos comprobar que el archivo **compo1.component.html** no se ha creado. Sin embargo, si abrimos nuestro archivo **compo1.component.ts,** observaremos que el contenido **HTML** se ha descrito en el campo **template** dentro de la metadata entre 2 acentos de tipo 'grave'(` , el que se halla en el corchete izquierdo):





```
@Component({
  selector: 'app-compo1',
  template: `
    <p>
      compo1 Works!
    </p>
  `,
  styleUrls: ['./compo1.component.css']
})
```

Seguidamente, modificamos **app.component.html** para añadir los tags **<app-compo1></app-compo1>** y así mostrar nuestro **compo1** y también, arrancamos nuestra app para ver cómo aparece en el navegador.

Para ello, en la ventana de **CMD** teclearemos:

```
C:\Ej100_angular\016_cmpInline>ng serve
```

Si abrimos nuestro navegador y tecleamos http://localhost:4200/, veremos cómo se muestra nuestra aplicación, mostrando el contenido de **app.component.html** (**app Works!**) y también, el contenido HTML creado por defecto en nuestro **compo1.component.ts** (compo1 works!).

4. Igual que hicimos en el ejercicio anterior, añadimos un poco de funcionalidad modificando el HTML de compi1 con el siguiente contenido:

```
  <p>compo1 works!</p>
  <div>
    <button (click)="saludar()">Saludar</button><br>
    <h1>{{ texto }}</h1>
  </div>
```

Modificaremos también al archivo **compo1.component.ts** para que la clase que se ha creado por defecto **Compo1Component** tenga lo siguiente:

```
export class Compo1Component {
  texto: string;
  saludar() { this.texto = "Hola Mundo en compo1";  }
}
```

Si pulsamos sobre el botón Saludar, observaremos cómo nos saluda.

017 Componentes: Styles inline

De forma similar a la descrita en el ejercicio anterior, la parte relacionada con **CSS** (lenguaje usado para definir la presentación de los documentos **HTML** o **XML**) también puede definirse dentro del propio componente usando el campo **style** de los metadatos, o bien mediante el empleo de una referencia a un fichero con dicha definición si usamos el campo **styleUrls** (el cual puede ser una colección, ya que es de tipo array).

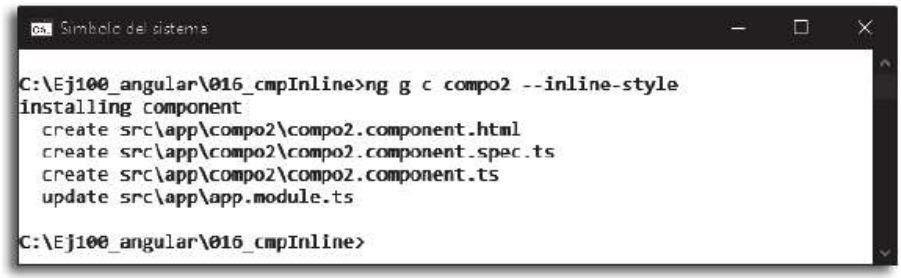
Importante

Utilice este método cuando el **CSS** que se desee definir sea realmente pequeño y no quiera definirlo en un archivo específico asociado al componente. En general, es mejor usar **styleUrls** para tener mejor organizado cada bloque de código.

En el siguiente ejercicio, añadiremos un componente al proyecto que creamos en el ejercicio anterior (016_cmpInline) al que definiremos su **style** mediante la opción **--inline-style**.

1. En primer lugar, nos ubicaremos en **ej100_angular\016_cmpInline** y crearemos un nuevo componente denominado **compo2** tecleando lo siguiente (**`ng generate component compo2 ----inline-style`** o de forma abreviada):

```
C:\Ej100_angular\016_cmpInline>ng g c compo2 --inline-style
```



Observaremos el resultado de la creación en la ventana de **CMD** y también la estructura que ha creado en el proyecto. En ambos casos, podemos comprobar que no se ha creado el archivo **compo2.component.css**. Sin embargo, si abrimos nuestro archivo **compo2.component.ts**, veremos que ha creado un campo vacío (**styles**) dentro de **metadata**, en el cual podremos definir nuestro contenido **CSS** entre dos acentos de tipo grave (`, el que se halla en el corchete izquierdo):

```
@Component({
  selector: 'app-compo2',
  templateUrl: './compo2.component.html',
  styles: []
})
```

Seguidamente, modificamos **app.component.html** para añadir los tags **<app-compo2></app-compo2>** y así mostrar nuestro **compo2**, y también arrancamos nuestra **app** para ver cómo aparece en el navegador.

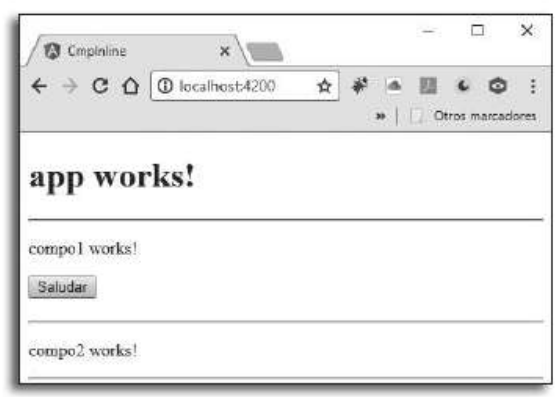
app.component.html

```
<h1>
  {{title}}
</h1>
<hr>
<app-compo1></app-compo1>
<hr>
<app-compo2></app-compo2>
<hr>
```

```
C:\Ej100_angular\016_cmpInline>ng serve
** NG Live Development Server is running on http://localhost:4200. **
Hash: f1097ibce6a43972d05e
Time: 11455ms
chunk    {0} polyfills.bundle.js, polyfills.bundle.map (polyfills) 232 kB {4} [initial] [rendered]
chunk    {1} main.bundle.js, main.bundle.map (main) 7.17 kB {3} [initial] [rendered]
chunk    {2} styles.bundle.js, styles.bundle.map (styles) 9.71 kB {4} [initial] [rendered]
chunk    {3} vendor.bundle.js, vendor.bundle.map (vendor) 2.63 MB [initial] [rendered]
chunk    {4} inline.bundle.js, inline.bundle.map (inline) 0 bytes [entry] [rendered]
webpack: Compiled successfully.
```

```
C:\Ej100_angular\016_cmpInline>ng serve
```

2. Si abrimos nuestro navegador y tecleamos http://localhost:4200/, veremos cómo se muestra nuestra aplicación, mostrando el contenido de **app.component.html** (**app Works!**), el contenido del **compo1** añadido en el ejercicio anterior (**compo1 works!** y su botón **Saludar**) y también el contenido **HTML** creado por defecto en nuestro **compo2.component.html** (**compo2 works!**).



3. Ahora, vamos a hacer una pequeña modificación en el **CSS** de **compo2** para definir algunas características de presentación. Por ejemplo, haremos que

los tags de tipo **<p>** muestren un tamaño de letra grande y que se aplique un fondo amarillo. También haremos que el texto comprendido entre los tags **<h2>** se muestre en azul. Así pues, los metadatos de nuestro **compo2** quedarán de la siguiente manera:

```
@Component({
  selector: 'app-compo2',
  templateUrl: './compo2.component.html',
  styles: [`
    p { font-size: xx-large ; background-color: yellow; }
    h2 { color: blue; }
    `]
})
```

Seguidamente, añadimos un pequeño texto en el archivo **compo2.component.html** para poder comprobar cómo se visualizan los tags **<h2>**. Para ello, modificaremos **dicho archivo** para que quede de la siguiente manera:

```
<p>
  compo2 works!
</p>
<h2>Texto en compo2</h2>
```

4. Por último, si observamos nuestro navegador, veremos que se muestra el contenido de los 3 componentes (app, compo1 y compo2) y, concretamente, en nuestro último componente apreciaremos las características de presentación definidas **inline**.

018 Componentes: Propiedades

En un componente, podemos definir propiedades que utilizaremos para mostrar información en las vistas o para condicionar el aspecto de las mismas. Para indicar al **DOM** que una de sus propiedades está ligada a una propiedad de nuestro componente, usaremos los corchetes **[ ]**, lo que denominaremos **binding de propiedades (property binding)**. En posteriores ejercicios se abordará el **DataBinding** con más profundidad. Podemos también hacer un **binding** con las clases **CSS** para condicionar un estilo al valor de una propiedad.

En el siguiente ejercicio, mostraremos unos cuantos ejemplos de cómo ligar algunas propiedades del **DOM** a propiedades de nuestro componente permitiendo así la visualización o no de un bloque, habilitar o deshabilitar un botón, definir un link a partir de un valor o de aplicar una clase **CSS** dependiendo del valor de una propiedad.

1. En primer lugar, nos ubicaremos en **ej100_angular** y crearemos el proyecto **cmpProp** mediante el comando **ng new**:

```
C:\Ej100_angular>ng new cmpProp
```

Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename cmpProp 018_cmpProp
```

A continuación, abrimos nuestro proyecto con nuestro editor (en nuestro caso, usamos **Atom**) y modificamos el archivo **styles.css** añadiendo una definición para las clases **fondo** (para **div**), **td** (para celdas de la tabla que incluiremos) y **tamB** para el tamaño de los botones:

```
.fondo { background: yellow; }
td { text-align: center;
     vertical-align: middle; }
.tamB { height:60px; width:200px }
```

Seguidamente, modificaremos nuestro archivo **app/app.component.ts** para que la clase **AppComponent** quede de la siguiente manera:

```
export class AppComponent {
  title = 'app works!';
  ponFondo = true;
  mostrar = true;
  habilitar = false;
  referencia = "http://www.google.com";
}
```

A continuación, modificaremos el fichero **app.component.html** para que tenga el siguiente contenido:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
    bootstrap/4.0.0-alpha.6/css/bootstrap.min.css" integrity="sha384-
    rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNUwEyJ"
    crossorigin="anonymous">
<h1>{{title}}</h1>
<div class="container">
 <table class="table table-bordered table-hover">
     <tr>
         <td><input type="checkbox" name="ponFondo"
    [(ngModel)]="ponFondo">Fondo<br><br></td>
         <td><div [class.fondo]="ponFondo">Hola</div></td>
     </tr>
    <tr>
         <td colspan="2"><a [href]="referencia" target="_blank">{{
    referencia }}</a></td>
     </tr>
```

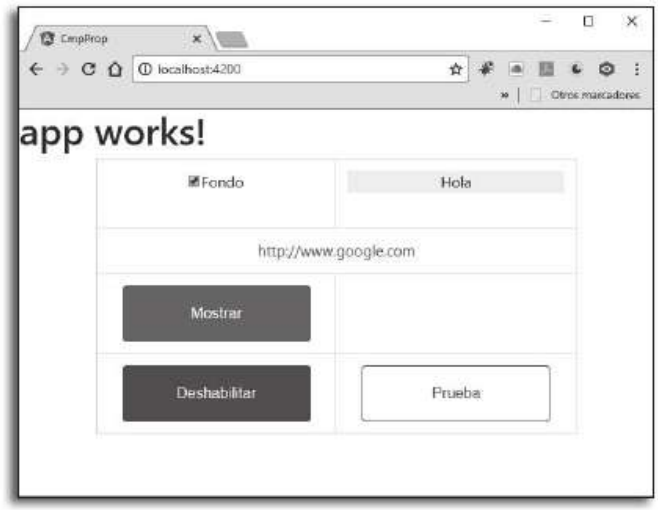
```
      <tr>
          <td><button class="btn btn-primary btn-success tamB"
    (click)="mostrar=!mostrar"><span>{{ mostrar ? 'Mostrar' : 'Ocultar'
    }}</span></button></td>
          <td><div [hidden]="mostrar"><h2>Se ve</h2></div></td>
      </tr>
      <tr>
          <td><button class="btn btn-primary tamB"
    (click)="habilitar=!habilitar">{{ habilitar ? 'Habilitar' :
    'Deshabilitar' }}</button></td>
          <td><button [disabled]="habilitar" class="btn btn-outline-
    primary tamB">Prueba</button></td>
      </tr>
 </table>
</div>
```

Observe que hemos añadido la siguiente referencia para poder utilizar **Bootstrap** y, de esta forma, añadir clases y funcionalidades a la aplicación:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
    bootstrap/4.0.0-alpha.6/css/bootstrap.min.css" integrity="sha384-
    rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNUwEyJ"
     crossorigin="anonymous">
```

Por último, ejecutamos la aplicación y vemos cómo se muestra en el navegador. Para ello, nos ubicamos en nuestro el directorio de la aplicación y tecleamos **ng serve**:

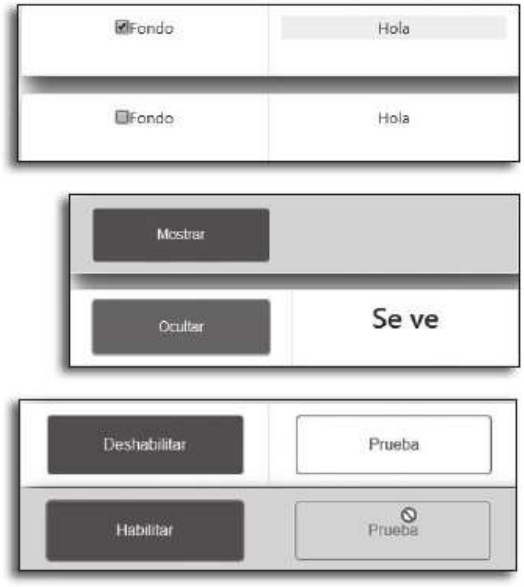
```
C:\Ej100_angular>cd 018_cmpProp
C:\Ej100_angular\018_cmpProp>ng serve
```



Una vez arrancada la aplicación, podemos marcar y desmarcar el check **Fondo** para ver cómo se aplica el fondo según tengamos marcado el check o no, podemos hacer clic sobre el link que en el componente definimos como http://www.google.com y veremos cómo se abre una pestaña con dicho URL, podemos pulsar sobre el botón **Mostrar** para exponer o no un determinado bloque o sobre el botón **Deshabilitar** para habilitar o no el botón con la etiqueta **Prueba**.

Importante

Podemos ligar propiedades de la vista con las de nuestro componente mediante el empleo de property binding.



019 Componentes: Test Unitarios

En general, los test son una parte muy importante del desarrollo, ya que son los que han de garantizar el buen funcionamiento de nuestras aplicaciones y, por tanto, evitar las posibles incidencias que pudieran aparecer una vez que el software esté en producción. Existen diversos tipos de test, pero los desarrolladores, en principio, deberían aportar sus test unitarios como parte de la documentación de sus desarrollos, demostrando el buen funcionamiento de los mismos. Es tal la importancia que se otorga a los test, que existe una metodología de desarrollo denominada **TDD** (**test-driven development**) o **desarrollo orientado o dirigido por pruebas**, que consiste en desarrollar inicialmente los test que han de superar los desarrollos y, a continuación, escribir el código que supere dichos test.

Importante

Testee sus aplicaciones para garantizar la calidad de las mismas y demostrar un nivel de calidad a sus clientes documentando los resultados.

Angular nos permite realizar test fácilmente ya que, por defecto, utiliza **Jasmine** (framework que permite probar código JavaScript) y **Karma** ("test runner" que permite automatizar algunas tareas de los frames de test). Es decir, sin configurar nada especialmente ya podríamos ejecutar un test justo después de crear un proyecto simplemente ejecutando **npm test** o **ng test**. La descripción de los test se apoya en ficheros denominados **spec**, que contienen un **describe** de las pruebas que queremos implementar y un **expect()** por cada una de las operaciones y resultados esperados de dichas pruebas. Incluye una API que permite realizar multitud de comparaciones (**toEqual**, **toBe**, **toContain**, etc.) y que puede consultar en https://www.npmjs.com/package/jasmine-expect.

En el siguiente ejercicio, elaboraremos un proyecto en el que incluiremos una clase para realizar una serie de operaciones matemáticas simples (**sumar**, **restar**, **multiplicar** y **dividir**) y a la que someteremos a un test para verificar qué parte de la misma está probada, falta por probar o puede contener un posible error. A pesar de que el tema de los test es amplio, consideramos que con este ejercicio podemos realizar una introducción suficiente a este apasionante apartado de la programación.

1. En primer lugar, nos ubicaremos en **ej100_angular** y crearemos el proyecto **cmpTest** mediante el comando **ng new**:

```
C:\Ej100_angular>ng new cmpTest
```

```
export class Operaciones {
  sumar(num1: number, num2: number): number{
    return num1 + num2;
  }
  restar(num1: number, num2: number): number{
    return num1 - num2;
  }
  multiplicar(num1: number, num2: number): number{
    return num1 * num2;
  }
  dividir(num1: number, num2: number): number{
    return num1 / num2;
  }
}
```

Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename cmpTest 019_cmpTest
```

A continuación, crearemos la clase **operaciones** usando **Angular CLI**. Para ello, nos ubicaremos en el directorio de la nueva aplicación y tecleare-mos lo siguiente:

```
C:\Ej100_angular>cd 019_cmpTest

C:\Ej100_angular\019_cmpTest>ng generate class operaciones
```

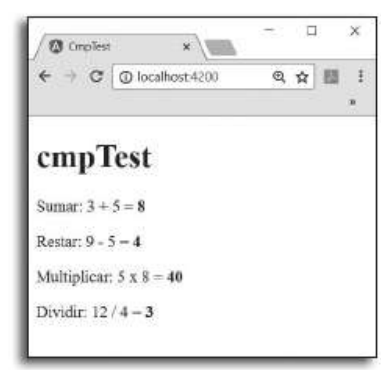
Ahora, abrimos nuestra aplicación con el editor que usemos habitualmente (p. ej., **Atom**) y abrimos la nueva clase **operaciones** para definir las operaciones que queremos implementar.

```
import { Component } from '@angular/core';
import { Operaciones } from './operaciones';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'cmpTest';
  res1:number;
  res2:number;
  res3:number;
  res4:number;
  ngOnInit(){
    let operaciones = new Operaciones();
    this.res1=operaciones.sumar(3,5);
    this.res2=operaciones.restar(9,5);
    this.res3=operaciones.multiplicar(5,8);
    this.res4=operaciones.dividir(12,4);
  }
}
```

2. Seguidamente modificaremos la clase **AppComponent** del fichero **app.component.ts** para importar la clase **Operaciones** y para que contenga la función **ngOnInit()** con una muestra de ejecución de las operaciones definidas en la clase anterior.

3. Vamos a modificar el fichero **app.component.html** para que contenga simplemente el título de la aplicación y algunos resultados tras el uso de nuestra clase.

```
<div>
  <h1>
    {{title}}
  </h1>
  <p>Sumar:  3 + 5 = <b>{{ res1 }}</b></p>
  <p>Restar:  9 - 5 = <b>{{ res2 }}</b></p>
  <p>Multiplicar:  5 x 8 = <b>{{ res3 }}</b></p>
  <p>Dividir: 12 / 4 = <b>{{ res4 }}</b></p>
</div>
```



4. Guardamos todo y arrancamos la aplicación con **ng serve** ubicándonos en **C:\Ej100_angular\019_cmpTest**.

5. A continuación, borraremos el fichero **app.component.spec** que se generó por defecto al crear la aplicación y, dentro de la carpeta **src/app**, crearemos el fichero **operaciones.spec.ts** donde definiremos nuestros test con la estructura **describre-it-expect**.

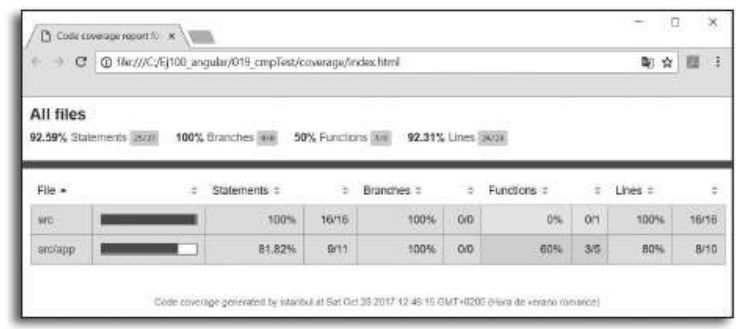
```
import { Operaciones } from './operaciones';
describe('Test para operaciones', () => {
  describe('Test para sumar y restar', () => {
    it("Suma 2 numeros", ()=>{
      let operaciones = new Operaciones();
      expect(operaciones.sumar(3,5)).toEqual(8);
    });
    it("Resta 2 numeros", ()=>{
      let operaciones = new Operaciones();
      expect(operaciones.restar(9,5)).toEqual(4);
    });
  });
});
```

6. Seguidamente, vamos a abrir otra sesión de **CMD** y, desde **C:\Ej100_angular\019_cmpTest**, ejecutaremos **ng test** para ejecutar todos los archivos de tipo **spec** que tengamos definidos. En este punto, observaremos que ya nos indica que se han ejecutado dos pruebas con éxito (**Executed 2 of 2 SUCCESS**) y que, al mismo tiempo, se ha abierto una sesión de Chrome (navegador por defecto en la configuración que puede cambiarse) con el resultado de las pruebas que se han lanzado. En esta pantalla, vemos los textos que hemos incluido en **describe** y también que, de los dos resultados esperados, ha habido 0 fallos (**2 specs**, **0 failures**).

7. Ahora, detenemos **ng test** pulsando **CTRL+C** y contestando **S** a la pregunta ¿Desea terminar el trabajo por lotes (S/N)? Una vez detenido, volveremos a lanzar el test con la opción **–code-coverage**:

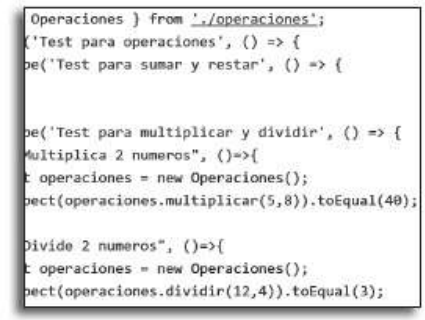
```
C:\Ej100_angular>ng test –code-coverage
```

Aparentemente, el resultado es el mismo que hemos comentado anteriormente, pero si observamos la carpeta del proyecto, veremos que se ha creado dentro de otra carpeta denominada **coverage**, donde se almacenan los elementos que nos permitirán consultar qué parte del código se ha cubierto. Si pulsamos sobre el fichero **index.html**, se muestra una página en la que puede analizarse cuál es el **porcentaje** de la aplicación que tenemos cubierto.



8. Si pulsamos sobre **src/app** para analizar qué parte nos falta, aparecerá otra pantalla que mostrará la clase operaciones que está incompleta respecto de las pruebas y, de nuevo, si pulsamos sobre operaciones, veremos cómo nos destaca el código pendiente de cubrir (**multiplicar** y **dividir**).

9. Vamos a completar los test que nos faltan añadiendo al fichero **operaciones.spec.ts** lo siguiente justo después del describe que tenemos para **sumar** y **restar**.



10. Si relanzamos de nuevo el test con la opción **–code-coverage**, veremos que ahora aparecen dos nuevos test y si refrescamos la página **index.html**, observaremos que ahora ya tenemos el **100%** de nuestra clase cubierto.

11. Para acabar, vamos a provocar un error para mostrar cómo se avisa el fallo de un test. Provoquemos el absurdo error de indicar que al sumar **3+5** el resultado es **9**:

```
expect(operaciones.sumar(3,5))
.toEqual(9)
```

Al relanzar de nuevo el test **(ng test)** veríamos cómo aparece un fallo **(1 failure)** indicando dónde se ha encontrado **(Test para operaciones Test para sumar y restar Suma 2 numeros)** y cuál era el valor esperado **(Expected 8 to equal 9)**.

020 Decoradores

Un decorador define qué partes posee un elemento (componente, directiva, etc.) y permite extender la funcionalidad de una función empleando otra función. Mediante los decoradores, los componentes quedan registrados y pueden utilizarse en otras partes de la aplicación gracias a la información añadida. En definitiva, el decorador es una implementación del patrón de diseño **decorator**, el cual permite añadir funcionalidad a un objeto dinámicamente evitando tener que crear un conjunto de clases que hereden de la inicial simplemente para añadir funcionalidad adicional. Existen cuatro tipos principales de decoradores:

Importante

La inmensa mayoría de los decoradores de uso frecuente nos los encontramos tras la creación de componentes, directivas, pipes, etc., que creamos con Angular CLI.

- De clase (p.e. @Component and @NgModule).
- De propiedades (en las clases, p. ej., @Input and @Output).
- De métodos (p.e. @HostListener).
- De parámetros (en el constructor, p. ej., @Inject).

Los nombres de los decoradores empiezan con el símbolo **@** seguido de un nombre que indica el elemento a decorar. Un ejemplo típico es el decorador **@Component**, el cual se utiliza para decorar componentes y puede emplearse siempre que importemos previamente la clase **Component** de **@angular/core**. Al crear un componente con **Angular CLI**, vimos cómo dicha sentencia se incluía de forma automática al inicio del archivo **nombreCompo.component.ts**. También hemos visto en ejercicios anteriores que, junto al decorador, se halla una metadata y que, en función del elemento a decorar, la misma varía. A continuación, mostramos una lista de decoradores de Angular y el módulo que se ha de importar desde **@angular/core** mediante la sentencia:

```
import { <MODULO> } from '@angular/core';
```

Por favor, consulte la documentación en **https://angular.io/** para obtener más detalles.

Decorador	Uso	Módulo a importar
@Input	Permite que los componentes reciban datos de un componente padre.	Input
@Output	Permite que los componentes envíen datos a su padre.	Output
@Attribute	Recupera el valor de un atributo disponible en el elemento host de este componente.	Attribute
@Hostlistener	Escucha el evento emitido por el host e invoca el método decorado.	HostListener
@HostBinding	Actualiza el elemento host si un enlace de propiedad cambia.	HostBinding
@Component	Indica cómo procesar, instanciar y usar un componente en ejecución.	Component
@Directive	Añade comportamiento a los elementos del DOM.	Directive
@Host	DI busca una dependencia en cualquier inyector hasta llegar al host.	Host
@Inject	Especifica una dependencia.	Inject
@Injectable	Indica a Angular que la clase puede ser usada con DI.	Injectable
@NgModule	Permite decorar módulos.	NgModule
@Optional	Parámetro que marca una dependencia como opcional. El inyector proporciona null si la dependencia no se encuentra.	Optional
@Pipe	Permite definir pipes.	Pipe
@Self	DI busca una dependencia propia, sin buscar hacia arriba en el árbol.	Self
@SkipSelf	DI busca dependencia en todo el árbol empezando por el padre.	SkipSelf

A continuación, realizaremos un ejercicio con **@Attribute** y **@Hostlistener** para mostrar el uso de decoradores.

1. En primer lugar, nos ubicaremos en **ej100_angular** y crearemos el proyecto **decora** mediante el comando **ng new**:

```
C:\Ej100_angular>ng new decora
```

Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename decora 020_decora
```

A continuación, desde el directorio **020_decora** crearemos un nuevo componente **compo1** tecleando (**ng generate component compo1** o de forma abreviada):

```
C:\Ej100_angular\020_decora>ng g c compo1
```

Ahora, abrimos la aplicación con nuestro editor y añadimos en **app.component.html** lo siguiente:

```
<app-compo1 miAtributo=”Valor de muestra para atributo”></
   app-compo1>
```

```
app.component.html
1  <h1>
2    {{title}}
3  </h1>
4  <app-compo1 miAtributo="Valor de muestra para atributo"></app-compo1>
```

De esta forma mostramos nuestro **compo1** al arrancar la aplicación y también definimos un atributo que consumiremos desde nuestro componente mediante **@Attribute**.

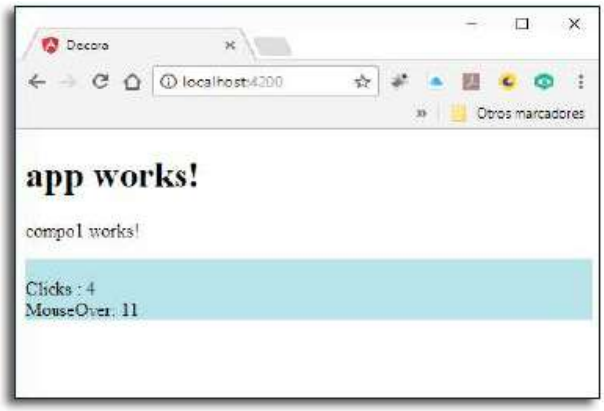
2. Seguidamente, modificaremos el archivo **compo1.component.html** para añadir un par de variables que muestren las veces que hacemos **clic** o que **pasamos el ratón sobre el componente** y usar de esta forma el decorador **@Hostlistener**. El contenido será el siguiente:

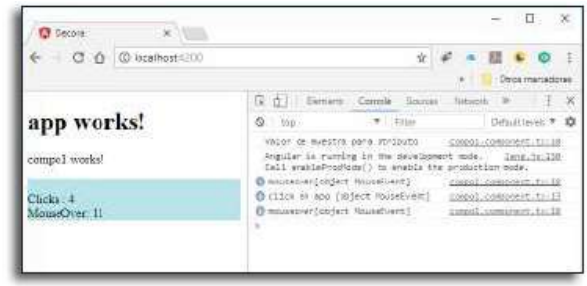
```
<p>compo1 works!</p>

<div style=”background-color:powderblue;”>

    <br> Clicks : {{ contaClick }}

    <br> MouseOver: {{ contaOver }}

</div>
```

En **compo1.component.ts**, modificaremos su contenido quedando de la siguiente manera:

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

@Component({

    selector: 'app-compo1',

    templateUrl: './compo1.component.html',

    styleUrls: ['./compo1.component.css']

})

export class Compo1Component {

    contaClick: number = 0;

    contaOver: number = 0;

    constructor( @Attribute('miAtributo') atributo) {console.
   log(atributo); }

    @HostListener('click', ['$event'])

    onHostClick(event: Event) {

        console.log("click en app " + event);

        this.contaClick += 1;

    }

    @HostListener('mouseover', ['$event'])

    onMouseOver(event: Event) {

        console.log("mouseover" + event);

        this.contaOver += 1;

    }

}
```

Por último, arrancamos la aplicación ubicándonos en **C:\Ej100_angular\020_decora** y tecleando **ng serve**. En el navegador, moveremos el ratón sobre el componente y haremos algunos clics para ver cómo se actualiza el contador. Abrimos las herramientas de desarrollador (en **Google Chrome** pulse **CTRL+MAYUS+I**) y en la **Console** vemos cómo se muestran los mensajes que indican el **Valor de muestra para atributo** obtenido con **@Attribute** y cada evento ejecutado **(MouseEvent)** del ratón.


Decora
localhost:4200
Otros marcadores
app works!
compo1 works!
Clicks : 4
MouseOver. 11


app works!
compo1 works!
Clicks : 4
MouseOver. 11

021 Comunicación entre componentes

La comunicación entre componentes puede realizarse de diversas maneras (servicios, observables, etc.), pero en este ejercicio, utilizaremos los decoradores **@Input** y **@Output** mencionados brevemente en ejercicios anteriores. Habíamos comentado que, mediante **@Input**, podemos pasar datos desde un componente **padre** hacia un componente **hijo**. Con **@Output**, realizaremos lo contrario. Pasaremos datos desde un **hijo** hacia su **padre**.

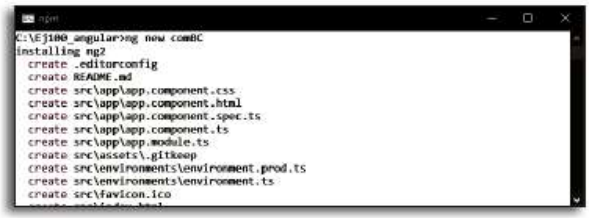
Importante

Utilice @Input y @Output para transferir datos de un componente "padre" a "hijo" y viceversa.

El siguiente ejercicio propone un proyecto muy simple en el que un componente **padre** se encarga de recoger dos valores que le pasa al hijo para realizar una operación sencilla (**suma**, **resta**, **multiplicación** y **división**) y, una vez obtenido el resultado, el mismo se devuelve al padre para que este se muestre en su vista.

En primer lugar, desde **ej100_angular**, crearemos el proyecto **comBC** mediante **ng new**:

```
C:\Ej100_angular>ng new comBC
```

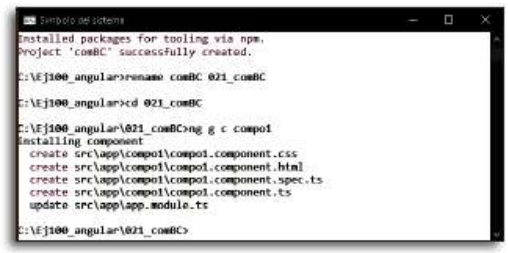


Seguidamente, renombraremos el proyecto de la siguiente forma:

```
C:\Ej100_angular>rename comBC 021_comBC
```

A continuación, desde el directorio **021_comBC** crearemos el componente **compo1** tecleando:

```
C:\Ej100_angular\021_comBC>
ng g c compo1
```



Abrimos la aplicación con nuestro editor y dejamos en **app.component.html** el siguiente contenido:

```
<div class="padre">
  <table border=1>
    <th colspan="2" style="text-align:center">Padre</th>
    <tr>
      <td>Valor1</td>
      <td><input class="form-control" [(ngModel)]="valor1"></td>
    </tr>
    <tr>
      <td>Valor2</td>
      <td><input class="form-control" [(ngModel)]="valor2"></td>
    </tr>
    <tr>
      <td>Resultado</td>
      <td><b>{{ resultadoP }}</b></td>
    </tr>
  </table>
</div>
<app-compo1          [valor1]="valor1"          [valor2]="valor2"
    (envRes)="captaResultado($event)">
</app-compo1>
```

En **app.component.ts** modificaremos la clase **AppComponent** para que tenga lo siguiente:

```
export class AppComponent {
    valor1: string = '10';
    valor2: string = '20';
    resultadoP: number;
    captaResultado(event) { this.resultadoP = event; }
}
```

Ahora, modificaremos **compo1.component.html** para añadir los botones con las operaciones:

```
<div class="hijo ">
    <table border=1>
        <th colspan="4" style="text-align:center">Hijo</th>
        <tr>
          <td><button class="tamB" (click)="suma()">+</button></
    td>
                <td><button  class="tamB"  (click)="resta()">-</
    button></td>
             <td><button class="tamB" (click)="multiplica()">*</
    button></td>
                <td><button  class="tamB"  (click)="divide()">/</
    button></td>
        </tr>
    </table>
</div>
```

Seguidamente, modificamos **compo1.component.ts** para que el **import** existente al inicio del fichero contenga lo siguiente:

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

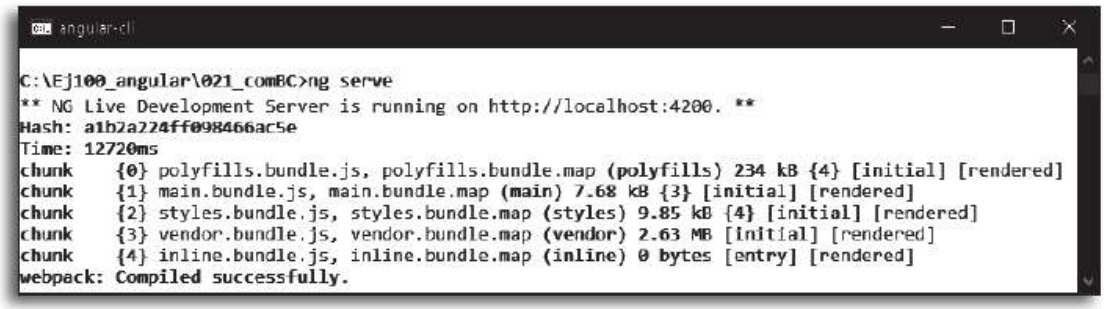
También modificaremos la clase **Compo1Component** con el siguiente contenido:

```
export class Compo1Component {
    @Input() valor1: string;
    @Input() valor2: string;
    aux1: number;
    aux2: number;
    @Output()
    envRes: EventEmitter<number> = new EventEmitter<number>();
    ngAfterContentChecked() {
        this.aux1 = parseFloat(this.valor1);
        this.aux2 = parseFloat(this.valor2);
    }
    suma()       { this.envRes.emit(this.aux1 + this.aux2); }
    resta()      { this.envRes.emit(this.aux1 - this.aux2); }
    multiplica() { this.envRes.emit(this.aux1 * this.aux2); }
    divide()     { this.envRes.emit(this.aux1 / this.aux2); }
}
```

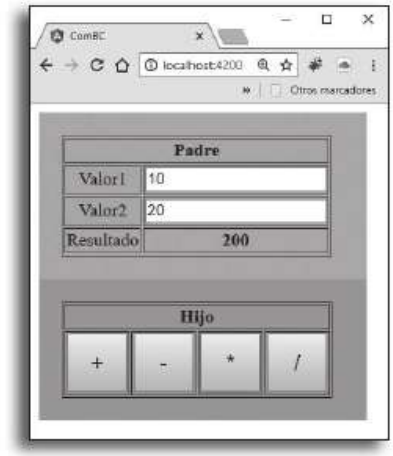
Por último, modificaremos el archivo **styles.css** con lo siguiente:

```
div{

    width:250px;

    padding: 20px;

    display: block;

    text-align: center;

}

.padre { background: #abe4b8; }

.hijo  { background: #5af0f0; }

.tamB  { height:53px; width:53px; font-size: 20px; }
```

Arrancamos la aplicación tecleando **ng serve** desde **C:\Ej100_angular\021_comBC** y vemos la aplicación en el navegador. Probaremos las distintas operaciones (**suma**, **resta**, **multiplicación** y **división**) para comprobar cómo el resultado llega al **padre** a través del evento que se lanza desde el **hijo** y que se captura desde el **padre**.




ComBC
localhost:4200
Otros marcadores
Padre
Valor1
10
Valor2
20
Resultado
-10
Hijo
+
-
*
/


ComBC
localhost:4200
Otros marcadores
Padre
Valor1
10
Valor2
20
Resultado
200
Hijo
+
-
*
/

022 Componentes: Ciclo de vida (Lifecycle hooks)

Las directivas y componentes poseen un ciclo de vida de tal manera que se crean, se actualizan y finalmente se destruyen. Angular ofrece los **lifecycle hooks** (o "ganchos" del ciclo de vida) para que los desarrolladores puedan aprovecharlos por mediación de interfaces cuyo nombre es idéntico al evento producido, pero anteponiéndole el prefijo **ng**. Las directivas poseen un ciclo más corto.

Importante

Podemos usar algunos métodos para aprovecharnos de las acciones que se realizan durante el ciclo de vida de un componente como, por ejemplo, inicializar variables o destruir recursos antes de que el componente sea destruido.

Los **lifecycle hooks** son los siguientes:

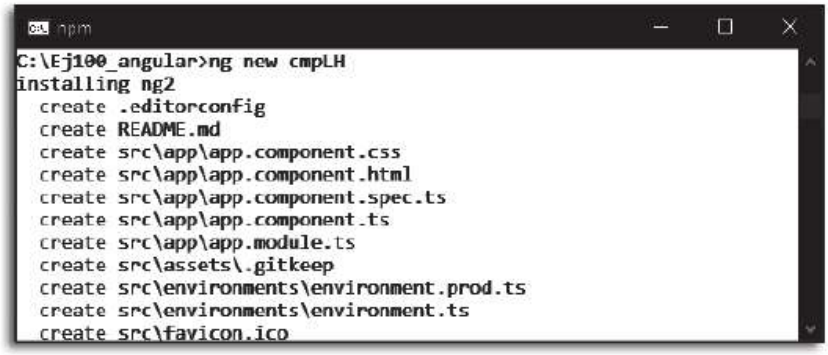
- **ngOnChanges:** cuando se detectan cambios en las propiedades de entrada. El método recibe un objeto **SimpleChanges** de los valores de propiedad actuales y anteriores.
- **ngOnInit:** tras la primera ejecución de **ngOnChanges**. Se llama solo una vez e inicializa el componente o directiva. Usamos este método para realizar inicializaciones justo después de la construcción del componente. Es un buen lugar para cargar datos.
- **ngDoCheck:** se usa para actuar sobre los cambios que Angular no captura y que pueden tener cierto interés más allá de un simple cambio de estado.
- **ngAfterContentInit:** después de inicializar el contenido del componente. Solo existe en los componentes.
- **ngAfterContentChecked:** tras cada comprobación del contenido del componente. Solo existe en los componentes. Comprueba el contenido visualizado en el componente.
- **ngAfterViewInit:** después de que las vistas del componente se inicialicen y después de **ngAfterContentChecked**.
- **ngAfterViewChecked:** después de cada comprobación de la vista de un componente. Se llama después de **ngAfterViewInit** y de cada **ngAfterContentChecked**.
- **ngOnDestroy:** antes de destruir el componente. Usamos este método para realizar la limpieza de recursos que

Angular no realiza automáticamente y es un buen lugar para avisar a otros componentes de que el componente en curso se va a destruir.

En el siguiente ejercicio realizaremos una pequeña aplicación en la que generaremos una traza de forma que podamos monitorizar cuál es la secuencia en la que sucede cada uno de estos momentos del ciclo de vida de un componente mediante la consola del navegador.

1. En primer lugar, nos ubicaremos en **ej100_angular** y crearemos el proyecto **cmpLH** mediante el comando **ng new**. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, ubicados en el directorio **Ej100_angular**, usaremos el comando **rename**. A continuación, nos ubicaremos en nuestro directorio **022_cmpLH** y crearemos un nuevo componente (**compo1**) con el comando **ng**.

```
C:\Ej100_angular>ng new cmpLH

C:\Ej100_angular>rename cmpLH 022_cmpLH

C:\Ej100_angular>cd 022_cmpLH

C:\Ej100_angular\022_cmpLH>ng g c compo1
```



Una vez creado el componente, abriremos la aplicación con nuestro editor y modificaremos el archivo **app.component.html** para incluir nuestro componente **compo1**:

```
<h1>{{title}}</h1>
<hr>
<div class="container bg"><br>
  En app.component <input class="form-control"
   [(ngModel)]="salidaPadre"><br>
</div>
<br><hr><app-compo1 [entradaHijo]="salidaPadre"></app-
   compo1>
```

Observemos que dentro de las etiquetas **<app-compo1>**, hemos indicado que queremos pasar a **compo1**, el valor de la variable **salidaPadre** de **app.component.ts** (que definiremos a continuación) dentro de la variable **entradaHijo** (que también definiremos en **compo1**).

2. Modificamos **app.component.ts** para que la clase **AppComponent** quede de la siguiente manera:

```
export class AppComponent {
  title = 'app works!';
  salidaPadre: string = "";
}
```

Modificamos el archivo **compo1/compo1.component.html** para que tenga el siguiente contenido:

```
<p>compo1 works!</p>
<div class="container bg">
    <br> En compo 1 <input class="form-control" [(ngModel)]="
   entradaHijo"><br><br>
    <div class="container">Hola {{ entradaHijo }}</div>
</div><br>
```

En la clase de **compo1** modificaremos el import existente para que tenga lo siguiente:

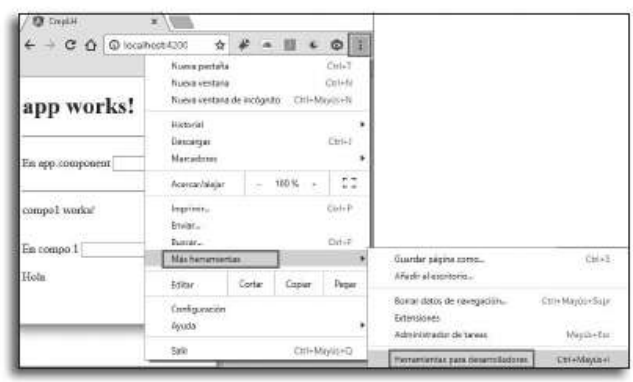
```
import { Component, OnInit, SimpleChanges, Input, OnChanges,
   ViewChild } from '@angular/core';
```

3. En la clase **Compo1Component** incluimos un método por cada momento del ciclo de vida:

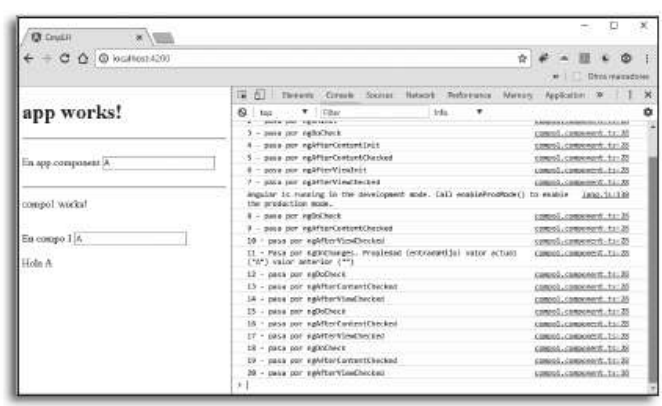
```
export class Compo1Component implements OnInit {

  @Input() entradaHijo: string = "";

  contador: number = 0;

  constructor() { }

  ngOnInit() { this.mostrar("pasa por ngOnInit"); }

  ngOnChanges(cambios: SimpleChanges) {

    for (let propiedad in cambios) {

      let cambio = cambios[propiedad];

      let actual = JSON.stringify(cambio.currentValue);

      let anterior = JSON.stringify(cambio.previousValue);

      this.mostrar("Pasa por ngOnChanges. Propiedad (" +
    propiedad + ") valor actual (" + actual + ") valor anterior
    (" + anterior + ")");

    }

  }

  ngDoCheck() { this.mostrar("pasa por ngDoCheck"); }

  ngAfterContentInit() { this.mostrar("pasa por
    ngAfterContentInit"); }

  ngAfterContentChecked() { this.mostrar("pasa por
    ngAfterContentChecked"); }

  ngAfterViewInit() { this.mostrar("pasa por ngAfterViewInit");
    }

  ngAfterViewChecked() { this.mostrar("pasa por
    ngAfterViewChecked"); }

  public mostrar(texto: string) {

    this.contador += 1;

    console.log(this.contador + " - " + texto);

  }

}
```

Ahora, ubicados en el directorio **022_cmpLH**, ejecutaremos **ng serve** y observaremos cómo se muestra nuestra aplicación en el navegador.

4. Una vez en el navegador, si estamos usando **Google Chrome**, mostraremos la **Console** acudiendo a **Más herramientas -> Herramientas para desarrolladores**.



5. De esta forma, podremos ver los displays de texto que se generan cada vez que ejecutamos la aplicación y cada vez que introducimos algún carácter en las cajas de texto de las vistas.



023 Directivas: Definición

Las directivas permiten añadir a HTML un comportamiento dinámico mediante una etiqueta o un selector. Dependiendo del tipo de directiva, podemos cambiar la apariencia o comportamiento de un elemento o incluso añadir o eliminar elementos del **DOM** (***document object model*** o **modelo de objetos del documento**).

Importante

Utilice las directivas para cambiar el aspecto de sus elementos y modificar la estructura del **DOM** dinámicamente.

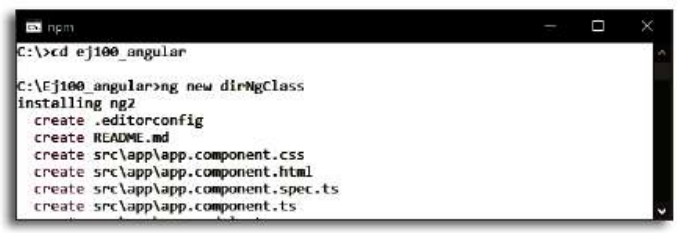
Básicamente, existen dos tipos de directivas:

- **Directivas estructurales:** alteran la estructura del DOM añadiendo, sustituyendo o eliminando elementos. Empiezan por asterisco (*).
  - **ngFor:** permite iterar sobre una lista de elementos y realizar diversas acciones en un HTML como, por ejemplo, fabricar listas.
  - **ngIf:** en base a una evaluación, crea o elimina elementos en el **DOM**.
  - **ngSwitch:** gestiona conjuntos de tags eliminando los que no cumplan una condición.
- **Directivas de atributo:** modifican la apariencia de un elemento o modifican su comportamiento.
  - **ngClass**: permite añadir o eliminar dinámicamente una o varias clases CSS en un elemento.
  - **ngModel**: habilita un mecanismo de **binding bi-direccional**.
  - **ngStyle**: permite asignar varios estilos **inline** a un elemento.

También podemos considerar que un componente es un tipo de directiva concreto al que se le ha dado un decorador propio por tratarse de un caso especial.

En el siguiente ejercicio se muestra cómo poder aplicar dos clases de forma condicional con **ngClass**.

1. En primer lugar, nos ubicaremos en **ej100_angular** y crearemos un proyecto denominado **dirNgClass**.

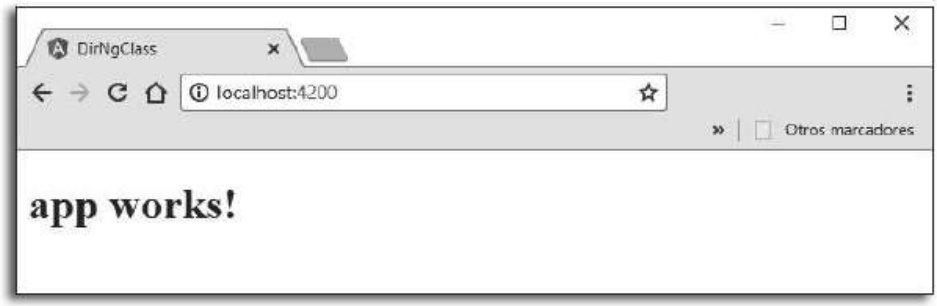


2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, ubicados en el directorio **Ej100_angular**, simplemente escribiremos lo siguiente:

```
C:\Ej100_angular>rename dirNgClass 023_dirNgClass
```

A continuación, podemos ubicarnos en nuestro nuevo directorio y arrancar la aplicación para verla en nuestro navegador. Para ello, teclearemos lo siguiente:

```
C:\Ej100_angular>cd 023_dirNgClass

C:\Ej100_angular\023_dirNgClass>ng serve
```



Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso, usamos **Atom**) y modificamos el archivo **styles.css** añadiendo una definición para las clases **fa** (fondo amarillo) y **lg** (letra grande):

```
.fa { background-color: rgb(255,246,51); }

.lg { font-size: 200%; }
```

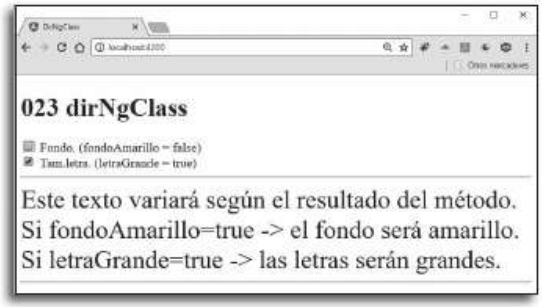
Seguidamente, modificaremos nuestro archivo **app.component.ts** para que la clase **AppComponent** quede de la siguiente manera:

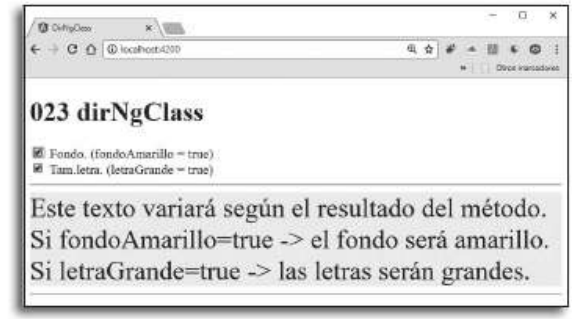
```
export class AppComponent {
  title = '023 dirNgClass';
  fondoAmarillo=false;
  letraGrande=false;
  checkFondo(){
    this.fondoAmarillo=!this.fondoAmarillo;
  }
  checkLetra(){
    this.letraGrande=!this.letraGrande;
  }
  asignaClases(){
    let classes = {
      fa: this.fondoAmarillo,
      lg: this.letraGrande
    };
    return classes;
  }
}
```

3. Por último, modificaremos el fichero **app.component.html** para que tenga el siguiente contenido:

```
<h1>{{title}}</h1>
<input type="checkbox" (change)="checkFondo()">
Fondo. (fondoAmarillo = {{fondoAmarillo}})<br>
<input type="checkbox" (change)="checkLetra()">
Tam.letra. (letraGrande = {{letraGrande}})<hr>
<div [ngClass]="asignaClases()">
  Este texto variará según el resultado del método.<br>
  Si fondoAmarillo=true -> el fondo será amarillo.<br>
  Si letraGrande=true -> las letras serán grandes.
</div>
<hr>
```

4. En el navegador, observaremos que nuestra aplicación muestra un texto con dos **checkboxs** que permiten cambiar el fondo de nuestro **div** y aumentar el tamaño de la letra según si los marcamos o no, o hacer ambas cosas. Pruebe a marcar y desmarcar para comprobar el efecto.





024 Directivas: ngIf

Importante

Utilizar ***ngIf** permite optimizar recursos al reducir el **DOM** y, con ello, facilitar la transferencia de recursos por la red.

Tal y como habíamos comentado en el ejercicio anterior, ***ngIf** permite añadir o eliminar un elemento al **DOM** si se cumple una determinada condición. A diferencia de la propiedad **visibility** de **CSS** donde se puede ocultar un elemento, **ngIf** lo elimina físicamente si la condición no se cumple, de forma que no consume recursos.

En el siguiente ejercicio, mostramos unos elementos **div** basándonos en ***ngIf** y en el valor asociado a unas variables de tipo Boolean. Dichos div se hallan anidados en una sencilla jerarquía que hemos llamado grupos, los cuales contienen subgrupos.

1. En primer lugar, nos ubicaremos en **ej100_angular** y crearemos el proyecto **dirNgIf** mediante el comando **ng new**:

```
C:\Ej100_angular>ng new dirNgIf
```

Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename dirNgIf 024_dirNgIf
```

A continuación, nos ubicaremos en **024_dirNgIf** y arrancaremos la aplicación para comprobar cómo funciona en nuestro navegador. Para ello, tecleamos lo siguiente:

```
C:\Ej100_angular>cd 024_dirNgIf

C:\Ej100_angular\024_dirNgIf >ng serve
```

Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso, usamos **Atom**) y modificamos el archivo **styles.css** añadiendo una definición para las clases **uno** (para **div** de tipo grupos), **dos** (para subgrupos) y **tamB** para el tamaño de los botones:

```
.uno{ border: 3px solid #F00; background-color:
   lightblue;}

.dos{ border: medium double #369; background-color:
   yellow; }

.tamB{ height:40px; width:300px; }
```

Seguidamente, modificaremos nuestro archivo **app.component.ts** para que la clase **AppComponent** quede de la siguiente manera:

```
export class AppComponent {

  title = '024 dirNgIf';

  grupo1=false;

  grupo2=false;

  subgrupo11=false;

  subgrupo12=false;

  subgrupo21=false;

  subgrupo22=false;

}
```

A continuación, modificaremos el fichero **app.component.html** para que tenga el siguiente contenido:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
    com/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css"
    integrity="sha384-rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/
    AZzGRnGxQQKnKkoFVhFQhNUwEyJ" crossorigin="anonymous">

<h1>{{title}}</h1>

<button type="button" class="btn btn-success tamB"
    (click)="grupo1=!grupo1">

  Grupo1. (grupo1 = {{grupo1}})</button><br>

<div *ngIf="grupo1" class="uno">Grupo1.<br>

  <button type="button" class="btn btn-primary tamB"
    (click)="subgrupo11=!subgrupo11">Subgrupo11. (subgrupo11 =
    {{subgrupo11}})</button><br>

  <div *ngIf="subgrupo11" class="dos">Subgrupo11<br></div>

  <button type="button" class="btn btn-primary tamB"
    (click)="subgrupo12=!subgrupo12">

    Subgrupo12. (subgrupo12 = {{subgrupo12}})</button><br>

  <div *ngIf="subgrupo12" class="dos">Subgrupo12<br></div>

</div>

<br>

<button type="button" class="btn btn-success tamB"
    (click)="grupo2=!grupo2">

  Grupo2. (grupo2 = {{grupo2}})</button><br>

<div *ngIf="grupo2" class="uno">Grupo2.<br>

  <button type="button" class="btn btn-primary tamB"
    (click)="subgrupo21=!subgrupo21">

    Subgrupo21. (subgrupo21 = {{subgrupo21}})</button><br>

  <div *ngIf="subgrupo21" class="dos">Subgrupo21<br></div>

  <button type="button" class="btn btn-primary tamB"
    (click)="subgrupo22=!subgrupo22">

    Subgrupo22. (subgrupo22 = {{subgrupo22}})</button><br>

  <div *ngIf="subgrupo22" class="dos">Subgrupo22<br></div>

</div>
```

Observe que hemos añadido la siguiente referencia para poder utilizar **Bootstrap** y de esta forma, añadir clases y funcionalidades a la aplicación:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
    com/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css"
    integrity="sha384-rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/
    AZzGRnGxQQKnKkoFVhFQhNUwEyJ" crossorigin="anonymous">
```

Por último, observaremos el aspecto de nuestra aplicación en el navegador. Si pulsamos sobre el botón del **Grupo1** veremos cómo se despliega el **div** asociado. Si pulsamos sobre **Subgrupo11** y **Subgrupo12** aparecen los **div** de dichos subgrupos. Si pulsamos sobre todos los botones veremos todos los div asociados.

025 Directivas: ngFor

La directiva **ngFor** permite insertar bloques de código **HTML** de forma dinámica, basándose en una lista de elementos (en general, arrays). La sintaxis general es la siguiente:

```
<tag *ngFor="let elemento of elementos"></tag>
```

donde tag es el elemento HTML que queremos crear según el número de iteraciones (p. ej., div, p, li, etc.) y elementos en la lista de elementos que iremos asignando uno a uno a elemento mediante let.

Importante

Utilice ***ngFor** para generar bloques HTML basándonos en el contenido de una lista de elementos.

Podemos averiguar el número del elemento que se está tratando si añadimos let i=index, tal y como se muestra a continuación:

```
<tag *ngFor="let elemento of elementos ; let i=index "></tag>
```

En primer lugar, nos ubicaremos en **ej100_angular** y crearemos el proyecto **dirNgFor**.

1. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename dirNgFor 025_dirNgFor
```

A continuación, nos ubicaremos en **025_dirNgFor** y arrancaremos la aplicación para comprobar cómo funciona en nuestro navegador. Para ello, teclearemos lo siguiente:

```
C:\Ej100_angular>cd 025_dirNgFor

C:\Ej100_angular\025_dirNgFor >ng serve
```

Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso, usamos **Atom**) y modificamos el archivo **app.component.ts** para que la clase **AppComponent** quede de la siguiente manera:

```
export class AppComponent {

  title = '025 dirNgFor';

  public nombres = [

    {nom:"uno",edad:10},

    {nom:"dos",edad:20},

    {nom:"tres",edad:30}

  ];

}
```

Podemos observar que hemos creado un pequeño **array** denominado nombres con tres objetos compuestos únicamente por el campo **nom** y **edad**.

2. A continuación, modificaremos el fichero **app.component.html** para que tenga el siguiente contenido:

```
<link rel="stylesheet" href="https://maxcdn.
   bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
   css/bootstrap.min.css" integrity="sha384-
   rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/
   AZzGRnGxQQKnKkoFVhFQhNUwEyJ" crossorigin="anonymous">
<h1>
  {{title}}
</h1>
<div class="container"><br>
  <ul class="list-group">
    <li class="list-group-item list-group-item-success"
   *ngFor="let nombre of nombres; let i=index">
      <p>{{i + 1}} - </p>
      <p>Nombre:  <b>{{ nombre.nom }}</b></p>
      <p>, edad: <b>{{ nombre.edad}}</b></p>
    </li>
  </ul>
</div>
```

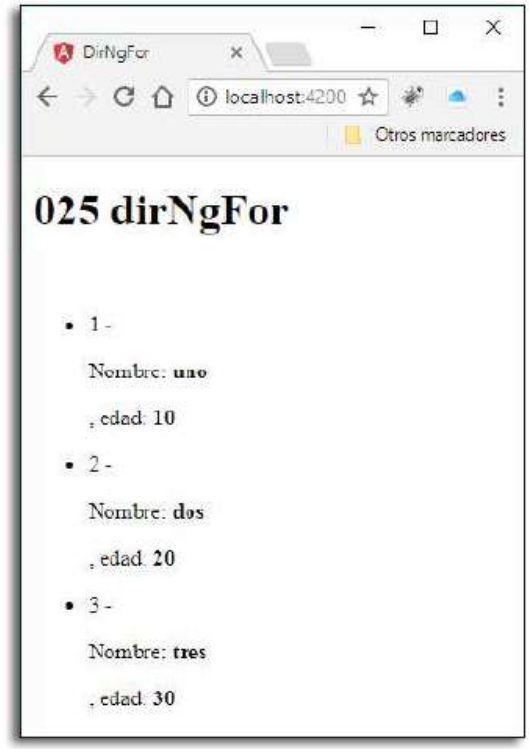
Observe que hemos añadido la siguiente referencia para poder utilizar **Bootstrap** y, de esta forma, añadir clases y funcionalidades a la aplicación:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
   com/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css"
   integrity="sha384-rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/
   AZzGRnGxQQKnKkoFVhFQhNUwEyJ" crossorigin="anonymous">
```

Ahora, observaremos el aspecto de nuestra aplicación en el navegador.

3. Si lo desea, comente la primera línea del archivo **app.component.html** (la de <link rel…) para comprobar el aspecto de la aplicación sin utilizar las clases de **Bootstrap**.


DirNgFor
localhost:4200
Otros marcadores
025 dirNgFor
1 -Nombre: uno, edad: 10
2 -Nombre: dos, edad: 20
3 -Nombre: tres, edad: 30


DirNgFor
localhost:4200
Otros marcadores
025 dirNgFor
1 - Nombre: uno , edad: 10
2 - Nombre: dos , edad: 20
3 - Nombre: tres , edad: 30

026 Directivas: ngSwitch

La directiva **ngSwitch** funciona como el clásico **switch** que podemos encontrar en multitud de lenguajes (p. ej., **JavaScript**). Es decir, evalúa una expresión y compara su resultado con una serie de valores añadiendo al **DOM** el bloque asociado en caso de coincidencia. Funciona con los siguientes elementos:

Importante

Utilice ***ngSwitch** cuando necesite usar varios **ngIf** que usan la misma expresión y quiera mostrar un bloque en el caso de que no se cumpla ninguna comparación con los valores existentes.

Keywords	Comentarios
`[ngSwitch]="variable"`	Expresión que se define como un atributo y que evalúa el contenido de variable. Utiliza la propiedad **binding**. Como variable, también puede usarse una interpolación (**{{ exp }}** como veremos en ejercicios posteriores).
`*ngSwitchCase`	Define la comparación que se va a realizar. Si se cumple, se muestra el elemento HTML asociado (div, p, li, etc.).
`*ngSwitchDefault`	Define el elemento por defecto que se va a mostrar cuando no hay ninguna coincidencia.

A continuación, mostramos una estructura simple:

```
<elemento [ngSwitch]="variable">
  <elemento *ngSwitchCase="'valor1'">…</elemento>
  <elemento *ngSwitchCase="'valor2'">…</elemento>
  <elemento *ngSwitchDefault>…</elemento>
</elemento>
```

También podríamos utilizar la siguiente sintaxis:

```
<elemento ngSwitch="{{ variable }}">
```

En el siguiente ejercicio vamos a mostrar un bloque u otro según el valor que asignemos a una variable al pulsar alguno de los botones incluidos en nuestra página.

1. En primer lugar, nos ubicaremos en **ej100_angular** y crearemos el proyecto **dirNgSwitch**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename dirNgSwitch 026_dirNgSwitch
```

A continuación, nos ubicaremos en **026_dirNgSwitch** y arrancaremos la aplicación para comprobar cómo funciona en nuestro navegador. Para ello, teclearemos lo siguiente:

```
C:\Ej100_angular>cd 026_dirNgSwitch

C:\Ej100_angular\026_dirNgSwitch>ng serve
```

Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso, usamos **Atom**) y modificamos el título de la aplicación dentro del archivo **app.component.ts** para que la clase **AppComponent** quede de la siguiente manera:

```
export class AppComponent {

  title = '026 dirNgSwitch';

}
```

Seguidamente, modificamos el archivo **styles.css** añadiendo una definición para los colores de fondo de un par de DIV (**bg** y **bg2**) dejando el siguiente contenido:

```
.bg { background-color: #F9E79F; }

.bg2 { background-color: #D5F5E3; }
```

A continuación, modificaremos el fichero **app.component.html** para que tenga el siguiente contenido:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
    bootstrap/4.0.0-alpha.6/css/bootstrap.min.css" integrity="sha384-
    rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNUwEyJ"
    crossorigin="anonymous">

<h1>{{title}}</h1>

<br>

<div class="container bg" [ngSwitch]="opcion">  Valor seleccionado:

  <div class="container text-success" *ngSwitchCase="'A'">Ha elegido la
    opcion: <b>A</b></div>

  <div class="container text-success" *ngSwitchCase="'B'">Ha elegido la
    opcion: <b>B</b></div>

  <div class="container text-success" *ngSwitchCase="'C'">Ha elegido la
    opcion: <b>C</b></div>

  <div class="container text-danger" *ngSwitchDefault><b>Sin selección</b></
    div>

</div>

<br>

<div class="container">

  <button class="btn btn-primary btn-lg" (click)="opcion='A'">A</button>

  <button class="btn btn-primary btn-md" (click)="opcion='B'">B</button>

  <button class="btn btn-primary btn-sm" (click)="opcion='C'">C</button>

</div>

<br>

<div class="container bg2"><h5>Boton pulsado : {{opcion}}</h5></div>
```

Observe que hemos incluido la siguiente referencia para poder utilizar **Bootstrap** y, de esta forma, añadir clases y funcionalidades a la aplicación:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
    com/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css"
    integrity="sha384-rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/
    AZzGRnGxQQKnKkoFVhFQhNUwEyJ" crossorigin="anonymous">
```

Hemos querido poner tres botones de diferentes tamaños para ver algunas posibilidades de **Bootstrap**.

3. Ahora, observaremos el aspecto de nuestra aplicación en el navegador. Vemos que, hasta que no pulsemos ningún botón, se mostrará el texto **Sin selección**.

4. Pulse cualquier botón para ver que, al cambiar de valor la variable **opcion**, también cambia el texto de Valor seleccionado.

027 Directivas: ngModel

ngModel es un mecanismo que permite actualizar los valores de las variables usadas en el componente y en el archivo HTML de forma bidireccional. Este tema se ve con más detalle en los ejercicios dedicados al **DATABINDING** (**One Way** y **Two Way**). Antes de poder utilizar **ngModel** hemos de importar el **FormsModule** y añadirlo en la lista de módulos importados en **app.module.ts**, tal y como explicamos en el ejercicio posterior:

Importante

Utilice ***ngModel** para actualizar de forma bidireccional el contenido de variables entre componentes y formularios o archivos HTML.

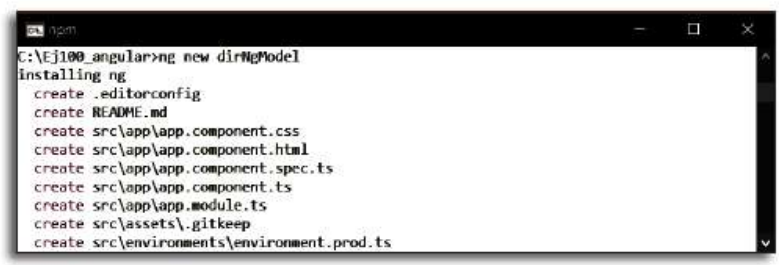
```
import { FormsModule } from '@angular/forms';

…

  imports: [ BrowserModule, FormsModule ],
```

El ejemplo más típico utiliza el elemento **INPUT** para mostrar su funcionamiento y su sintaxis general es la siguiente:

```
<input [(ngModel)]="variable">
```

A la notación **[()]** también se le conoce como **banana in a box** por la forma gráfica resultante al imaginar que los corchetes representan una caja y los paréntesis una banana. En el siguiente ejemplo, vamos a trabajar con dos elementos de tipo **INPUT** donde observaremos que, al escribir en cualquiera de ellos, se copiará el texto al otro convirtiendo a minúsculas todo el texto en uno de ellos y a mayúsculas en el otro. Al mismo tiempo, mostraremos mediante interpolación el contenido de ambos **INPUT**. Hemos incluido un ***ngIF** para que el texto resultante de la concatenación solo se muestre si hay algún carácter introducido en cualquier **INPUT**.

1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos el proyecto **dirNgModel** utilizando **ng new**.



2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename dirNgModel 027_dirNgModel
```

A continuación, nos ubicaremos en **027_dirNgModel** y arrancaremos la aplicación para comprobar cómo funciona en nuestro navegador. Para ello, teclearemos lo siguiente:

```
C:\Ej100_angular>cd 027_dirNgModel

C:\Ej100_angular\027_dirNgModel>ng serve
```

Tal y como habíamos anticipado al principio, hemos de modificar también el archivo **app.module.ts** para que contenga las importaciones comentadas y, en definitiva, lo siguiente:

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

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

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

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

@NgModule({

  declarations: [ AppComponent ],

  imports: [ BrowserModule, FormsModule ],

  providers: [],

  bootstrap: [AppComponent],

})

export class AppModule { }
```

Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso, usamos **Atom**) y modificamos el archivo **app.component.ts** para que la clase **AppComponent** quede de la siguiente manera:

```
export class AppComponent {
  title   = '027 dirNgModel';
  nombre  = '';
  nombre2 = '';
  cambiaNombre() {
    this.nombre=this.nombre.toLowerCase();
    this.nombre2=this.nombre.toUpperCase();
  }
  cambiaNombre2() {
    this.nombre2=this.nombre2.toUpperCase();
    this.nombre=this.nombre2.toLowerCase();
  }
}
```

Seguidamente, modificamos el archivo **styles.css** añadiendo una definición para los colores de fondo de los div existentes (**bg** y **bg2**) dejando el siguiente contenido:

```
.bg { background-color: #F9E79F; }

.bg2 { background-color: #D5F5E3; }
```

A continuación, modificaremos el fichero **app.component.html** para que tenga el siguiente contenido:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
    bootstrap/4.0.0-alpha.6/css/bootstrap.min.css" integrity="sha384-
    rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNUwEyJ"
    crossorigin="anonymous">
<h1>{{title}}</h1>
<br>
<div class="container bg">
  <br>
  <input class="form-control" [(ngModel)]="nombre"
    (keyup)="cambiaNombre($event)"><br>
  <input class="form-control" [(ngModel)]="nombre2"
    (keyup)="cambiaNombre2($event)"><br>
</div>
<br>
<div *ngIf="nombre" class="container bg2">
  <h3>Hola {{ nombre }} / {{ nombre2 }}</h3>
</div>
```

Observemos cómo se ve ahora nuestra aplicación en el navegador.

3. Si introducimos el texto **JuAnTo** en cualquiera de las cajas de texto correspondientes a los **INPUT**, veremos cómo en la primera caja aparece todo el texto en minúsculas y en la segunda, en mayúsculas.

028 Directivas: ngStyle

ngStyle permite asignar varios estilos **inline** a un elemento con la definición de una serie de estilos o bien al asociar una función que devuelve uno o varios estilos en función de una serie de condiciones. La sintaxis general es:

```
[ngStyle]="{'propiedad1': variable, 'propiedad2': 'valor2'}"
```

Por ejemplo:

```
[ngStyle]="{'color': color, 'font-weight': 'bold'}"
```

Si asociamos una función usaríamos:

```
[ngStyle]="funcion()"
```

Por ejemplo:

```
[ngStyle]="defineEstilos()"
```

En el siguiente ejercicio, mostraremos un **div** en el que usamos **ngStyle** aplicando unos estilos más o menos fijos que solo se apoyan en una variable para determinar el color del texto y, a continuación, mostraremos un div al que asociamos una función que evaluará varias condiciones y dará como resultado un conjunto de estilos que se aplicarán simultáneamente. Al igual que en el ejercicio anterior, tendremos que importar **FormsModule** para utilizar **ngModel** (como veremos a lo largo del ejercicio).

Importante

Utilice ***ngStyle** cuando tenga que aplicar diferentes estilos en función de una serie de condiciones.

1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos el proyecto **dirNgStyle** utilizando **ng new**.

```
C:\>cd Ej100_angular

C:\Ej100_angular>ng new dirNgStyle
installing ng
  create .editorconfig
  create README.md
  create src\app\app.component.css
  create src\app\app.component.html
  create src\app\app.component.spec.ts
  create src\app\app.component.ts
  create src\app\app.module.ts
  create src\assets\.gitkeep
```

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename dirNgStyle 028_dirNgStyle
```

A continuación, nos situaremos en **028_dirNgStyle** y arrancaremos la aplicación para comprobar cómo funciona en nuestro navegador. Para ello, teclearemos lo siguiente:

```
C:\Ej100_angular>cd 028_dirNgStyle

C:\Ej100_angular\028_dirNgStyle>ng serve
```

Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso, usamos **Atom**) y, tal y como habíamos anticipado al principio, hemos de modificar el archivo **src/app/app.module.ts** para que contenga las importaciones comentadas y, en definitiva, lo siguiente:

```
import …

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

…

  imports: [ BrowserModule, FormsModule ],

…

})

export class AppModule { }
```

Seguidamente modificamos el archivo **app.component.ts** para que la clase **AppComponent** quede de la siguiente manera:

```
export class AppComponent {

  title   = '028 dirNgStyle';

  color = 'blue';

  hayLetraGrande:boolean  = false;

  hayColorFondo:boolean  = false;

  hayLetraColor:boolean  = false;

  hayLetraItalica:boolean  = false;

  defineEstilos(){

      let styles= {

          'font-size': this.hayLetraGrande ? '300%' : 'initial',

          'background-color': this.hayColorFondo ? '#58FA58' :
    '#FFFFFF',

          'color': this.hayLetraColor ? 'red' : 'black',

          'font-style': this.hayLetraItalica ? 'italic' : 'normal'

      };

      return styles;

  }

}
```

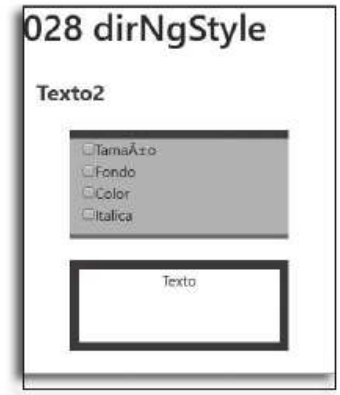
Seguidamente, modificamos el archivo **styles.css** al añadir una definición para cada uno de los **div** existentes (**div1** y **div2**), dejando el siguiente contenido:

```
.div1 {

border-top: 10px solid blue; border-bottom: 5px solid red;
    height: 120px; width: 70%;

background-color: #A9E2F3;

}

.div2 {

border: 10px solid green; height: 100px; width: 70%;
    background-color: #58FA58;

text-align: center;

}
```

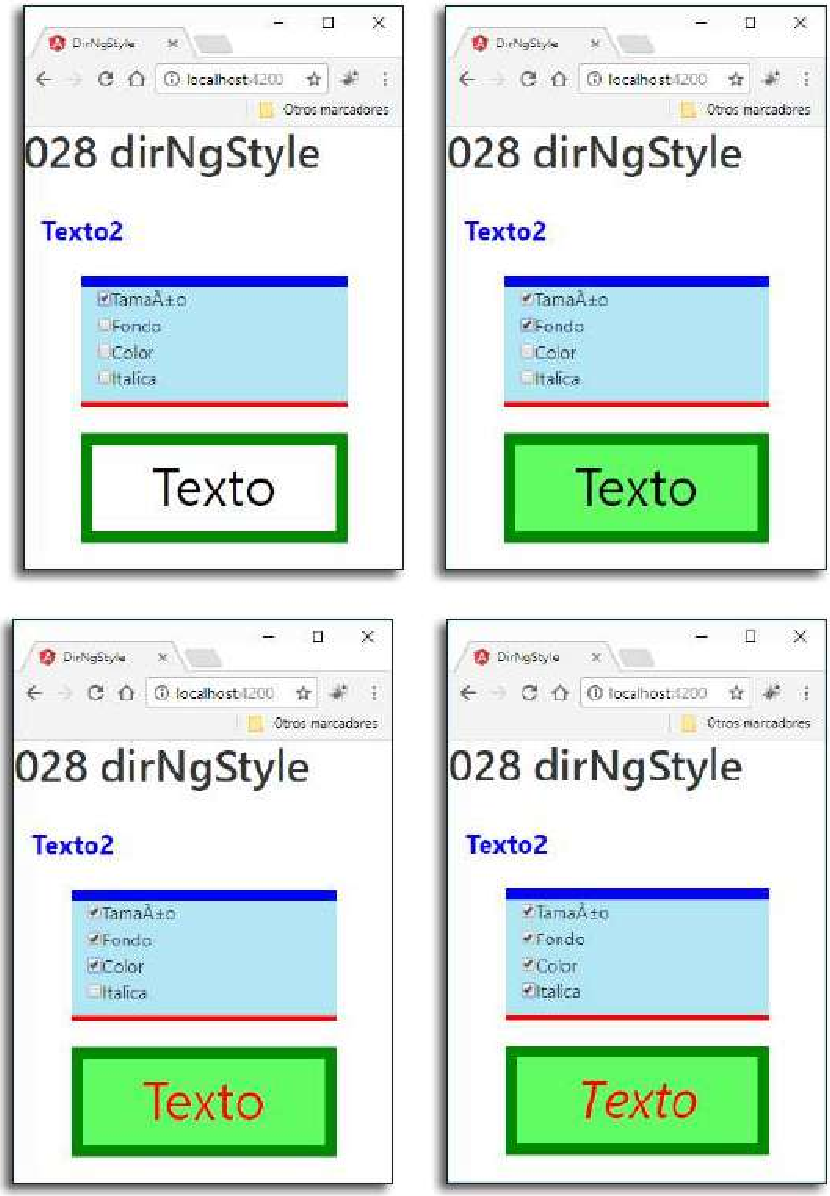
A continuación, modificaremos el fichero **app.component.html** para que tenga el siguiente contenido:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
    bootstrap/4.0.0-alpha.6/css/bootstrap.min.css" integrity="sha384-
    rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNUwEyJ"
    crossorigin="anonymous">

<h1>{{title}}</h1><br>

<div class="container" [ngStyle]="{'color': color, 'font-weight':
    'bold', 'font-size': '150%'}">Texto2</div>

<br>

<div class="container div1">

  <input type="checkbox" name="hayLetraGrande"
    [(ngModel)]="hayLetraGrande">Tamaño<br>

  <input type="checkbox" name="hayColorFondo"
    [(ngModel)]="hayColorFondo">Fondo<br>

  <input type="checkbox" name="hayLetraColor"
    [(ngModel)]="hayLetraColor">Color<br>

  <input type="checkbox" name="hayLetraItalica"
    [(ngModel)]="hayLetraItalica">Italica<br>

</div>

<br>

<div class="container div2" [ngStyle]="defineEstilos()">Texto</div>
```

Observemos cómo se ve ahora nuestra aplicación en el navegador.



3. Si marcamos los **checkbox** (**Tamaño**, **Fondo**, **Color**, **Italica**) veremos cómo varían los estilos del bloque inferior.


DirNgStyle
localhost:4200
Otros marcadores
028 dirNgStyle
Texto2
TamaÃ±o
Fondo
Color
Italica
Texto

029 Directivas: Mezcla

En el siguiente ejercicio realizaremos una aplicación en la que se combinan las directivas mostradas en los ejercicios anteriores. Mostraremos una colección de números y un par de **checkbox** para escoger la visualización de números pares o impares, ambos o ninguno. En primer lugar, se muestra un botón mediante el cual podemos usar ***ngIf** visualizando (o no) el **div** principal con los **checkbox** y **números**. Para los **checkbox**, usaremos **ngClass** para llamar a la función **clActividad()** que aplicará una clase cambiando el color (**activo** o **inactivo**). A continuación, usaremos el ***ngFor** para crear un **div** por cada uno de los números existentes en un array de números definido en el componente. Luego, usaremos **ngSwitch** para determinar el valor a comparar en ***ngSwitchCase**. Dicho valor será el resto de dividir el índice de cada elemento dentro de la lista entre dos para saber si es **par** o **impar**. Según sea par o impar, se aplicará un color con **ngStyle**.

Importante

Utilice las directivas siempre que pueda para simplificar su trabajo y dejar que sea Angular quien aplique la parte mecánica que sea posible para alterar el DOM.

Al igual que en los ejercicios anteriores, tenemos que importar el **FormsModule** y añadirlo en la lista de módulos importados en **app.module.ts,** tal y como explicaremos a lo largo del ejercicio.

1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos el proyecto **dirNgMezcla** utilizando **ng new**.

2. Seguidamente, renombraremos el proyecto ubicándonos en **Ej100_angular** y escribiendo lo siguiente:

```
C:\Ej100_angular>rename dirNgMezcla 029_dirNgMezcla
```

A continuación, nos situaremos en **029_dirNgMezcla** y arrancaremos la aplicación para comprobar cómo funciona en nuestro navegador. Para ello, teclearemos lo siguiente:

```
C:\Ej100_angular>cd 029_dirNgMezcla

C:\Ej100_angular\029_dirNgMezcla>ng serve
```

Ahora, abrimos nuestro proyecto y modificamos el archivo **src/app/app.module.ts** para que contenga las importaciones comentadas y, en definitiva, lo siguiente:

```
…

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

…

  imports: [ BrowserModule, FormsModule ],

…
```

Seguidamente modificamos **app.component.ts** para que la clase **AppComponent** quede así:

```
export class AppComponent {

  title   = '029 dirNgMezcla';

  mostrar = false;

  numeros:number [] = [1,2,3,4,5,6];

  pares:boolean = true; impares:boolean = true;

  colorPar = 'blue'; colorImpar = 'red';

  clActividad(valor){

    let classes = { activo: valor, inactivo: !valor };

    return classes;

  }

}
```

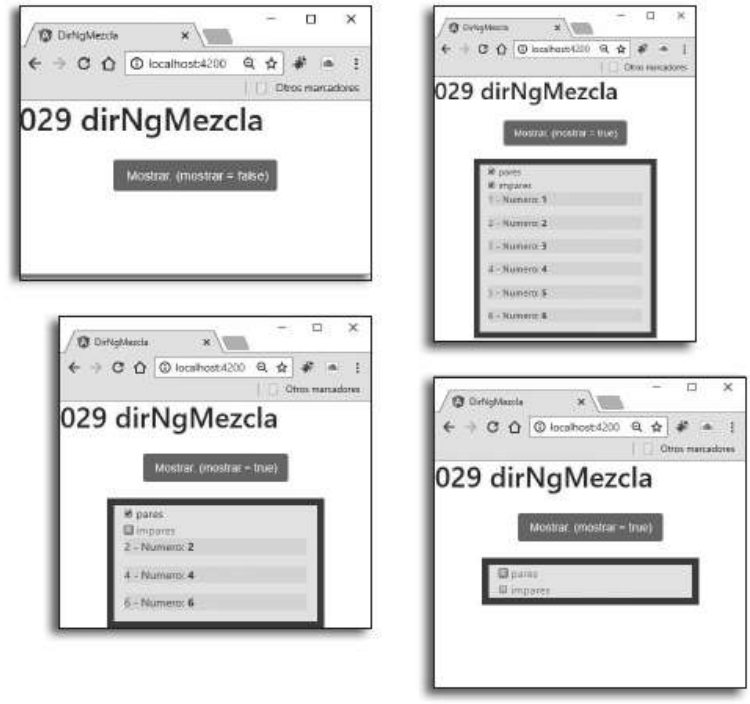
Seguidamente, modificamos **styles.css** añadiendo diversas definiciones para div, textos y el botón:

```
.divBg { border: 10px solid green; width: 70%; background-color:
   #F3F781; }
.tamB { height:40px; width:50%; }
.activo  { color: blue; }
.inactivo{ color:red;    }
```

A continuación, modificaremos el fichero **app.component.html** para que tenga el siguiente contenido:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
   com/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css"
   integrity="sha384-rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/
   AZzGRnGxQQKnKkoFVhFQhNUwEyJ" crossorigin="anonymous">
<h1>{{title}}</h1><br>
<div class="container" style="text-align:center;">
  <button type="button" class="btn btn-success tamB"
    (click)="mostrar=!mostrar;pares=true;impares=true">
    Mostrar. (mostrar = {{mostrar}})</button><br><br>
  </div>
  <div *ngIf="mostrar" class="container divBg">
    <input type="checkbox" [(ngModel)]="pares" name="pares">
    <span [ngClass]="clActividad(pares)">pares</span><br>
    <input type="checkbox" [(ngModel)]="impares" name="impares">
    <span [ngClass]="clActividad(impares)">impares</span><br>
    <div class="list-group-item-info" *ngFor="let numero of
   numeros; let i=index">
      <div [ngSwitch]="i%2">
        <div *ngSwitchCase="'1'">
          <p *ngIf="pares"><span [ngStyle]="{'color': colorPar}">
            {{i + 1}}</span> - Numero:  <b>{{ numero }}</b></p>
        </div>
        <div *ngSwitchCase="'0'">
          <p *ngIf="impares"><span [ngStyle]="{'color':
   colorImpar}">
            {{i + 1}}</span> - Numero:  <b>{{ numero }}</b></p>
        </div>
      </div>
    </div>
  </div>
```

Observemos cómo se ve ahora nuestra aplicación en el navegador. Si pulsamos el botón Mostrar, se visualiza el **div** principal con los números. Si desmarcamos los impares, vemos que el texto se pone rojo y solo se visualizan los pares. Por último, si desmarcamos los pares, solo se visualizan los **checkbox**.



030 Pipes: Uso, parametrización y encadenamientos

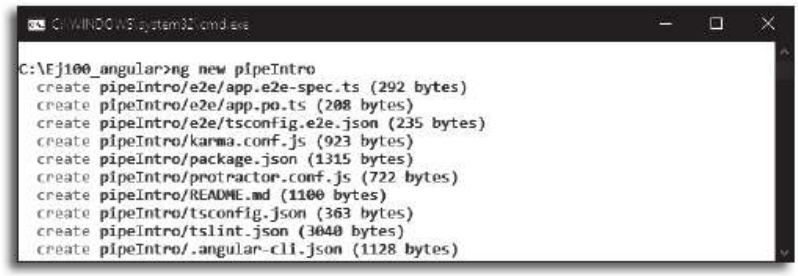
Importante

Los pipes nos permitirán realizar transformaciones de datos desde los mismos templates o documentos HTML.

Cuando las aplicaciones muestran datos a los usuarios, raramente los muestran tal y como llegan desde los servicios u otras fuentes. Normalmente, lo que hacen es aplicar algún tipo de transformación antes de visualizarlos. Por ejemplo, una aplicación normalmente no mostraría una fecha como "Sun June 12 1977 00:00:00 GMT+0200 (Central European Summer Time)", sino que lo haría como "June 12, 1977".

Para realizar estas transformaciones tan comunes, Angular ofrece los **pipes**. Los pipes nos permitirán realizar las transformaciones desde los mismos templates o documentos html. Vamos a ver unos ejemplos:

1. Nos ubicaremos en **ej100_angular**, y crearemos el proyecto tecleando **ng new pipeIntro**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio al ejecutar **rename pipeIntro 030_pipeIntro**, y finalmente pondremos en marcha la aplicación tras ejecutar **ng serve** desde la carpeta **030_pipeIntro**.



2. Ahora abriremos nuestro proyecto con nuestro editor (en nuestro caso, usamos **Atom**) y añadiremos el siguiente código en los siguientes archivos sobrescribiendo todo su contenido.

`src/app/app.component.ts`

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


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  fecha1 = new Date(1988, 3, 15); //15 de Abril del 1988
}
```

src/app/app.component.html

```
<p>Fecha (sin pipe)  : {{ fecha1 }}</p>
<p>Fecha (DatePipe): {{ fecha1 | date }}</p>
```

En el navegador veremos lo siguiente:

```
Fecha (sin pipe) : Fri Apr 15 1988 00:00:00 GMT+0200 (Hora de
    verano romance)
Fecha (DatePipe): Apr 15, 1988
```

Todos los pipes funcionan de igual manera: dentro de la expresión de interpolación ("{{...}}"), justo después de la propiedad del componente, se añade un operador de pipe '|' y, a continuación, se añade el tipo de pipe a usar.

Angular incorpora todo un conjunto de pipes para que puedan usarse desde cualquier template: **DatePipe** para formatear una fecha, **UpperCasePipe** para transformar texto a mayúscula, **LowerCasePipe** para transformar texto a minúscula, **CurrencyPipe** para formatear un número como moneda, **PercentPipe** para formatear un número como porcentaje, etc. Pero, además, también nos permite crear nuevos **pipes personalizados.**

Por otra parte, un pipe puede aceptar **parámetros**. Estos se pasarán a la pipe a través del operador ':'. Si el pipe aceptara varios parámetros, los separaríamos con ':'. Ejemplo: "{{ ... | currency:'USD':false }}".

El valor de un parámetro puede ser una expresión de template válida, una cadena de caracteres o una propiedad/función de un componente. Esto significa que el cambio de formato puede controlarse con data binding (mecanismos que permiten la comu-

nicación entre un componente y su template) de la misma manera en la que controlábamos "fecha1" en el ejemplo anterior. Veamos un ejemplo:

3. En este ejemplo vamos a usar una función de un componente como valor de parámetro de un DatePipe. Está función devolverá ShortDate o FullDate, parámetros válidos para el DatePipe, según el valor de la propiedad "idFormatoFecha", que podremos modificar mediante un botón. Para ello, añadiremos el siguiente código y lo pondremos a prueba desde el navegador:

src/app/app.component.ts

```
export class AppComponent {

…

  idFormatoFecha = true; // true == shortDate ; false == fullDate

  get formatoFecha()     { return this.idFormatoFecha ? 'shortDate' :
    'fullDate'; }

  cambiarFormatoFecha() { this.idFormatoFecha = !this.idFormatoFecha;
    }

}
```

src/app/app.component.html

```
…

<p>Fecha (DatePipe con '{{ formatoFecha }}'): {{ fecha1 |
   date:formatoFecha }}</p>

<button (click)="cambiarFormatoFecha()">Cambiar formato</button>
```

Por último, comentar que los **pipes pueden encadenarse** para realizar diversas transformaciones una detrás de otra. El encadenamiento de pipes lo realizaremos usando el operador '|' como separador. Veamos un ejemplo:

4. Para este último ejemplo añadiremos el siguiente código y comprobaremos desde el navegador cómo se aplican los pipes DatePipe y UpperCasePipe uno detrás de otro.

src/app/app.component.html

```
<p>Fecha (DatePipe con 'fullDate' y UpperCasePipe): {{ fecha1 |
    date:'fullDate' | uppercase }}</p>
```

031 Pipes: DatePipe, UpperCasePipe y LowerCasePipe

Angular por defecto incorpora los pipes DatePipe, UpperCasePipe, LowerCasePipe, DecimalPipe, CurrencyPipe y PercentPipe para que puedan ser usados desde cualquier template.

En este ejercicio veremos cómo usar DatePipe, UpperCasePipe y LowerCasePipe a través de diversos ejemplos. El resto, los veremos en el siguiente ejercicio.

Importante

Utilice DatePipe y todo su abanico de parametrizaciones posibles para realizar los cambios de formato de fechas que necesite.

DatePipe

DatePipe nos permite formatear fechas. Se usa de la siguiente manera:

date_expression | date[:format]

donde:

- *date_expression* es un objeto fecha, un número (milisegundos respecto fecha UTC epoch [1 de enero de 1970]) o un cadena de caracteres con formato ISO (http://www.w3.org/TR/NOTE-datetime).
- *format* es una cadena de caracteres que identifica el formato con que el que se quiere visualizar la fecha. Puede contener alguno de los valores predefinidos validos: medium, short, fullDate, o una combinación de símbolos ('y': año, 'M': mes, etc.) para crear nuevos formatos. Consulte https://angular.io/api/common/DatePipe para más información.

UpperCasePipe

UpperCasePipe transforma un texto a mayúsculas.

LowerCasePipe

LowerCasePipe transforma un texto a minúsculas.

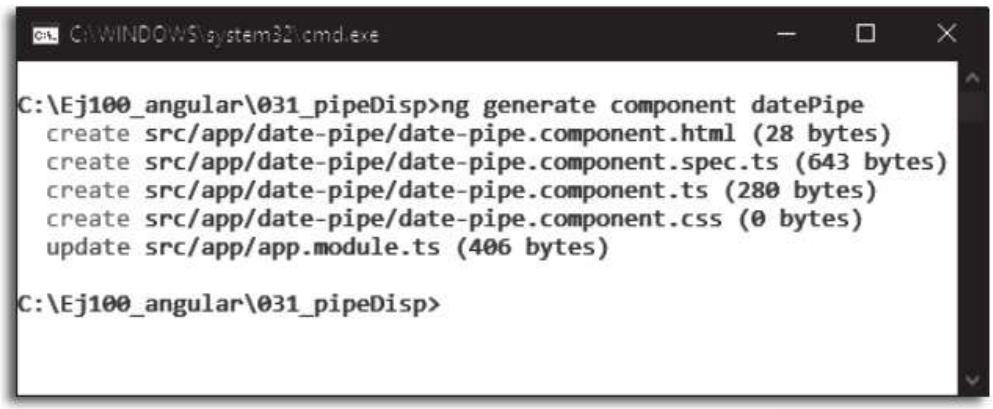
Vamos a ver ejemplos de estos pipes.

1. Nos ubicaremos en **ej100_angular**, y crearemos el proyecto tecleando **ng new pipeDisp**. Seguidamente lo renombraremos para que haga referencia

a nuestro número de ejercicio ejecutando **rename pipeDisp 031_pipeDisp**, y finalmente pondremos en marcha la aplicación ejecutando **ng serve** desde la carpeta **031_pipeDisp**.

2. Primero crearemos el componente a partir del cual realizaremos el ejercicio

```
C:\Ej100_angular\031_pipeDisp>ng generate component datePipe
```



Seguidamente abriremos el proyecto con nuestro editor **Atom**, y en el template principal de la aplicación pondremos el siguiente código para que cargue nuestro componente. Podremos comprobarlo desde el navegador.

src/app/app.component.html

```
<app-date-pipe></app-date-pipe>
```

Finalmente añadiremos los ejemplos en el código. Desde el navegador, podremos ver el resultado.

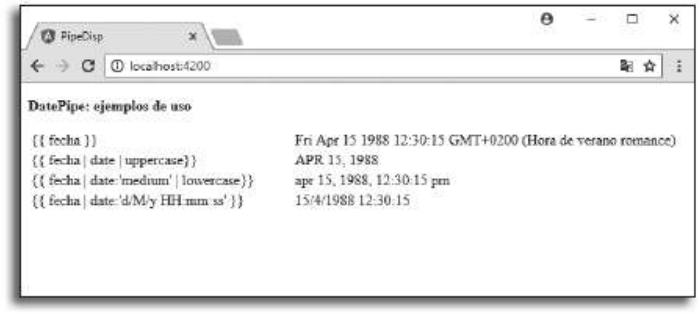
src/app/date-pipe/date-pipe.component.ts

```
export class DatePipeComponent implements OnInit {

  fecha: Date = new Date(1988, 3, 15, 12, 30, 15); //15 de Abril del
    1988 12:30:15

…
```

src/app/date-pipe/date-pipe.component.html

```
<div>

    <p><b>DatePipe: ejemplos de uso</b></p>

    <table style="width:100%">

      <colgroup><col style="width:40%"><col style="width:60%"></
    colgroup>

      <tr>
```

```
        <td ngNonBindable>{{ fecha }}</td>
        <td>{{ fecha }}</td>
      </tr>
      <tr>
        <td ngNonBindable>{{ fecha | date | uppercase}}</td>
        <td>{{ fecha | date | uppercase}}</td>
      </tr>
      <tr>
        <td ngNonBindable>{{ fecha | date:'medium' | lowercase}}</td>
        <td>{{ fecha | date:'medium' | lowercase}}</td>
      </tr>
      <tr>
        <td ngNonBindable>{{ fecha | date:'d/M/y HH:mm:ss' }}</td>
        <td>{{ fecha | date:'d/M/y HH:mm:ss' }}</td>
      </tr>
    </table>
</div>
```



032 Pipes: DecimalPipe, CurrencyPipe y PercentPipe

En este ejercicio seguiremos analizando los pipes que Angular incorpora. En este caso, analizaremos los pipes DecimalPipe, CurrencyPipe y PercentPipe. Todos ellos parten de un número como valor de entrada.

Importante

A veces, desde una pipe, se usan funcionalidades de otras pipes, como es el caso de CurrencyPipe y PercentPipe respecto a DecimalPipe.

DecimalPipe

Nos permite formatear un número como texto. Se usa de la siguiente manera:

number_expression | number[:digitInfo]

donde:

- *digitInfo* es una cadena de texto con el siguiente formato:

  {minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

donde:

- *minIntegerDigits* es el número mínimo de enteros que se van a usar. Por defecto vale 1.
- *minFractionDigits* es el número mínimo de decimales que se van a usar. Por defecto vale 0.
- *maxFractionDigits* es el número máximo de decimales que se van a usar. Por defecto vale 3.

CurrencyPipe

CurrencyPipe nos permite formatear un número como moneda. Se usa de la siguiente manera:

number_expression | currency[:currencyCode[:symbolDisplay[:digitInfo]]]

donde:

- *currencyCode* es el código de moneda ISO 4217, como USD para el dólar y EUR para el euro.

- *symbolDisplay:* con true visualizaremos el símbolo ($ o €), y con false el código (USD o EUR).
- *digitInfo:* mirar DecimalPipe para ver su descripción.

PercentPipe

Nos permite formatear un número como porcentaje. Se usa de la siguiente manera:

number_expression | percent[:digitInfo]

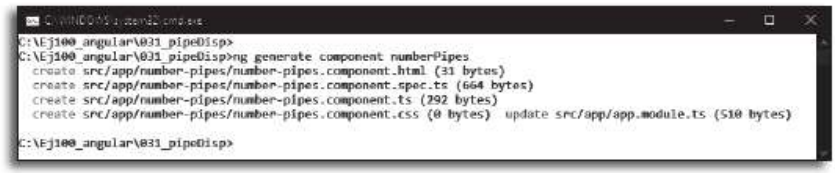
donde:

- *digitInfo*: mirar DecimalPipe para ver su descripción**.**

Vamos a ver ejemplos de estos pipes.

1. Desde la carpeta del proyecto creado en el anterior ejercicio, creamos el componente.

```
C:\Ej100_angular\031_pipeDisp>ng generate component numberPipes
```



Seguidamente abriremos el proyecto con nuestro editor **Atom**, y en el template principal de la aplicación, pondremos el siguiente código para que cargue nuestro nuevo componente. Desde el navegador podremos comprobarlo.

src/app/app.component.html

```
<app-number-pipes></app-number-pipes>
```

Por último, añadiremos el código en el componente. En el navegador podremos ver el resultado.

src/app/number-pipes/number-pipes.component.ts

```
export class NumberPipesComponent implements OnInit {

  pi: number = 3.141592;

…
```

src/app/number-pipes/number-pipes.component.html

```
<div>

    <p><b>CurrencyPipe, PercentPipe: ejemplos de uso</b></p>

    <table style="width:100%">

      <colgroup><col style="width:40%"><col style="width:60%"></
    colgroup>

      <tr>

        <td ngNonBindable>{{pi}}</td>

        <td>{{pi}}</td>

      </tr>

      <tr>

        <td ngNonBindable>{{pi | currency:'EUR':false}}</td>

        <td>{{pi | currency:'EUR':false}}</td>

      </tr>

      <tr>

        <td ngNonBindable>{{pi | currency:'EUR':true:'4.3-3'}}</td>

        <td>{{pi | currency:'EUR':true:'4.3-3'}}</td>

      </tr>

      <tr>

        <td ngNonBindable>{{pi | percent}}</td>

        <td>{{pi | percent}}</td>

      </tr>

      <tr>

        <td ngNonBindable>{{pi | percent:'4.3-3'}}</td>

        <td>{{pi | percent:'4.3-3'}}</td>

      </tr>

    </table>

</div>
```


PipeDisp
localhost:4200
CurrencyPipe, PercentPipe: ejemplos de uso
{{pi}} 3.141592
{{pi | currency:'EUR':false}} EUR3.14
{{pi | currency:'EUR':true:'4.3-3'}} €0,003.142
{{pi | percent}} 314.159%
{{pi | percent:'4.3-3'}} 0,314.159%

033 Pipes: Pipes personalizados

Como hemos visto en anteriores capítulos, Angular ofrece un conjunto de pipes que permiten formatear los datos antes de mostrarlos al usuario. Sin embargo, podría pasar que este conjunto de pipes no fuera suficiente para realizar los cambios de formato que necesitamos. Por suerte, Angular ofrece las herramientas necesarias para crear nuevos pipes personalizados. En este ejercicio veremos cómo hacerlo.

Importante

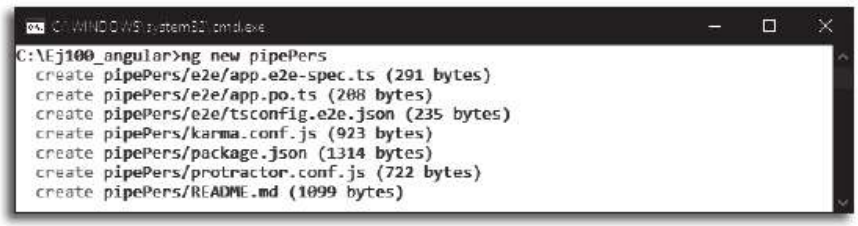
Angular ofrece todas la herramientas necesarias para crear pipes según nuestras necesidades. De esta manera podemos centralizar en los pipes todas las tareas de transformación que necesitemos.

En el ejemplo crearemos un pipe que realice una división entera, o sea, que lleve a cabo una división y, como resultado, devuelva el cociente y resto como números enteros sin decimales. A través del operador "|" obtendrá el dividendo y por parámetro se le pasará el divisor:

number_expression | divisionEntera:divisor

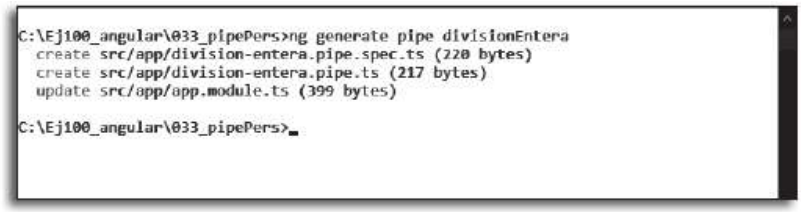
Vamos a empezar el ejercicio:

1. Nos ubicaremos en **ej100_angular**, y crearemos el proyecto tecleando **ng new pipePers**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio ejecutando **rename pipePers 033_pipePers**, y finalmente pondremos en marcha la aplicación ejecutando **ng serve** desde la carpeta **033_pipePers**.



2. Ahora crearemos el nuevo pipe **divisionEnteraPipe**. Para ello ejecutaremos lo siguiente:

```
C:\Ej100_angular\033_pipePers>ng generate pipe divisionEntera
```



La anterior ejecución habrá creado los archivos necesarios para el nuevo pipe y añadido una referencia en app.module.tst para que pueda usarse desde cualquier template, aunque, de momento, sin realizar ningún tipo de transformación. Para que se lleve a cabo, tendremos que implementar el código necesario. El desarrollo se realizará íntegramente en el archivo **division-entera.pipe.ts**. Este archivo lo encontraremos así:

src/app/division-entera.pipe.ts

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


@Pipe({
  name: 'divisionEntera'
})
export class DivisionEnteraPipe implements PipeTransform {


  transform(value: any, args?: any): any {
    return null;
  }


}
```

De forma resumida, podemos definir un pipe como una clase "decorada" con metadatos de pipe. La implementación de cualquier pipe personalizado tiene los siguientes requerimientos:

- La clase del pipe debe implementar el método **transform** de la interface **PipeTransform**. Este método toma el valor que "canaliza" y un número variable de parámetros de cualquier tipo, y devuelve un valor transformado.
- Debe usarse el decorador **@Pipe** para que Angular sepa que es un pipe. En el decorador, indicaremos el nombre del pipe. Este será el que deberemos usar desde los templates para usarlo.

Una vez analizado el código que se ha generado, seguiremos con la implementación.

3. Añadiremos el siguiente código, que definirá la funcionalidad de nuestro pipe:

src/app/division-entera.pipe.ts

```
…
  transform(value: number, divisor: string): string {
    let div = parseFloat(divisor);
    let cociente = Math.floor(value/div);
    let resto = value % div;

    return "Cociente: " + cociente + " # Resto: " + resto;
  }
```

Finalmente, para probar el pipe, añadiremos el siguiente código. Desde el navegador podremos probar nuestro nuevo pipe modificando los valores de dividendo y divisor.

src/app/app.component.ts

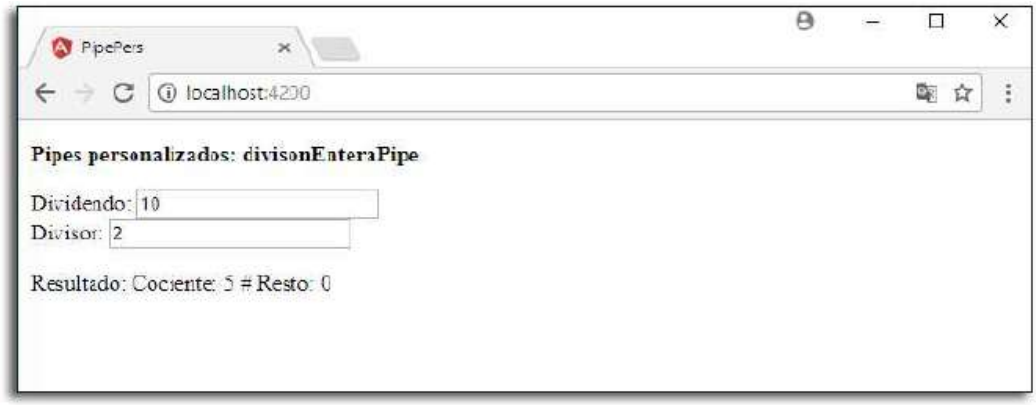
```
…
export class AppComponent {
  dividendo = 10;
  divisor = 2;
…
```

src/app/app.component.html

```
<p><b>Pipes personalizados: divisionEnteraPipe</b></p>
<div>Dividendo: <input [(ngModel)]="dividendo"></div>
<div>Divisor: <input [(ngModel)]="divisor"></div>
<p>  Resultado: {{dividendo | divisionEntera: divisor}}</p>
```

src/app/app.module.ts

```
…
import { FormsModule } from '@angular/forms';
…
@NgModule({
  declarations: […],
  imports: […,FormsModule],…
```


PipePers
localhost:4200
Pipes personalizados: divisonEnteraPipe
Dividendo: 10
Divisor: 2
Resultado: Cociente: 5 # Resto: 0


PipePers
localhost:4200
Pipes personalizados: divisonEnteraPipe
Dividendo: 17
Divisor: 2
Resultado: Cociente: 8 # Resto: 1

034 Pipes: Puros e impuros

Existen dos categorías de pipes: **puros e impuros**. La principal diferencia entre ellos radica en la manera que tiene Angular de gestionar su ejecución.

En este sentido, Angular ejecutará un **pipe puro** cuando detecte cambios en la variable de entrada, o sea, un cambio de valor si la variable es de tipo primitivo (string, number, boolean, symbol), o un cambio de referencia si la variable es una referencia a un objeto (date, array, function, object). Por lo tanto, Angular no ejecutaría un pipe de este tipo si, por ejemplo, la variable de entrada fuera de tipo array y le hubiéramos añadido un elemento.

Por el contrario, Angular ejecutará un **pipe impuro** en cada ciclo de detección de cambios (al interaccionar con cualquier elemento de la aplicación) sin importar si ha habido cambios o no en la variable de entrada o parámetros del pipe.

Todos los pipe que habíamos visto hasta ahora eran de tipo puro. De hecho, **todo pipe es puro por defecto**. Si queremos crear un pipe impuro, deberíamos indicarlo de la siguiente manera:

```
@Pipe({
  name: '...'
  pure: false
})
...
```

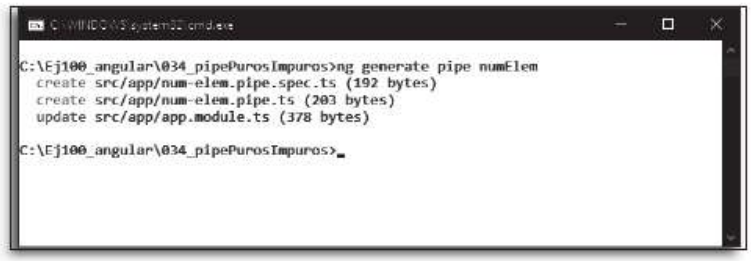
Aunque usar pipes impuros puede parecer más ventajoso, el hecho es que su uso requiere el consumo de muchos más recursos que pueden ralentizar la aplicación. Por tanto, siempre **es preferible usar pipes puros**.

Vamos a realizar un ejercicio que nos ayudará a entender estas diferencias. Crearemos un nuevo pipe, **numElem**, que devolverá el número de elementos de un array. Usaremos este pipe en un contador que nos indicará el número de elementos de un array al cual podremos añadir elementos desde la misma aplicación.

1. Nos ubicaremos en **ej100_angular** y crearemos el proyecto tecleando **ng new pipePurosImpuros**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio ejecutando **rename pipePurosImpuros 034_pipePurosImpuros**, y finalmente pondremos en marcha la aplicación ejecutando **ng serve** desde la carpeta **034_pipePurosImpuros**.

2. Seguidamente crearemos el pipe **numElem**. Para ello, ejecutaremos el siguiente comando:

```
C:\Ej100_angular\034_pipePurosImpuros>ng generate pipe numElem
```



y en el archivo **num-elem.pipe.ts**, añadimos el siguiente código en la función *transform*(…).

src/app/num-elem.pipe.ts

```
…

  transform(cadena: any[]): number {

    return cadena.length;

  }
```

El resto de código de la aplicación sería el siguiente:

src/app/app.component.ts

```
…

export class AppComponent {

  fechas: Date[] = [];

  anadirFecha() {    this.fechas.push(new Date());  }

}
```

src/app/app.component.html

```
<p><b>Pipes: Puros e Impuros</b></p>

<p>Contador: {{fechas | numElem}} </p>

<button (click)="anadirFecha()">Añadir fecha</button>

<div *ngFor="let fecha of (fechas)">

  <p>{{fecha | date:'medium'}}</p>

</div>
```

Importante

Utilice los pipes impuros de forma controlada. Consumen muchos más recursos que los pipes puros y puede afectar negativamente a la experiencia de usuario.

Al probar la aplicación desde el navegador, veremos que el contador no se actualiza al pulsar "Añadir fecha". El motivo es el comentado al principio del ejercicio: el pipe es puro, y como Angular no detecta cambios en la referencia del array de fechas, cuando se añaden nuevas fechas no ejecuta el pipe y el contador no se actualiza.

Como podrá ver a continuación, podemos solucionar el problema de dos maneras distintas. Aunque la opción preferible será la de declarar el pipe numElem como impuro (opción b) Al hacerlo vemos que el contador ya funciona.

a. Crear un nuevo array de fechas con la nueva fecha añadida, y asignarla a la original.

src/app/app.component.ts

```
…
  fechas: Date[] = [];
  anadirFecha() {
    this.fechas.push(new Date());
    let fechas2 = this.fechas.slice();
    this.fechas = fechas2;
  }
}
```

b. Declarar el pipe como impuro.

src/app/num-elem.pipe.ts

```
…
@Pipe({
  name: 'numElem',
  pure: false
})
…
```

035 Pipes: AsyncPipe

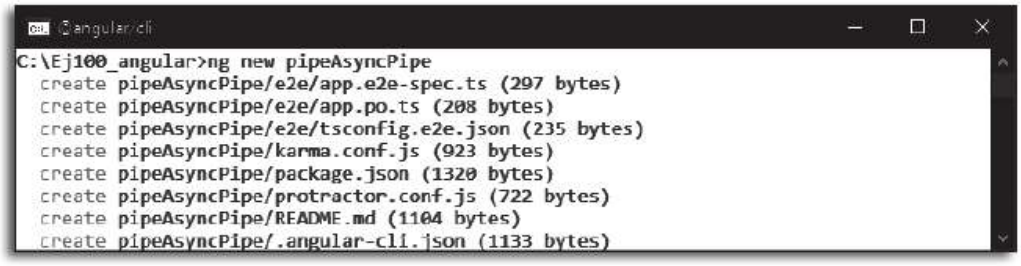
AsyncPipe es un tipo de pipe impuro. Su característica principal es que acepta objetos **Promise** u **Observable** como objetos de entrada. En el mundo de las aplicaciones web, la velocidad es esencial, por lo que el uso de **programación asíncrona** para aprovechar el tiempo y evitar esperas innecesarias es imprescindible. Promise y Observable son dos de las principales herramientas que tenemos para gestionar este tipo de programación. Consulte los capítulos 51, 52 y 53 para más información.

Importante

El uso de AsyncPipe, aparte de permitirnos ahorrar código, nos ofrece mayor seguridad al realizar de forma automática acciones como "unsubscribe" sobre los observables, que si no se realizaran, podrían provocar problemas de Memory Leaks.

En este ejercicio veremos como AsyncPipe nos facilita el uso de estas herramientas. Crearemos una aplicación primero sin usar AsyncPipe y luego empleándolo. La aplicación pondrá en marcha dos temporizadores de 2 y 4 segundos, respectivamente. Al terminar, cada uno de ellos devolverá un texto que acabaremos visualizando por el template. El primer temporizador será controlado mediante Promise y el segundo mediante Observable.

1. Nos ubicaremos en **ej100_angular**, y crearemos el proyecto tecleando **ng new pipeAsyncPipe**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio ejecutando **rename pipeAsyncPipe 035_pipeAsyncPipe**, y finalmente pondremos en marcha la aplicación ejecutando **ng serve** desde la carpeta **035_pipeAsyncPipe**.



2. Seguidamente añadiremos el código del ejemplo. En esta primera versión no usaremos AsyncPipe.

src/app/app.component.ts

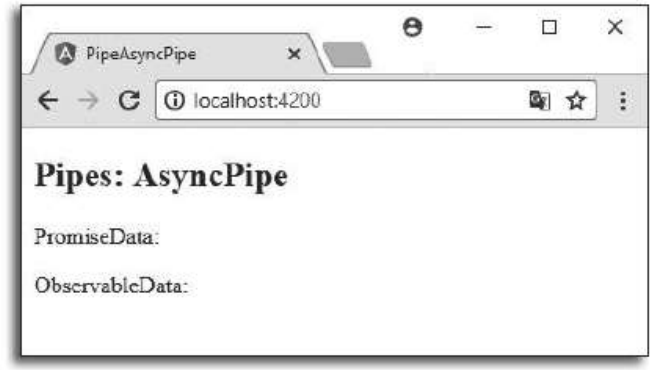
```
…
import { Observable } from 'rxjs/Observable';
import { Subscription } from "rxjs/Subscription";


@Component({…})
export class AppComponent {
  promiseData:string;
  observableData:string;
  observableSubs: Subscription = null;


  getPromise(){
    return new Promise<string>(function(resolve, reject) {
        setTimeout(() => {resolve("Timer1 finalizado");}, 2000);
    });
  }
  getObservable() {
    return new Observable<string>(observer => {
        setTimeout(() => {observer.next("Timer2 finalizado");}, 4000);
    });
  }
  constructor() {
    this.getPromise().then( v => this.promiseData = v );
    this.observableSubs = this.getObservable()
                                  .subscribe( v => this.observableData =
   v );
  }
  ngOnDestroy(){ if (this.observableSubs) this.observableSubs.
   unsubscribe();  }
}
```

src/app/app.component.html

```
<h2>Pipes: AsyncPipe</h2>

<p>PromiseData: {{ promiseData }}</p>

<p>ObservableData: {{ observableData }}</p>
```

Una vez añadido el código, podremos comprobar su funcionamiento en el navegador. Por pantalla veremos el texto devuelto por los temporizadores a medida que terminen. La secuencia sería la que aparece en las imágenes.

Recargando la página con F5, podremos reactivar los temporizadores.



3. Seguidamente modificaremos el código para que se haga uso de AsyncPipe.

src/app/app.component.ts

```
…
export class AppComponent {
  promiseObj: Promise<string>;
  observableObj: Observable<string>;

  getPromise(){…}
  getObservable() {…}

  constructor() {
    this.promiseObj = this.getPromise();
    this.observableObj = this.getObservable();
  }
}
```

src/app/app.component.html

```
<h2>Pipes: AsyncPipe</h2>
<p>PromiseData: {{ promiseObj | async }}</p>
<p>ObservableData: {{ observableObj| async }}</p>
```

Como podemos observar, a AsyncPipe le podemos pasar directamente los objetos Promise y Observable, por lo que ya no es necesario extraer los valores que devuelven y exponerlos en variables para que puedan ser mostrados en el template mediante interpolación (data binding). Por otra parte, tampoco hace falta hacer el "subscribe" y "unsubscribe" al objeto Observable. AsyncPipe se encarga de hacerlo por nosotros.

036 Pipes: JsonPipe

JsonPipe es otro pipe de tipo impuro. Convierte un objeto a cadena JSON. Se usa de la siguiente manera:

expression | json

JsonPipe se usa en multitudes de situaciones, sin embargo, cabe destacar su uso en procesos de depuración de código, ya que permite analizar el contenido de cualquier objeto para comprender el comportamiento del código y buscar posibles errores.

Importante

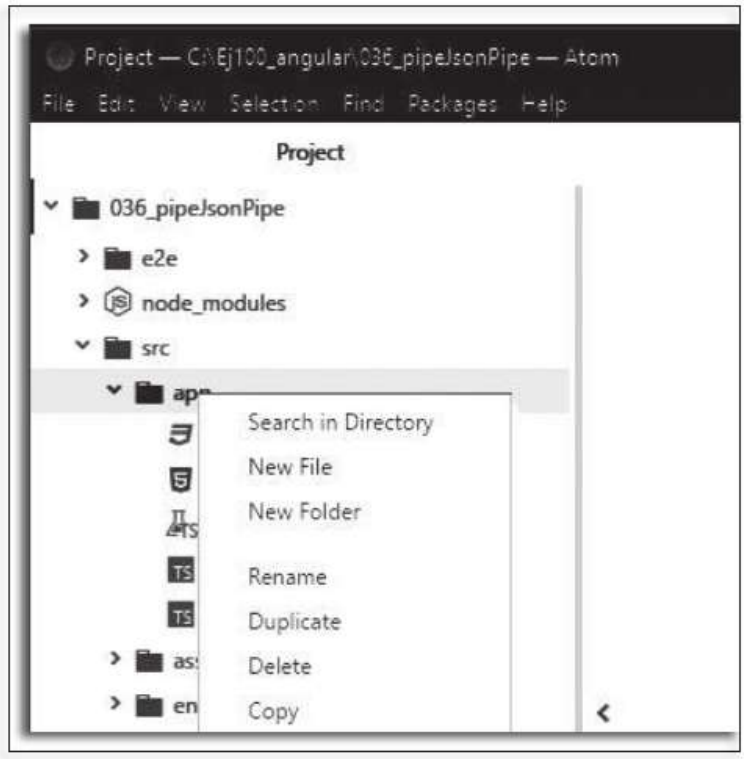
JsonPipe permite transformar un objeto a cadena JSON. Esto puede ser muy útil en tareas de depuración de código.

En este ejercicio crearemos una aplicación que nos ayudará a entender el funcionamiento de JsonPipe. La aplicación tendrá dos clases, **Libro** y **Escritor**, a partir de las cuales se crearán diversos objetos. Luego, la misma aplicación mostrará por el navegador el contenido de todos estos objetos usando JsonPipe.

1. Primero de todo nos ubicaremos en **ej100_angular**, y crearemos el proyecto tecleando **ng new pipeJsonPipe**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio ejecutando **rename pipeJsonPipe 036_pipeJsonPipe**, y finalmente pondremos en marcha la aplicación ejecutando **ng serve** desde la carpeta **036_pipeJsonPipe**.

```
C:\Ej100_angular>
C:\Ej100_angular>
C:\Ej100_angular>ng new pipeJsonPipe
  create pipeJsonPipe/e2e/app.e2e-spec.ts (296 bytes)
  create pipeJsonPipe/e2e/app.po.ts (208 bytes)
  create pipeJsonPipe/e2e/tsconfig.e2e.json (235 bytes)
  create pipeJsonPipe/karma.conf.js (923 bytes)
  create pipeJsonPipe/package.json (1319 bytes)
  create pipeJsonPipe/protractor.conf.js (722 bytes)
  create pipeJsonPipe/README.md (1103 bytes)
  create pipeJsonPipe/tsconfig.json (363 bytes)
  create pipeJsonPipe/tslint.json (3040 bytes)
```

2. Ahora abriremos el proyecto con **Atom**, y crearemos las clases **Libro** y **Escritor**. Para ello, nos situaremos sobre la carpeta **app**, pulsaremos botón derecho, y seleccionaremos "New File". La clase **Libro** la crearemos como "libro.ts" y la clase **Escritor** como "escritor.ts".



3. Una vez creados los ficheros de las clases, añadiremos su código:

src/app/libro.ts

```
export class Libro {

  constructor(public titulo:string, public tematica:string) {

  }

}
```

src/app/escritor.ts

```
import {Libro} from './libro';


export class Escritor {

    constructor(public id:number, public nombre:string,  public
   fecha:Date, public libros:Libro[]) {

    }

}
```

Seguidamente crearemos los objetos en el componente. Para ello, añadiremos el siguiente código:

src/app/app.component.ts

```
…
import {Escritor} from './escritor';
import {Libro} from './libro';


@Component({… })
export class AppComponent {


  libro1: Libro = new Libro('Mucho ruido y pocas nueces', 'Comedia');
  libro2: Libro = new Libro('Romeo y Julieta', 'Drama');
  librosArray: Libro[] = [this.libro1, this.libro2];


  escritor = new Escritor (1, 'William Shakespeare', new Date(1564,
    3, 26), this.librosArray);
}
```

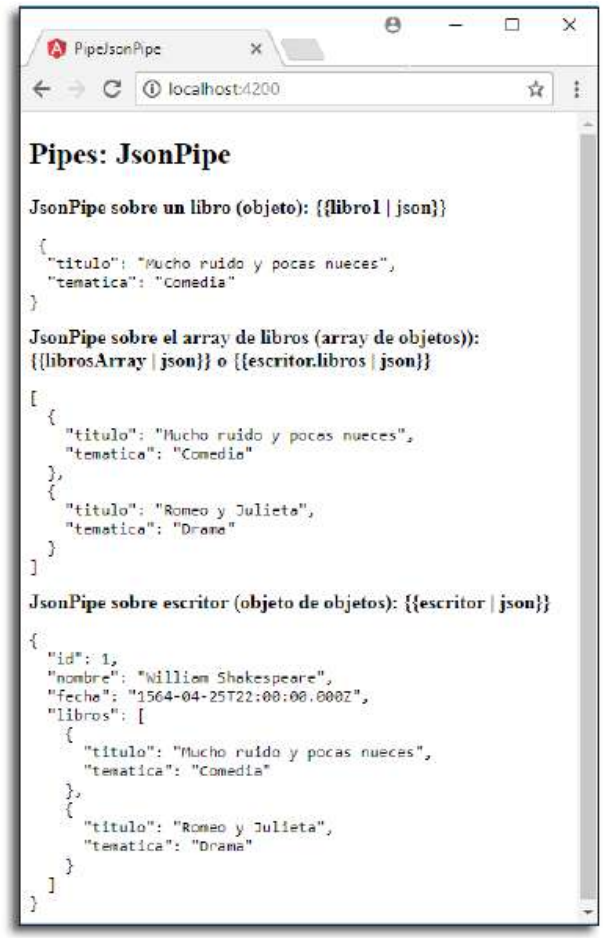
Y, finalmente, añadiremos el código en el template que nos permitirá visualizar el contenido de todos esos objetos. Usaremos JsonPipe para visualizar el contenido de un objeto (Libro), de un array de objetos (Libros) y de un objeto de objetos (Escritor).

src/app/app.component.html

```
<h2>Pipes: JsonPipe</h2>
<b ngNonBindable>JsonPipe sobre un libro (objeto): {{libro1 |
    json}}</b>
<pre> {{libro1 | json}} </pre>
<b ngNonBindable>JsonPipe sobre el array de libros (array de
    objetos)): {{librosArray | json}} o {{escritor.libros | json}}</
    b>
<pre>{{escritor.libros | json}}</pre>
<b ngNonBindable>JsonPipe sobre escritor (objeto de objetos):
    {{escritor | json}}</b>
<pre>{{escritor | json}}</pre>
```

En el anterior código, cabe destacar el uso de la directiva **ngNonBindable**. Esta directiva le indica a Angular que no compile el contenido del tag. Esto nos permite mostrar por pantalla el literal "{{…}}" sin que sea procesado por Angular. Por otra parte, hay que destacar el uso del tag **<pre>**, que nos permite visualizar la cadena JSON con saltos de línea para facilitar su análisis.

4. Desde el navegador podremos ver el resultado de todo el código anterior



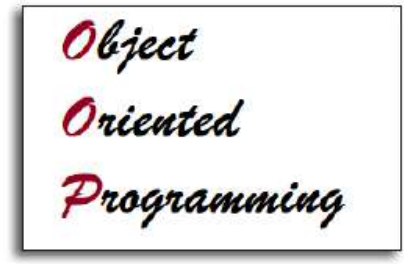
037 Modelos de datos y mock data (datos simulados) (parte I)

Tal como habíamos visto en el capítulo "005 TypeScript. Introducción…", el lenguaje de programación **TypeScript** nos permite realizar una programación JavaScript **orientada a objetos** con código simple y limpio. Esto significa que podremos aplicar una programación orientada a objetos en nuestras aplicaciones Angular y aprovecharnos de todas la ventajas que esto supone.

Importante

La programación orientada a objetos aplicada en nuestras aplicaciones Angular mejorará su reusabilidad, mantenimiento y fiabilidad.

De forma resumida podríamos definir la **programación orientada a objetos** como un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representan una instancia de alguna clase. Las clases representan entidades o conceptos. Cada una de ellas es un modelo que define un conjunto de variables y métodos. Además, todas ellas son miembros de una jerarquía de clases unidas mediante relaciones de herencia.



Algunas de las ventajas de la programación orientada a objetos son:

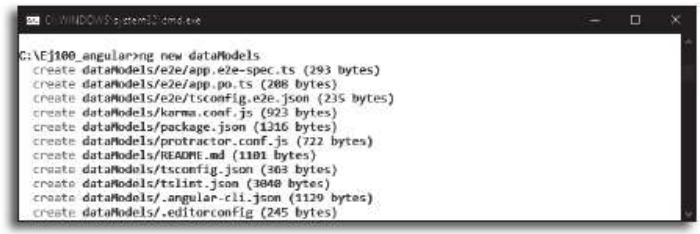
- **Reusabilidad.** El diseño adecuado de las clases permite que se puedan usar en distintas partes del programa y en otros proyectos.
- **Mantenimiento.** Los programas orientados a objetos son más sencillos de leer y comprender, ya que permiten ocultar detalles de implementación dejando visibles solo aquellos detalles más relevantes. Todo esto facilita el trabajo de realizar modificaciones y detectar posibles problemas.
- **Fiabilidad.** Al dividir el problema en partes más pequeñas permite que podamos probarlas de manera independiente y aislar fácilmente los posibles errores que puedan surgir.

En este ejercicio veremos cómo enfocar un programa Angular a la programación orientada a objetos trabajando con **modelos de datos**.

Primero crearemos una aplicación Angular con un único componente Biblioteca que visualizará el contenido de una variable interna de tipo array con distintos elementos que serían los libros.

En la segunda parte de este ejercicio modificaremos la aplicación creando el modelo de datos Libro, reestructurando el código y el **mock data** (datos simulados), y comentando las ventajas de esos cambios.

1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos el proyecto tecleando **ng new dataModels**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio, ejecutando **rename dataModels 037_dataModels** y, finalmente, pondremos en marcha la aplicación ejecutando **ng serve** desde la carpeta **037_dataModels**.



2. Ahora crearemos el componente **biblioteca** mediante el comando siguiente:

```
C:\Ej100_angular\037_dataModels>ng generate component biblioteca
```

Seguidamente abriremos el proyecto con Atom y editaremos el componente biblioteca añadiendo la variable con los libros:

src/app/biblioteca/biblioteca.component.ts

```
…
export class BibliotecaComponent implements OnInit {

  libros = [{
    "id": 1,
    "titulo": "El Quijote",
    "autor": "Cervantes"
  },{
    "id": 2,
    "titulo": "Hamlet",
    "autor": "Shakespeare"
  }];
…
```

El siguiente paso será modificar el template del componente biblioteca para visualizar el contenido de la variable anterior. Para ello abriremos **biblio-**

teca.component.html y sustituiremos todo el código que tenga por el siguiente:

src/app/biblioteca/biblioteca.component.html

```
<h1>Biblioteca:</h1>
<ul>
  <li *ngFor="let libro of libros">
    <h2>{{libro.titulo | uppercase}}</h2>
    <p class="description">{{libro.autor}}</p>
  </li>
</ul>
```

En este código aparecen ciertos elementos de Angular como son las directivas ("*ngFor") y los pipes ("...libro.titulo | uppercase..."). Para más información, consulte los capítulos dedicados a las directivas y a los pipes.

3. Finalmente, solo faltará que nuestra aplicación cargue el componente biblioteca en la página principal. Para ello, simplemente añadiremos la referencia a nuestro componente en el template del componente principal de la aplicación: **app.component.html**. Sustituiremos su contenido por el siguiente:

```
<app-biblioteca></app-biblioteca>
```

4. En este punto, desde el navegador (**http://localhost:4200**) ya podremos ver que nuestra aplicación muestra los libros de la biblioteca.

038 Modelos de datos y mock data (datos simulados) (parte II)

Vamos a continuar con la aplicación que habíamos creado en el anterior ejercicio.

En este ejercicio crearemos el **modelo de datos Libro** añadiendo las modificaciones necesarias, reestructurando el código y **mock data** (datos simulados), comentando las ventajas de esos cambios.

Importante

El uso de variables tipadas (con tipo de datos definido) de TypeScript aporta mayor veracidad al código. Permite la detección de errores en el tiempo de compilación.

1. En primer lugar, abriremos el proyecto 037_dataModels con el editor (**Atom**) y pondremos en marcha la aplicación ejecutando **ng serve** desde línea de comandos y dentro de la carpeta **037_dataModels**.

2. Ahora crearemos el modelo de datos **Libro**. Desde el editor Atom nos situaremos en la carpeta **app**, pulsaremos el botón derecho del ratón, seleccionaremos New File, y crearemos el fichero con el nombre **libro.model.ts**. Dentro del nuevo fichero añadiremos el siguiente código:

src/app/libro.model.ts

```
export class Libro {
  id: number;
  titulo: string;
  autor: string;
}
```

Como vemos, un modelo de datos básicamente es una **clase en JavaScript**.

También podemos observar cómo, gracias a **TypeScript**, podemos definir el tipo (number, string, etc.) de cada una de las distintas propiedades del modelo. Esto permitirá al compilador validar nuestro código y asegurarse de que usamos esas propiedades de manera adecuada.

3. A continuación, modificaremos el componente biblioteca para que haga uso del nuevo modelo de datos Libro. El cambio será muy sencillo, simplemente

añadiremos la referencia al modelo de datos Libro, y le diremos a TypeScript que trate la variable libros como una cadena de Libros:

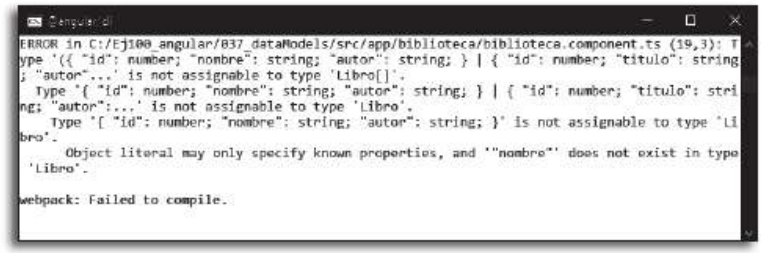
src/app/biblioteca/biblioteca.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Libro } from '../libro.model';
…
export class BibliotecaComponent implements OnInit {

  libros: Libro[] = [{
    "id": 1,
    "titulo": "El Quijote",
…
```

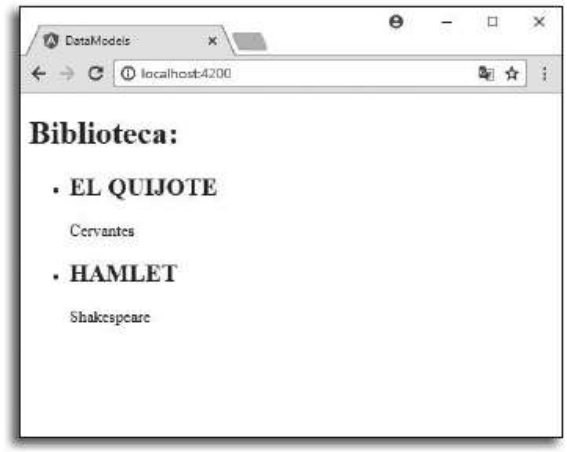
Como hemos comentado en el punto anterior, al definir un tipo para la variable libros (en este caso, cadena de Libros), el compilador es capaz de validar que se esté usando esta variable de forma correcta. Si, por ejemplo, modificáramos la propiedad "titulo" por "nombre", de alguno de los libros el compilador daría error.

src/app/biblioteca/biblioteca.component.ts

```
…
  libros: Libro[] = [{
    "id": 1,
    "nombre": "El Quijote",
…
```

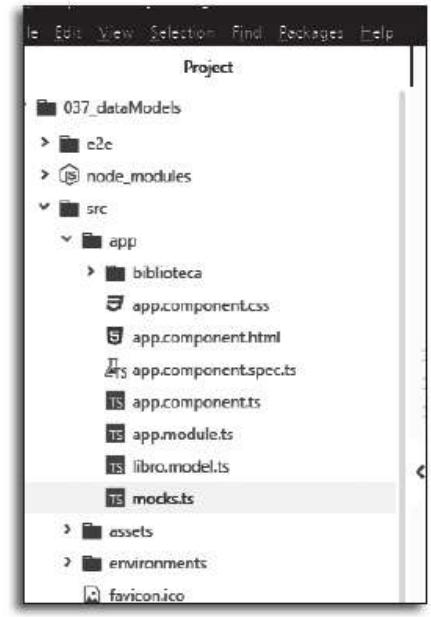


4. Desde el navegador (http://localhost:4200) podremos verificar que todo funciona.



Normalmente, una aplicación Angular haría una llamada a un web service (API) para obtener los datos que necesita que, en el caso de nuestra aplicación, serían los libros. Por lo tanto, es una buena práctica **quitar los datos de prueba (o mock data) de nuestros modelos y componentes** y dejarlos en ficheros aparte. Siguiendo esta recomendación, vamos a realizar cambios en nuestra aplicación.

5. Desde el editor Atom, nos situaremos en la carpeta **app**, pulsaremos el botón derecho del ratón, seleccionaremos New File, y crearemos el fichero **mocks.ts**. Dentro, añadiremos lo siguiente:



src/app/mocks.ts

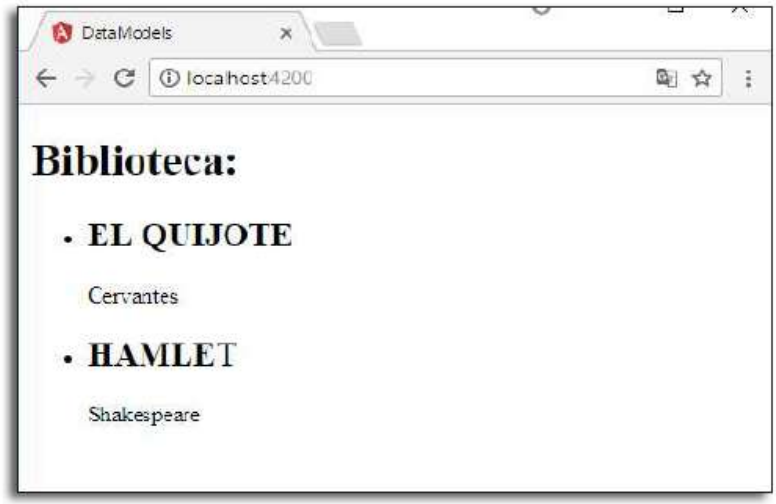
```
import { Libro } from './libro.model';

export const LIBROS: Libro[] = [{
  "id": 1,
  "titulo": "El Quijote",
  "autor": "Cervantes"
},{
  "id": 2,
  "titulo": "Hamlet",
  "autor": "Shakespeare"
}];
```

6. Finalmente modificaremos el componente biblioteca para que cargue los datos mock desde ese fichero:

src/app/biblioteca/biblioteca.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Libro } from '../libro.model';
import { LIBROS } from '../mocks';
…
export class BibliotecaComponent implements OnInit {

  libros: Libro[];
…
  ngOnInit() {
    this.libros = LIBROS;
  }
}
```

7. Desde el navegador (http://localhost:4200) podremos verificar que todo funciona bien.



039 Librerías. Enumeración de librerías

Al igual que en otros lenguajes, las librerías, entre otras muchas ventajas, nos permiten organizar el código y reutilizarlo en otras aplicaciones. En este sentido, **Angular** utiliza un conjunto de librerías "de serie" que aportan funcionalidad en diferentes aspectos como veremos a continuación, pero, además, facilita el uso de otras librerías de terceros que pueden aportarnos soluciones que simplemente hemos de importar y utilizar.

Importante

Importe solo las librerías que necesite para optimizar el código y hacer que la aplicación sea más eficiente al cargar solo los recursos que se vayan a utilizar.

Las librerías más importantes de Angular y su cometido son las siguientes:

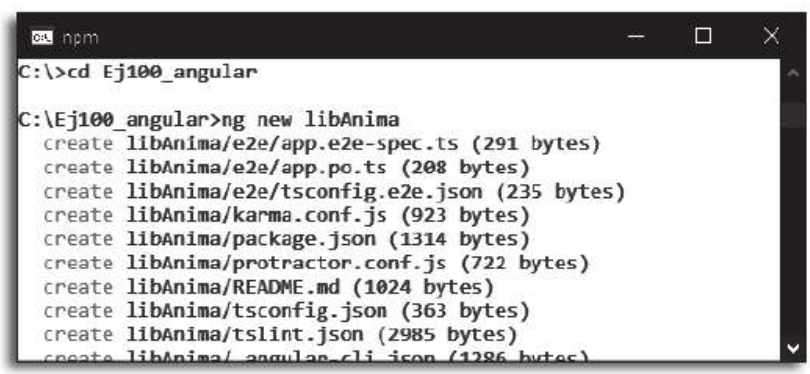
@angular/animations	Permite realizar animaciones similares a las realizadas con CSS (efectos, transiciones entre páginas, etc.) integrando la lógica de animación con el resto de códigos de la aplicación.
@angular/cli	Permite trabajar con **Angular-CLI** inicializando, desarrollando y manteniendo aplicaciones Angular a partir de la creación de componentes, directivas, etc.
@angular/common	Proporciona **directivas**, **pipes** y **servicios** comúnmente utilizados.
@angular/compiler	Librería del compilador que permite convertir nuestro código y nuestras plantillas antes de ejecutar y representar la aplicación.
@angular/compiler_cli	Librería del compilador para **Node.js** invocada por los comandos **build** y **serve** de **Angular CLI**.
@angular/core	Es el principal repositorio de fuentes de **Angular**. Incluye todos los decoradores de **metadatos**, **componentes**, **directivas**, **inyección de dependencia**, etc. Imprescindibles para la creación de aplicaciones.

@angular/forms	Proporciona **directivas** y **servicios** para crear **forms** tanto **template-driven** como **reactive-driven** (vistos más adelante).
@angular/http	Permite usar **http**.
@angular/language-service	Permite analizar las plantillas de los componentes proporcionando sugerencias, chequeo de sintaxis, chequeo de errores, etc., detectando automáticamente el tipo de archivo tratado.
@angular/platform-browser	Librería usada para navegadores web que ayuda a procesar el **DOM** e incluye el método **bootstrapStatic()** para arranque de aplicaciones precompiladas con **AOT**.
@angular/platform-browser-dynamic	Librería usada para navegadores web que incluye proveedores y métodos para compilar y ejecutar la aplicación en el cliente utilizando el compilador **JIT** y que contiene el código del lado del cliente procesando las plantillas (enlaces, componentes, etc.), además de la inyección de dependencia reflexiva.
@angular/router	Librería que permite navegar entre páginas de la aplicación al cambiar el URL del navegador.

A lo largo del libro, tendremos la oportunidad de trabajar con diversas librerías. En este ejercicio, vamos a realizar un ejemplo sencillo del uso de animaciones como excusa para utilizar la librería de **@angular/animations**.

1. En primer lugar, nos ubicaremos en **ej100_angular** y crearemos el proyecto **libAnima** mediante el comando **ng new**:

```
C:\Ej100_angular>ng new libAnima
```

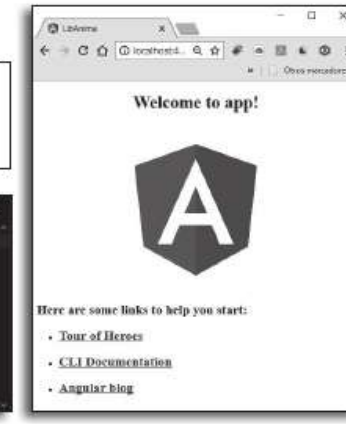


Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename libAnima 039_libAnima
```

A continuación, nos ubicaremos en **039_libAnima** y arrancaremos la aplicación para comprobar cómo funciona en nuestro navegador. Para ello, teclearemos lo siguiente:

```
C:\Ej100_angular>cd 039_libAnima

C:\Ej100_angular\039_libAnima>ng serve
```



Ahora, abrimos el proyecto con nuestro editor (en nuestro caso, usamos **Atom**) y modificamos el archivo **app.component.html** para que incluya un div denominado **caja**, el evento click que llame a la función **cambioEstado()** y un **trigger** asociado a la variable **estado**.

```
<div class="container">
  <h1>{{title}}</h1>
  <div class="container" id="caja"
       (click)="cambioEstado()"
       [@estado]="estado">
    <h2>JAG</h2>
  </div>
</div>
```

2. Seguidamente, modificaremos el archivo **app.module.ts** para importar el módulo **BrowserAnimationsModule**.

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

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

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

3. A continuación, modificaremos el archivo **app.component.ts** para importar las siguientes funciones contenidas en la librería **@angular/animations**:

```
import { trigger, state, style, animate, transition } from '@
    angular/animations';
```

Por otra parte, definiremos un **trigger** denominado **estado** que contempla dos estados: **activo** e **inactivo**. Por tanto, modificaremos el decorador **@Component** para definir la animación y la clase **AppComponent** para poner el título"'**039 libAnima**", definir la variable **estado** y definir la función **cambioEstado()** que permita alternar el valor de la variable estado entre los valores **activo** e **inactivo**. Como puede observarse, cada estado define un **color** de **fondo**, un **color** para el **texto** y una **escala** (p. ej., **scale(1)**) además de un **efecto** en la **transición** entre **estados** (**ease-in**, **ease-out**) que se realiza durante un **tiempo** determinado (**1.000 ms**).

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  animations: [
    trigger('estado', [
      state('inactivo', style({
        backgroundColor: '#FFFF00',
        transform: 'scale(1)',
        color:'blue'
      })),
      state('activo',   style({
        backgroundColor: '#013ADF',
        transform: 'scale(2.0)',
        color:'white'
      })),
      transition('inactivo => activo', animate('1000ms ease-in')),
      transition('activo => inactivo', animate('1000ms ease-out'))
    ])
  ]
})
```

```
export class AppComponent {
  title = '039 libAnima';
  estado:string='inactivo';
  cambioEstado(){
    this.estado = ( this.estado == "activo" ) ? 'inactivo' : 'activo';
  }
}
```

4. Por último, definiremos los estilos que usará nuestro **div** modificando el fichero **styles.css**.

```
#caja {
  width: 50px;
  height: 50px;
  margin: 40px;
  padding:15px;
  display: block;
  text-align: center;
  border-style: double;
  border-radius: 20px;
}
```

5. Guárdelo todo y abra su navegador para teclear el URL http://localhost:4200/ y comprobar el resultado.

6. Haga clic sobre el **div** y observe cómo cambia de **tamaño** y **color**.

040 One Way Data Binding (hacia el DOM): Interpolación, Property Binding y Class Binding

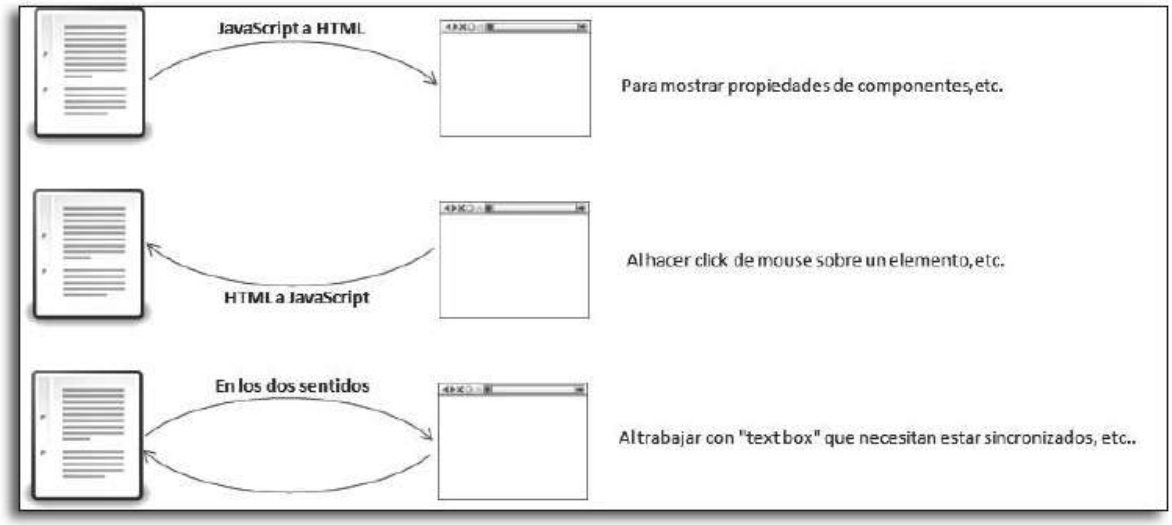
Cuando un navegador web carga un documento (vista), genera una estructura de objetos que llamamos DOM. El DOM puede alterarse mediante JavaScript para cambiar dinámicamente los contenidos y el aspecto de la página.

El Data Binding es un concepto primordial en Angular. Permite definir la comunicación entre un componente y el DOM, lo que hace que sea muy fácil definir aplicaciones interactivas sin preocuparnos de la actualización de datos en DOM (push) o la obtención de datos desde el DOM (pull).

Importante

Interpolación, Property Binding y Class Binding son distintos mecanismos de comunicación de datos desde JavaScript (TypeScript compilado) hacia HTML, pero no para el sentido contrario.

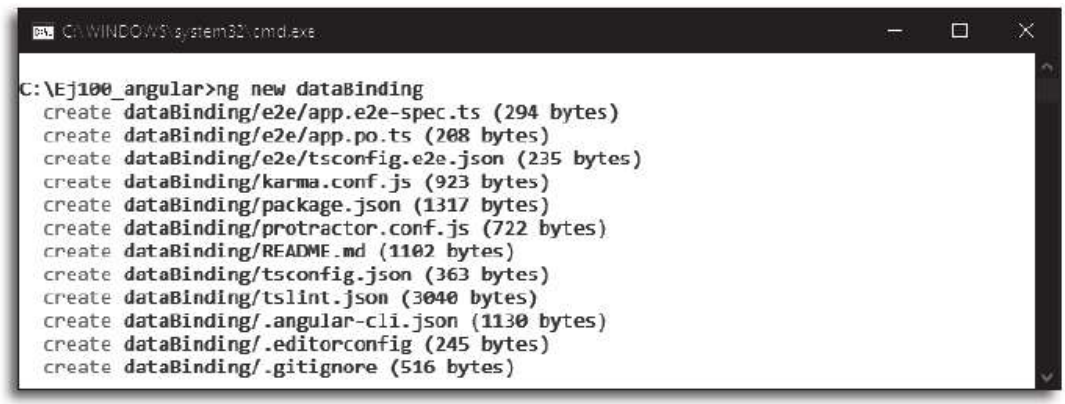
Hay tres tipos distintos de Data Binding que difieren en la forma que fluyen los datos. Y, para cada uno de ellos, Angular ofrece distintos mecanismos:



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

En este ejercicio y en los dos siguientes, veremos ejemplos de todos ellos. Crearemos una aplicación que mostrará un artículo con una disponibilidad determinada de stock, y una serie de opciones para añadir/quitar unidades del "carrito de compra". Esta aplicación se irá desarrollando a lo largo de estos tres ejercicios haciendo uso de todos estos mecanismos.

1. Nos ubicaremos en **ej100_angular**, y crearemos el proyecto tecleando **ng new dataBinding**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio ejecutando **rename dataBinding 040_dataBinding**, y finalmente pondremos en marcha la aplicación ejecutando **ng serve** desde dentro la carpeta **040_dataBinding**. A continuación, abriremos el proyecto con **Atom**.



2. Seguidamente añadiremos los links de **Bootstrap** (https://v4-alpha.getbootstrap.com) para poder hacer uso de sus hojas de estilo. Consulte el ejercicio "087 Bootstrap Documentacion" para más información.

3. La **interpolación** es la forma más sencilla de visualizar propiedades de un componente en su template. Solamente hace falta poner el nombre de la propiedad entre **{{ … }}** en el template.

   Mediante el siguiente código, crearemos nuestro artículo en el componente principal de la aplicación, y con interpolación, mostraremos sus datos en el template. La imagen referenciada en el artículo, "assets/old-books.jpg", la encontrará en las fuentes del ejercicio. Debe dejarla en la carpeta **assets** del proyecto.

src/app/app.component.ts

```
…
export class AppComponent {
  Libro = {"titulo": "Hamlet", "autor": "William Shakespeare",
"precio": 21.30, "stock": 5, "cantidad": 0, "imagen": "assets/old-
books.jpg"};
}
```

src/app/app.component.html

```
<div class="container-fluid">
    <div class="card" style="width: 20rem;">
      <div class="card-block">
        <h4 class="card-title">{{Libro.titulo}}</h4>
        <h6 class="card-subtitle mb-2 text-muted">{{Libro.autor}}</h6>
      </div>
      <ul class="list-group list-group-flush">
        <li class="list-group-item">Unidades disponibles: {{Libro.
stock}}</li>
        <li class="list-group-item">Precio: {{Libro.precio |
currency:'EUR':true}}</li>
      </ul>
    </div>
  </div>
```

4. Cuando necesitemos enlazar propiedades de componente a propiedades de elementos del DOM podemos usar **Property Binding** como alternativa a la interpolación (que también podría usarse en estos casos). Para usarlo pondremos la propiedad del elemento del DOM entre "[...]" asignándole la propiedad del componente.

   Continuando nuestro ejercicio, haremos uso de Property Binding para visualizar la imagen del artículo:

src/app/app.component.html

```
…
<img class="card-img-top img-fluid" [src]="Libro.imagen"
height="122" alt="Imagen no encontrada">
<div class="card-block">
…
```

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

   Para terminar el ejercicio, haremos uso de Class Binding para añadir la clase **CSS** "aviso" (poner letras en rojo) al indicador de "unidades disponibles" en el caso de que no haya stock. Para ello, añadiremos:

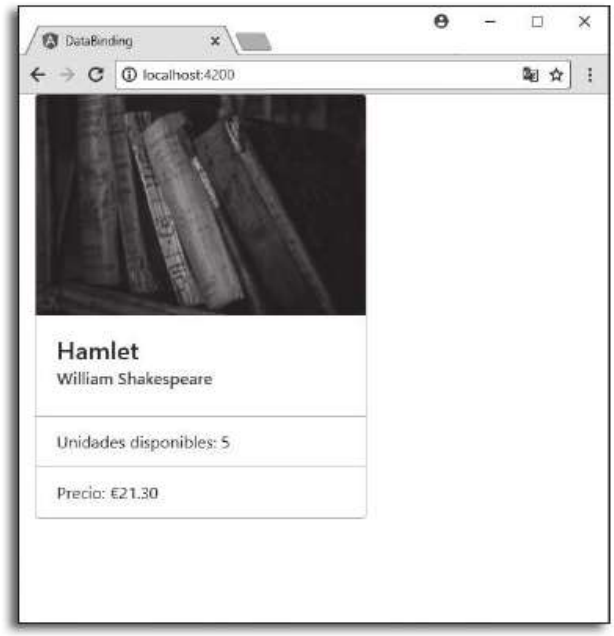
src/app/app.component.css

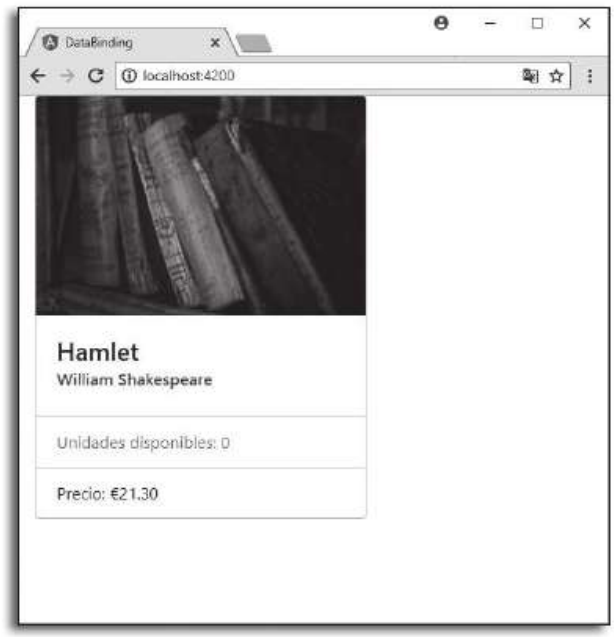
```
.aviso{ color: red; }
```

src/app/app.component.html

```
…
<li class="list-group-item" [class.aviso]="Libro.
stock==0">Unidades disponibles: {{Libro.stock}}</li>
…
```

6. Desde el navegador (http://localhost:4200/) podrá ver el estado de nuestra aplicación. Puede cambiar valores del artículo, incluido poner a 0 "Libro. enStock", y comprobar cómo se actualiza la vista.





041 One Way Data Binding (desde el DOM): Event Binding y $event

En este ejercicio analizaremos los mecanismos que ofrece Angular para el envío de datos del DOM al componente: Event Binding y $event. Son mecanismos de comunicación de un solo sentido, de la vista al componente, por eso los llamamos mecanismos de One Way Data Binding.

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

Importante

Con Event Binding podemos gestionar cualquier evento estándar del DOM desde nuestros componentes. Con el uso de $event podemos tener acceso al objeto event para tener más detalle del evento que queramos gestionar.

```
<div (mouseover)="funcionX()">
<input type="text" (keydown)="funcionY()">
<form (submit)="funcionZ()">
```

Algunas veces necesitaremos información adicional en relación al evento que estamos controlando, por lo que necesitaremos tener acceso al objeto event que se haya producido. Este procedimiento lo haremos pasando el objeto event al método de nuestro componente con **$event**. Veamos un ejemplo de cómo usarlo en el caso que quisiéramos saber qué tecla se ha pulsado al producirse un evento de tecla pulsada (Keydown):

```
<input type="text" (keydown)="funcionY($event)">

funcionY(event) {  console.log("Tecla pulsada: " + event.keyCode); }
```

Continuando la aplicación que habíamos empezado en el anterior ejercicio, le añadiremos las opciones para añadir/quitar unidades del "carrito de compra" utilizando los mecanismos Event Binding y $event.

1. Desde línea de comandos nos situaremos en **ej100_angular/040_dataBinding**, y ejecutaremos la aplicación con **ng serve**. A continuación, abriremos el proyecto con **Atom**.

2. En el template del componente añadiremos los botones de añadir/quitar unidades del "carrito de compra" junto a un visor con la cantidad de unidades

que tenemos en el carrito. Mediante **Event Binding** gestionaremos el evento click sobre los botones. Este es el código que hay que añadir:

src/app/app.component.ts

```
export class AppComponent {
…
    downCantidad(libro){
      if (libro.cantidad > 0 ) libro.cantidad--;
    }
    upCantidad(libro){
      if (libro.cantidad < libro.stock ) libro.cantidad++;
    }
}
```

src/app/app.component.html

```
…
     </ul>
     <div class="text-center">
       <button (click)="downCantidad(Libro)">-</button>
       {{Libro.cantidad}}
       <button (click)="upCantidad(Libro)">+</button>
     </div>
…
```

Como vemos en el navegador, los botones de quitar/añadir unidades funcionan correctamente. Sin embargo, podemos añadir una mejora. Podemos hacer uso de **Property Binding**, visto en el anterior capítulo, para deshabilitar los botones cuando al hacer clic sobre ellos no tenga ningún efecto debido a las restricciones que hemos puesto en *downCantidad(libro)* y *upCantidad(libro)*. Esto lo realizaremos añadiendo el siguiente código:

src/app/app.component.html

```
<button [disabled]="Libro.cantidad<=0" (click)="downCantidad(Libro)">
-</button>
{{Libro.cantidad}}
<button [disabled]="Libro.cantidad>=Libro.stock"
(click)="upCantidad(Libro)">+</button>
```

Finalmente, para ver un ejemplo de uso de **$event** junto a Event Binding, capturaremos el evento de movimiento de ratón sobre la imagen e indicaremos por log en qué posición se encuentra el ratón (recuerde que para ver los logs debe hacer clic sobre "Más Herramientas/Herramientas para desarrolladores" del menú de Google Chrome). Para ello, añadiremos el siguiente código:

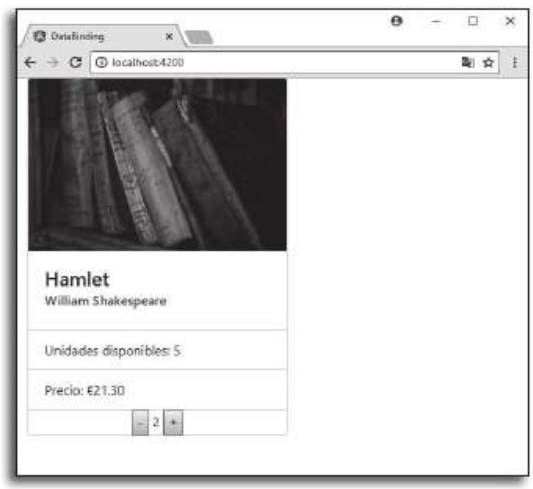
src/app/app.component.ts

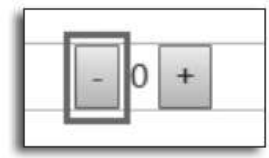
```
export class AppComponent {
…
  getCoord(event) {  console.log(event.clientX + ", " + event.
clientY); }
}
```

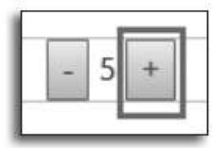
src/app/app.component.html

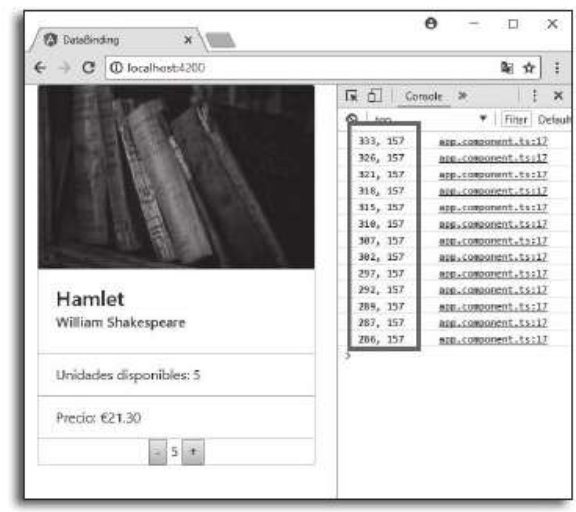
```
<img class="card-img-top img-fluid" [src]="Libro.imagen"
(mousemove)="getCoord($event)" height="122" alt="Imagen no
encontrada">
```

Desde el navegador (http://localhost:4200/) podrá realizar pruebas de la aplicación. Pulsando los botones podrá comprobar cómo se quitan/añaden unidades en el "carrito", podrá comprobar cómo se deshabilitan los botones cuando la cantidad es 0 o cuando se llega al stock disponible y por último podrá ver la posición del ratón cuando pase el puntero por encima de la imagen.









042 Two Way Data Binding (hacia-desde el DOM): FormsModule y [(ngModel)]

Hay ciertas situaciones en las que necesitaremos comunicación entre componente y DOM en los dos sentidos al mismo tiempo. Por ejemplo, este sería el caso de un elemento "input" del DOM. Por una parte, necesitaríamos que se actualizara con el valor de una determinada propiedad del componente y, por la otra, necesitaríamos que escuchara eventos (introducción de datos por parte del usuario) actualizando el valor de esa propiedad.

Importante

Con [(ngModel)] podemos realizar Two Way Data Binding, cosa que significa que una propiedad de componente se mantendrá sincronizada tanto si se modifica desde dentro del componente (JavaScript) como si se hace desde la vista (HTML).

Combinando los mecanismos de One Way Data Binding que hemos visto anteriormente podríamos conseguir ese resultado. Veamos cómo podríamos conseguirlo combinando Property Binding y Event Binding en un elemento "input" destinado a la entrada de un nombre de usuario:

```
<input type="text" [value]="username"
                   (input)="username = $event.target.value">
```

Sin embargo, Angular nos ofrece una mejor manera de hacerlo. Se trata de **ngModel**. Veamos cómo quedaría el código anterior con ngModel:

```
<input type="text" [(ngModel)]="username">
```

Como vemos, su sintaxis es mucho más sencilla y clara. Únicamente hay que tener en cuenta que deberemos importar a nuestro proyecto el módulo FormsModule para poder hacer uso de ngModel.

Por otra parte, hay que tener en cuenta que, a diferencia de los mecanismos de One Way Data Binding, ngModel solo acepta propiedades de datos de componente. Si lo inicializáramos con una función de componente (p. ej.: [(ngModel)]="nombreCompleto()") se produciría un error.

Para continuar la aplicación que habíamos empezado en los anteriores ejercicios, vamos a ver cómo usaríamos ngModel al añadir un nuevo elemento "input" para poder introducir directamente la cantidad de unidades del artículo que deseemos.

1. Desde línea de comandos nos situaremos en ej100_angular/040_dataBinding, y ejecutaremos la aplicación con ng serve. A continuación, abriremos el proyecto con Atom.

2. Abriremos app.module.ts e importaremos el modulo FormsModule. También será necesario ponerlo en "imports:[..]" de @NgModule para que pueda usarse en toda la aplicación.

src/app/app.module.ts

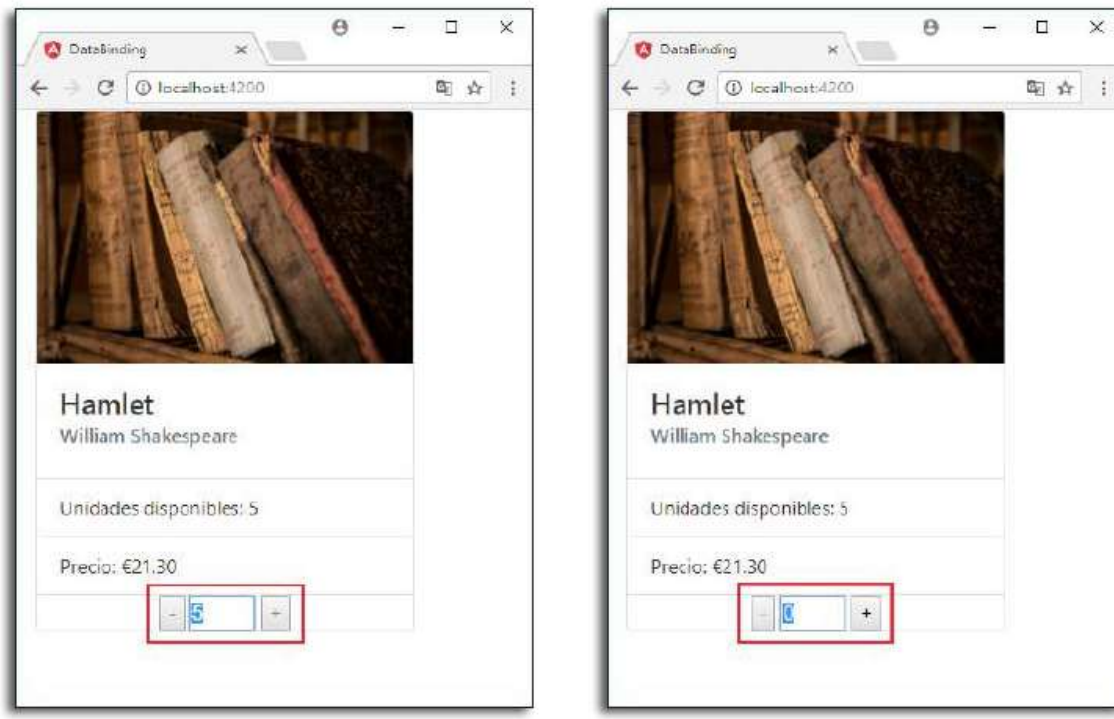
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';


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


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

A continuación, sustituiremos el visor de cantidad "{{Libro.cantidad}}" que teníamos en la vista por un elemento "input" que nos permita introducir la cantidad manualmente. Este elemento "input" lo vincularemos a la propiedad "Libro.cantidad" mediante **[(ngModel)]**. Esto lo realizaremos añadiendo el siguiente código:

src/app/app.component.html

```
…
<button [disabled]="Libro.cantidad<=0"
(click)="downCantidad(Libro)">-</button>
<input type="text" [(ngModel)]="Libro.cantidad" size="2">
<button [disabled]="Libro.cantidad>=Libro.stock"
(click)="upCantidad(Libro)">+</button>
…
```

Desde el navegador (http://localhost:4200/) podrá comprobar cómo se produce el *Two Way Data Binding*, o sea, la sincronización de datos en los dos sentidos, entre la propiedad "Libro.cantidad" del componente y el nuevo campo cantidad de la vista:



a. Si a través del elemento "input" modifica la cantidad a un valor igual o superior al stock (5), verá como se deshabilita el botón de incrementar cantidad que se enlaza directamente con la propiedad "Libro.cantidad" del componente. Lo mismo pasará con el botón decrementar cantidad si introduce un valor igual o inferior a 0.
b. Por otra parte, si usa los botones de decrementar/incrementar cantidad verá como el elemento "input" muestra el nuevo valor de la propiedad "Libro.cantidad" del componente.

043 Routing: Introducción y configuración básica (parte I)

Como todos sabemos, la navegación web se realiza mediante la siguiente serie de acciones:

- Si introducimos una dirección URL en la barra de direcciones del navegador, cambiaremos de página.
- Si hacemos clic en un vínculo de una página, cambiaremos de página.
- Si hacemos clic sobre los botones de avance y retroceso del navegador, navegaremos hacia atrás y hacia delante a través del historial de páginas que haya visto.

Importante

El servicio Router nos servirá para gestionar las rutas y sus vistas asociadas a nuestra aplicación.

Toda aplicación Web se adapta a esta forma de navegación a través de algún sistema para gestionar sus rutas (direcciones URL) y vistas asociadas (páginas HTML). En el caso de Angular, esta gestión la haremos con el servicio **Angular Router**.

El servicio Router no es accesible por defecto. Tiene su propio paquete sobre el cual deberemos realizar la siguiente importación:

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

Las rutas que gestionaremos con Angular Router serán relativas a una ruta raíz. Esta la configuraremos con **"<base href>"** dentro de la etiqueta "head" de "src/index.html". Para los proyectos creados con Angular cli, mediante "ng new ...", esta configuración ya está realizada. Se asigna "/" como ruta raíz.

index.html

```
1  <!doctype html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8">
5    <title>Routing</title>
6    <base href="/">
7
```

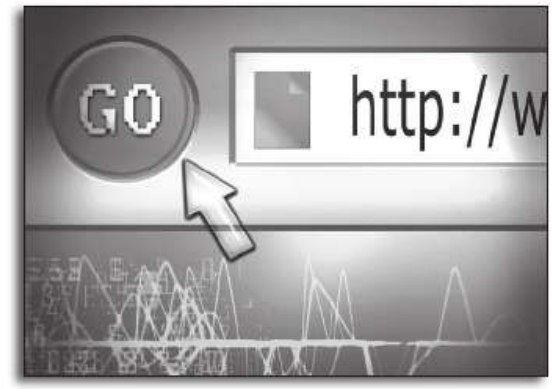
El servicio Router necesita que le indiquemos una **configuración de rutas** con la que trabajar. Esta la realizaremos en dos pasos:

1. Creación de la cadena de rutas donde indicaremos la asociación "ruta · componente":

```
const appRoutes: Routes = [

  { path: <path1>, component:<componente a mostrar> },

  { path: <path2>, component:<componente a mostrar> },

  ...,

  { path: '', redirectTo: <pathX>, pathMatch: 'full' },

  { path: '**', component:<componente a mostrar> }];
```

2. Asignación de la cadena de rutas al servicio Router mediante el método **RouterModule.forRoot(*Routes*)**.

La configuración indica qué componentes deben visualizarse por el navegador para distintos path de ruta. Y esto es lo que realiza el servicio Router. Cuando en la barra de direcciones del navegador entremos una dirección URL determinada, el servicio intentará hacer "match" de esa dirección con los path de rutas configuradas. Si lo consigue, mostrará el componente que corresponda.



Esto significa que para una situación como la siguiente, el servicio Router visualizaría el componente "UsuarioListaComponent".

Dirección URL	**Configuración de rutas**	**Componente seleccionado**
localhost:4200/ Usuario	const appRoutes: Routes = [{ path: 'Usuario', component:UsuarioListaComponent },…	*UsuarioListaComponent*

En la cadena de rutas de ejemplo que hemos visto antes, aparecen dos path con unas características particulares:

- '': El servicio Router seleccionará esta ruta cuando el URL este vacía (p. ej., "localhost:4200/"). Normalmente esta situación sucederá al acceder a la aplicación por primera vez.

- **'**'**: El servicio Router seleccionará esta ruta cuando el URL no coincida con ninguna configuración anterior.

Para redireccionar rutas haremos uso de **redirecTo**. En el ejemplo que veremos más adelante, se realizará una redirección a otra ruta en el caso que el URL esté vacío ("http://localhost:4200"). Cuando se usa el redirecTo, es obligatorio añadir un valor para pathMatch para indicar al servicio cómo debe realizar el "match" de un URL con el path de ruta.

El servicio Router siempre recorre la configuración de rutas de arriba abajo. Por lo tanto, es **importante el orden de las rutas en la configuración**. Siempre deberíamos poner las más específicas arriba y abajo las más generales. Esto implica que la configuración de path '**', por su significado, siempre tiene que estar en última posición, ya que el servicio Router ignorará cualquier otra configuración que haya por debajo.

Una vez el servicio Router haya seleccionado el componente adecuado, lo visualizará por pantalla. Concretamente lo situará donde se encuentre la etiqueta **router-outlet**:

```
<router-outlet></router-outlet>
```

app.component.html

```
1  <div class="container">
2
3  <router-outlet></router-outlet>
4
5  </div>
6
```

Con una configuración de rutas básica como la que hemos visto, solamente tendremos una etiqueta "router-outlet" en el template principal de la aplicación. Sin embargo, tal como veremos más adelante, podríamos tener rutas anidadas en la configuración y, en ese caso, tendríamos varias etiquetas router-outlet.

Como puede observar, la navegación por la aplicación consistiría en pasar de una vista a otra sin salir nunca de la página HTML que tuviera la etiqueta "router-outlet". Por tanto, realmente estaríamos navegando en una aplicación de una sola

página o SPA (single-page application), concepto del que habíamos hablado en el capítulo 02 “Introducción a las aplicaciones SPA”. Consulte ese capítulo si desea más información.

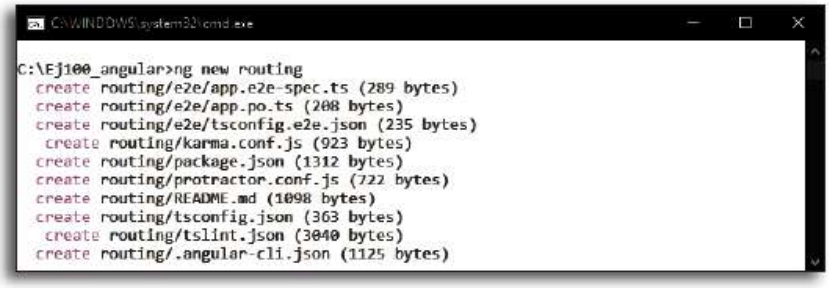
044 Routing: Introducción y configuración básica (parte II)

Una vez vista la introducción al servicio Router, vamos a crear una aplicación para poner en práctica todo lo aprendido. La aplicación consistirá en un sencillo gestor de Libros y Autores. Tendremos una vista libros y una para autores, y pasaremos de una a otra cambiando el URL de la barra de direcciones del navegador.

Importante

Las rutas de la configuración siempre deben ordenarse de más específicas a más genéricas. En este sentido, siempre debe situarse la configuración "**" (cualquier URL) al final de todo.

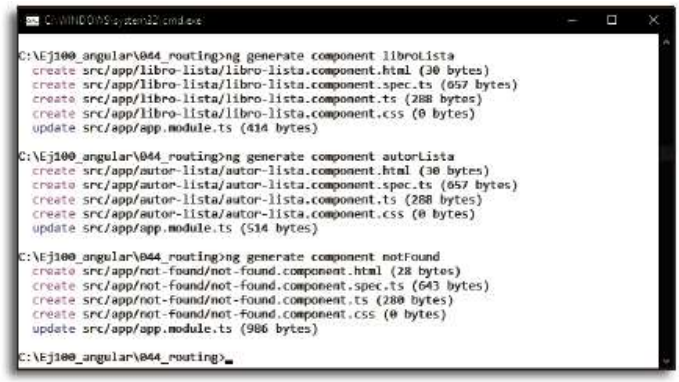
1. Nos ubicaremos en **ej100_angular**, y crearemos el proyecto tecleando **ng new routing**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio ejecutando **rename routing 044_routing**, y finalmente pondremos en marcha la aplicación ejecutando **ng serve** desde la carpeta **044_routing**. A continuación, abriremos el proyecto con el editor **Atom**.



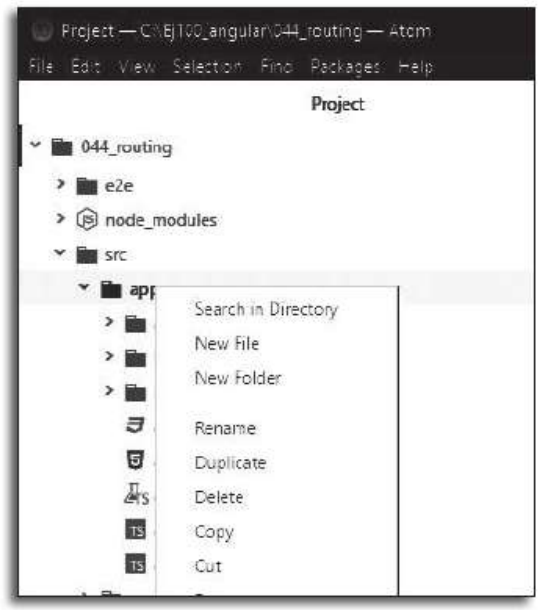
2. Seguidamente añadiremos los links de **Bootstrap** (https://v4-alpha.getbootstrap.com) en index.html para poder hacer uso de sus hojas de estilo. Consulte "087 Bootstrap. Introducción" para obtener más información.

3. Desde la línea de comandos, crearemos los componentes **LibroLista**, **AutorLista** y **NotFound** con los que trabajará el servicio Router. El componente NotFound se visualizará por pantalla cuando el servicio Router no pueda resolver la ruta.

```
C:\Ej100_angular\044_routing>ng generate component libroLista
C:\Ej100_angular\044_routing>ng generate component autorLista
C:\Ej100_angular\044_routing>ng generate component notFound
```



En este punto, desarrollaremos el componente **libroLista**. Para ello, previamente crearemos el modelo de datos Libro **(libro.model.ts)** y un fichero mock data **(mocks.ts)** con un conjunto de libros de prueba. Esto lo realizaremos mediante la opción *New File* del menú que aparece al pulsar el botón derecho del ratón sobre la carpeta **app**. Luego, añadiremos el siguiente código:



src/app/libro.model.ts

```
export class Libro {  id: number;  titulo: string;  autor: string;}
```

src/app/mocks.ts

```
import{ Libro } from'./libro.model';


export const LIBROS: Libro[] = [{"id": 1,"titulo": "El Quijote","autor":
    "Cervantes"},    {  "id": 2,"titulo": "Hamlet","autor": "Shakespea-
    re"}];
```

Seguidamente realizaremos las siguientes modificaciones en el componente **libroLista** para que visualice esos libros:

src/app/libro-lista/libro-lista.component.ts

```
import{ Libro } from'../libro.model';
import { LIBROS } from '../mocks';
…
export class LibroListaComponent implements OnInit {
  libros: Libro[];
…
  ngOnInit() {    this.libros = LIBROS;  }
}
```

src/app/libro-lista/libro-lista.component.html

```
<h4>Libros:</h4>
<ul>
  <li *ngFor="let libro of libros">
  {{ libro.titulo }}
  </li>
</ul>
```

4. Una vez creados los componentes, pasaremos a realizar la configuración del servicio Router. El primer paso será realizar la importación del servicio en "app.module.ts":

src/app/app.module.ts

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

A continuación, realizaremos la **configuración de rutas** en el mismo fichero:

src/app/app.module.ts

```
const appRoutes: Routes = [
    { path: 'libros', component: LibroListaComponent },
    { path: 'autores', component: AutorListaComponent },
    { path: '', redirectTo: '/libros', pathMatch: 'full' },
    { path: '**', component: NotFoundComponent }
 ];
 @NgModule({
   declarations: […],
   imports: […,RouterModule.forRoot(appRoutes)],
…
```

Y, finalmente, pondremos la etiqueta **router-outlet** para indicar al servicio Router dónde realizar las visualizaciones de los componentes. Para ello, ponga el siguiente código en "app.component.html":

src/app/app.component.html

```
<div style="text-align:center">

    <h3>Servicio Router: ejemplos de uso</h3>

</div>

<router-outlet></router-outlet>
```

5. La primera parte del ejercicio termina aquí. Desde el navegador puede realizar las pruebas para validar el código. Para el URL "localhost:4200/", será redireccionado a "localhost:4200/libros", donde visualizará la vista de libros, para "localhost:4200/autores", visualizará la vista de autores, y para cualquier otra ruta, visualizará la vista de "Not Found".

045 Routing: RouterLinks

Llegados a este punto, nuestro servicio Router ya estaría gestionando correctamente los cambios de direcciones URL de la barra de direcciones del navegador. Sin embargo, la navegación en una aplicación web principalmente se realiza desde la misma aplicación haciendo clics sobre enlaces a otras vistas.

Importante

Las directivas RouterLink nos permiten crear enlaces a partes específicas de nuestras aplicaciones.

En HTML, estos enlaces se realizan mediante la etiqueta <a> y su atributo "href", que sería donde pondríamos la dirección URL. Sin embargo, al haber creado un servicio Router, nos interesa que todo el control de rutas lo realice el mismo servicio. Por eso Angular dispone de las directivas **RouterLink.** Si en las etiquetas <a> sustituimos los atributos "href" por directivas RouterLink, conseguiremos que el servicio Router tome el control de todos estos elementos.

Los path configurados en RouterLink pueden ser **estáticos**:

```
<a routerLink="/grupo/a/usuario/alberto" routerLinkActive="active-link">…</a>
```

o **dinámicos**, donde usamos una cadena con literales y variables para formar el path dinámicamente:

```
<a [routerLink]=['/grupo', grupoId, 'usuario', usuarioNombre]>…</a>
```

En el primero de los ejemplos anteriores, se hace uso de la directiva **RouterLinkActive**. Esta directiva sirve para aplicar una clase **CSS** al elemento cuando la ruta del link esté activa. De esta manera podremos distinguir qué enlace, de todos los visibles en la aplicación, es el que está activo.

Por otra parte, el primer segmento del path puede venir precedido por **"/"**, **"../"** o **"./"**. Esto indica a Angular en qué parte de árbol de rutas realizar el link del path. Si, por el URL activo, el que aparece en la barra de direcciones, fuese "localhost:4200/libros" y quisiéramos desplazarnos a "localhost:4200/libros/comics", en routeLink podríamos poner el path "/libros/comics" o "./comics" o "comics".

Continuando la aplicación que habíamos empezado en el ejercicio anterior, añadiremos dos enlaces en la página principal de la aplicación usando RouterLink. Uno nos llevará a la vista de los libros y el otro a la de los autores.

1. Desde la línea de comandos nos situaremos en **ej100_angular/044_routing**, y ejecutaremos la aplicación con **ng serve**. A continuación, abriremos el proyecto con **Atom**.

2. A continuación, editaremos el template del componente principal de la aplicación y añadiremos los dos enlaces usando **RouterLink**. Todas las clases CSS usadas en el siguiente código forman parte de las hojas de estilo de Bootstrap.

src/app/app.component.html

```
…
<ul class="nav">
  <li class="nav-item">
    <a class="nav-link active" routerLink="/libros">Libros</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" routerLink="/autores">Autores</a>
  </li>
</ul>

<router-outlet></router-outlet>
```

Desde el navegador puede realizar pruebas de los enlaces. Al hacer clic sobre uno u otro link verá cómo cambia la vista y la dirección URL de la barra de direcciones del navegador.

3. El siguiente paso será usar la directiva **RouterLinkActive** para añadir una clase **CSS** al enlace cuando este haya sido seleccionado. De esta manera, en cualquier momento podremos ver a qué enlace pertenece la vista que tenemos en pantalla.

4. Para ello, primero crearemos la clase **CSS active-link** en app.component.css, o sea, en la hoja de estilos asociada al componente principal de la aplicación. Lo único que hará esta clase **CSS** será dejar las letras en color naranja.

src/app/app.component.css

```
.active-link {

    color: orange;

}
```

Finalmente añadiremos el uso de RouterLinkActive en los enlaces creados:

src/app/app.component.html

```
…

<ul class="nav">

  <li class="nav-item">

    <a class="nav-link active" routerLink="/libros" router-
    LinkActive="active-link">Libros</a>

  </li>

  <li class="nav-item">

    <a class="nav-link" routerLink="/autores" routerLinkActi-
    ve="active-link">Autores</a>

  </li>

</ul>


<router-outlet></router-outlet>
```

5. Desde el navegador podrá comprobar cómo cambia el color del enlace a naranja cuando lo selecciona.

046 Routing: Rutas con parámetros y ActivatedRoute

En una configuración de rutas podríamos tener rutas con parámetros, es decir, rutas con una parte fija y otra variable. Por ejemplo:

```
const appRoutes: Routes = [

  { path: 'libro/:id', component:<componente a
    mostrar> ,  ...,
```

Con esta configuración, el servicio Router seleccionaría esta primera línea de configuración para direcciones URL como: "/libro/2, "/libro/aa" o "/libro/4vd", pero no para direcciones como "libro/2/opiniones" o "libro/3/productos-relacionados". Luego, el componente seleccionado accedería al valor del parámetro ":id" para realizar las acciones que convenga.

Importante

Utilice ":" para indicar los parámetros de las rutas que configure. Acceda a ActivatedRoute para consultar los valores de esos parámetros y otra información relacionada con la ruta activa.

Para obtener el valor de los parámetros que pueden venir por URL, el componente utiliza el objeto ActivatedRoute. ActivatedRoute contiene información de la ruta asociada al componente que actualmente tengamos cargado en la etiqueta "router-outlet", por lo tanto, información de la actual ruta activa. Si desea más información de ActivatedRoute consulte https://angular.io/api/router/ActivatedRoute.

Para continuar con la aplicación que habíamos empezado en los ejercicios anteriores, vamos a crear el código necesario para que se puedan seleccionar libros de la lista de libros y ver su detalle. El detalle lo gestionará un nuevo componente, LibroDetalle, al que asociaremos a una nueva ruta URL: "libros/:id". El parámetro ":id" indicará el libro del cual deseamos ver su detalle.

1. Desde línea de comandos nos situaremos en **ej100_angular/044_routing**, y ejecutaremos la aplicación con **ng serve**. A continuación, abriremos el proyecto con **Atom**.

2. Sin salir de línea de comandos, crearemos el componente **LibroDetalle**.

```
C:\Ej100_angular\044_routing>ng generate component libroDetalle
```

3. Seguidamente, en la configuración de rutas, añadiremos la nueva ruta y componente:

src/app/app.module.ts

```
const appRoutes: Routes = [

  { path: 'libros', component: LibroListaComponent },

  { path: 'libros/:id', component: LibroDetalleComponent },

...
```

En el componente **LibroDetalle** añadiremos el código para primero **obtener el identificador** del libro (parámetro ":id" del URL) y luego para **leer el detalle** de ese libro.

Para obtener los parámetros de la ruta, ActivateRoute nos proporciona el observable **paramMap**. Para obtenerlos, simplemente nos suscribiremos al observable a la espera de recibirlos. De esta manera recibiremos el identificador de libro. En el capítulo "052 Servicios: Gestión asíncrona con observables" veremos con más detalle los observables, pero, en lo que respecta a este ejercicio, no es necesario que profundicemos más en el tema.

src/app/libro-detalle/libro-detalle.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import{ Libro } from'../libro.model';
import { LIBROS } from '../mocks';


…
export class LibroDetalleComponent implements OnInit {
  libro: Libro;

  constructor( private route: ActivatedRoute) { }

  ngOnInit() {
    this.route.paramMap
      .subscribe((params: ParamMap) => {
        let id = +params.get('id');
        this.libro =  LIBROS.find(item => item.id === id);
      });
  }
}
```

En el template del componente añadiremos el código para visualizar el detalle del libro.

src/app/libro-detalle/libro-detalle.component.html

```
<h4>Libro detalle:</h4>
<dl class="dl-horizontal">
  <dt>Identificador</dt><dd>{{ libro.id }}</dd>
  <dt>Titulo</dt><dd>{{ libro.titulo }}</dd>
  <dt>Autor</dt><dd>{{ libro.autor }}</dd>
</dl>
```

4. Y, finalmente, añadiremos un **enlace o link** al componente LibroDetalle para cada uno de los libros del componente **LibroLista**:

src/app/libro-lista/libro-lista.component.html

```
...
  <li *ngFor="let libro of libros">
    <a [routerLink]="['/libros', libro.id]">
      {{ libro.titulo }}
    </a>
  </li>
...
```

Desde el navegador puede realizar pruebas de los diferentes enlaces creados.

047 Routing: child routes

Angular nos permite crear árboles de configuraciones de rutas mediante **child routes**:

```
{ path: 'actor/:id', component: LibroDetalleComponent,
    children: [
      { path: 'biografia', component: ActorBiografiaComponent },
      { path: 'filmografia', component: ActorFilmografiaComponent }
    ]
},…
```

Los path de configuraciones hijos son **relativos** a los de las configuraciones padre. Por ejemplo, "/actor/:id/biografia" o "/actor/:id/filmografia" en el código anterior. Por otra parte, en el template de todo componente padre siempre habrá que añadir la etiqueta **router-outlet**. Allí será donde el servicio Router visualice los componentes hijos.

Al usar child routes, podemos tener varias rutas activas (ActivatedRoute) al mismo tiempo durante la navegación. En esta situación, es probable que desde un componente cualquiera necesitemos acceder a una ruta determinada del árbol de rutas activas para, por ejemplo, leer parámetros. Angular nos proporciona dos maneras de acceder al árbol de rutas activas. La primera consiste en partir de una **ruta activa (ActivatedRoute)** cualquiera, y usar sus propiedades **parent** y **children** para desplazarnos por el árbol. Y la segunda consiste en usar la propiedad **RouterState**. Esta propiedad nos permite obtener el árbol de rutas activas en cualquier momento y lugar de la aplicación.

Importante

Con child routes podemos crear árboles de rutas que faciliten la gestión de rutas de nuestras aplicaciones.

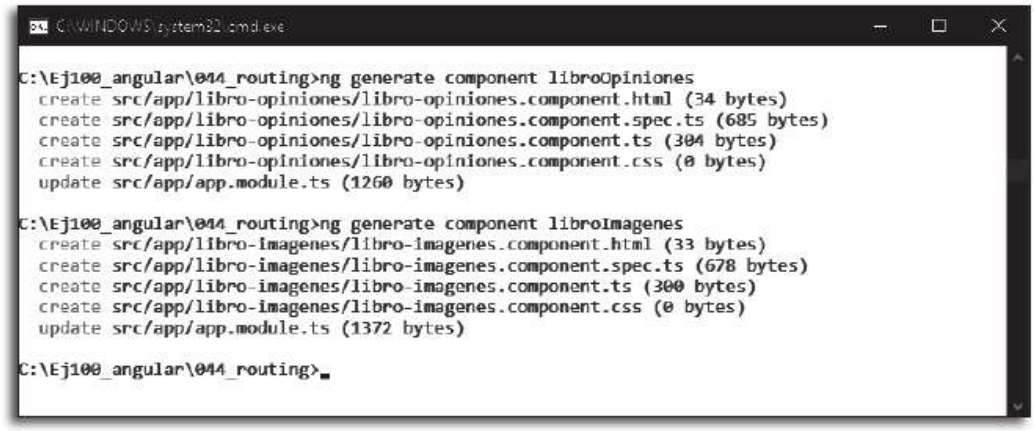
Continuando la aplicación que habíamos empezado en los ejercicios anteriores, vamos a añadir dos child routes, "opiniones" y "imágenes", a la ruta "libros/:id". Las dos rutas estarán asociadas a dos nuevos componentes, "LibroOpiniones" y "LibroImagenes", respectivamente. No entraremos a desarrollar estos dos componentes,

pero sí que les añadiremos el código necesario para obtener el identificador de libro del URL.

1. Desde línea de comandos nos situaremos en **ej100_angular/044_routing**, y ejecutaremos la aplicación con **ng serve**. A continuación, abriremos el proyecto con **Atom**.

2. Sin salir de la línea de comandos, crearemos el componente **LibroOpiniones y LibroImagenes**.

```
C:\Ej100_angular\044_routing>ng generate component libroOpiniones

C:\Ej100_angular\044_routing>ng generate component libroImagenes
```



3. Seguidamente, en la configuración de rutas, añadiremos las nuevas **child routes**:

src/app/app.module.ts

```
…
    { path: 'libros/:id', component: LibroDetalleComponent,
        children: [
          { path: 'imagenes', component: LibroImagenesComponent },
          { path: 'opiniones', component: LibroOpinionesComponent },
          { path: '', redirectTo: 'imagenes', pathMatch: 'full' },
          { path: '**', component: NotFoundComponent }
        ]},
…
```

En el template del componente padre, **LibroDetalle**, añadiremos dos enlaces a estas nuevas rutas y la etiqueta **router-outlet** para visualizar sus componentes asociados:

src/app/libro-detalle/libro-detalle.component.html

```
…
<ul class="list-inline">
  <li class="list-inline-item"><h4>Información adicional:</h4></li>
  <li class="list-inline-item"><a routerLink="imagenes" routerLinkAc-
    tive="active-link">Imágenes</a></li>
  <li class="list-inline-item"><a routerLink="opiniones" router-
    LinkActive="active-link">Opiniones</a></li>
</ul>
<router-outlet></router-outlet>
```

src/app/libro-detalle/libro-detalle.component.css

```
.active-link { color: orange;}
```

4. A continuación, añadiremos el código necesario en LibroImagenes para cargar el identificador de libro que serviría al componente para cargar las imágenes relacionadas. Para ello, haremos uso de ActivatedRoute primero para acceder al **ActivatedRoute parent**, y luego para obtener el valor del parámetro.

src/app/libro-imagenes/libro-imagenes.component.ts

```
import { ActivatedRoute, ParamMap } from '@angular/router';
…
export class LibroImagenesComponent implements OnInit {
  idLibro: number;

  constructor(    private route: ActivatedRoute  ) { }
  ngOnInit() {
    this.route.parent.paramMap
      .subscribe((params: ParamMap) => {
        this.idLibro = +params.get('id');
      });
  }
}
```

src/app/libro-imagenes/libro-imagenes.component.html

```
<p>(Imagenes del libro con identificador: {{idLibro}})</p>
```

Desde el navegador puede realizar pruebas de los diferentes enlaces creados.

048 Inyección de dependencias (DI)

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

Analicémoselo con un ejemplo. Imaginen el siguiente código. Se trata de una clase llamada **Musico** que utiliza métodos de un objeto de tipo "Instrumento".

Importante

La inyección de dependencias es un patrón de diseño que nos permite aislar las dependencias de nuestras clases consiguiendo unas clases más fuertes, flexibles y probables.

```
import { Instrumento }      from './instrumento';


export class Musico {
  public instrumento: Instrumento;
  constructor() {
    this.instrumento = new Instrumento();
  }
  tocar(): string {     return this.instrumento.tocar();   }
}
```

Esta clase **Musico** presenta los siguientes problemas:

- **Frágil:** si por ejemplo se realizaran cambios sobre el constructor de "Instrumento" añadiendo un parámetro obligatorio, la clase **Musico** fallaría a no ser que se adaptara su código.
- **Inflexible:** no es posible asignarle otro tipo de "Instrumento", ni tampoco es posible que comparta un mismo "Instrumento" a la vez con otro "Musico". Siempre será necesario adaptar su código.
- **Difícil para realizar test:** los test sobre "Musico" son peligrosos ya que estamos a merced de las dependencias escondidas que no vemos: no sabemos qué puede suponer crear un objeto de tipo "Instrumento" en un entorno de pruebas…

Aplicando la **inyección de dependencias,** la clase **Musico** quedaría de la siguiente manera:

```
import { Instrumento }      from './instrumento';


export class Musico {


  constructor(public instrumento: Instrumento) {   }


  tocar(): string {     return this.instrumento.tocar();   }
}
```

Y el consumidor de la clase realizaría lo siguiente:

```
import { Musico } from './musico';
import { Instrumento }      from './instrumento';
...
let musico = new Musico(new Instrumento());
console.log(musico.tocar());
```

Las definiciones de las dependencias ahora están en el constructor. La clase **Musico** ya no crea el objeto "Instrumento", ahora lo consume. De esta manera, quien hace uso de **Musico** es quien tiene todo el control sobre sus dependencias. Por tanto, podemos realizar acciones como las que comentábamos antes sin que el código de la

clase **Musico** se vea afectado. Por ejemplo, podríamos crear una nueva versión de la clase **Instrumento** añadiendo un parámetro obligatorio en su constructor:

```
export class Instrumento2 {

  constructor(public nombre: string) {}

  tocar(): string {…}
}
```

y probarla en nuestra clase **Musico** sin realizar más modificaciones:

```
//let musico = new Musico(new Instrumento());
let nombre = "piano";
let musico = new Musico(new Instrumento2(nombre));
```

A pesar de las ventajas comentadas, ahora el consumidor se ve obligado a crear todas las dependencias de la clase **Musico** cuando antes no tenía que hacerlo. Para solucionar este y otros temas relacionados, existen los **frameworks de inyección de dependencias**. Angular dispone de su propio framework al respecto. Entre otras cosas, nos permitirá que mediante el uso de inyectores **(injectors)** y proveedores **(providers)** configuremos el sistema para que el consumidor simplemente tenga que realizar lo siguiente para crear un objeto **Musico**:


```
let musico = injector.get(Musico);
console.log(musico.tocar());
```

Tanto el consumidor como la clase **Musico** solamente preguntan por lo que necesitan y el inyector lo entrega.

En los siguientes capítulos describiremos en detalle del framework de inyección de dependencias de Angular.

049 Servicios: Definición y uso mediante la inyección de dependencias (parte I)

Un **servicio Angular** es una clase que encapsula algún tipo de funcionalidad común entre los diferentes componentes de la aplicación, como podría ser, por ejemplo, el acceso a datos. La extracción de funcionalidades comunes de los componentes para crear servicios supone toda una serie de ventajas:

- Evitamos la repetición innecesaria de código en los componentes, por lo que su código acaba siendo más ligero y centrado en el soporte a la vista.
- Podemos crear servicios mock (de prueba) que faciliten el análisis de los componentes que los usan.
- Mejoramos el control y mantenimiento de esas funcionalidades comunes.

Importante

El framework de inyección de dependencias de Angular permite aislar las dependencias de las clases, pero también permite aislarlas del que las consume.

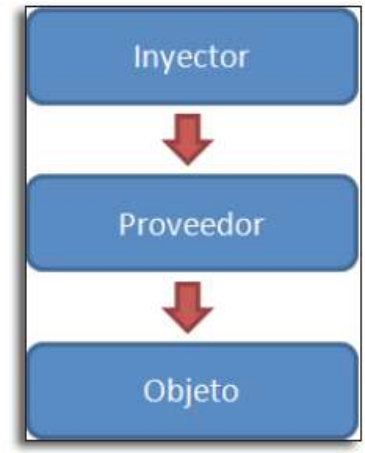
Vamos a ver cómo sería el formato estándar de cualquier servicio a través de un ejemplo:

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


@Injectable()
export class LoggerService {
  log(message: string) {    console.log(message);   }
  constructor() { }
}
```

Como vemos no es más que una clase estándar donde se le ha añadido el decorador **@Injectable()**. De momento dejaremos el análisis de este decorador para más adelante.

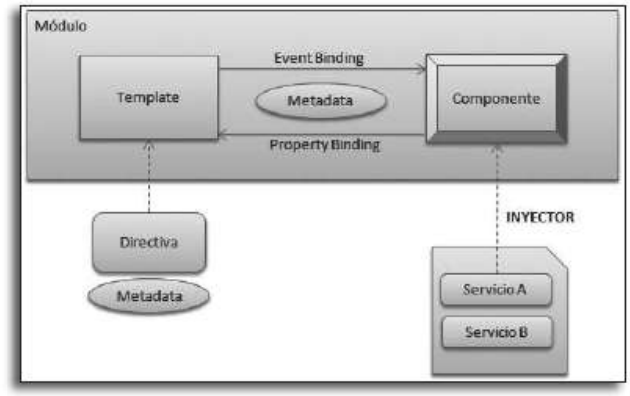
Los servicios son suministrados a los componentes mediante el patrón de diseño **inyección de dependencias** que hemos visto en el capítulo anterior. Angular incorpora un framework al respecto. Básicamente, este framework nos facilita la gestión de dependencias mediante **inyectores** (injectors) y **proveedores** (pro-

viders). Estos elementos los podríamos definir de la siguiente manera:



- Un **inyector** mantiene una colección de servicios previamente instanciados. Si se requiere un servicio aun no instanciado, el inyector crea una instancia mediante un proveedor y lo agrega a la colección.
- Un **proveedor** es algo que puede crear o devolver una instancia de servicio. Normalmente es la clase del servicio en sí. Para que el inyector pueda hacer uso de los proveedores, estos deben estar registrados. Estos registros pueden hacerse a nivel de módulo o componente, dependiendo de la disponibilidad que se le quiera dar al servicio.

En el diagrama general de la arquitectura Angular puede observar dónde se situaría el inyector y su colección de servicios previamente instanciados.



A partir de aquí, vamos a ver cómo utilizaríamos todas estas herramientas para que los componentes pudieran usar el servicio de ejemplo que hemos creado antes (LoggerService).

Durante el proceso de inicialización de la aplicación, Angular ya crea un inyector de forma automática. Lo que tenemos que hacer nosotros es **registrar el proveedor de nuestro servicio** para que el inyector pueda usarlo. Dado el tipo de servicio del ejemplo, lo registraremos en ngModule para que pueda usarse en toda la aplicación. Esto lo haríamos de la siguiente manera:

```
@NgModule({ ..., providers: [LoggerService], ...})

export class AppModule { }
```

En este caso hemos realizado el registro del proveedor de la forma más básica, sin embargo, hay que tener en cuenta que Angular nos ofrece muchas opciones al respecto. Veamos algunos ejemplos:

- `providers: [LoggerService]`... O ... `[{ provide: LoggerService, useClass: LoggerService }]`

  Es el método de registro que hemos visto antes.

- `[{ provide: LoggerService, useClass: LoggerService2 }]`

  Le decimos al inyector que instancie otra versión del servicio.

Finalmente haríamos la inyección del servicio en el componente que necesitara usarlo:

```
@Component({...})
export class EjemploComponent implements OnInit {
  constructor(private loggerService: LoggerService){  }
  ngOnInit() {    this.loggerService.log("hola");  }
}
```

La combinación del tipo de parámetro del constructor (*LoggerService)*, el decorador @Component y la información de los proveedores registrados, indican al inyector de Angular que inyecte una instancia de "LoggerService" cuando se cree un nuevo "EjemploComponent". **Por lo tanto, como consumidores de "EjemploComponent", el framework también nos aísla de sus dependencias.**

Desde el punto de vista de un inyector, las dependencias son **"singletons"**. Esto significa que, por defecto, el inyector creará una única instancia de cada clase, la cual será compartida por los distintos componentes.

Por último, analizaremos un poco el decorador **@Injectable()** que ponemos en las clases de nuestros servicios. El decorador @Injectable() marca una clase como disponible para un inyector para poder instanciarla. Sin este decorador, el inyector informaría de un error si intentara instanciarla. Los inyectores también son responsables de instanciar componentes; sin embargo, no hace falta poner @Injectable() porque @Component, y otros decoradores como @Directive o @Pipe, son subtipos de @Injectable().

En el siguiente ejercicio pondremos en práctica todos estos conceptos creando y usando un servicio.

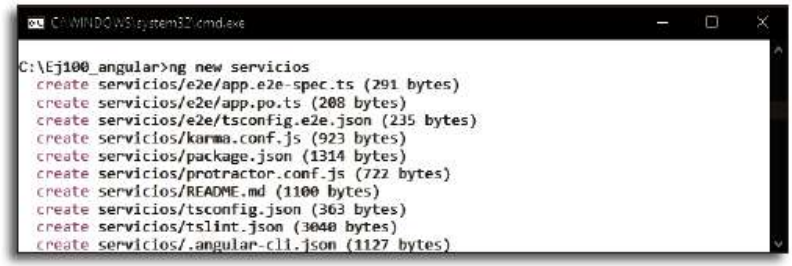
050 Servicios: Definición y uso mediante inyección de dependencias (parte II)

En este ejercicio pondremos en práctica los servicios y la inyección de dependencias que hemos visto en los anteriores capítulos. Aplicaremos todos estos conceptos en una nueva aplicación extraída en parte de la aplicación de gestión de Libros que habíamos desarrollado en los ejercicios dedicados al "Router".

Importante

Los servicios nos permiten extraer funcionalidades comunes de nuestros componentes dejándolos más centrados en el soporte a la vista.

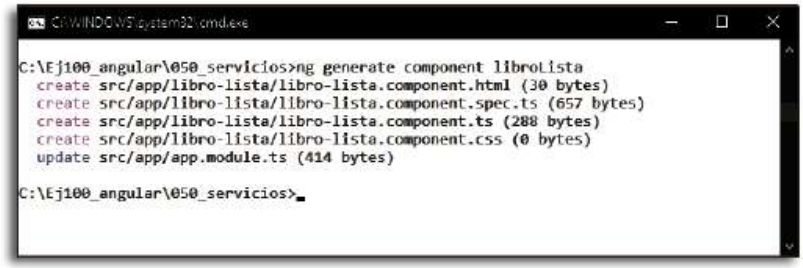
1. Nos ubicaremos en **ej100_angular**, y crearemos el proyecto tecleando **ng new servicios**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio ejecutando **rename servicios 050_servicios**, y finalmente pondremos en marcha la aplicación ejecutando **ng serve** desde dentro la carpeta **050_servicios**. A continuación, abriremos el proyecto con el editor **Atom**.



2. Seguidamente añadiremos los links de **Bootstrap** (https://v4-alpha.getbootstrap.com) en index.html para poder hacer uso de sus hojas de estilo. Consulte "087 Bootstrap: Introducción" para obtener más información.

3. Desde la línea de comandos, crearemos el componente **LibroLista**.

```
C:\Ej100_angular\050_servicios>ng generate component libroLista
```



Añadiremos su código y crearemos los ficheros **libro.model.ts** y **mocks.ts** que necesita:

src/app/libro-lista/libro-lista.component.ts

```
import{ Libro } from'../libro.model';

import { LIBROS } from '../mocks';

…

export class LibroListaComponent implements OnInit {

  libros: Libro[];

  ngOnInit() {    this.libros = LIBROS;  }

}
```

src/app/libro-lista/libro-lista.component.html

```
<ul>

  <li *ngFor="let libro of libros">{{ libro.titulo }}</li>

</ul>
```

src/app/libro.model.ts

```
export class Libro {  id: number;  titulo: string;  autor: string;}
```

src/app/mocks.ts

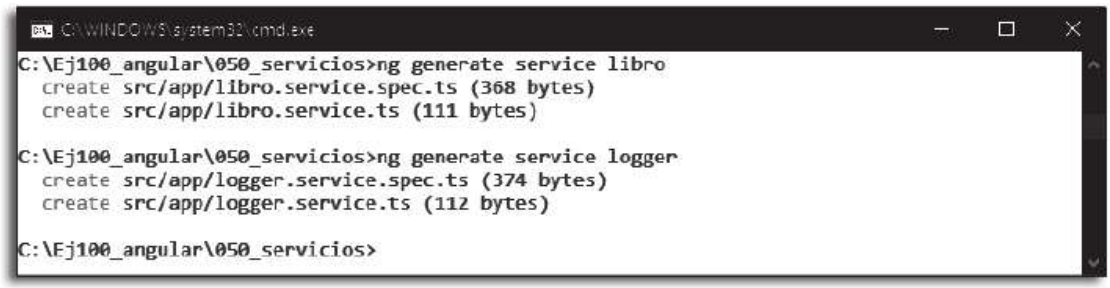
```
import{ Libro } from'./libro.model';

export const LIBROS: Libro[] = [{"id": 1,"titulo": "El Quijo-
    te","autor": "Cervantes"},    {  "id": 2,"titulo": "Hamlet","au-
    tor": "Shakespeare"}];
```

Seguidamente añadiremos LibroLista al componente principal para poder visualizarlo en el navegador:

src/app/app.component.html

```
<div class="container">

  <app-libro-lista></app-libro-lista>

</div>
```

4. El componente LibroLista obtiene los libros por sí solo. Esta funcionalidad previsiblemente será usada por otros componentes, por lo que lo mejor será dejarlo como servicio. Vamos a crearlo. También crearemos un segundo servicio para grabar log que será usado por el primero:




```
C:\Ej100_angular\050_servicios>ng generate service libro
C:\Ej100_angular\050_servicios>ng generate service logger
```

Para poder hacer uso de los servicios, el primer paso será **registrar su proveedor**. En este caso, lo haremos a nivel de módulo para que todos sus componentes puedan usarlos:

src/app/app.module.ts

```
…
import { LoggerService }         from './logger.service';
import { LibroService }         from './libro.service';


@NgModule({…, providers: [LoggerService,LibroService],…})
export class AppModule { }
```

A continuación, añadiremos su código. Además, para el servicio Libro, también le inyectaremos el servicio Logger, ya que lo usará:

src/app/logger.service.ts

```
…
@Injectable()
export class LoggerService {
  constructor() { }
  log(message: string) {
     console.log("(" + new Date().toLocaleTimeString() + ") " + mes-
   sage);}
}
```

src/app/libro.service.ts

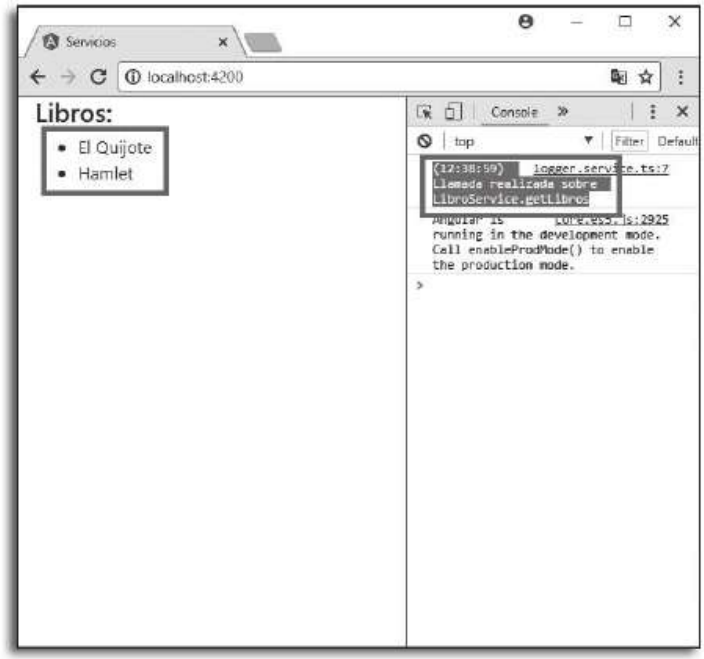
```
…

@Injectable()

export class LibroService {

  constructor(private loggerService: LoggerService) { }

  getLibros() {

    this.loggerService.log("Llamada realizada sobre LibroService.
   getLibros");

    return LIBROS;}

}
```

Finalmente modificaremos nuestro componente para que use el servicio Libro:

src/app/libro-lista/libro-lista.component.ts

```
…

@Component({…})

export class LibroListaComponent implements OnInit {

  libros: Libro[];

  constructor(private libroService: LibroService){  }

  ngOnInit() {    this.libros = this.libroService.getLibros();  }

}
```

5. Desde el navegador podrá comprobar su funcionamiento. Recuerde que para visualizar log debe habilitar la opción "Más herramienta/Herramientas para desarrolladores" de Google Chrome.


Servicios
localhost:4200
Libros:
El Quijote
Hamlet
Console
top
Filter
Default
(12:38:59) logger.service.ts:7
Llamada realizada sobre
LibroService.getLibros
Angular is core.mjs:2925
running in the development mode.
Call enableProdMode() to enable
the production mode.

051 Servicios: Gestión asíncrona con promesas

Los dos servicios que hemos creado en el ejercicio anterior, LoggerService y LibroService, devuelven respuestas inmediatas, por lo que se han gestionado de forma **síncrona**. Sin embargo, muchas veces tendremos servicios que realizarán llamadas a servidores remotos cuyas respuestas no sabemos cuánto tiempo tardarán en llegar. Esos servicios no pueden gestionarse de la misma forma, ya que produciríamos bloqueos inaceptables en la aplicación. Esos servicios hay que gestionarlos de forma **asíncrona**. De esta manera, el usuario podrá seguir trabajando mientras se ejecutan los servicios.

Importante

Para mantener un buen nivel de fluidez en nuestras aplicaciones, siempre deberíamos trabajar con servicios asíncronos. Las promesas son una buena herramienta para su implementación.

Las **promesas** es una de las herramientas que nos permite la programación asíncrona. Tal como su nombre indica, una promesa la podríamos definir como un valor que se espera en un tiempo futuro no definido después de ejecutar, de forma asíncrona, una operación. Su sintaxis sería la siguiente:

```
new Promise( /* ejecutor */ function(resolver, rechazar) { ... } );
```

La función ejecutor empieza a ejecutarse de forma asíncrona justo en el momento en el que se crea la promesa. La función ejecutor, al terminar, deberá llamar a la función **resolver** para resolver la promesa con el valor obtenido, o llamar a la función **rechazar** para rechazarla con el error producido. Finalmente, mediante los métodos "**then**" y "**catch**" de la promesa, gestionaremos esa resolución y rechazo, respectivamente. Vea en el ejemplo cómo gestionamos la promesa devuelta por el método getUsuario.

```
servicioUsuarios.getUsuario(1)
  .then(function(usuario){})
  .catch(function(msg){});
```

Una llamada al método “then” de una promesa devuelve otra promesa a la que también podemos llamar a su método “then”, y así sucesivamente. Esto nos permitirá **encadenar promesas**.

Vamos a ver unos ejemplos que realizaremos sobre la aplicación que habíamos creado en el anterior ejercicio. Modificaremos “getLibros” de “LibroService” para que devuelva una promesa de los Libros, y le añadiremos un retraso de 5 segundos para simular una llamada a un servidor remoto.

1. Desde línea de comandos nos situaremos en **ej100_angular/050_servicios**, y ejecutaremos la aplicación con **ng serve**. A continuación, abriremos el proyecto con el editor **Atom**.

2. Editaremos el código del servicio “LibroService” y realizamos las modificaciones comentadas:

src/app/libro.service.ts

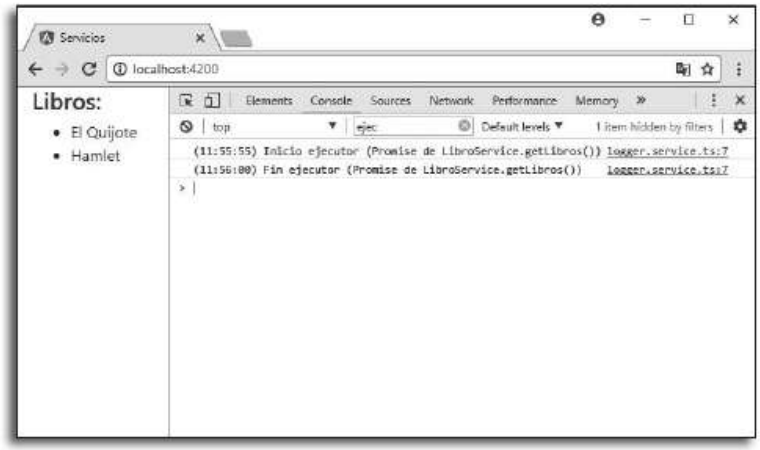
```
…

@Injectable()

export class LibroService {

  constructor(private loggerService: LoggerService) { }

  getLibros(): Promise<Libro[]> {

    return new Promise<Libro[]>( (resolve, reject) => {

      this.loggerService.log("Inicio ejecutor (Promise de LibroService.
   getLibros())");

      setTimeout(() => {

        this.loggerService.log("Fin ejecutor (Promise de LibroService.
   getLibros())");

        resolve(LIBROS);

      }, 5000);

    });}

}
```

Finalmente modificaremos el componente para que gestione la promesa:

src/app/libro-lista/libro-lista.component.ts

```
…
@Component({…})
export class LibroListaComponent implements OnInit {
  libros: Libro[];
  constructor(private libroService: LibroService){  }
  ngOnInit() {
    this.libroService.getLibros().then(libros => this.libros = libros);
  }
}
```

Desde el navegador podremos comprobar su funcionamiento. Recuerde que para ver los "logs" debe activar la opción "Más herramientas/Herramientas para desarrolladores" de Google Chrome.



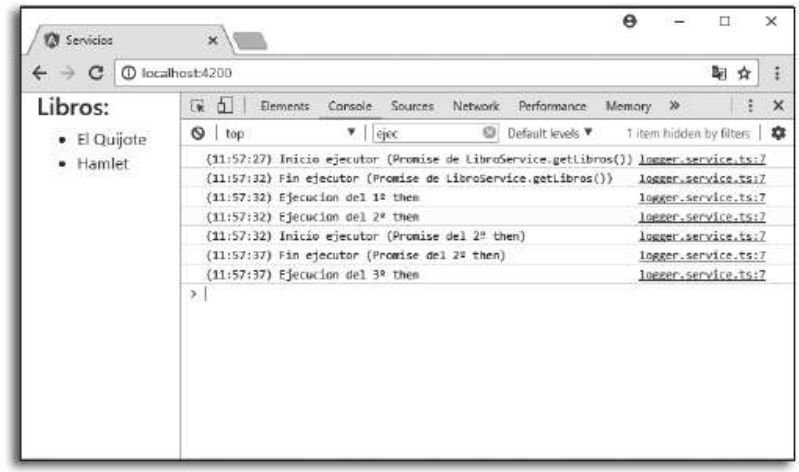
3. Para probar el **encadenamiento de promesas**, realizaremos los siguientes cambios:

src/app/libro-lista/libro-lista.component.ts

```
@Component({…})
export class LibroListaComponent implements OnInit {
  libros: Libro[];
  constructor(private libroService: LibroService, private loggerService:
    LoggerService){  }
  ngOnInit() {
    this.libroService.getLibros()
    .then( libros => {
              this.loggerService.log("Ejecucion del 1° then");
              return libros;})
```

```
    .then( libros => {
              this.loggerService.log("Ejecucion del 2° then");
              return new Promise<Libro[]>((resolve, reject) => { // (*)
                  this.loggerService.log("Inicio ejecutor (Promise del 2°
  then)");
                  setTimeout(() => {
                    this.loggerService.log("Fin ejecutor (Promise del 2°
  then)");
                    resolve(libros);
                  }, 5000);
              });})
    .then( libros => {
              this.loggerService.log("Ejecucion del 3° then");
              this.libros = libros;});
  }
}
```

Desde el navegador y su consola podremos comprobar la secuencia de ejecuciones.



052 Servicios: Gestión asíncrona con observables (Librería RxJs) (parte I)

Los **observables de las librerías RxJS** (reactive extensions for JavaScript) son otra herramienta que nos permitirá programación asíncrona. Veamos cuáles son las principales diferencias respecto a las **promesas** que veíamos en el anterior ejercicio:

- Una promesa siempre devolverá un valor o un error, mientras que un observable puede emitir múltiples valores en el tiempo. Por este motivo, a veces se le define como un flujo de datos o "stream".
- Una promesa no puede cancelarse, en cambio un observable sí. Si el valor esperado por una promesa (p. ej., el resultado de una llamada a un servicio) ya no es necesario, no hay manera de cancelar esa promesa. Siempre acabará llamándose a su "callback" de éxito o rechazo. En un observable no ocurre lo mismo, ya que siempre podemos anularlo cancelando la suscripción previa que se habría realizado.

Importante

Los observables proporcionan y amplían las características de las promesas. Permiten manejar 0, 1 o N eventos, y pueden cancelarse.

La sintaxis de una observable sería la siguiente:

```
vObs = new Observable( /* ejecutor */

            observer => { ... } /*o lo que es lo mismo: function(observer) {
        ... } */

    );

vObsSubs = vObs.subscribe(

  value => ...,

  error => ...,

  () => ...

);
```

La función ejecutor empieza a ejecutarse de forma asíncrona cuando un observador (u "observer") realiza una **suscripción** en el observable. Esta suscripción se traduce en una ejecución al método "subscribe(observer)" del observable. En el caso que se realizara otra suscripción al mismo observable, se pondría en marcha otro hilo de ejecución asíncrona distinta.

Una vez realizada la suscripción, el observador (u "observer") recibirá cualquier dato emitido por el observable. En la suscripción al observable, es posible pasar una instancia del objeto observador que previamente haya creado, o puede pasar directamente las **tres funciones de retorno o "callback"** que lo definen: la primera es ejecutada al recibir nuevos valores, la segunda al recibir un error, y la última al recibir la notificación de finalización. Toda esa información será emitida por el observable usando los siguientes métodos del observador: **next**(), **error**() o **complete**(). Para cancelar una suscripción ejecutaremos **"unsuscribe**()**"** sobre la suscripción. Esta cancelación siempre debe realizarse en suscripciones de observables que emitan valores de forma indefinida sin terminar nunca. En caso contrario, podrían producirse memory leaks (problemas de memoria).

Los observables aquí explicados son los llamados *cold observables,* no emiten valores hasta que un observador se suscribe a ellos. Sin embargo, también existen los *hot observables,* que emiten valores aunque no haya suscriptores. Para más información consulte: http://reactivex.io/rxjs/manual/overview.html.

Continuando la aplicación de los anteriores ejercicios, vamos a crear un nuevo servicio "LibroObservableService" con un método "getLibros" que devuelva un observable. El observable no devolverá todos los libros de golpe como hacía el servicio "LibroService", sino que irá devolviendo cadenas de libros cada segundo y medio añadiendo un nuevo libro en cada envío. Al final, enviará la notificación de finalización.

1. Desde línea de comandos nos situaremos en **ej100_angular/050_servicios**, y ejecutaremos la aplicación con **ng serve**. A continuación, abriremos el proyecto con el editor **Atom**.

2. Desde la línea de comandos, crearemos el servicio **LibroObservableService**:

```
C:\Ej100_angular\050_servicios>ng generate service libroObservable
```

   Y le añadiremos el siguiente código:

src/app/libro-observable.service.ts

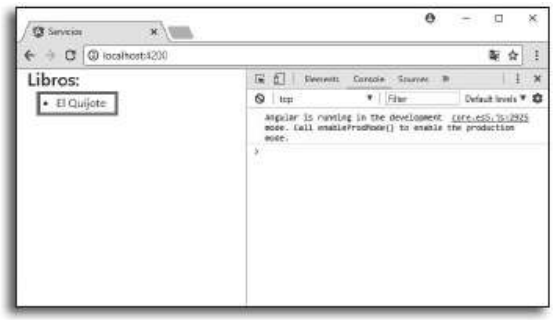
```
@Injectable()
export class LibroObservableService {
  libros: Libro[];

  constructor() { }
  getLibros(): Observable<Libro[]> {
    return new Observable<Libro[]>( observer => {
      let libros: Libro[] = [];
      observer.next([]);
      LIBROS.forEach((libro, index) => {
        setTimeout(() => {
          libros.push(libro);
          observer.next(libros);
        }, (index + 1) * 1500);
      });
      setTimeout(() => {observer.complete();}, (LIBROS.length + 1) *
    1500);
     });
  }
}
```

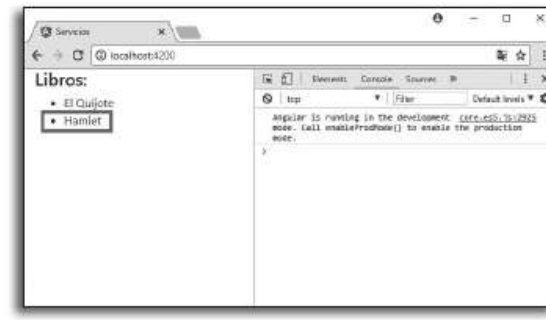
A continuación registraremos el proveedor del servicio en **app.module.ts**, y finalmente modificaremos nuestro componente **libro-lista.component.ts** para que lo use:

src/app/libro-lista/libro-lista.component.ts

```
  ngOnInit() {
    this.observableSubs= this.libroObservableService.getLibros()
    .subscribe(
      libros => this.libros = libros,
      error => console.log(error),
      () => console.log("this.libroObservableService.getLibros()
    FINALIZADO")
    );
  }
  ngOnDestroy(){ if (this.observableSubs) this.observableSubs.un-
    subscribe();  }
}
```

3. Desde el navegador y su consola podremos comprobar la secuencia de elementos devueltos por el observable cada segundo y medio.



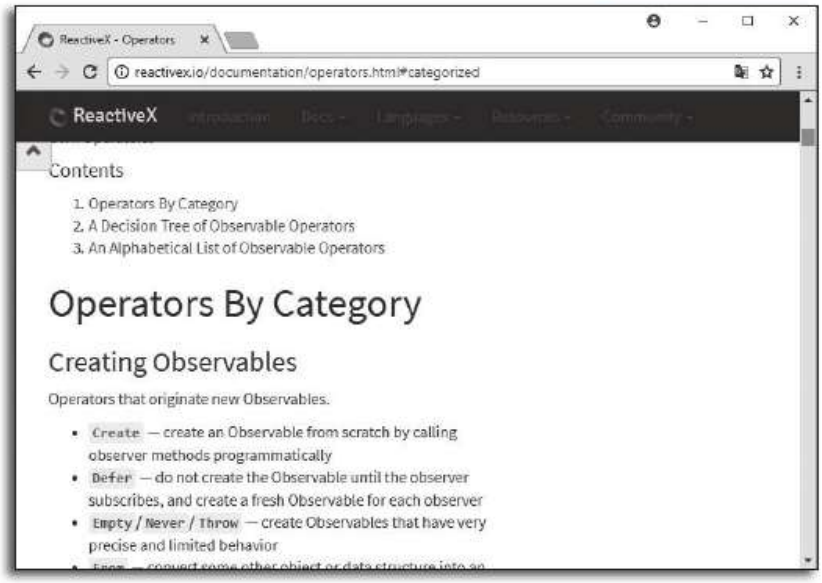


053 Servicios: Gestión asíncrona con observables (Librería RxJs) (parte II)

Las librerías **RxJS** (reactive extensions for JavaScript) incluyen toda una serie de **operadores** que, combinados con los observables, pueden llegar a ser muy útiles. El abanico de operadores es muy amplio, por lo que para facilitar su comprensión se suelen clasificar por categorías. En este capítulo veremos algunas de las categorías y operadores más usados. Si desea más información, le recomendamos que consulte la documentación oficial: http://reactivex.io/documentation/operators.html.

Importante

Gracias a los observables y a sus operadores, la librería RxJs (reactive extension for JavaScript) es una de las herramientas más importantes que hay en el mercado a la hora de trabajar con flujos de datos asíncronos.



Para cada uno de los operadores comentados hay un pequeño ejemplo. Si lo desea, puede implementarlos en un nuevo proyecto para ponerlos a prueba. Recuerde que para crear el proyecto seguiremos estos pasos:

1. Nos ubicaremos en **ej100_angular**, y crearemos el proyecto tecleando **ng new rxjs**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio ejecutando **rename rxjs 053_rxjs**, y finalmente pondremos en marcha la aplicación ejecutando **ng serve** desde la carpeta **053_rxjs**. A continuación, abriremos el proyecto con el editor **Atom**.

2. Los ejemplos los implementaremos en **app.component.ts**. Para verificarlos, sobre los observables realizaremos **suscripciones** que graben en log los valores obtenidos. Recuerde visualizar los log activando: "Más Herramientas/ Herramientas para desarrolladores" del menú del Google Chrome.

Operadores de creación

El operador **create** crea un observable que ejecutará la función específica cuando un observador se suscriba.

```
vObs = Observable.create( /* ejecutor */ observer => { ... });
```

En el anterior capítulo lo habíamos visto hacer de otra forma ("vObs = new Observable (..)"), pero el resultado final es el mismo. En los dos casos obtenemos un observable con una función configurada.

El operador **interval** crea un observable que emite números secuenciales (0,1,...) cada cierto intervalo de tiempo según lo que le hayamos indicado por parámetro. Vea el siguiente ejemplo:

```
var source = Observable.interval(1000); //Emite secuencia 0, 1, 2, ... cada 1
    segundo
```

El observador que se haya suscrito recibirá un número secuencial cada segundo. Mirar imagen.

El operador **of** crea un observable que emite los valores que se le hayan pasado por parámetro y finaliza emitiendo la notificación de finalización.

```
var source = Observable.of('a','b','c'); //Emite secuencia 'a', 'b' y 'c'
```

El operador **from** permite crear un observable a partir de distintos objetos y tipos de datos. Las posibilidades son muy amplias. A continuación, puede ver un ejemplo donde partimos de una promesa que se resuelve en dos segundos devolviendo una cadena de caracteres, y otro donde partimos de un array:

```
var source = Observable.from(
  new Promise<string>( (resolve, reject) => {
      setTimeout(() => { resolve("Valor resuelto por la Promise."); }, 2000);
  })
);
var source = Observable.from([{nombre: 'Miguel', edad: 30}, {nombre: 'Juan',
   edad: 35}]);
```

Operadores de transformación

El operador **map** transforma los valores emitidos por un observable aplicando una función, y devuelve un observable que emite los valores transformados.

```
var source = Observable.interval(1000).map(x => 2 * x); //Emite secuencia 0,
   2, 4, ...
```

Operadores de combinación

El operador **merge** nos permite combinar varios observables en uno solo fusionando sus emisiones:

```
var source1 = Observable.interval(1000);
var source2 = Observable.interval(1000).map(x => 10 * x);
var source3 = source1.merge(source2); //Emite secuencia 0, 0, 1, 10, 2,
   20,...
```

En este ejemplo, el observador que se haya suscrito a "source3" recibirá 0 y 0, el primer segundo, 1 y 10, el siguiente segundo, … y así sucesivamente. .

El operador **concat** es similar a merge, pero en este caso el observable resultado de la fusión emite los valores de los distintos observables de forma ordenada: primero los del primer observable, luego del segundo, etc.

```
var source1 = Observable.of('a','b','c');
```

```
var source2 = Observable.of('d','e','f');

var source3 = source1.concat(source2); //Emite secuencia a, b, c, d, e, f
```

Operadores de utilidad

El operador **do** nos permite realizar una acción para cada uno de los valores emitidos por un observable, pero sin posibilidad de modificarlos. El operador devuelve un observable idéntico al de entrada.

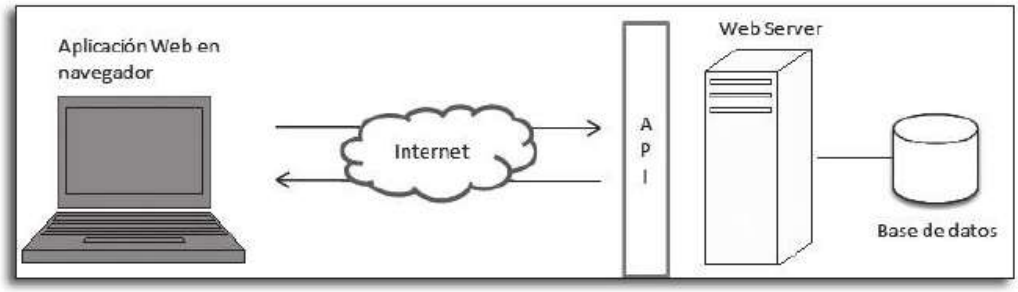
```
var source = Observable.of('a','b','c').do(x => console.log(x));

//Emite secuencia 'a', 'b' y 'c', y muestra estos valores por log
```

054 HttpClient: Introducción e instalación

Una aplicación web puede dividirse en dos grandes bloques: **Front-end** y **Back-end**. El Front-end es la parte que se muestra a través del navegador web, o sea, la parte que se encarga de interactuar con el usuario para recoger datos e información. El Back-end, por su parte, se ejecuta en un servidor y recoge a través de una API de servicios toda esa información para gestionarla (guardarla en una base de datos, realizar comprobaciones, etc.).

Importante

HttpClient es un servicio con métodos para realizar peticiones HTTP en aplicaciones Angular.



En la mayoría de casos, la comunicación entre aplicaciones Front-end y servicios Back-end se realiza mediante el **protocolo HTTP**. Para realizar este tipo de comunicación, las aplicaciones se apoyan en API como la interfaz **XMLHttpRequest** y la API **fetch(),** que son soportadas por la mayoría de navegadores web y permiten al Front-end realizar peticiones HTTP.

Angular, por su parte, nos ofrece el servicio HttpClient. **HttpClient** es un servicio que proporciona una API simplificada, construida sobre la interfaz XMLHttpRequest, para realizar peticiones HTTP y procesar sus respuestas. En este sentido, el servicio ofrece toda una serie de métodos para realizar los distintos tipos de peticiones HTTP que existen:

• get • post • put • delete	• patch • head • jsonp

El servicio también aporta otros beneficios muy interesantes como son el soporte a la realización de test, soporte a la intercepción de peticiones y respuesta, una mejor gestión de errores, etc.

HttpClient forma parte del módulo **HttpClientModule** del paquete **@angular/common/http** que tendremos que incluir en nuestra aplicación Angular. Hay que tener en cuenta que HttpClient está disponible en **Angular 4.3.X y versiones posteriores**. Para versiones anteriores existía un servicio parecido que se llamaba simplemente HTTP.

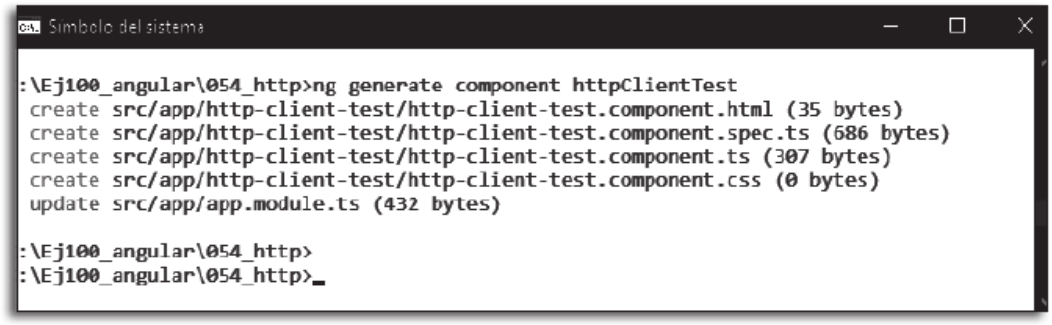
Una vez realizada la introducción empezaremos nuestro ejercicio. En esta primera parte únicamente crearemos una aplicación con un único componente que habilitaremos para que pueda hacer uso del servicio HttpClient. En los siguientes ejercicios ya le añadiremos nuevas funcionalidades.

1. Nos ubicaremos en **ej100_angular**, y crearemos el proyecto tecleando **ng new http**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio ejecutando **rename http 054_http**, y finalmente pondremos en marcha la aplicación ejecutando **ng serve** desde dentro la carpeta **054_http**. A continuación, abriremos el proyecto con **Atom**.

2. Seguidamente añadiremos los links de **Bootstrap** (https://v4-alpha.getbootstrap.com) en index.html para poder hacer uso de sus hojas de estilo. Consulte "087 Bootstrap: Introducción" para obtener más información.

3. Desde línea de comandos crearemos el componente **HttpClientTest** con el que trabajaremos:

```
C:\Ej100_angular\054_http>ng generate component httpClientTest
```



A continuación, lo añadiremos en el template del componente principal para poder visualizarlo en el navegador:

src/app/app.component.html

```
<div class="container">
    <h3>Servicio HttpClient: ejemplos de uso</h3>
    <app-http-client-test></app-http-client-test>
</div>
```

Para poder usar el servicio HttpClient en nuestros componentes, lo primero que tendremos que hacer es **incluir el módulo HttpClientModule** en nuestra aplicación Angular. Para ello, importaremos HttpClientModule en el módulo raíz de nuestra aplicación:

src/app/app.module.ts

```
…
import {HttpClientModule} from '@angular/common/http';

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

Finalmente, inyectaremos el servicio HttpClient en nuestro componente para que pueda usarlo:

src/app/http-client-test/http-client-test.component.ts

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

import { HttpClient } from '@angular/common/http';



@Component({…})

export class HttpClientTestComponent implements OnInit {

  constructor(private http: HttpClient) { }

  ngOnInit() {  }

}
```

055 HttpClient: Operaciones Get y Post

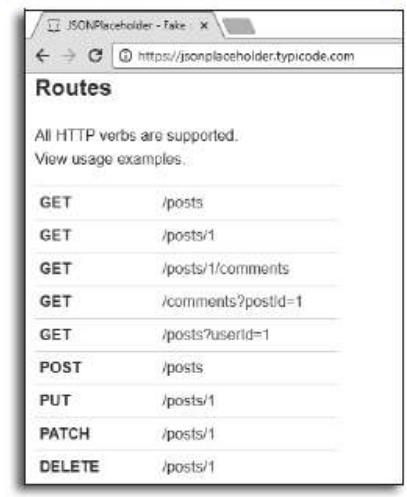
En este ejercicio analizaremos los métodos get() y post() de HttpClient. Estos métodos los utilizaremos para realizar peticiones HTTP tipo "get" y "post", respectivamente.

- **GET():** para obtener un recurso del servidor.
- **POST():** para crear un recurso en el servidor.

Para realizar pruebas con estas peticiones HTTP necesitaríamos un servidor remoto que ofreciera una **API** sobre la cual lanzarlas. Sin embargo, la creación de un backend con una API no es el objetivo de este ejercicio. Por este motivo haremos uso de JSONPlaceholder (https://jsonplaceholder.typicode.com/), una API REST pública para realizar test. Usando esta API podremos gestionar "publicaciones", que las podríamos definir como ítems compuestos por un título, cuerpo y un identificador de usuario.

Importante

Utilizaremos el método get() para obtener un recurso del servidor y post() para crearlo.



En este ejercicio usaremos las siguientes rutas de la API:

Método	URL	Descripción
GET	/posts	Obtención de todas las publicaciones
POST	/posts	Crear nueva publicación

Para ver cómo funcionan los métodos get() y post(), vamos a crear ejemplos de uso en la aplicación que habíamos creado en el ejercicio anterior. Vamos a crear una función para obtener todas las publicaciones mediante el método get(), y otra para crear una nueva publicación mediante el método post(). También añadiremos botones en el template para poder llamar a estas funciones. Hay que tener en cuenta que la gestión que realiza el servidor remoto JSONPlaceholder con las peticiones de actualización es simulada. Por tanto, al realizar un post() recibiremos el identificador de la nueva publicación; sin embargo, si a continuación realizamos un get(), no recibiremos esa nueva publicación.

1. En primer lugar, abriremos el proyecto 054_http con el editor **(Atom)**, y pondremos en marcha la aplicación ejecutando **ng serve** desde la línea de comandos y desde la carpeta **054_http**.

2. A continuación, añadimos las modificaciones en el componente:

src/app/http-client-test/http-client-test.component.ts

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

@Component({…})
export class HttpClientTestComponent implements OnInit {
  resultadoPeticion;
  constructor(private http: HttpClient) { }
  ngOnInit() { this.get(); }

  get() {
    this.http.get('https://jsonplaceholder.typicode.com/posts')
      .subscribe( data => { this.resultadoPeticion = data; } );
  }
  post(){
    this.http.post('https://jsonplaceholder.typicode.com/posts',
      {
      title: 'Previsión Viernes.',
      body: 'Parcialmente soleado.',
      userId: 1
      })
      .subscribe( data => { this.resultadoPeticion = data; } );
  }
}
```

src/app/http-client-test/http-client-test.component.html

```
<hr>
<h5>Petición:</h5>
<div class="btn-group" role="group" aria-label="Basic example">
  <button type="button" class="btn btn-secondary"
(click)="get()">Get</button>
  <button type="button" class="btn btn-secondary"
(click)="post()">Post</button>
</div>
<hr>
<h5>Resultado:</h5>
<pre>{{resultadoPeticion|json}}</pre>
```

Finalmente, desde el navegador (http://localhost:4200) podrá probar las peticiones pulsando el botón que desee en cada caso.

Analizando la sintaxis de la llamada, seguramente ya habrá intuido que tanto get() como post() devuelven un **observable** sobre el que realizamos una suscripción. Y así es. Este y el resto de métodos de HttpClient para realizar peticiones HTTP devuelven observables de **tipo *cold***. Esto significa lo siguiente:

- La petición no se realiza hasta que se hace una suscripción al observable devuelto.
- Cada suscripción implicará una petición nueva.

Si desea más información de los observables consulte el capítulo “052 Servicios: Gestión asíncrona con observables”.

Por otra parte, las API no siempre nos devolverán datos en formato JSON. Si este fuera el caso, deberíamos indicarlo al realizar la petición con la parametrización correspondiente. Vea un ejemplo:

```
this.http.get('/assets/ficheroTexto.txt', {responseType: 'text'})
.subscribe(data => console.log(data));
```

056 HttpClient: Operaciones put, patch y delete

Continuando el análisis de métodos de HttpClient, en este ejercicio analizaremos los métodos put(), patch() y delete(), que usaremos para realizar peticiones HTTP de tipo "put", "path" y "delete", respectivamente.

- **PUT():** para actualizar un recurso del servidor.
- **PATCH():** para actualizar, de forma parcial, un recurso del servidor. Lo utilizaríamos cuando, por ejemplo, solamente necesitáramos actualizar una propiedad del recurso.
- **DELETE():** para eliminar un recurso del servidor.

Importante

Utilizaremos put() y patch() para actualizar recursos del servidor de forma total o parcial, respectivamente, y delete() para eliminarlos.

Vamos a probar todos estos métodos sobre la aplicación con la que estábamos trabajando anteriormente. Para ello, haremos uso de las siguientes rutas de la API REST de JSONPlaceholder (https://jsonplaceholder.typicode.com/):

Método	URL	Descripción
PUT	/posts/:id	Actualizar una publicación.
PATCH	/posts/:id	Actualizar parcialmente una publicación. Actualizar un subconjunto de propiedades.
DELETE	/posts/:id	Eliminar una publicación.

En los tres casos, ":id" correspondería al identificador de la publicación sobre la cual queremos realizar la acción. Vamos a realizar los cambios:

1. En primer lugar, abriremos el proyecto 054_http con el editor **(Atom)**, y pondremos en marcha la aplicación ejecutando **ng serve** desde la línea de comandos y desde dentro la carpeta **054_http**.

2. A continuación, añadimos las modificaciones en el componente:

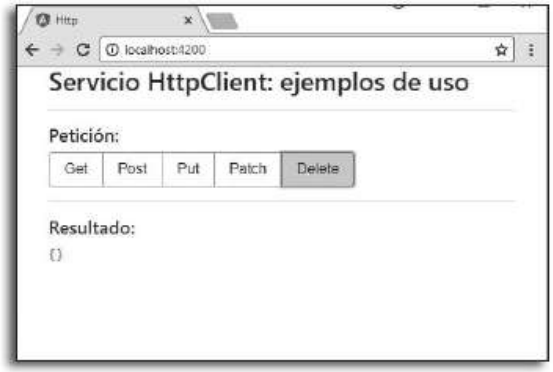
src/app/http-client-test/http-client-test.component.ts

```
…
@Component({…})
export class HttpClientTestComponent implements OnInit {
  resultadoPeticion;

  constructor(private http: HttpClient) { }
  …
  put(){
    this.http.put('https://jsonplaceholder.typicode.com/posts/1',
      {
      id: 1,
      title: 'Previsión Lunes',
      body: 'Lluvias.',
      userId: 1
      })
      .subscribe( data => { this.resultadoPeticion = data; } );
  }
  patch(){
    this.http.patch('https://jsonplaceholder.typicode.com/posts/1',
      {
      body: 'Soleado.'
      })
      .subscribe( data => { this.resultadoPeticion = data; } );
  }
  delete(){
    this.http.delete('https://jsonplaceholder.typicode.com/posts/1')
      .subscribe( data => { this.resultadoPeticion = data; } );
  }
}
```

src/app/http-client-test/http-client-test.component.html

```
<hr>
<h5>Petición:</h5>
<div class="btn-group" role="group" aria-label="Basic example">
…
  <button type="button" class="btn btn-secondary" (click)="put()">Put</
button>
  <button type="button" class="btn btn-secondary"
(click)="patch()">Patch</button>
  <button type="button" class="btn btn-secondary"
(click)="delete()">Delete</button>
</div>
<hr>
<h5>Resultado:</h5>
<pre>{{resultadoPeticion|json}}</pre>
```

Finalmente, desde el navegador (http://localhost:4200) ya podrá probar las peticiones pulsando el botón que desee en cada caso.



Por último, indicar que todas las peticiones HTTP que hemos visto también pueden realizarse mediante un único método genérico: **request()**. Veamos, por ejemplo, cómo podríamos realizar una petición "get" mediante el método request() en sustitución del método get() que habíamos visto en el capítulo anterior:

```
/*
this.http.get('https://jsonplaceholder.typicode.com/posts')
  .subscribe( data => { this.resultadoPeticion = data; } );
*/
this.http.request('GET','https://jsonplaceholder.typicode.com/posts')
  .subscribe( data => { this.resultadoPeticion = data; } );
```

057 HttpClient: Configuraciones adicionales sobre las peticiones HTTP

En este ejercicio analizaremos algunas configuraciones adicionales que podemos aplicar a las peticiones HTTP antes de que se envíen al servidor remoto. Los ejemplos que veremos los implementaremos sobre la aplicación creada en los anteriores ejercicios, por lo que recuerde abrir el proyecto 054_http con el editor (Atom) y poner en marcha la aplicación ejecutando "ng serve" desde 054_http en línea de comandos.

Importante

Con HttpParams podremos añadir parámetros a nuestras peticiones HTTP realizadas con HttpClient. Y con HttpHeaders, cabeceras personalizadas.

Añadir parámetros en el URL

Normalmente, en las API existen algunos "endpoint" que aceptan parámetros que condicionan la operativa que se va a realizar. Por ejemplo, en el caso de la API REST JSONPlaceholder (https://jsonplaceholder.typicode.com/) que hemos usado en los anteriores ejercicios, existe una ruta mediante la cual podemos indicar que solamente queremos las publicaciones de un determinado usuario:

Método	URL	Descripción
GET	/posts	Obtención de todas las publicaciones
GET	/posts?userId=:id	Obtención de las publicaciones del usuario con userId = :id

Con HttpClient, podemos realizar llamadas con parámetros usando la clase **HttpParams**. Veamos cómo usarla a través de un ejemplo:

1. Sobre nuestra aplicación, crearemos una nueva llamada get() para la ruta "/posts?userId=:id" de JSONPlaceholder, indicando que queremos las publicaciones del usuario con "userId" = 9. Una vez introducido el código, realice la llamada desde el navegador y compruebe su funcionamiento.

src/app/http-client-test/http-client-test.component.ts

```
import { Component, OnInit } from '@angular/core';
import { HttpClient, HttpParams } from '@angular/common/http';

@Component({…})
export class HttpClientTestComponent implements OnInit {
…
  get_param() {
    const params = new HttpParams().set('userId', '9');
    this.http.get('https://jsonplaceholder.typicode.com/posts',
{params})
      .subscribe( data => { this.resultadoPeticion = data; } );
  }
}
```

src/app/http-client-test/http-client-test.component.html

```
…
<button type="button" class="btn btn-secondary" (click)="get_
param()">Get (userId=9) </button>
…
```

HttpParams es **inmutable**, lo que significa que las llamadas a sus métodos no modifican el objeto, sino que devuelven de nuevo. Por este motivo, en el anterior

ejemplo tenemos la creación y configuración del objeto HttpParams en una misma instrucción. Un código como el siguiente no crearía un objeto HttpParams como el anterior, sino que crearía dos objetos HttpParams (aunque, para el segundo, no habríamos guardado ninguna referencia): el primer objeto no tendría parámetros y el segundo tendría uno.

```
const params = new HttpParams();
params.set('param1', "value1");
```

Añadir cabecera personalizada

En ciertos contextos necesitaremos añadir cabeceras personalizadas en nuestras peticiones HTTP. Un claro ejemplo de ello se produce en determinados sistemas de seguridad. En ellos, necesitaremos añadir una cabecera de autorización en todas las peticiones HTTP que realicemos. En esa cabecera, añadiríamos un "token" (o password) que permitiría al servidor remoto identificarnos para que este haga sus gestiones o las deniegue.

Con HttpClient, podemos añadir cabeceras personalizadas usando la clase **HttpHeaders**. Veamos un ejemplo:

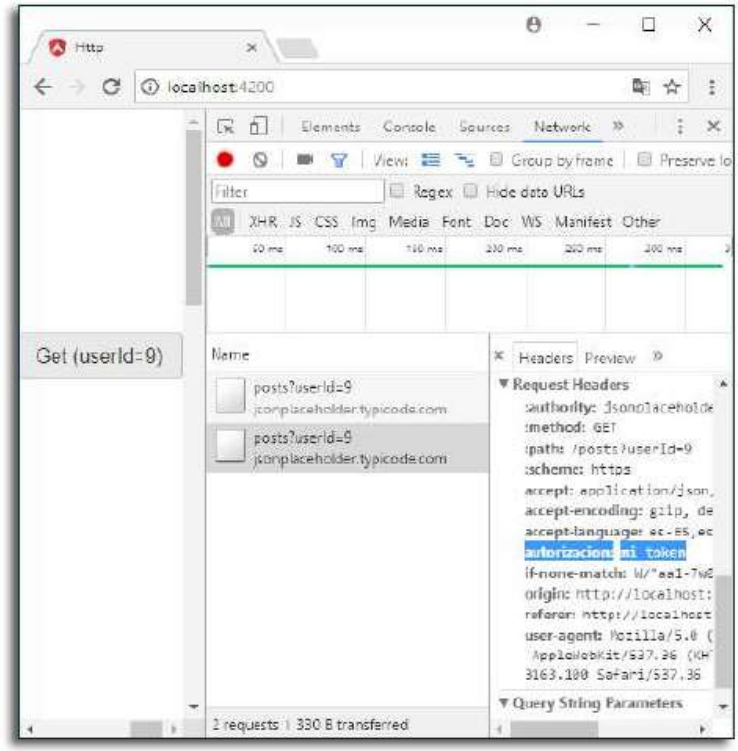
2. Añadiremos una cabecera personalizada con un "token" de autorización sobre el método get_param() que habíamos creado en el punto anterior.

src/app/http-client-test/http-client-test.component.ts

```
import { Component, OnInit } from '@angular/core';
import { HttpClient, HttpParams, HttpHeaders } from '@angular/common/
http';

@Component({…})
export class HttpClientTestComponent implements OnInit {
…
  get_param() {
    const headers = new HttpHeaders().set('Autorizacion', 'mi token');
    const params = new HttpParams().set('userId', '9');
    this.http.get('https://jsonplaceholder.typicode.com/posts',
{headers, params})
      .subscribe( data => { this.resultadoPeticion = data; } );
  }
}
```

Para comprobar que se ha añadido la cabecera personalizada utilizaremos los log de Google Chrome, por lo que antes de realizar la llamada desde

el navegador, deberá seleccionar la opción “Más herramientas/Herramientas para desarrolladores/Network” de Google Chrome.



Hay que tener en cuenta que, igual que pasaba con la clase **HttpParams**, HttpHeaders también es inmutable.

058 HttpClient: Gestión de respuestas y errores de peticiones HTTP

En este ejercicio analizaremos distintas gestiones que podemos realizar sobre respuestas de peticiones HTTP. Los ejemplos que veremos los implementaremos en la aplicación creada en los anteriores ejercicios, por lo que recuerde abrir el proyecto 054_http con el editor (Atom) y poner en marcha la aplicación ejecutando "ng serve" desde 054_http en línea de comandos.

Importante

Realice una buena gestión de errores de sus peticiones HTTP para tener bien monitorizadas sus aplicaciones.

Definir tipos para las respuestas

Normalmente, una petición HTTP devolverá una respuesta en formato JSON que HttpClient se encargará de parsear en un objeto. Para que ese objeto tenga propiedades vinculadas a los distintos elementos de la respuesta y así facilitarnos su gestión, será necesario que indiquemos a HttpClient el tipo o forma que tiene que tener. Veamos cómo podemos hacerlo implementándolo en las llamadas de nuestra aplicación:

1. Primero crearemos la interface que defina nuestra respuesta, que en nuestro caso es una publicación. Crearemos el archivo **post.ts** y le añadiremos el siguiente código:

src/app/post.ts

```
export interface Post {
  userId: number;
  id: number;
  title: string;
  body: string;
}
```

2. A continuación, usaremos la interface para definir el tipo de respuesta en cada una de las llamadas. Esto, por ejemplo, nos permitirá que en la llamada a "post" podamos mostrar por log el identificador de la publicación creada accediendo a la propiedad "id" del objeto data:

src/app/http-client-test/http-client-test.component.ts

```
import { Post } from '../post';
…
    this.http.get<Post[]>('https://jsonplaceholder.typicode.com/
posts')
      .subscribe( data => { this.resultadoPeticion = data; } );
…
    this.http.post<Post>('https://jsonplaceholder.typicode.com/
posts',{…})
      .subscribe( data => { this.resultadoPeticion = data;
                            console.log("Id. de la nueva publicación:
" + data.id)} );
…
```

Desde el navegador, pulsando la opción "post", podrá comprobar su funcionamiento.

Leer la respuesta completa

Por defecto, los métodos de HttpClient solamente devuelven la parte que forma el cuerpo (body) de la respuesta. Sin embargo, en algunos casos puede ser que nos interese ver la respuesta entera para consultar la cabecera o algún código de estado. Veamos cómo lo podríamos hacer.

3. Vamos a indicar en la llamada get() que queremos recibir la respuesta entera. Fíjese que, ahora, en la variable "resultadoPetición" debemos asignarle "data.body" y no "data".

src/app/http-client-test/http-client-test.component.ts

```
…
this.http.get<Post[]>('https://jsonplaceholder.typicode.com/posts',
{observe:'response'})
   .subscribe( data => { this.resultadoPeticion = data.body; console.
    log(data);} );
```

Desde el navegador podrá comprobar que al pulsar "get" recibimos el mensaje entero.

Gestión de errores

Los métodos de HttpClient devuelven observables, y tal como explicábamos en el capítulo de los observables ("052 Servicios: Gestión asíncrona con observables"), la gestión de errores se realizará añadiendo un controlador de error en la suscripción. Veamos un ejemplo.

1. Vamos a ponerlo en práctica primero añadiendo un control de error en la llamada a "put", y luego provocando un error fijando la actualización de una publicación inexistente (id publicación = 1.000):

src/app/http-client-test/http-client-test.component.ts

```
…
this.http.put<Post>('https://jsonplaceholder.typicode.com/
posts/1000',{…})
.subscribe(
  data => { this.resultadoPeticion = data; },
  err  => { console.log(err); });
```

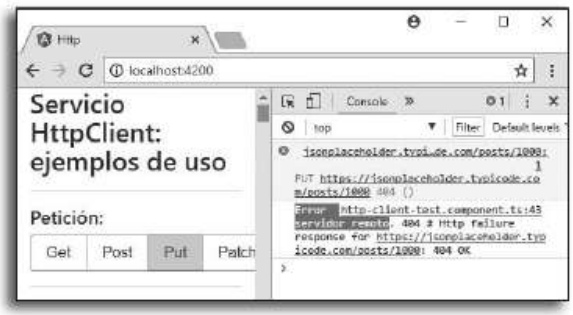
El error devuelto es de tipo **HttpErrorResponse**, y podemos obtener más detalles de ese error accediendo a sus propiedades. Por otra parte, en una petición HTTP se pueden producir dos grandes tipos de errores: error de cliente o red, o sea, error en el canal de comunicación, y error devuelto por el servidor remoto. Es recomendable que se gestionen por separado. Veamos cómo hacerlo.

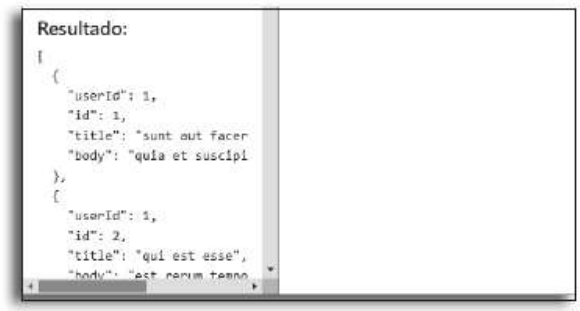
1. Para los errores de cliente o red, la propiedad "error" del objeto HttpErrorResponse es de tipo **Error**, por lo que para gestionar los dos tipos de error añadiremos el siguiente código:

src/app/http-client-test/http-client-test.component.ts

```
…
this.http.put<Post>('https://jsonplaceholder.typicode.com/
posts/1000',{…})
.subscribe(
  data => { this.resultadoPeticion = data; },
  (err: HttpErrorResponse)  => {
    if (err.error instanceof Error){
      console.log('Error cliente o red:', err.error.message);
    } else {
      console.log(`Error servidor remoto. ${err.status} # ${err.
message}`);
    }
  });
```

Pulsando la opción "put" de la aplicación podrá comprobar cómo se gestiona correctamente el error de publicación inexistente *(http 404 not found)*.





059 HttpClient: Intercepción de peticiones y respuestas

Una característica importante del módulo HttpClient ("HttpClientModule") es la **intercepción**, la posibilidad de crear interceptores de mensajes entre el frontend y backend para poderlos modificar, monitorizar, etc.

Para crear un interceptor declararemos una clase decorada con @Injectable() y que implemente la interfaz **HttpInterceptor**. La implementación de la interfaz requiere que añadamos el método **intercept()** en nuestra clase. Veamos un ejemplo de interceptor que simplemente realiza un log de la petición interceptada:

Importante

Los interceptores nos permitirán interceptar los mensajes transmitidos entre aplicación y backend para poder modificarlos y monitorizarlos.

```
@Injectable()
export class TestInterceptor implements
HttpInterceptor {
  intercept (req: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
      console.log(req);
      return next.handle(req);
    }
}
```

El método "intercept" tiene dos parámetros de entrada: req (petición) y next ("siguiente controlador"). El procedimiento sería el siguiente. El interceptor realiza las acciones que convengan sobre la petición interceptada (req) y realiza la llamada "next.handle(req)". Esta llamada tiene que realizarse para que el resto de interceptores (en caso que los tuviéramos) procesen la petición y para que esta acabe realizándose. Por último indicar que la llamada "next.handle(req)" devuelve un observable para la respuesta de la petición. Este observable también deberá devolverse por nuestro método "intercept".

Vamos a practicar con los interceptores en nuestra aplicación. Crearemos dos interceptores. El primero añadirá un "token" de autenticación en la cabecera de la petición. El segundo simplemente mostrará esta petición y su respuesta en el log. En el capítulo "057 HttpClient: Configuraciones adicionales sobre las peticiones HTTP" ya habíamos visto cómo añadir una cabecera de autenticación en una petición; sin

embargo, aquí, al hacerlo desde de un interceptor, podremos aplicarlo de forma automática para todas las peticiones que realicemos.

1. Primero abriremos el proyecto 054_http con el editor **(Atom)**, y, desde la carpeta **054_http**, pondremos en marcha la aplicación ejecutando **ng serve**.

2. Desde el editor (Atom), nos situaremos sobre la carpeta **app** y crearemos el fichero **testInterceptor.ts**. Dentro, añadiremos el código del primer interceptor:

src/app/testInterceptor.ts

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

import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest}
from '@angular/common/http';

import { Observable } from 'rxjs/observable';



@Injectable()

export class TestInterceptor implements HttpInterceptor {

  intercept (req: HttpRequest<any>, next: HttpHandler):

    Observable<HttpEvent<any>> {

      const modReq = req.clone({headers: req.headers.set('tI_
Autorizacion','token')});

      return next.handle(modReq);

    }

}
```

De este código cabe destacar lo siguiente. Las clases **HttpRequest** y **HttpResponse** son **inmutables**, no podemos modificarlos. Por este motivo, las modificaciones las realizamos sobre un clon o duplicado de la petición original. Esto es así para evitar que, en operaciones de reenvío, peticiones modificadas por la cadena de interceptores vuelvan a entrar, ya modificadas, por la misma cadena.

3. Seguidamente crearemos el segundo interceptor “testInterceptor2.ts” con el siguiente código:

src/app/testInterceptor2.ts

```
…
@Injectable()
export class TestInterceptor2 implements HttpInterceptor {
  intercept (req: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
      console.log(req);
      return next
        .handle(req)
        .do(event => { if (event instanceof HttpResponse) { console.
log(event);} });
    }
}
```

El operador do() de la librería RxJS agrega un efecto secundario a un observable sin afectar los valores en la secuencia. Aquí lo usamos para detectar el evento HttpResponse, o sea, la respuesta a la petición realizada, y mostrarlo por el log.

4. Para activar los interceptores en nuestra aplicación deberemos **configurarlos en el modulo principal**. Esto lo haremos de la siguiente forma:

src/app/app.module.ts

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';
…
@NgModule({ declarations: […], imports: […],
  providers:[{ provide: HTTP_INTERCEPTORS, useClass: TestInterceptor,
multi: true },
              { provide: HTTP_INTERCEPTORS, useClass: TestInterceptor2,
multi: true }],…
})…
```

El orden establecido en la configuración de los interceptores determinará su **orden de ejecución**, por tanto, la petición que veremos en el log ya contendrá la cabecera de seguridad.

5. Ejecutando las distintas llamadas implementadas podrá comprobar el funcionamiento de los interceptores. Por ejemplo, pulsando "post", podrá comprobar cómo aparecen petición y respuesta en el log. Además, si analiza la petición, podrá comprobar que incluye la cabecera de seguridad


Http
localhost:4200
Servicio HttpClient: ejemplos de uso
Petición:
Get
Post
Put
Patch
Resultado:
{
"title": "Previsión Viernes.
"body": "Parcialmente soleado
"userId": 1,
"id": 101
}
Elements
Console
top
Filter
Default levels
testInterceptor2.ts:10
HttpRequest {url:
"https://jsonplaceholder.typicode.com/post
testInterceptor2.ts:13
HttpResponse {headers: HttpHeaders, status
"https://jsonplaceholder.typicode.com/post
body:
body: "Parcialmente soleado."
id: 101
title: "Previsión Viernes."
userId: 1
__proto__: Object
headers: HttpHeaders {normalizedNames: M
ok: true
status: 201
statusText: "OK"
type: 4
url: "https://jsonplaceholder.typicode.c
__proto__: HttpResponseBase
Id. de http-client-test.component.ts:27
la nueva publicación: 101


Http
localhost:4200
Servicio HttpClient: ejemplos de uso
Petición:
Get
Post
Put
Patch
Resultado:
{
"title": "Previsión Viernes.
"body": "Parcialmente soleado
"userId": 1,
"id": 101
}
Elements
Console
top
Filter
Default levels
testInterceptor2.ts:10
HttpRequest {url: "https://jsonplaceholder.typicode.com/post",
"json", …}
body: {title: "Previsión Viernes.", bo
headers: HttpHeaders
headers: Map(1)
size: (...)
__proto__: Map
[[Entries]]: Array(1)
0: {"ti_autorizacion" => Array(1
length: 1
lazyInit: null
lazyUpdate: null
normalizedNames: Map(1) {"ti_autoriz
__proto__: Object
method: "POST"
params: HttpParams {updates: null, clo
reportProgress: false
responseType: "json"
url: "https://jsonplaceholder.typicode
urlWithParams: "https://jsonplaceholde
withCredentials: false
__proto__: Object
testInterceptor2.ts:13
HttpResponse {headers: HttpHeaders, sta
ok: true, …}
Id. http-client-test.component.ts:27
de la nueva publicación: 101

060 HttpClient: Combinación y sincronización de peticiones HTTP. Eventos de progreso

Los ejemplos que veremos a continuación, los implementaremos sobre la aplicación creada en los anteriores ejercicios, por lo que recuerde abrir el proyecto 054_http con el editor (Atom) y poner en marcha la aplicación ejecutando "ng serve" desde 054_http en línea de comandos.

Importante

Utilice todo el potencial de la librería RxJs para gestionar las peticiones HTTP según sus necesidades.

Lanzar peticiones HTTP en paralelo combinando su resultado

Para lanzar varias peticiones HTTP en paralelo y combinar su resultado podemos usar el operador **forkjoin** de la librería RxJs. Veamos un ejemplo:

1. Crearemos una nueva llamada para lanzar dos "get" sobre las publicaciones 4 y 5.

src/app/http-client-test/http-client-test.component.ts

```
peti_paral() {
    Observable.forkJoin(
       this.http.get<Post>('https://jsonplaceholder.typicode.com/
posts/4').delay(3000),
       this.http.get<Post>('https://jsonplaceholder.typicode.com/
posts/5')
    ).subscribe( data => { this.resultadoPeticion = data;} );
  }
```

src/app/http-client-test/http-client-test.component.html

```
<button type="button" class="btn btn-secondary" (click)="peti_
paral()">Peti. Paral.</button>
```

Una vez introducido el código, realice la llamada desde el navegador y compruebe su funcionamiento. Podrá comprobar que, pasados 3 segundos (debido al delay del primer "get"), se visualizan los datos.

"forkjoin" nos permite combinar varios observables, obtenidos de peticiones HTTP, en un nuevo observable. Este último emitirá una cadena con el último valor emitido por cada uno de los observables cuando todos ellos hayan acabado. Si los observables emitiesen más de un valor, solamente se tendría en cuenta el último.

Lanzar peticiones HTTP en secuencia

Un caso de uso común es el de lanzar una petición HTTP, recoger su resultado y, finalmente, usarlo para realizar otra petición HTTP. Esta secuencia de peticiones la podemos hacer usando el operador **switchMap** de la librería RxJs. Veamos un ejemplo:

1. Crearemos una nueva llamada para obtener una publicación, modificarla y transferirla:

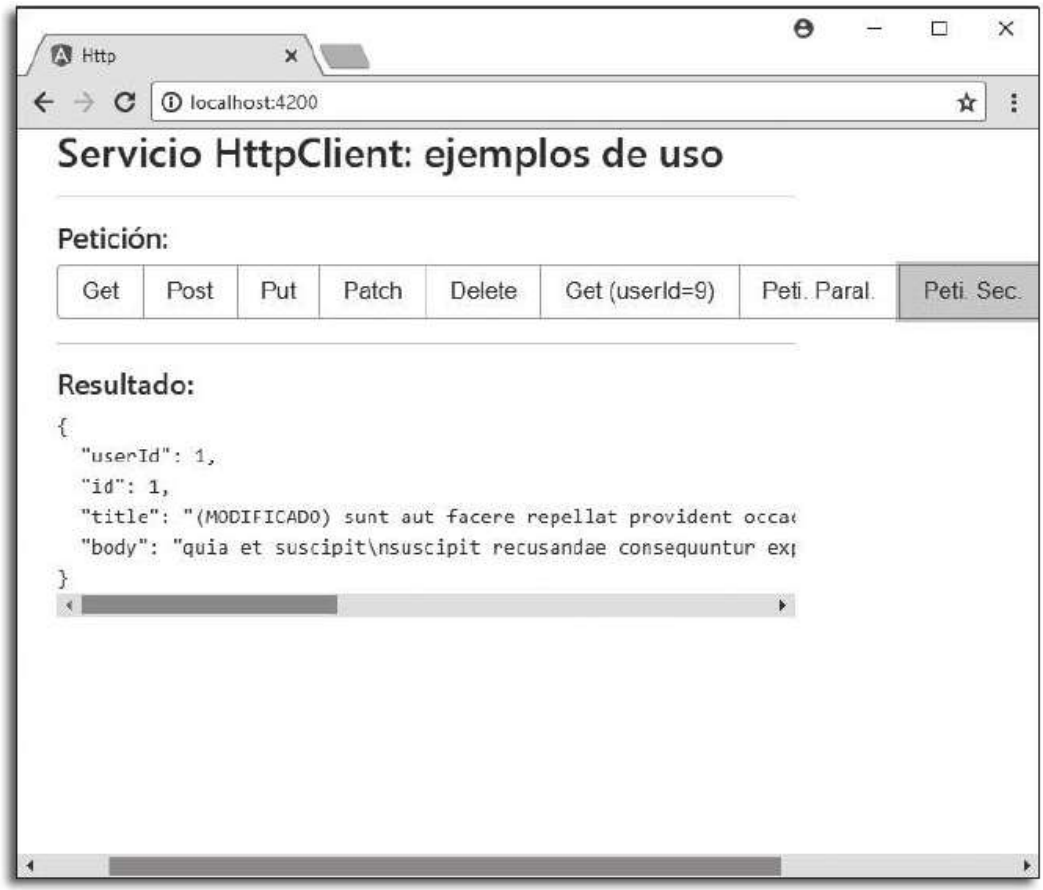
src/app/http-client-test/http-client-test.component.ts

```
peti_sec() {
    this.http.get<Post>('https://jsonplaceholder.typicode.com/
posts/1')
      .switchMap( data => {
         data.title = "(MODIFICADO) " + data.title;
         return
            this.http.put<Post>('https://jsonplaceholder.typicode.
com/posts/1',data)})
      .subscribe( data => { this.resultadoPeticion = data;} );
   }
```

src/app/http-client-test/http-client-test.component.html

```
<button type="button" class="btn btn-secondary" (click)="peti_
sec()">Peti.Sec.</button>
```

Una vez introducido el código, realice la llamada desde el navegador y compruebe su funcionamiento.



"switchMap" es parecido al operador "map" que vimos en el capítulo 053. Sin embargo, debemos usar "switchMap" cuando la función de transformación devuelva un observable y "map" cuando devuelva un valor.

Eventos de progreso

Hasta este momento todas las peticiones HTTP que hemos visto se realizaban de forma muy rápida porque el tamaño de los mensajes transmitidos era muy pequeño. Sin embargo, habrá veces que necesitemos transferir un mensaje mucho más grande que necesite cierto tiempo. En estos casos, siempre es bueno poder indicar al usuario el progreso de esa transferencia. Para poder hacerlo, deberemos capturar los eventos de progreso y lo haremos realizando las peticiones con el método genérico "**request(HttpRequest)**" (si desea más información consulte el capítulo 056) y la opción **reportProgres**. Veamos un ejemplo:

1. Crearemos una nueva llamada que realizará un "post" y capturaremos los eventos:

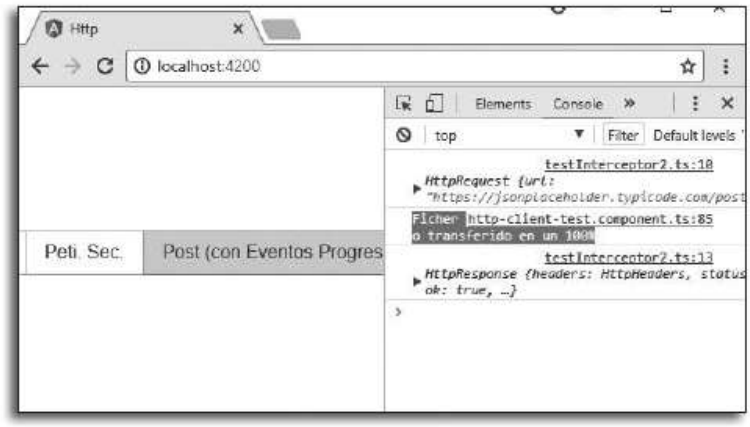
src/app/http-client-test/http-client-test.component.ts

```
post_prgEvents() {
    const request = new HttpRequest("POST", "https://
jsonplaceholder.typicode.com/posts", {title: 'Crítica de
la película', body: 'Me ha gustado mucho.', userId: 1},
{reportProgress: true});
    this.http.request(request)
        .subscribe(
            event => {
                if (event.type === HttpEventType.UploadProgress) {
                    const percentDone = Math.round(100 * event.
loaded / event.total);
                    console.log(`Fichero transferido en un
${percentDone}%`);
                } else if (event.type === HttpEventType.Response) {
                    this.resultadoPeticion = event.body;
                }
            }
        );
  }
```

src/app/http-client-test/http-client-test.component.html

```
<button type="button" class="btn btn-secondary" (click)="post_
prgEvents()">Post (con Eventos Progreso)</button>
```

Compruebe su funcionamiento. El mensaje del ejemplo es muy pequeño y con un solo log se indica que el fichero se ha transferido al 100%, pero con uno de más grande, veríamos más log de progreso de transferencia.



061 Forms: Introducción

Los formularios son uno de los elementos más importantes en las aplicaciones de gestión ya que son, en esencia, los que nos permiten obtener y mostrar información para registrarla o realizar acciones a partir de la misma.

En HTML disponemos de un buen conjunto de elementos mediante los cuales podemos fabricar nuestros formularios, entre los que destacamos los siguientes: **input**, **select**, **radio button**, **textarea**, **checkbox**, **buttons**, etc.

Importante

Use **form-group** para incluir campos en formularios asociando a cada elemento su etiqueta (**label**) en el mismo grupo.

Los inputs poseen una gran variedad de tipos donde, además de los clásicos **text**, **password** y **number**, podemos enumerar los siguientes: **color**, **date**, **datetime-local**, **email**, **month**, **range**, **search**, **tel**, **time**, **url**, **week**, etc. En función del tipo de input indicado, se facilita el tratamiento del tipo de dato relacionado con el mismo. Por ejemplo, si indicamos un tipo **number**, veremos que en el input solo se pueden introducir números.

Cada elemento puede llevar asociados una serie de eventos mediante los cuales podemos realizar acciones, ya sea desde el propio template o desde su componente asociado.

Los atributos más clásicos que pueden asociarse a muchos de los tipos son: **value**, **disabled**, **max**, **maxlength**, **min**, **pattern**, **readonly**, **required**, **size**, **step**.

Una buena práctica es incluir cada campo en un div al que asociaremos la clase **form-group** y en el que definamos una etiqueta y el propio control a utilizar. Al control le asociaremos también la clase **form-control**. Veamos un ejemplo:

```
<div class="form-group">
    <label for="codigo">Codigo</label>
    <input type="text" class="form-control">
</div>
```

En el siguiente ejercicio realizaremos una aplicación en la que incluiremos algún ejemplo de los elementos comentados anteriormente.

1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos el proyecto tecleando **ng new formIntro**.

2. Seguidamente, renombraremos el proyecto ubicándonos en **Ej100_angular**, y escribiendo lo siguiente:

```
C:\Ej100_angular>rename formIntro 061_formIntro
```

A continuación, abrimos el proyecto con nuestro editor y modificamos el título de la aplicación que se halla en la clase **AppComponent** del fichero **app.component.ts**:

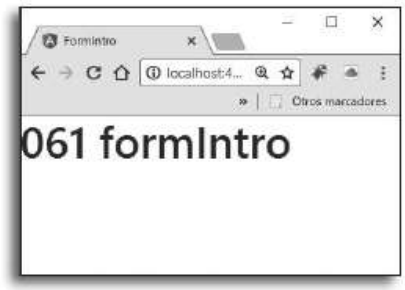
```
export class AppComponent {
  title    = '061 formIntro';
}
```

Para usar las clases de **Bootstrap**, modificaremos el fichero **app.component.html** para que, de momento, tenga el siguiente contenido:

```
<link rel="stylesheet" href="https://maxcdn.
   bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
   css/bootstrap.min.css" integrity="sha384-
   rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/
   AZzGRnGxQQKnKkoFVhFQhNUwEyJ"
   crossorigin="anonymous">
<h1>{{title}}</h1><br>
```

Ahora nos ubicaremos en **061_formIntro** y arrancaremos la aplicación tecleando lo siguiente:

```
C:\Ej100_angular>cd 061_formIntro
C:\Ej100_angular\061_formIntro >ng serve
```



A continuación, incluiremos los tags correspondientes al form y, dentro de los mismos, añadiremos el primer control de nuestro formulario que será por ejemplo “código”:

```
<div class=”container p-1 mx-3 marco”>
    <form>
        <div class=”form-group”>
            <label for=”codigo”>Codigo</label>
            <input type=”text” class=”form-control”>
        </div>
    </form>
</div>
```

La clase marco la definiremos también en nuestro archivo **styles.css** con el siguiente contenido:

```
.marco {
    border: 3px solid blue;
}
```

Guardamos y actualizamos el navegador para ver cómo queda.

3. A continuación, añadiremos unos cuantos controles más para tener un formulario con una muestra de los controles más típicos. Así pues, añadimos un control para los campos **nombre** (input text), **edad** (input number)), **opción** (select), **sexo** (radiobutton), **comentarios** (textarea), **activar** (checkbox) y **submit** (button).

4. Guardamos y actualizamos el navegador para ver cómo queda.

```
<div class="form-group">
    <label for="nombre">Nombre</label>
    <input type="text" class="form-control">
</div>
```

```
<div class="form-group">
  <label for="edad">Edad</label>
  <input type="number" class="form-control">
</div>
```

```
<div class="form-group">
  <label for="opcion">Opcion</label>
  <select class="form-control">
    <option>Uno</option>
    <option>Dos</option>
    <option>Tres</option>
  </select>
</div>
```


```
<label>Sexo</label><br>
<div class="form-group">
    <label><input class="mx-1" type="radio" checked="true">Hombre</label>
    <label><input class="mx-1" type="radio">Mujer</label>
</div>
```

```
<div class="form-group">
    <label for="comentarios">Comentarios</label>
    <div class="textarea-btn-holder form-group form-group-replace">
        <textarea class="form-control text-comment"></textarea>
    </div>
</div>
```

```
<div class="form-group">
    <label><input class="mx-1" type="checkbox" checked="true" >Activar</label>
</div>
```

```
<button type="submit" class="btn btn-primary m-x-2">Submit</button>
```

5. Seguidamente, añadimos el atributo **placeholder** al campo nombre para indicar al usuario qué es lo que hay que introducir en el mismo:

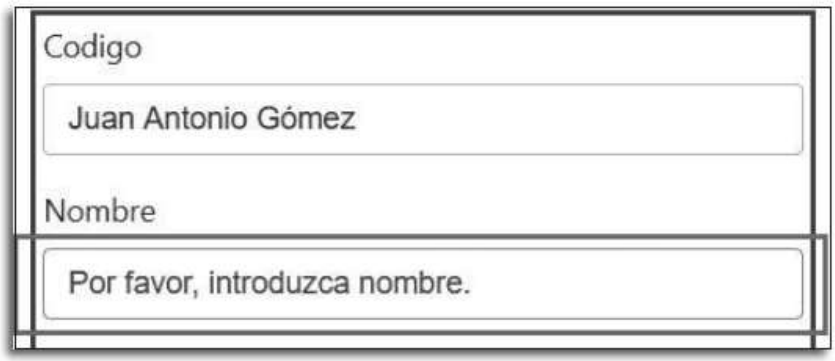
```
<input type="text" class="form-control" placeholder="Por
   favor, introduzca nombre.">
```

Añadimos al código una restricción para que no se puedan introducir más de 6 caracteres usando **maxlength**:

```
<input type="text" class="form-control" maxlength="6">
```

Incluimos un valor por defecto, un máximo y mínimo para la edad añadiendo los atributos **value**, **min** y **max**:

```
<input type="number" class="form-control" value="20"
   min="15" max="65">
```



062 Forms: Obtención de valores

Para acceder a los valores introducidos en los controles de un formulario, podemos utilizar diversos mecanismos. Uno de los mecanismos es el uso de variables locales que hagan referencia a los controles. Para poder trabajar con las propiedades de un control añadimos en su tag la definición **#nomVar**, la propiedad **name** y **ngModel**. Por ejemplo, para un supuesto control denominado **nombre**, usaríamos una expresión como la siguiente:

Importante

Use binding para poder modificar el valor de una propiedad de forma bidireccional (del template a la clase del componente y viceversa).

```
<input type="text" class="form-control" ngModel name="nombre"
   #nom>
```

Podemos comprobar que es posible utilizar **nom** en nuestro template mostrando su valor mediante interpolación (**nom {{ nom.value }}**). Otra posibilidad es realizar un **binding** (**two-way data binding**) para poder modificar el valor de una propiedad de forma bidireccional. Adaptando el ejemplo anterior, podríamos crear una variable **name** en nuestro archivo **component.ts** asociado al template y en el template definir lo siguiente:

```
<input type="text" class="form-control" [(ngModel)]="nombre"
   name="nombre">
```

En el siguiente ejercicio, usaremos los controles del ejercicio anterior **(Forms_Introduccion_Controles)**.

1. Nos ubicamos en **Ej100_angular**, crearemos el proyecto tecleando **ng new formValores** y, seguidamente, renombraremos el proyecto ubicándonos en **Ej100_angular**, y escribiendo lo siguiente:

```
C:\Ej100_angular>rename formValores 062_formValores
```

A continuación, copiamos los archivos **src/app/app.component.html** y **src/styles.css** del ejercicio anterior **(formIntro)** y los pegamos en nuestro proyecto **062_formValores** reemplazando los existentes.

2. Abrimos el proyecto y modificamos el título de la aplicación (dentro de **app.component.ts**):

```
export class AppComponent {

  title    = '062 formValores';

}
```

Ahora nos ubicaremos en **062_formValores**, arrancaremos la aplicación y teclearemos lo siguiente:


```
C:\Ej100_angular>cd 062_formValores

C:\Ej100_angular\061_formIntro >ng serve
```

A continuación, vamos a añadir una variable local para código definiendo lo siguiente:

```
<input type="text" class="form-control" maxlength="6"
    name="codigo" ngModel #cod>
```

Envolvamos el div que contiene el formulario en un **row** para mostrar 2 div: el div de formulario y el div que mostrará valores de la siguiente manera:

```
<div class="row p-2">

    <div class="container p-1 marco col-5">

        <form>

<!-- Contiene el form actual -->

        </form>

    </div>

    <div class="container col-5">

<!-- Para mostrar contenido de las variables
   -->

    </div>

</div>
```

El div que contiene el formulario ocupará 5 columnas (col-5) y en el div de variables incluiremos:

```
Cod: {{ cod.value }}
```

Guardamos el archivo y observamos como se muestra el contenido del campo código.

3. Ahora, obtendremos el valor del campo **codigo** y usando "**two-way data binding**" creando una variable en la clase **AppComponent** de la siguiente manera:

```
export class AppComponent {

  title = '062 formValores';

  codigo: string = "";

}
```

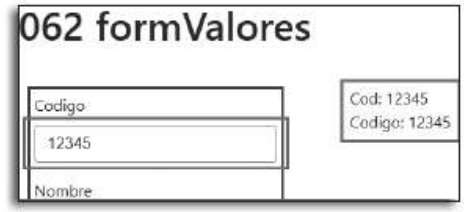
En nuestro tag input, introduciremos lo siguiente:

```
<input type="text" class="form-control" maxlength="6"
   [(ngModel)]="codigo" name="codigo" #cod>
```

En el **div** donde mostramos valores, tendremos:

```
<div class="container col-5">

    <!-- Para mostrar contenido de las variables -->

    Cod: {{ cod.value }}<br>

    Codigo: {{ codigo }}

</div>
```

Guárdelo todo y observe cómo se ve en el navegador.



4. A continuación, crearemos una función **(mostrarVar())** en nuestro componente, la cual, simplemente, mostrará por consola el contenido de la variable codigo de nuestro formulario y que asociaremos al evento **change** del control codigo. En la clase **AppComponent** añadiremos lo siguiente:

```
export class AppComponent {

  title = '062 formValores';

  codigo: string = "";

  mostrarVar() {

    console.log("Codigo (" + this.codigo + ")");

  }

}
```

En el tag **input** asociado al codigo añadiremos:

```
<input type="text" class="form-control" maxlength="6"
    [(ngModel)]="codigo" name="codigo" #cod
    (change)="mostrarVar()">
```

Muestre la consola (herramientas de desarrollador) del navegador (p. ej., en **Google Chrome** pulse **ALT+MAYUS+I**) y visualice el contenido de código al introducir un valor y cambiar el foco.

5. Defina también nombre, edad, opción, sexo, comentarios y activar, añadiendo en cada uno el **binding** (**[(ngModel)]= "campo"**), name (**name="campo"**) y la llamada a la función **mostrarVar()** mediante el evento **change**. En el componente, añada una variable por cada campo a tratar, defina un valor por defecto y muéstrelos en la consola. Añada la función **onSubmit** para asociarla al evento **Submit** del formulario.

```
<div class="form-group">
  <label for="nombre">Nombre</label>
  <input type="text" class="form-control"
  placeholder="Por favor, introduzca nombre."
  [(ngModel)]="nombre" name="nombre"
  (change)="mostrarVar()">
</div>
```

```
<div class="form-group">
  <label for="edad">Edad</label>
  <input type="number" class="form-control"
  value="20" min="15" max="65" placeholder="Introduzca edad (15-65)"
  [(ngModel)]="edad" name="edad"
  (change)="mostrarVar()">
</div>
```

```
<div class="form-group">
  <label for="opcion">Opcion</label>
  <select class="form-control"
  [(ngModel)]="opcion" name="opcion"
  (change)="mostrarVar()">
    <option value="1">Uno</option>
    <option value="2">Dos</option>
    <option value="3">Tres</option>
  </select>
</div>
```

```
<label>Sexo</label><br>
<div class="form-group">
  <label><input class="mx-1" type="radio"
    [(ngModel)]="sexo" name="sexo" value="hombre"
    (change)="mostrarVar()">Hombre</label>
  <label><input class="mx-1" type="radio"
    [(ngModel)]="sexo" name="sexo" value="mujer"
    (change)="mostrarVar()">Mujer</label>
</div>
```

```
<div class="form-group">
  <label for="comentarios">Comentarios</label>
  <div class="textarea-btn-holder form-group form-group-replace">
    <textarea class="form-control text-comment"
    [(ngModel)]="comentarios" name="comentarios"
    (change)="mostrarVar()"></textarea>
  </div>
</div>
```

```
<div class="form-group">
  <label><input class="mx-1" type="checkbox"
    [(ngModel)]="activar" name="activar"
    (change)="mostrarVar()" >Activar</label>
</div>
```

```
export class AppComponent {
  title = '062 formValores';
  codigo:string="";
  nombre:string="";
  edad:number = 20;
  opcion:string="2";
  comentarios:string="";
  sexo:string = "hombre";
  activar:boolean = true;
  mostrarVar(){
    console.log(
      "Codigo ("+ this.codigo + ") "
      + "Nombre (" + this.nombre+ ") "
      + "Edad (" + this.edad+ ") "
      + "Opcion (" + this.opcion+ ") "
      + "Comentarios (" + this.comentarios+ ") "
      + "Sexo (" + this.sexo+ ") "
      + "Activar (" + this.activar+ ")"
    );
  }
  onSubmit(){
    console.log("Submit");
  }
}
```

6. Muestre a través de la interpolación el contenido de las variables en el **div** creado a tal efecto.

```
<div class="container col-5">
  <!-- Para mostrar contenido de las variables -->
  Cod: {{ cod.value }}<br>
  Codigo: {{ codigo }}<br>
  Nombre: {{ nombre }}<br>
  Edad: {{ edad }}<br>
  Opcion: {{ opcion }}<br>
  Sexo: {{ sexo }}<br>
  Comentarios: {{ comentarios }}<br>
  Activar: {{ activar }}
</div>
```

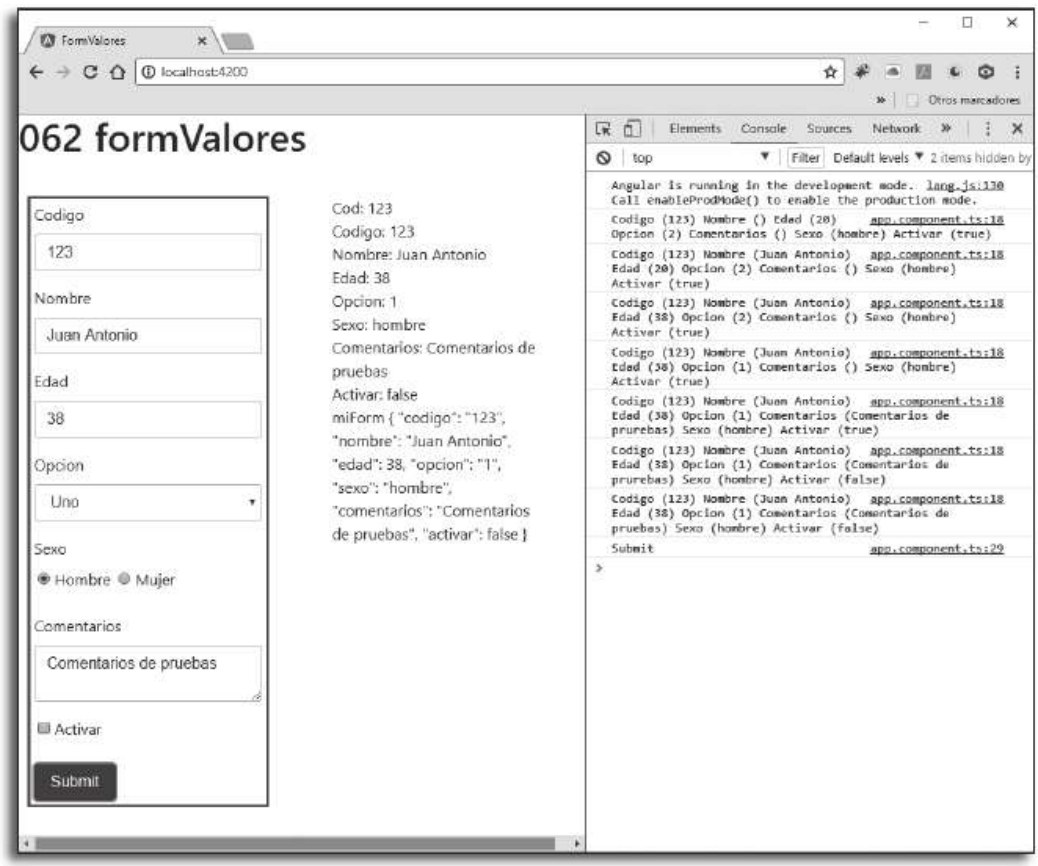
7. Por último, asocie al tag **form** la llamada a **onSubmit** mediante el evento **ngSubmit** y cree una variable local (**miForm**) para el formulario:

```
<form (ngSubmit)="onSubmit()" #miForm="ngForm">
```

Muestre la variable **miForm (miForm {{ miForm.value | json }})** aplicando un **pipe** con **json** para ver el contenido del objeto en el **div** donde venimos mostrando los valores:

```
    Activar: {{ activar }}<br>

    miForm {{ miForm.value | json }}

</div>
```

Añada un valor a cada campo y compruebe cómo se muestra su contenido en el template y en la consola. Pruebe el botón **Submit** para ver que se ejecuta la función de nuestro componente **onSubmit**.



063 Forms: Estado de los objetos

El estado de los objetos viene determinado por un conjunto de clases que nos permitirá averiguar si los objetos de un formulario han sido tratados o modificados o sin son válidos o no respecto de sus posibles restricciones.

En el ejercicio destinado a validaciones veremos algunos de sus posibles usos, pero en este ejercicio, nos vamos a dedicar a presentarlos y explicar cuál es la información que nos aportan. Los estados más importantes de un objeto están representados por alguna de las siguientes clases:

Importante

Compruebe el estado cuando quiera realizar alguna validación o detectar alguna situación respecto a un control.

Clases	Estado del control
ng-touched	Ya ha tenido el foco en algún momento.
ng-untouched	Aún no ha tenido el foco.
ng-dirty	Se ha hecho alguna modificación.
ng-pristine	Aun no se ha modificado nada.
ng-valid	Posee un contenido válido.
ng-invalid	Posee un contenido inválido.
ng-pending	Validaciones asíncronas pendientes.

Un objeto posee varias clases asociadas simultáneamente, aunque algunas de ellas son incompatibles entre sí como pueden ser **ng-touched** con **ng-untouched**, **ng-dirty** con **ng-pristine**, o **ng-valid** con **ng-invalid**.

En el siguiente ejercicio realizaremos una aplicación sencilla en la que incluiremos algunos controles que nos ayudarán a entender el funcionamiento de estas clases.

1. Nos ubicaremos en **Ej100_angular** y crearemos el proyecto tecleando **ng new formEstado**.

2. Seguidamente, renombraremos el proyecto ubicándonos en **Ej100_angular**, y escribiendo lo siguiente:

```
C:\Ej100_angular>rename formEstado 063_formEstado
```

A continuación, abrimos el proyecto con nuestro editor y modificamos el título de la aplicación que se halla en la clase **AppComponent** del fichero **app.component.ts**:

```
export class AppComponent {
  title    = '063 formEstado';
}
```

Para usar las clases de **Bootstrap**, incluiremos en el fichero **app.component.html** lo siguiente:

```
<link rel="stylesheet" href="https://maxcdn.
   bootstrapcdn.com/bootstrap/4.0.0-alpha.6/
   css/bootstrap.min.css" integrity="sha384-
   rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/
   AZzGRnGxQQKnKkoFVhFQhNUwEyJ"
   crossorigin="anonymous">
<h1>{{title}}</h1><br>
```

Ahora nos ubicaremos en **063_formEstado** y arrancaremos la aplicación tecleando lo siguiente:

```
C:\Ej100_angular>cd 063_formEstado
C:\Ej100_angular\061_formIntro >ng serve
```

A continuación, incluiremos un div con los tags correspondientes al form y, dentro de los mismos, añadiremos el primer control de nuestro formulario que será, por ejemplo, **codigo**. También añadiremos otro div para mostrar las clases asociadas al control mediante **className** y también su propiedad **value**:

```
<div class="container-fluid m-1 p-1 marco">
    <form #miForm="ngForm">
        <div class="form-group">
            <label for="codigo">Codigo</label>
            <input type="text" class="form-control"
   name="codigo" ngModel #codigo required>
        </div>
    </form>
</div>
<div class="container-fluid fondo p-1 m-1">
    Codigo clases: <b>{{ codigo.className }}</b><br>
    Codigo: <b>{{ codigo.value }}</b>
</div>
```

Vemos que también hemos declarado una variable local para el form mediante **#miForm=”ngForm”**. Observe que, en esta ocasión, hemos incluido solo la propiedad **ngModel** y la variable local **#codigo** para podernos referir al control. Las clases **marco** y **fondo** las definiremos en nuestro archivo **styles.css** con el siguiente contenido:

```
.marco {
    border: 3px solid green;
}
.fondo {
    border: 1px solid red;
    background-color: lightblue;
}
```

Guardamos y actualizamos el navegador para ver cómo queda.

3. A continuación, añadiremos algunos controles más (**nombre**, **email** y **movil**) para tener un formulario más variado y aprovechamos para agrupar los campos **email** y **movil** y mostrar un ejemplo de agrupación. Para agrupar varios campos, los incluiremos en un div al que asignaremos la propiedad **ngModelGroup** y también una variable local (**#contacto**) con la que reconocer dicha agrupación:

```
<div ngModelGroup=”contacto” #contacto=”ngModelGroup”>
```

```
<div class="form-group">
  <label for="nombre">Nombre</label>
  <input type="text" class="form-control" name="nombre" ngModel #nombre>
</div>
```

```
<div class="form-group">
  <label for="email">Email</label>
  <input type="email" class="form-control" name="email" ngModel #email>
</div>
```

```
<div class="form-group">
  <label for="movil">Movil</label>
  <input type="number" class="form-control" name="movil" ngModel #movil>
</div>
```

```
<div ngModelGroup="contacto" #contacto="ngModelGroup">
  <div class="form-group">
    <label for="email">Email</label>
    <input type="email" class="form-control" name="email" ngModel #email>
  </div>
  <div class="form-group">
    <label for="movil">Movil</label>
    <input type="number" class="form-control" name="movil" ngModel #movil>
  </div>
</div>
```

En el div donde mostramos las clases asociadas a cada control y su contenido, añadiremos una línea para cada control.

```
<div class="container-fluid fondo p-1 m-1">
  Codigo clases: <b>{{ codigo.className }}</b><br>
  Nombre clases: <b>{{ nombre.className }}</b><br>
  Email clases: <b>{{ email.className }}</b><br>
  Movil clases: <b>{{ movil.className }}</b><br>
  Codigo: <b>{{ codigo.value }}</b><br>
  Nombre: <b>{{ nombre.value }}</b><br>
  Email : <b>{{ email.value }}</b><br>
  Movil : <b>{{ movil.value }}</b><br>
  Form  : <b>{{ miForm.value | json }}</b>
</div>
```

4. Guardamos y actualizamos el navegador para ver cómo queda.

5. Podemos observar que el código muestra las variables: **form-control ng-untouched ng-pristine ng-invalid**. **ng-invalid** es mostrado porque hemos incluido el atributo **required** y, de momento, está vacío. **ng-pristine** se muestra porque el contenido del control no ha variado desde que se mostró el formulario por primera vez. Y **ng-untouched** se muestra porque aún no ha tenido el foco en ningún momento.

6. Si hacemos clic con el ratón sobre el campo codigo e introducimos el valor '**1**', observamos como ahora aparecen otras clases, ya que el control ha sido tocado (**ng-touched**), se ha modificado su valor inicial (**ng-dirty**) y se ha convertido en válido (**ng-valido**), ya que es requerido y no está vacío.

7. Rellenemos el resto de controles (**nombre**, **email** y **movil**) y veamos cómo muestran sus valores el div del pie de página y también cómo se visualiza el contenido de cada uno de los controles del form gracias a la sentencia **{{ miForm.value | json }}**.

064 Forms: Validaciones

Las validaciones en los formularios son imprescindibles para poder recoger la información con la calidad necesaria antes de enviarla para su procesamiento posterior. Podemos realizar tantas validaciones como sean necesarias en cada uno de los controles que queramos y habilitar o deshabilitar el botón **Submit** del formulario dependiendo de si hay o no errores en el formulario.

En el siguiente ejercicio crearemos un formulario sencillo solo con dos campos que nos permitirán analizar los detalles básicos y más importantes sobre las validaciones.

Importante

Valide cada una de las introducciones de los formularios para garantizar la calidad de los datos recogidos y no envíe el formulario hasta que esté completamente validado.

1. Nos ubicaremos en **Ej100_angular** y crearemos el proyecto tecleando **ng new formValida**.

2. Seguidamente, renombraremos el proyecto ubicándonos en **Ej100_angular**, y escribiendo lo siguiente:

```
C:\Ej100_angular>rename formValida 064_formValida
```

Abrimos el proyecto y modificamos el título de la aplicación en el fichero **app.component.ts**:

```
export class AppComponent {

  title    = '064 formValida';

}
```

Para usar las clases de **Bootstrap**, incluiremos al principio del fichero **app.component.html** lo siguiente:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
   com/bootstrap/4.0.0-alpha.6/css/bootstrap.
   min.css" integrity="sha384-rwoIResjU2yc3z8GV/
   NPeZWAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNUwEyJ"
   crossorigin="anonymous">

<h1>{{title}}</h1><br>
```

Importamos **FormsModule** en el archivo **app.module.ts** y lo añadimos al bloque de **imports**.

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

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

3. A continuación, incluiremos 2 div en nuestro archivo **app.component.html**. Uno para definir el formulario y un control (código); otro para mostrar algunas propiedades de los controles y del formulario en sí. Tienen el siguiente contenido:

```
<div class="container-fluid marco p-1 m-1">
  <form #miForm="ngForm">
    <div class="form-group">
      <label for="codigo">Codigo</label>
      <input type="text" class="form-control"
   name="codigo"
      ngModel #codigo="ngModel" #_codigo required>
    </div>
  </form>
</div>
<div class="container-fluid fondo p-1 m-1">
  <!-- Mostrar propiedades -->
  Codigo clases: <b>{{ _codigo.className }}</b><br>
  Codigo: <b>{{ codigo.value }}</b><br>
</div>
```

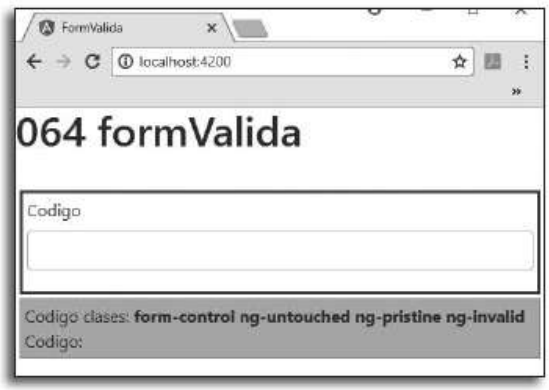
En el archivo **styles.css** incluimos un par de definiciones como las siguientes para los div:

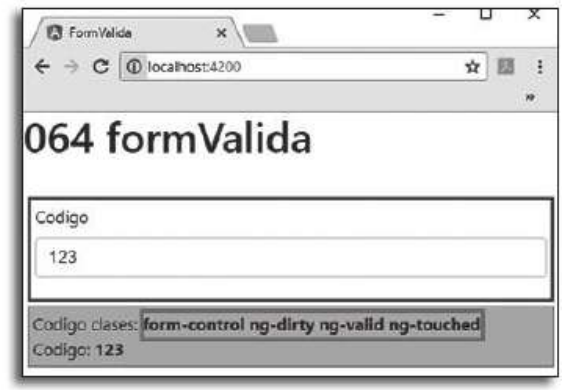
```
.marco { border: 3px solid green; }
.fondo { border: 1px solid red; background-color:
   lightblue; }
```

Ahora nos ubicaremos en **064_formValida** y arrancaremos la aplicación tecleando lo siguiente:

```
C:\Ej100_angular>cd 064_formValida
C:\Ej100_angular\061_formIntro >ng serve
```

Observe las clases mostradas en el div inferior y vea cómo cambian al introducir un valor en el campo **codigo**.

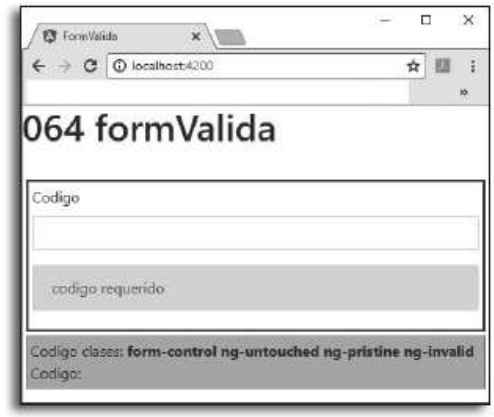




4. Cuando el campo **codigo** está vacío, se muestra la clase **ng-invalid**. Eso denota que existe un error. Para verlo, añadimos un div y con ***ngIf** lo mostramos, dependiendo de si hay o no error. Así pues, vamos a añadir debajo del control de **codigo**, lo que resaltamos a continuación:

```
...
    <input type="text" class="form-control"
   name="codigo"
      ngModel #codigo="ngModel" #_codigo required>
    </div>
    <div *ngIf="codigo.invalid" class="alert alert-
   danger">
      codigo requerido
    </div>
```

Si el código es erróneo, se mostrará una alerta.



5. Para evitar que la alerta se muestre por primera vez, añadiremos la condición de que haya tenido el foco al menos una vez. Para ello, modificaremos el ***ngIf** de la siguiente manera:

```
<div *ngIf="codigo.invalid && codigo.touched"
   class="alert alert-danger">
```

Ahora no se muestra la alerta hasta que seleccionamos el control y lo abandonamos vacío.

6. A continuación, añadimos otro requisito más al código para tener 2 posibles errores. Exigimos que el código tenga una longitud mínima de 3 caracteres añadiendo **minlength="3"**:

```
ngModel #codigo="ngModel" #_codigo required
   minlength="3">
```

Ahora, matizaremos a qué error se refiere cuando se detecte que el código es inválido. Verificaremos si el código ha sido modificado o no. El código de nuestras alertas quedará de la siguiente manera:

```
   <div *ngIf="codigo.invalid && (codigo.dirty ||
   codigo.touched)">
      <!-- hay errores -->
      <div *ngIf="codigo.errors.required" class="alert
   alert-danger">
         codigo requerido
      </div>
      <div [hidden]="!codigo.errors.minlength"
   class="alert alert-danger">
         Debe tener 3 caracteres como mínimo.
      </div>
    </div>
```

Además de ***ngIf** visto para **required**, también podríamos usar **[hidden]** para mostrar o no el error del mínimo de caracteres dependiendo de si se da o no. Guarde el archivo y compruebe cómo funciona.

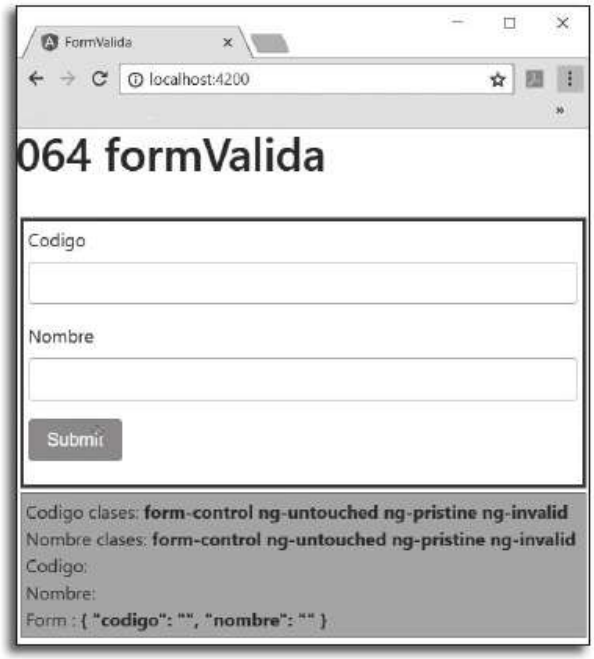
7. Añadimos un botón **Submit** antes del cierre del formulario para poderlo habilitar o deshabilitar en función de si todo el formulario es válido.

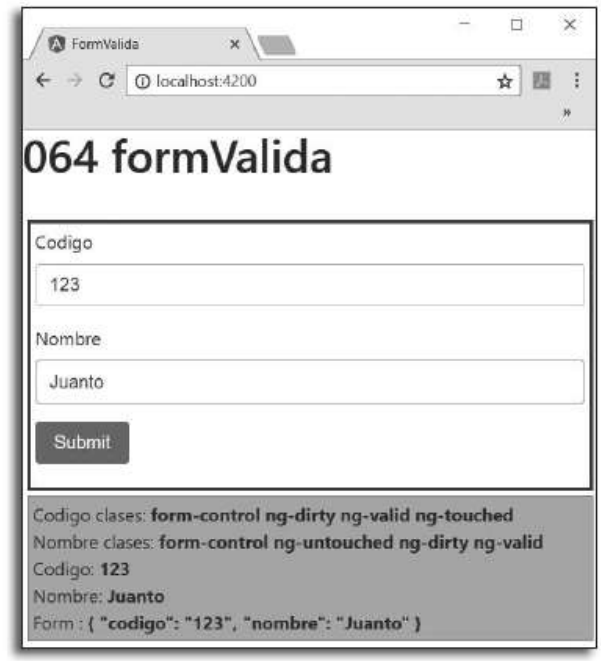
```
        nombre requerido
      </div>
    </div>
    <div class="form-group">
      <button type="submit" class="btn btn-success"
      [disabled]="!miForm.form.valid">Submit</button>
    </div>
  </form>
</div>
```

8. Añadimos también un nuevo campo denominado **nombre** para tener dos posibles controles a validar y modificamos el div inferior (mostrando valores) para añadir **nombre** y el contenido de todo el **form**.

```
<div class="container-fluid fondo p-1 m-1">
  <!-- Mostrar propiedades -->
  Codigo clases: <b>{{ _codigo.className }}</b><br>
  Nombre clases: <b>{{ _nombre.className }}</b><br>
  Codigo: <b>{{ codigo.value }}</b><br>
  Nombre: <b>{{ nombre.value }}</b><br>
  Form  : <b>{{ miForm.value | json }}</b>
</div>
```

9. Observe que el botón **Submit** está deshabilitado gracias a la instrucción **[disabled]="!miForm.form.valid"**. Efectivamente, ambos campos son requeridos y ahora están vacíos. Introduzca un valor aceptable en cada campo y compruebe cómo se habilita el botón **Submit**.


FormValida
localhost:4200
064 formValida
Codigo
Nombre
Submit
Codigo clases: form-control ng-untouched ng-pristine ng-invalid
Nombre clases: form-control ng-untouched ng-pristine ng-invalid
Codigo:
Nombre:
Form : { "codigo": "", "nombre": "" }


FormValida
localhost:4200
064 formValida
Codigo
123
Nombre
Juanto
Submit
Codigo clases: form-control ng-dirty ng-valid ng-touched
Nombre clases: form-control ng-untouched ng-dirty ng-valid
Codigo: 123
Nombre: Juanto
Form : { "codigo": "123", "nombre": "Juanto" }

065 Forms: Validaciones personalizadas

En ocasiones, necesitamos realizar personalizaciones algo más complicadas que las mencionadas en ejercicios anteriores. Para estos casos, podemos crear una validación personalizada en la que, mediante una función, controlemos todos los detalles que sean necesarios. La idea es crear un archivo adicional que contenga la directiva que usaremos en la validación e importarla en el archivo **app.module.ts**.

En el siguiente ejercicio crearemos una validación especial que consistirá en permitir introducir solo letras mayúsculas en el campo sobre el que la apliquemos.

Para ello, utilizaremos de base el formulario desarrollado en el ejercicio anterior **(formValida)** y le añadiremos nuestra validación especial al campo nombre.

1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos el proyecto tecleando **ng new formValidaEsp**.

2. Seguidamente, renombraremos el proyecto ubicándonos en **Ej100_angular**, y escribiendo lo siguiente:

```
C:\Ej100_angular>rename formValidaEsp 065_formValidaEsp
```

   Ahora aprovecharemos algunos ficheros del ejercicio anterior **(formValida)** y copiaremos los siguientes archivos para pegarlos en nuestro proyecto en sus respectivas carpetas:

```
..\src\ styles.css
..\src\app\app.component.html
..\src\app\app.module.ts
```

   A continuación, abrimos el proyecto y modificamos el título de la aplicación en el fichero **app.component.ts**:

```
export class AppComponent {
  title    = '065 formValidaEsp';
}
```

Guardemos el archivo y abramos el navegador para comprobar que nuestro punto de partida es idéntico al punto en el que finalizó el ejercicio anterior y que la copia ha ido perfectamente.

3. Ahora, vamos a crear la directiva que nos permitirá realizar nuestra validación especial. Para ello, crearemos un nuevo fichero denominado **src\app\validacaracteres.directive.ts** con el siguiente contenido:

```
import { Directive } from '@angular/core';
import { AbstractControl, NG_VALIDATORS } from "@angular/
   forms";

function filtrarCaracteres(caracter:AbstractControl){
  if (caracter.value == null) return null;
  var contenido = caracter.value;
  for(var i = 0; i < contenido.length; i++){
    var letra = contenido.substr(i,1);
    var valor = letra.charCodeAt(0);
    if (!(valor >=65 && valor <= 90)){
      return {filtrarCaracteres: true};
    }
  }
  return null;
}

@Directive({
  selector: '[filtrar-caracteres]',
  providers:[
    {provide: NG_VALIDATORS, multi: true, useValue:
   filtrarCaracteres}
  ]
})
export class FiltrarCaracteres {
}
```

Lo más destacable en este archivo es la función **filtrarCaracteres** que será la que nos permitirá analizar cada uno de los caracteres introducidos en el

campo que estemos validando y determinar si dicho carácter es una letra mayúsculas o no. Para ello, nos basaremos en el código **ASCII** de cada carácter de forma que, si no está comprendido entre **65** y **90**, indicaremos que es un error **(filtrarCaracteres: true)**.

Importante

Use validaciones personalizadas cuando necesite realizar validaciones complejas que no puedan resolverse con las validaciones que Angular propone por defecto.

4. Una vez creada la directiva, hemos de incluirla en el archivo **app.module.ts**.

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

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

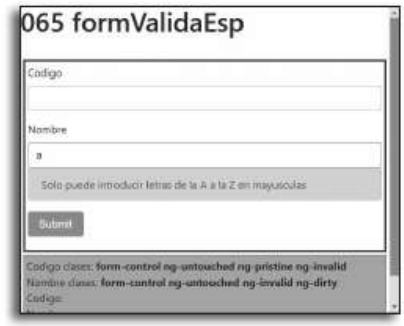
5. Seguidamente, añadimos nuestra directiva al campo nombre para que actué la validación de forma que nuestro input quedará de la siguiente manera:

```
<input type="text" class="form-control" name="nombre"
 ngModel #nombre="ngModel" #_nombre required filtrar-
   caracteres>
```

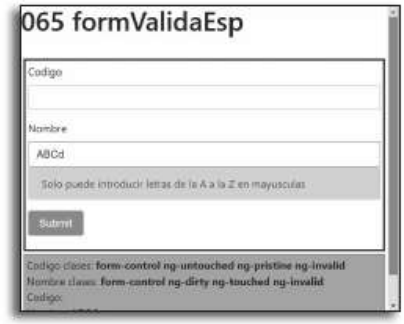
Ahora, solo nos queda añadir una alerta que se muestre cuando se introduzca algún carácter que no sea una letra en mayúsculas.

```
<div class="form-group">
  <label for="nombre">Nombre</label>
  <input type="text" class="form-control" name="nombre"
  ngModel #nombre="ngModel" #_nombre required filtrar-caracteres>
  <div *ngIf="nombre.invalid && (nombre.dirty || nombre.touched)">
    <div *ngIf="nombre.errors.required" class="alert alert-danger">
      Nombre requerido
    </div>
    <div *ngIf="nombre.errors.filtrarCaracteres" class="alert alert-danger">
      Solo puede introducir letras de la A a la Z en mayusculas
    </div>
  </div>
</div>
```

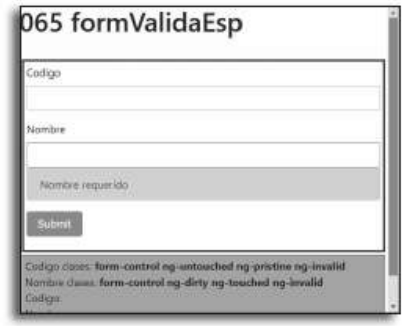
6. Guarde el archivo y compruebe que, al introducir una letra en minúsculas, aparece un error indicando que no es posible.



7. Observe también que el error aparece si introducimos un carácter inválido en cualquier posición del campo.



8. Deje el campo vacío y compruebe que también se muestra el error de "**Nombre requerido**".



9. Observe también que el botón **Submit** no se habilitará hasta que todo el **form** contenga valores válidos.

066 Forms: Reactive

Los **reactive forms** o **model-driven** se diferencian de los **template-driven** en que la creación de los controles se hace en la clase del componente. Otra diferencia importante es que los **reactive forms** son síncronos mientras que los **template-driven** son asíncronos. Por otra parte, los reactive forms permiten añadir controles dinámicamente y facilitan los test unitarios.

En la plantilla, simplemente definiremos los tags **HTML** correspondientes a los inputs, pero estos quedarán relacionados con la clase del componente gracias a **formControlName** y **formGroupName** para los controles y grupos de controles, respectivamente.

Importante

Utilice **reactive forms** cuando necesite crear controles dinámicamente y/o requiera realizar pruebas unitarias, ya que será mucho más fácil.

En el siguiente ejercicio crearemos un formulario sencillo con cuatro campos de los cuales dos los trataremos de forma individual, y al final del ejercicio los otros dos formarán un grupo.

1. Nos ubicaremos en **Ej100_angular** y crearemos el proyecto tecleando **ng new formReactive**.

2. Seguidamente, renombraremos el proyecto ubicándonos en **Ej100_angular**, y escribiendo lo siguiente:

```
C:\Ej100_angular>rename formReactive 066_formReactive
```

Abrimos el proyecto y modificamos el título de la aplicación en el fichero **app.component.ts**:

```
export class AppComponent {

  title    = '066 formReactive';

}
```

Para usar las clases de **Bootstrap**, incluiremos al principio del fichero **app.component.html** lo siguiente:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
   com/bootstrap/4.0.0-alpha.6/css/bootstrap.min.css" in-
   tegrity="sha384-rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/
   AZzGRnGxQQKnKkoFVhFQhNUwEyJ" crossorigin="anonymous">

<h1>{{title}}</h1><br>
```

Importamos **ReactiveFormsModule** (y **HttpModule** si no lo tenemos) en el archivo **app.module.ts** y lo añadimos al bloque de **imports**.

3. En **app.component.html** definiremos nuestro **form** y un primer control para **codigo** añadiendo detrás del título lo siguiente:

```
<div class="container-fluid marco p-1 m-1">

  <form>

    <div class="form-group">

      <label for="codigo">Codigo</label>

      <input type="text" class="form-control" #codigo>

    </div>

  </form>

</div>

<div class="container-fluid fondo p-1 m-1">

  Codigo: {{ codigo.value }}<br>

</div>
```

A continuación, incluiremos el siguiente código en el fichero **styles.css**:

```
.marco { border: 3px solid green; }

.fondo { border: 1px solid red;background-color: light-
   blue; }
```

Seguidamente, en el archivo **app.component.ts** hemos de importar:

```
import { FormGroup, FormBuilder } from '@angular/forms';
```

Dentro de la clase **AppComponent**, definiremos una variable para el form y usaremos el constructor para inyectar **FormBuilder** y poder crear un objeto que represente los controles que va a tener nuestro formulario que, por el momento, solo tiene un código. La clase **AppComponent** contendrá lo siguiente:

```
export class AppComponent {

  title = '066 formReactive';

  form: FormGroup;

  constructor(private fctrl:FormBuilder ){

    this.form=fctrl.group({

      codigo:''

    })

  }

}
```

Vemos la variable **form** que representa nuestro form y, dentro del constructor, creamos un grupo de controles que asignaremos a dicho form.

4. Ahora, solo nos queda modificar el tag **form** de la siguiente manera:

```
  <form [formGroup]='form'>
```

Y el control de código para añadir **formControlName**, dejándolo así:

```
<input type="text" class="form-control" formControlNa-
   me="codigo" #codigo>
```

Salve todos los archivos y arranque la aplicación ubicándonos en **C:\Ej100_angular\066_formReactive** tecleando **ng serve**. Introduzca algún valor para ver cómo funciona.

5. A continuación, detrás del control **codigo**, insertaremos los campos **nombre**, **aficion1** y **aficion2** en **app.component.html**. También añadiremos el botón **Submit**.

```
<div class="form-group">
  <label for="nombre">Nombre</label>
  <input type="text" class="form-control"
  formControlName="nombre" #nombre>
</div>
```

```
<div class="form-group">
  <label for="aficion1">Aficion 1</label>
  <input type="text" class="form-control"
  formControlName="aficion1" #aficion1>
</div>
```

```
<div class="form-group">
  <label for="aficion2">Aficion 2</label>
  <input type="text" class="form-control"
  formControlName="aficion2" #aficion2>
</div>
```

```
<div class="form-group">
  <button type="submit" class="btn btn-success">Submit</button>
</div>
```

6. En el div inferior, añadiremos la visualización de los valores de los controles añadidos y del formulario en general.

```
<div class="container-fluid fondo p-1 m-1">
  Codigo: {{ codigo.value }}<br>
  Nombre: {{ nombre.value }}<br>
  Aficion 1: {{ aficion1.value }}<br>
  Aficion 2: {{ aficion2.value }}<br>
  {{ form.value | json }}
</div>
```

7. En **app.component.ts**, también hemos de añadir la definición de los controles nuevos.

```
export class AppComponent {
  title = '066 formReactive';
  form: FormGroup;
  constructor(private fctrl:FormBuilder ){
    this.form=fctrl.group({
      codigo:'',
      nombre:'',
      aficion1:'',
      aficion2:''
    })
  }
}
```

8. Guarde el archivo y compruebe cómo funciona en el navegador introduciendo algunos valores.

9. Vamos a dar funcionalidad al botón **Submit** y, para ello, modificaremos el tag **form** para que contenga lo siguiente:

```
<form [formGroup]='form'  (ngSubmit)="onSubmit()">
```

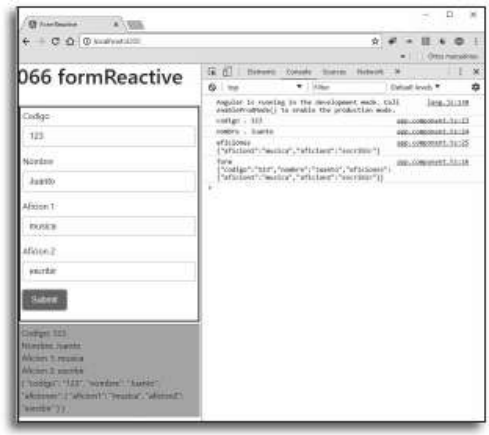
En **app.component.ts**, añadimos la función **onSubmit()** dentro de la clase **AppComponent**.

```
onSubmit() {
  console.log("codigo . " + this.form.controls['codigo'].value);
  console.log("nombre . " + this.form.controls['nombre'].value);
  console.log("aficion1 " + this.form.controls['aficion1'].value);
  console.log("aficion2 " + this.form.controls['aficion2'].value);
  console.log(JSON.stringify(this.form.value));
 }
```

10. Introduzca algunos datos y pulse **Submit** para ver cómo se muestran dichos datos en la consola.

11. Si quisiéramos agrupar los controles **aficion1** y **aficion2** en uno grupo denominado **aficiones**, simplemente tendríamos que modificar el **template** para añadir un div que los agrupara, incluir la propiedad **formGroupName**, y modificar **app.component.ts** para añadir esta agrupación en la creación del grupo y también en **onSubmit** para su display en la consola. Observe

cómo hemos definido unos valores por defecto al definir los controles en el constructor de **AppComponent**.




```
<div formGroupName="aficiones">
  <div class="form-group">
    <label for="aficion1">Aficion 1</label>
    <input type="text" class="form-control"
    formControlName="aficion1" #aficion1>
  </div>
  <div class="form-group">
    <label for="aficion2">Aficion 2</label>
    <input type="text" class="form-control"
    formControlName="aficion2" #aficion2>
  </div>
</div>
```

```
constructor(private fctrl:FormBuilder ){
  this.form=fctrl.group({
    codigo:'123',
    nombre:'Juanto',
    aficiones: fctrl.group({
      aficion1:'musica',
      aficion2:'escribir'
    })
  })
}
```

```
onSubmit() {
  console.log("codigo . " + this.form.controls['codigo'].value);
  console.log("nombre . " + this.form.controls['nombre'].value);
  console.log("aficiones " + JSON.stringify(this.form.controls['aficiones'].value));
  console.log("form " + JSON.stringify(this.form.value));
}
```

12. Observe también que, al pulsar **Submit**, se muestran los valores en la consola.

067 Forms: Reactive validaciones

Las validaciones en los **reactive forms** se declaran en la definición de los controles con la siguiente sintaxis:

```
campo:['valor_defecto',Validators.XXX]
```

Si solo necesita una validación, **XXX** se indica dicha validación (p. ej., **Validators.required**). Sin embargo, si necesitamos aplicar varias validaciones, **XXX** contendrá la palabra **compose** y la sintaxis será la siguiente:

```
codigo:['valor_defecto',Validators.compose([
  Validators.validacion1,
  Validators.validacion2,
  Validators.validacion3
])],
```

En el siguiente ejercicio crearemos un formulario aplicando validaciones a dos campos diferentes usando en cada uno las dos sintaxis comentadas. Para la realización de este ejercicio, usaremos como punto de partida el ejercicio anterior **(formReactive)** al que le quitaremos los controles de aficiones para simplificar el código comentado.

Importante

Utilice las validaciones para asegurar la calidad de los datos introducidos en un formulario. Puede utilizar en cada control, tantas validaciones como considere necesarias.

1. Nos ubicaremos en **Ej100_angular** y crearemos el proyecto tecleando **ng new formRValida**.

2. Seguidamente, renombraremos el proyecto ubicándonos en **Ej100_angular**, y escribiendo lo siguiente:

```
C:\Ej100_angular>rename formRValida 067_formRValida
```

Copiaremos los siguientes archivos del ejercicio anterior y los pegaremos dentro de nuestro nuevo proyecto en sus respectivas ubicaciones:

```
src\styles.css
src\app\app.component.html
src\app\app.component.ts
src\app\app.module.ts
```

Una vez copiados, abrimos el proyecto y modificamos el título en el fichero **app.component.ts**:

```
export class AppComponent {
  title    = '067 formRValida;
}
```

En **app.component.html** eliminamos todo lo referente a aficiones, es decir, el div que agrupaba los campos **aficion1** y **aficion2** y también eliminamos la visualización de sus valores en el div inferior. Por tanto, solo quedarán los controles código y nombre. Dejaremos el botón **Submit** tal y como está. A continuación, en el archivo **app.component.ts** hemos de importar la clase **Validators**:

```
import { FormGroup, FormBuilder, Validators } from '@
   angular/forms';
```

Dentro de la clase **AppComponent**, dejaremos lo siguiente:

```
export class AppComponent {
  title = '067 formRValida';
  form: FormGroup;
  constructor(private fctrl:FormBuilder ){
    this.form=fctrl.group({
      codigo:'',
      nombre:''
    })
  }
  onSubmit() {
    console.log("codigo . " + this.form.controls['codi-
   go'].value);
    console.log("nombre . " + this.form.controls['nom-
   bre'].value);
    console.log(JSON.stringify(this.form.value));
  }
}
```

Arrancamos la aplicación y comprobamos nuestro formulario con los dos campos previstos.

3. A continuación, añadiremos un primer filtro al campo **nombre** para que este sea obligatorio, así:

```
this.form=fctrl.group({
  codigo:'',
  nombre:['',Validators.required]
})
```

En el archivo **app.componment.html**, debajo del input asociado al **nombre**, añadiremos un div que se visualizará condicionado a un ***ngIf** que verificará si el campo **nombre** tiene errores y si ha tenido el foco al menos una vez **(touched)** o ha sido modificado **(dirty)**. En caso de error, añadiremos una alerta:

```
<div class="form-group">
  <label for="nombre">Nombre</label>
  <input type="text" class="form-control"
  formControlName="nombre" #nombre>
  <div *ngIf="form.controls.nombre.errors && (form.
controls.nombre.touched||form.controls.nombre.dirty)"
        class="alert alert-danger">
      Nombre es obligatorio.
  </div>
</div>
```

Añadiremos también la visualización de los errores asociados al campo nombre en el div inferior.

```
<div class="container-fluid fondo p-1 m-1">
  Codigo: {{ codigo.value }}<br>
  Nombre: {{ nombre.value }}<br>
  Errores nombre {{form.controls.nombre.errors | json}}<br>
  {{ form.value | json }}
</div>
```

4. En el navegador, nos situamos en el campo nombre y lo dejamos vacío (p. ej., pulsando **TAB**) y observamos que aparece el error.

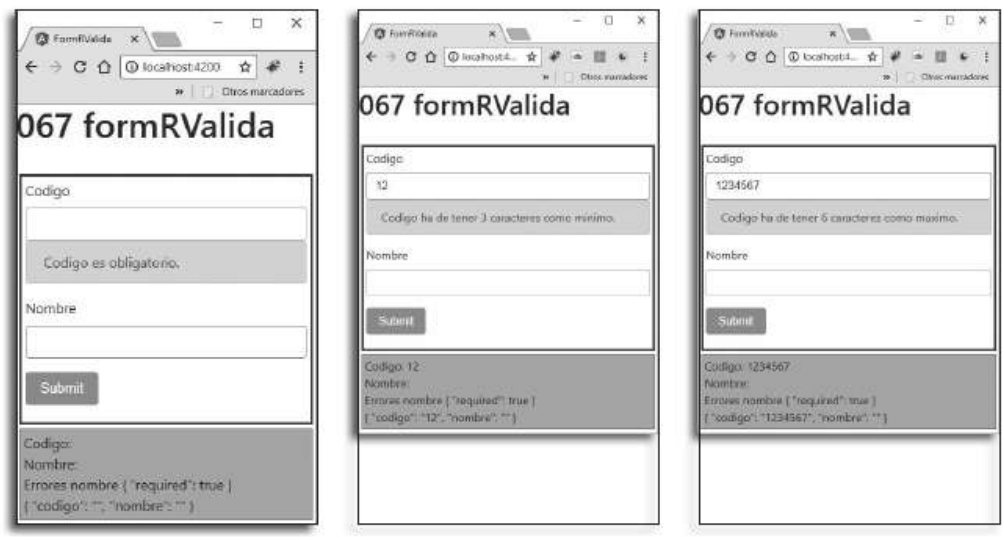
A continuación, vamos a añadir algunas validaciones al campo código. En esta ocasión, tendremos que usar **Validators.compose** y definirlo de la siguiente manera:

```
constructor(private fctrl: FormBuilder) {
  this.form = fctrl.group({
    codigo: ['', Validators.compose([
      Validators.required,
      Validators.minLength(3),
      Validators.maxLength(6)
    ])],
    nombre: ['', Validators.required]
  })
}
```

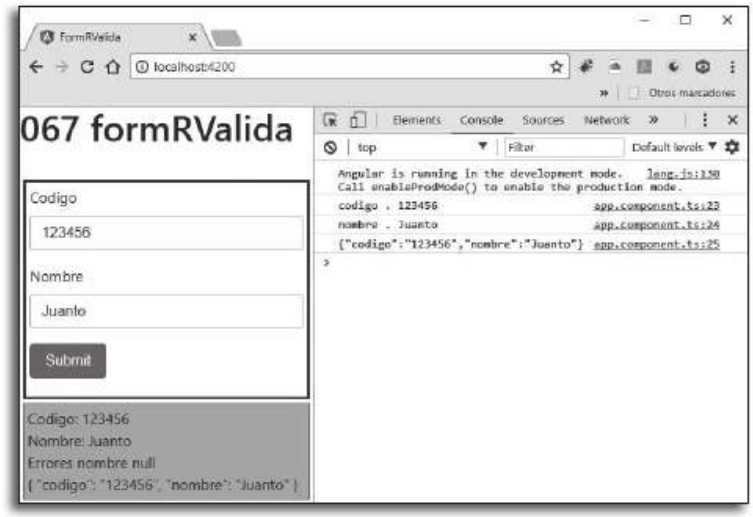
En este caso, además de indicar que el código también es obligatorio, validaremos que tenga **3** caracteres como **mínimo** y **6** como **máximo**. En el template, hemos de incluir debajo del input de **codigo** los div con los errores asociados.

```
<label for="codigo">Codigo</label>
<input type="text" class="form-control"
formControlName="codigo" #codigo>
<div *ngIf="form.controls.codigo.errors
            (form.controls.codigo.touched||form.controls.codigo.dirty)">
  <div *ngIf="form.controls.codigo.errors.required"
       class="alert alert-danger">
    Codigo es obligatorio.
  </div>
  <div *ngIf="form.controls.codigo.errors.minlength"
       class="alert alert-danger">
    Codigo ha de tener 3 caracteres como minimo.
  </div>
  <div *ngIf="form.controls.codigo.errors.maxlength"
       class="alert alert-danger">
    Codigo ha de tener 6 caracteres como maximo.
  </div>
</div>
```

5. Podemos comprobar que existe un primer div que se mostrará cuando existan errores **(errors)** y al mismo tiempo, el control haya tenido el foco alguna vez **(touched)** o haya sido modificado **(dirty)**. Dentro de este div, tenemos un div para cada una de las posibles alertas: **required**, **minlength** y **maxlength**.



6. Podemos comprobar que, si introducimos un **codigo** válido y un **nombre**, el botón **Submit** se activa y si abrimos las herramientas de desarrollador (p. ej., con Google Chrome **CTRL+MAYUS+I**) veremos en la consola que se visualiza el contenido de los controles.


FormRValida
localhost:4200
Otros marcadores
067 formRValida
Codigo
123456
Nombre
Juanto
Submit
Codigo: 123456
Nombre: Juanto
Errores nombre null
{ "codigo": "123456", "nombre": "Juanto" }
Elements
Console
Sources
Network
top
Filter
Default levels
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
lang.js:130
codigo . 123456
app.component.ts:23
nombre . Juanto
app.component.ts:24
{"codigo":"123456","nombre":"Juanto"}
app.component.ts:25

068 Forms: Reactive validaciones personalizadas

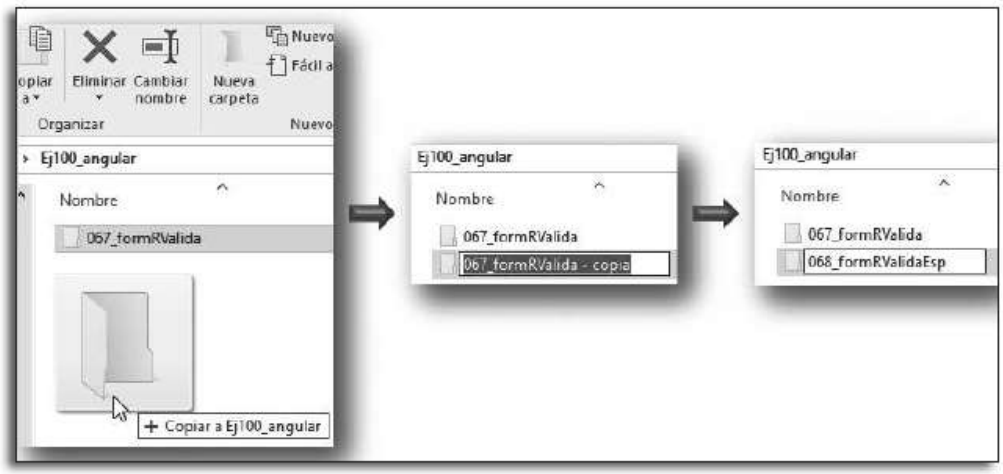
De forma similar a como vimos en el caso de los forms **template-driven**, es posible realizar validaciones personalizadas apoyándonos en una clase auxiliar que disponga de una función, la cual podemos desarrollar para que verifique de forma personalizada cualquier entrada de cualquier campo del formulario. En este caso, usaremos la misma función que fabricamos en el ejercicio **formValidaEsp** pero con algunos pequeños cambios. Recuerde que dicha función verificaba que todas las letras introducidas en el campo fuesen mayúsculas de forma que, si no lo son, generará un error.

Importante

Use validaciones personalizadas cuando necesite filtrar entradas complejas que requieran de una funcionalidad específica.

Para la realización de este ejercicio, usaremos como punto de partida el ejercicio anterior **(formRValida)** al que le añadiremos más funcionalidad relacionada con nuestra validación personalizada.

1. Para facilitar el desarrollo del ejercicio, copiaremos el ejercicio anterior **067_formRValida** sobre **068_formRValidaEsp**. Podemos hacerlo a través del propio **Explorador de Windows**.



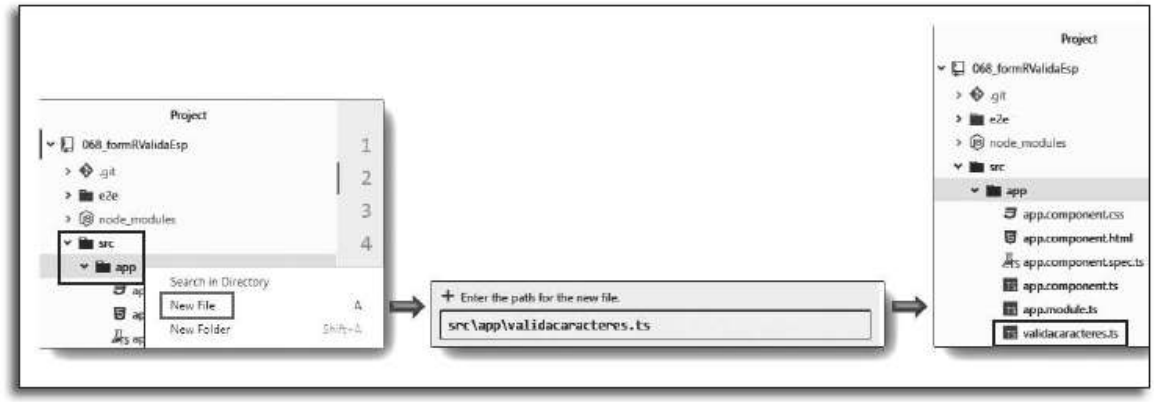
2. Una vez copiado, abrimos el proyecto y modificamos el título en el fichero **app.component.ts**:

```
export class AppComponent {

  title    = '068 formRValidaEsp';

}
```

Abrimos una ventana de **CMD**, nos ubicamos en el directorio de la aplicación y la arrancamos con **ng serve**:

```
C:\Ej100_angular\068_formRValidaEsp>ng serve
```

A continuación, creamos un fichero llamado **validacaracteres.ts** bajo la carpeta **app\src**.



3. Dentro del mismo, importaremos **AbstractControl** y crearemos la clase **FiltrarCaracteres** en la que definiremos el método que usaremos para nuestra validación y que llamaremos **filtrarCaracteres**. Así pues, el contenido del archivo **validacaracteres.ts** será el siguiente:

```
import { AbstractControl } from "@angular/forms";


export class FiltrarCaracteres {

    static filtrarCaracteres(caracter: AbstractControl) {

        if (caracter.value == null) return null;

        var contenido = caracter.value;

        for (var i = 0; i < contenido.length; i++) {

            var letra = contenido.substr(i, 1);

            var valor = letra.charCodeAt(0);

            if (!(valor >= 65 && valor <= 90)) {

                return { filtrarCaracteres: true };

            }

        }

        return null;

    }

}
```

Lo que hacemos en el método es analizar cada uno de los caracteres recibidos en el parámetro **caracter** y ver si su valor **ASCII** está comprendido entre **65** y **90**, que son los valores correspondientes a **A** y **Z**, respectivamente.

4. Seguidamente, modificaremos el fichero **app.component.ts** para importar la clase recién creada para la validación:

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

import { FormGroup, FormBuilder, Validators } from '@angu-
   lar/forms';

import { FiltrarCaracteres } from "./validacaracteres";


@Component({

…
```

Dentro de **app.component.ts**, modificaremos también el constructor para añadir al nombre la nueva validación, de forma que el constructor quedará de la siguiente manera:

```
    constructor(private fctrl: FormBuilder) {

        this.form = fctrl.group({

            codigo: ['', Validators.compose([

                Validators.required,

                Validators.minLength(3),

                Validators.maxLength(6)

            ])],

            nombre: ['', Validators.compose([

                Validators.required,

                FiltrarCaracteres.filtrarCaracteres

            ])],

        })

    }
```

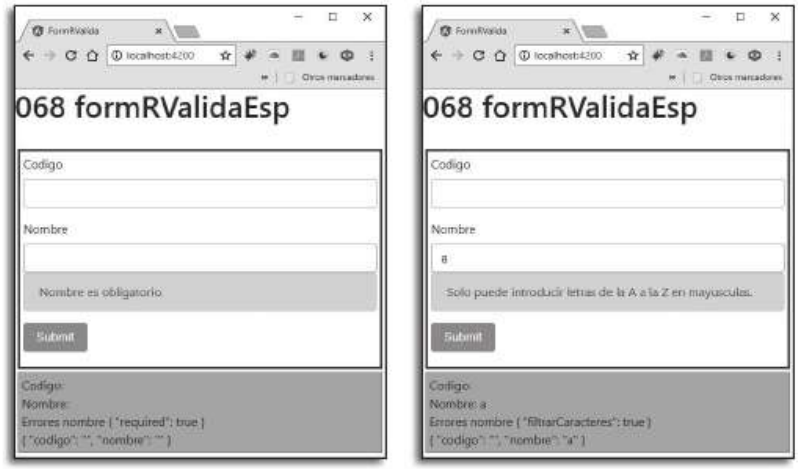
Es decir, conservamos la validación **required** y añadimos la validación especial **filtrarCaracteres**.

5. Por último, tendremos que añadir las nuevas alertas en el template modificando el div asociado a los errores del nombre de la siguiente manera:

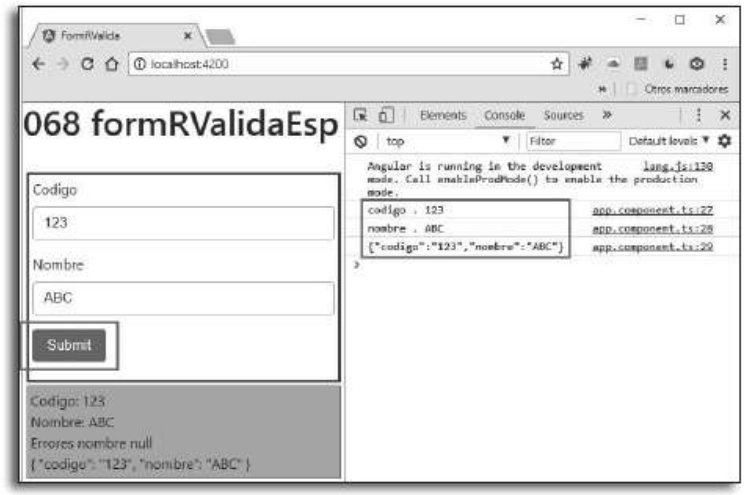
```
<div class="form-group">

    <label for="nombre">Nombre</label>

    <input type="text" class="form-control" formControl-
   Name="nombre" #nombre>

    <div *ngIf="form.controls.nombre.errors && (form.
   controls.nombre.touched||form.controls.nombre.dir-
   ty)">

        <div *ngIf="form.controls.nombre.errors.requi-
   red" class="alert alert-danger">

            Nombre es obligatorio.

        </div>
```

```
        <div *ngIf="form.controls.nombre.errors.filtrar-
    Caracteres" class="alert alert-danger">

            Solo puede introducir letras de la A a la Z
    en mayusculas.

        </div>

     </div>

</div>
```

Guardamos el archivo y vemos que, en el navegador, al situarnos sobre el campo **nombre** y pulsar **TAB** dejándolo vacío aparece el error de **Nombre es obligatorio**, como ya teníamos en el ejercicio anterior. También podemos comprobar que si introducimos un carácter que no sea una letra en mayúsculas, bien sea al principio del campo o en cualquier posición del contenido del campo, nos mostrará el error de **"Solo puede introducir letras de la A a la Z en mayusculas"**.



6. Introduzca valores correctos en codigo y nombre y compruebe que se habilita el botón **Submit**, de forma que, al pulsarlo, se muestra en la consola de las **herramientas del desarrollador** del navegador (p. ej., **CTRL+MAYUS+I** en **Google Chrome**) los valores introducidos en el formulario.

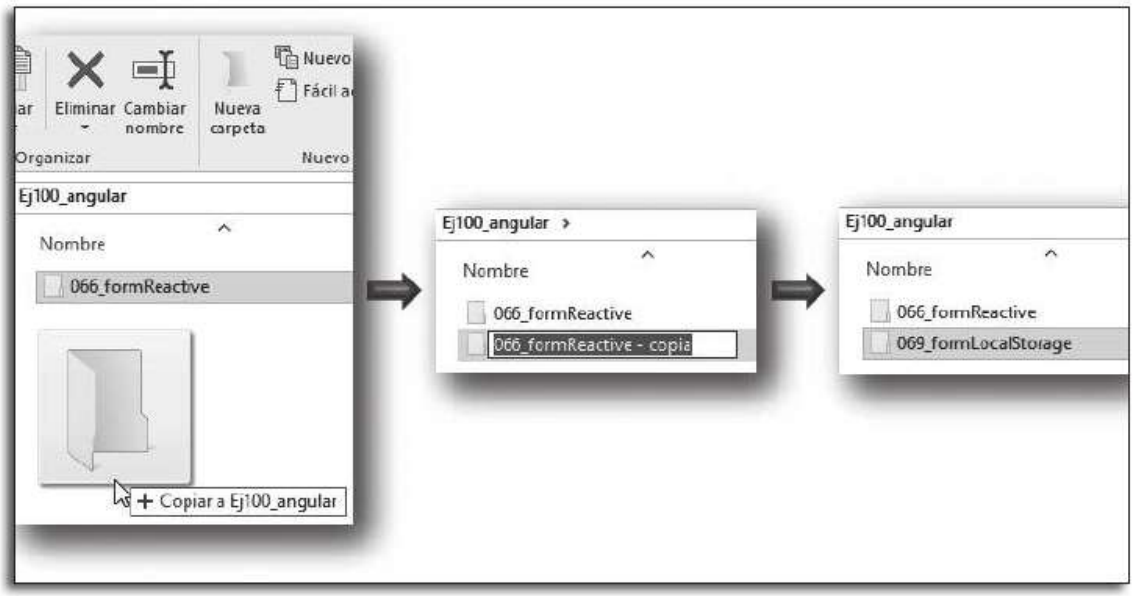


069 Forms: LocalStorage

Una de las tareas de los formularios es la de recoger datos y mostrarlos. Para ello, generalmente se utilizan bases de datos que tratamos mediante servicios. Estos temas se tratan en otros ejercicios de este libro, pero a modo de recurso sencillo queríamos mostrar la posibilidad de almacenar datos en lo que se denomina **LocalStorage** que viene a sustituir a las antiguas cookies y que, en definitiva, se trata de un espacio que ofrecen los navegadores para almacenar entre **2,5** y **5** Mb de información. Podemos analizar el contenido de este espacio accediendo a las herramientas de desarrolladores (en el caso de **Google Chrome** en el apartado **Application->Storage->Local Storage**).

Para la realización de este ejercicio, usaremos como punto de partida el ejercicio anterior **(066_formReactive)** al que le añadiremos la funcionalidad necesaria para poder grabar y leer información de la **LocalStorage**.

1. Para facilitar el desarrollo del ejercicio, copiaremos el ejercicio anterior **066_formReactive** sobre **069_formLocalStorage**. Podemos hacerlo a través del propio **Explorador de Windows**.



2. Una vez copiado, abrimos el proyecto y modificamos el título en el fichero **app.component.ts**:

```
export class AppComponent {

  title    = '069 formsLocalStorage';

}
```

Abrimos una ventana de **CMD**, nos ubicamos en el directorio de la aplicación y la arrancamos con **ng serve**:


```
C:\Ej100_angular\069_formsLocalStorage>ng serve
```

A continuación, modificaremos el archivo **app.component.ts** para añadir dos métodos que permitan grabar y leer datos sobre la **LocalStorage** de nuestro navegador. Primero, añadiremos el método **grabarDatos()** con el siguiente contenido:

```
grabarDatos(){

    localStorage.setItem("codigo", this.form.control-
   s['codigo'].value);

    localStorage.setItem("nombre", this.form.control-
   s['nombre'].value);

    localStorage.setItem("aficiones", JSON.stringify(-
   this.form.controls['aficiones'].value));

}
```

Para almacenar un valor, vemos que usamos la expresión **localStorage.setItem(clave, valor)**.

3. Seguidamente, añadimos la función **leerDatos()** con lo siguiente:

```
leerDatos(){
    this.form.patchValue({
        'codigo': localStorage.getItem("codigo") ?
            localStorage.getItem("codigo") : ""
    });
    this.form.patchValue({
        'nombre': localStorage.getItem("nombre") ?
            localStorage.getItem("nombre") : ""
    });
    this.form.patchValue({
        'aficiones': JSON.parse(localStorage.getItem("afi-
   ciones")) ?
            JSON.parse(localStorage.getItem("aficiones"))
   : ""
    });
}
```

Para leer los datos, usamos la expresión **localStorage.getItem(clave)**. En este caso usamos el operador **Elvis** (**?**) para inicializar el campo con un nulo (“”) en caso de que esté vacío en **LocalStorage**.

4. Para asignar el valor al control usamos el método **patchValue** del objeto **form (FormGroup)**.

5. Ahora, incluiremos la llamada a **grabarDatos()** en el método **onSubmit()** justo después de la visualización de los datos por consola:

Importante

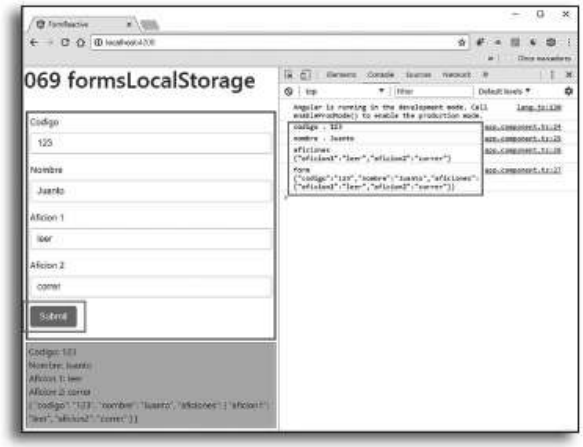
LocalStorage puede permitirnos almacenar cierta información (como preferencias de usuario o ciertas características asociadas al puesto de trabajo) de forma sencilla sin necesidad de acudir a ninguna BBDD.

```
onSubmit() {
    console.log("codigo . " + this.form.controls['codi-
   go'].value);
    console.log("nombre . " + this.form.controls['nom-
   bre'].value);
    console.log("aficiones " + JSON.stringify(this.form.
   controls['aficiones'].value));
    console.log("form " + JSON.stringify(this.form.va-
   lue));
    this.grabarDatos();
}
```

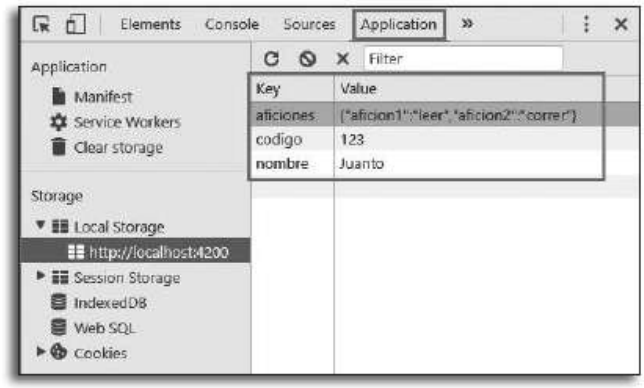
Incluiremos también la llamada a **leerDatos()** al final del constructor y aprovecharemos para eliminar el valor por defecto que tenían los controles en su inicialización:

```
constructor(private fctrl:FormBuilder){
    this.form = fctrl.group({
        codigo: '',
        nombre: '',
        aficiones: fctrl.group({
            aficion1: '',
            aficion2: ''
        })
    })
    this.leerDatos();
}
```

A continuación, abrimos las herramientas de desarrolladores (p. ej., en **Google Chrome CTRL+MAYUS+I**). Introducimos un valor en cada control y pulsamos en **Submit** para ver los datos introducidos en la consola.

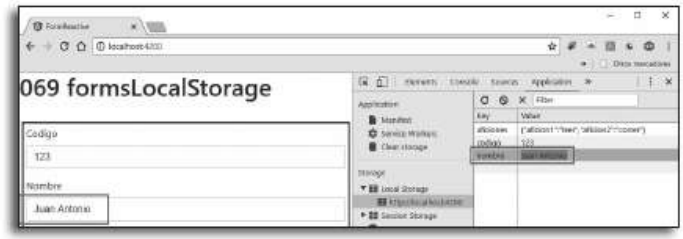


6. Dentro de las herramientas de desarrollador del navegador, buscamos el apartado **Application** y, dentro del mismo, el apartado de **Storage** -> **LocalStorage** -> http://localhost:4200.



7. Vamos a modificar el **nombre** mediante las herramientas de desarrollo haciendo doble clic sobre el mismo y poniendo otro contenido (p. ej., cambio **"Juanto"** por **"Juan Antonio"**).

8. Si recargamos la aplicación en el navegador, observaremos como ahora se muestra el nuevo **nombre**, lo que nos demuestra que, además de **grabar**, podemos **leer**.



070 MEAN: Desarrollos con MongoDB, Express, Angular y Node.js

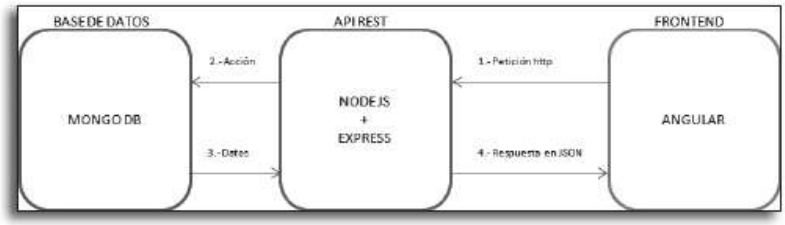
Tal como comentábamos en los primeros capítulos del libro, la principal característica de las aplicaciones single-page application (SPA) que desarrollamos con Angular es que la mayor parte de su funcionalidad se lleva al cliente (navegador Web). El código en servidor se usa básicamente para proveer de servicios a nuestro código cliente para, por ejemplo, dar acceso a una base de datos.

Y así es si repasamos las aplicaciones que hemos ido desarrollando hasta este momento. En ningún caso, salvo en los ejercicios relacionados con el servicio HttpClient, hemos necesitado de un backend.

En la serie de ejercicios que empezamos ahora usaremos MEAN stack para desarrollar una aplicación entera con un front-end Angular y un back-end de servicios REST de acceso a una base de datos.

Importante

MEAN stack define el uso de las tecnologías MongodbB Express, Angular y Node para el desarrollo web. Su principal característica es que todas estas tecnologías tienen el JavaScript como lenguaje de programación común.



En el mundo del desarrollo web **MEAN stack** define el uso de un conjunto concreto de tecnologías que permiten el desarrollo de todas y cada una de las partes de una aplicación. Estas tecnologías son: **MongoDB, Express.js, Angular, Node.js**. Todas ellas se empezaron a usar de forma conjunta por un motivo muy simple y práctico: todas ellas utilizan **JavaScript** como lenguaje de programación. Esto facilita mucho el desarrollo y es el principal motivo de éxito de MEAN stack.

Vamos a ver un poco en qué consisten las tecnologías MongoDB y Express que no habíamos visto hasta ahora.

MongoDB

Tradicionalmente, las aplicaciones siempre han trabajado con **bases de datos relacionales** como son

Oracle, Sql Server, MySQL o PostgreSQL. Cada una de ellas tiene sus particularidades, pero al fin y al cabo todas ellas trabajan con tablas, columnas, claves primarias, consultas SQL, etc.

Estos sistemas son muy fiables, sin embargo, pueden tener problemas de rendimiento y escalabilidad en aplicaciones que gestionan volúmenes muy grandes de datos y que son usadas por muchos usuarios a la vez.

Es por este motivo que aparecieron las **bases de datos NoSQL**. De entrada, la principal diferencia respecto a las bases de datos es que NoSQL es una forma de almacenamiento **no estructurado**: no hay tablas, no hay registros ni tampoco consultas SQL.

Existen muchos sistemas de bases de datos NoSQL y la mayoría son de código abierto. Cada uno de ellos tiene sus particularidades y pueden ser muy diferentes entre ellos, sin embargo podemos considerar que existen cuatro tipos diferentes según como almacenan y gestionan los datos: orientadas a documentos, orientadas a columnas, de clave-valor y en grafo.

En MEAN stack la base de datos usada es MongoDB. **MongoDB** es una base de datos orientada a **documentos**, es decir, los datos se guardan en documentos y no en registros. Estos documentos se almacenan en BSON, que es una representación binaria de JSON.

Una de las principales diferencias con las bases de datos relacionales es que los documentos de una misma **colección** (concepto similar al de tabla en una base de datos relacional) no tienen por qué seguir un esquema. Por ejemplo, para una colección "Usuarios" podríamos tener dos documentos como los siguientes:

```
{ Nombre: "Julian", Hijos: [{ Nombre:"Isabel", Edad:6},{ Nombre:"Alberto",
Edad:2}]}
{ Nombre: "Ramon", Apellidos: "Lopéz", Hijos: 3}
```

Como vemos, los esquemas son totalmente diferentes. El segundo documento tiene un campo nuevo, y el campo "Hijos" es de tipo distinto al del otro documento. A pesar de esto, es importante diseñar un esquema para nuestras colecciones para facilitar su gestión. De hecho, el esquema lo definirán las consultas que vayamos a realizar con más frecuencia. Ejemplo: db.Usuarios.find({Nombre: "Ramon"});

Express

Express es un framework de desarrollo de aplicaciones web para la plataforma Node.js. Su configuración básica es la siguiente:

- Creación de la aplicación Express y configuración del puerto para la entrada de peticiones de clientes.

```
const app = express();
var server = app.listen(8080, function(){ console.log('Listening... ');
});
```

- Configuración de las llamadas a funciones middleware para el procesamiento de las peticiones. Ejemplo:

```
app.use((req, res, next) => {  console.log('Petición recibida.');
next();});
app.get('/usuarios', function(req, res, next){
  //res.send('<Datos de usuarios>');
  var err = new Error('<Datos del error>'); next(err);
});
app.post('/usuarios', function(req, res, next){ res.send('Usuario
creado');});
app.use((err, req, res, next) => { console.log('Error!'); res.status(500).
send(err.message);});
```

Una llamada a una función de middleware consta de los siguientes elementos: método HTTP para el que se aplica la función, vía de acceso (ruta) para la que se aplica la función, y la propia función middleware.

Las funciones middleware tienen acceso al objeto petición (req), al objeto respuesta (res) y a la siguiente función middleware (next), y las tareas que suelen realizar son las de:

- Ejecutar código.
- Realizar cambios en la petición (req) y respuesta (res).
- Finalizar el ciclo petición/respuesta respondiendo al cliente mediante los métodos del objeto "res".
- Invocar la siguiente función middleware en la pila ejecutando "next()".

Cuando Express recibe una petición, se ejecuta la primera función middleware que aplique según método HTTP y ruta de la petición. Por ejemplo, para una petición "post" en "/usuarios", el código anterior ejecutaría la primera y tercera funciones middleware, y el cliente acabaría recibiendo el texto "Usuario creado". Y para una petición "get" en "/usuarios", el mismo código ejecutaría la primera, segunda y cuarta funciones middleware, y el cliente acabaría recibiendo un error de tipo 500 (internal server error) con el texto "<Datos del error>".

071 MEAN: Creación de la aplicación Express

Este es el primero de una serie de ejercicios que nos permitirá construir una sencilla aplicación MEAN para poder realizar la gestión (alta, baja, consulta y modificación) de "tareas" almacenadas en una tabla de base de datos MongoDB.

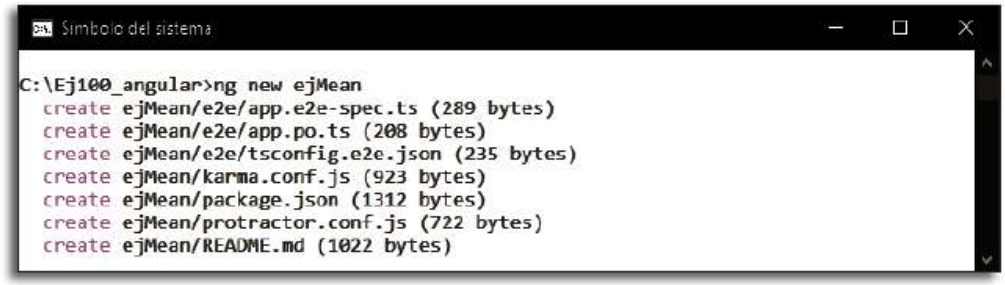
El backend de la aplicación lo desarrollaremos con Express.js, y estará formada por una API de servicios REST para la gestión de las "tareas" de la base de datos MongoDB. Por otra parte, el frontend lo desarrollaremos con Angular, y estará formada por dos vistas: una principal con el listado de tareas y las distintas opciones de gestión, y otra para crear o modificar una tarea.

Importante

Cualquier cambio en el frontend, o sea, en la aplicación Angular, requiere un "ng build" para actualizar los archivos de la carpeta **/dist**.

En este primer ejercicio construiremos la estructura básica de la aplicación. Partiendo de un proyecto Angular, configuraremos una aplicación Express.js que hará las funciones de servidor web. La aplicación gestionará todas las peticiones que le lleguen desde el navegador, ya sea para dar acceso a la API de servicios, o para enviar la aplicación Angular al navegador.

1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos el proyecto **ejMean**.



2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio:

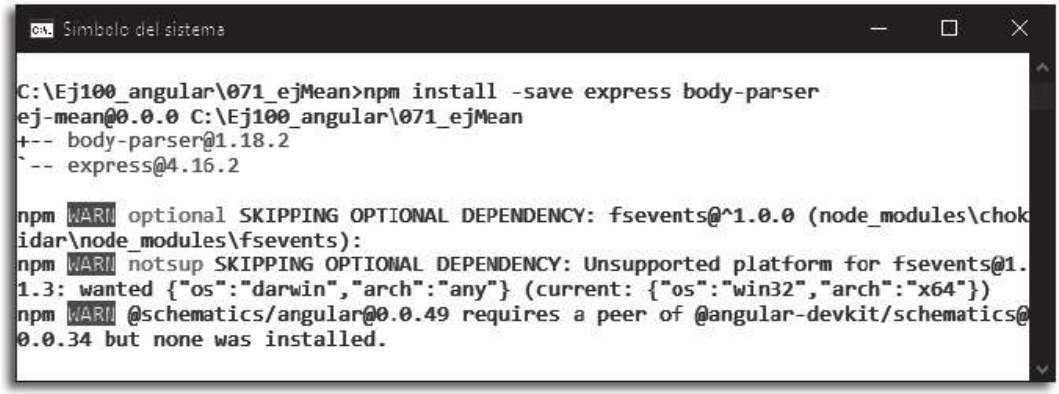
```
C:\Ej100_angular>rename ejMean 071_ejMean
```

3. A continuación, nos ubicaremos en **071_ejMean** y arrancaremos la aplicación para comprobar cómo funciona en nuestro navegador. Para ello, teclearemos lo siguiente:

```
C:\Ej100_angular>cd 071_ejMean

C:\Ej100_angular\071_ejMean>ng serve
```

4. Desde la ventana de CMD, realizaremos un **ctrl+c** para parar la aplicación. Una vez termine, pasaremos a instalar los módulos **express** y **body-parser** en nuestra aplicación:

```
C:\Ej100_angular\071_ejMean>npm install -save express
body-parser
```



5. Una vez terminada la instalación, abriremos nuestro proyecto con nuestro editor (en nuestro caso **Atom**), y crearemos el archivo **server.js** en la raíz de nuestro proyecto. Luego, añadiremos su código:

server.js

```
// Get dependencies
const express = require('express');
const path = require('path');
const http = require('http');
const bodyParser = require('body-parser');

//////////////////////////////////////////////////////
// Creamos la aplicacion express y la configuramos...
const app = express();

// Parsers for POST data
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

//Cfg. del directorio 'dist' como directorio estatico.
//En este directorio tendremos los archivos obtenidos en el
build de nuestra aplicación Angular
app.use(express.static(path.join(__dirname, 'dist')));

//Cfg. de las rutas
app.get('/api', (req, res) => {
  res.send('La API funciona');
});

app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, 'dist/index.html'));
});

//Cfg. del puerto de escucha
const port = process.env.PORT || '3000';
app.set('port', port);

//Creamos el servidor http con la aplicación express y abrimos
puerto
const server = http.createServer(app);
server.listen(port, () => console.log(`API running on
localhost:${port}`));
```

6. El código anterior establece una simple **aplicación Express con las siguientes características**:

a. Puerto de escucha de peticiones: puerto 3000.
b. La devolución de un texto para las peticiones que le lleguen por la ruta **"/api"**.
c. La devolución de la página **dist/index.html** (página principal de nuestra aplicación Angular) para las peticiones que le lleguen por cualquier otra ruta.

7. Antes de ejecutar nuestra aplicación Express, deberemos realizar un **build** para generar la archivos de la aplicación Angular en la carpeta **/dist**:

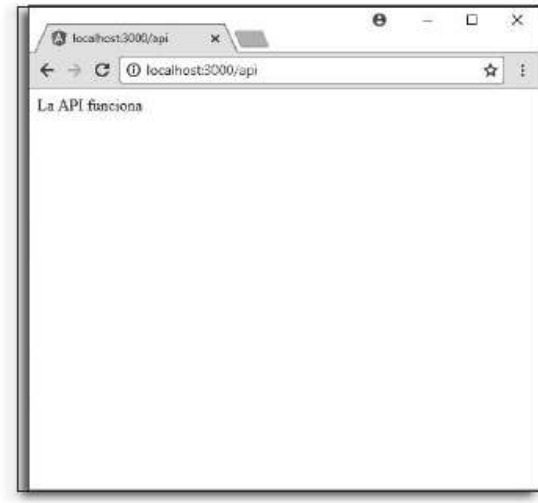
```
C:\Ej100_angular\071_ejMean>ng build
```

Es importante tener en cuenta que, a partir de ahora, siempre que modifiquemos la aplicación Angular, deberemos realizar un "ng build" para actualizar la carpeta **/dist**, de tal manera que la aplicación Express envíe siempre la última versión al navegador. Es recomendable salir del editor Atom al hacerlo, ya que si no pueden producirse errores de permisos.

8. Finalmente ejecutaremos nuestra aplicación Express de la siguiente manera:

```
C:\Ej100_angular\071_ejMean>node server.js
```

9. Desde el navegador llamaremos a la aplicación por el puerto 3000. Por defecto, la aplicación Express nos devolverá la aplicación Angular. Pero si a la ruta añadimos "/api", nos devolverá el texto codificado.



072 MEAN: Instalación y configuración de MongoDB

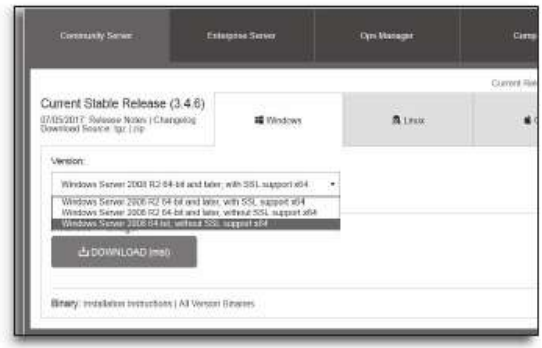
En este ejercicio nos centraremos en el subsistema MongoDB del MEAN stack, o sea, en el tipo de base de datos que se usa en las aplicaciones MEAN. Descargaremos MongoDB, lo instalaremos, configuraremos y finalmente añadiremos el código necesario en nuestra aplicación para que se conecte a MongoDB.

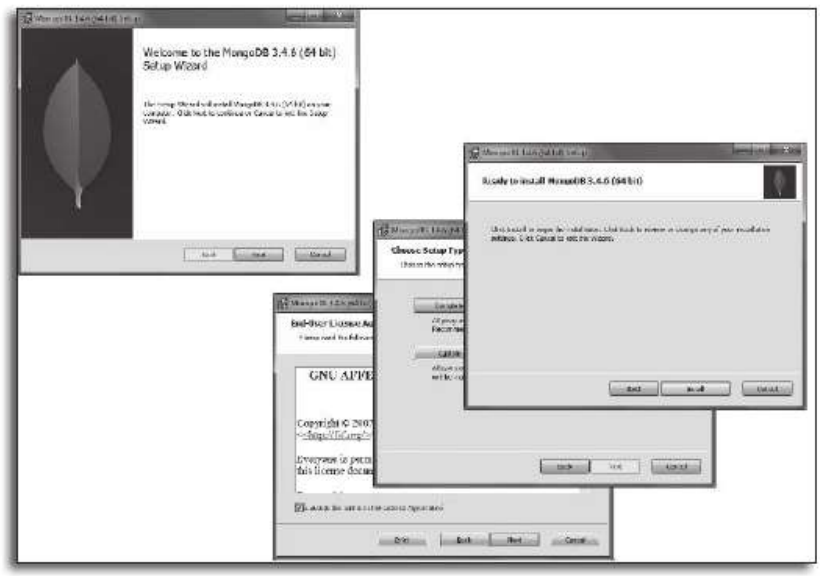
Importante

En **c:\mongodb\data\log\mongod.log** tenemos los log del servidor MongoDB. Entre otras cosas podemos ver las conexiones/desconexiones, errores, etc.

1. El primer paso será descargar el instalador de MongoDB en nuestro ordenador desde la siguiente página: https://www.mongodb.org/downloads#production

   En esta página, seleccionaremos la opción que más nos interese según nuestro SO. En nuestro caso, seleccionaremos la opción **Windows Server 2008 64-bit, without SSL support x64** que es la versión para Windows Vista o superior. Una vez seleccionada la opción adecuada, descargaremos el instalador y lo ejecutaremos seleccionando las opciones por defecto.





2. Una vez instalado el producto, pasaremos a crear el **servicio Windows MongoDB**. Este servicio nos permitirá que el servidor de base de datos MongoDB se ponga en marcha automáticamente cada vez que reiniciemos el ordenador.

3. Para ello, primero **crearemos la siguiente estructura de directorios** desde una ventana CMD o, si se prefiere, desde un Explorador de Windows.

```
C:\>mkdir mongodb\data\db mongodb\data\log
```

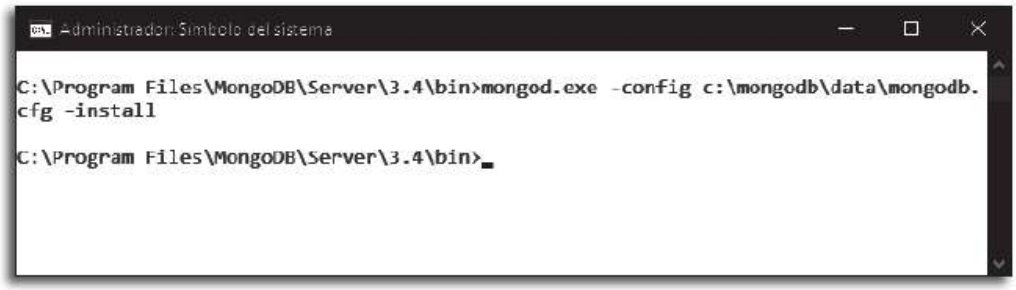
4. A continuación, crearemos el fichero **C:\mongodb\data\mongodb.cfg** con el siguiente código:

```
systemLog:

    destination: file

    path: C:\mongodb\data\log\mongod.log

storage:

    dbPath: C:\mongodb\data\db
```

Este es el fichero de configuración que se aplicará a nuestro servidor MongoDB. En él estamos indicando las carpetas y ficheros que se usarán para los datos y ficheros **log**.

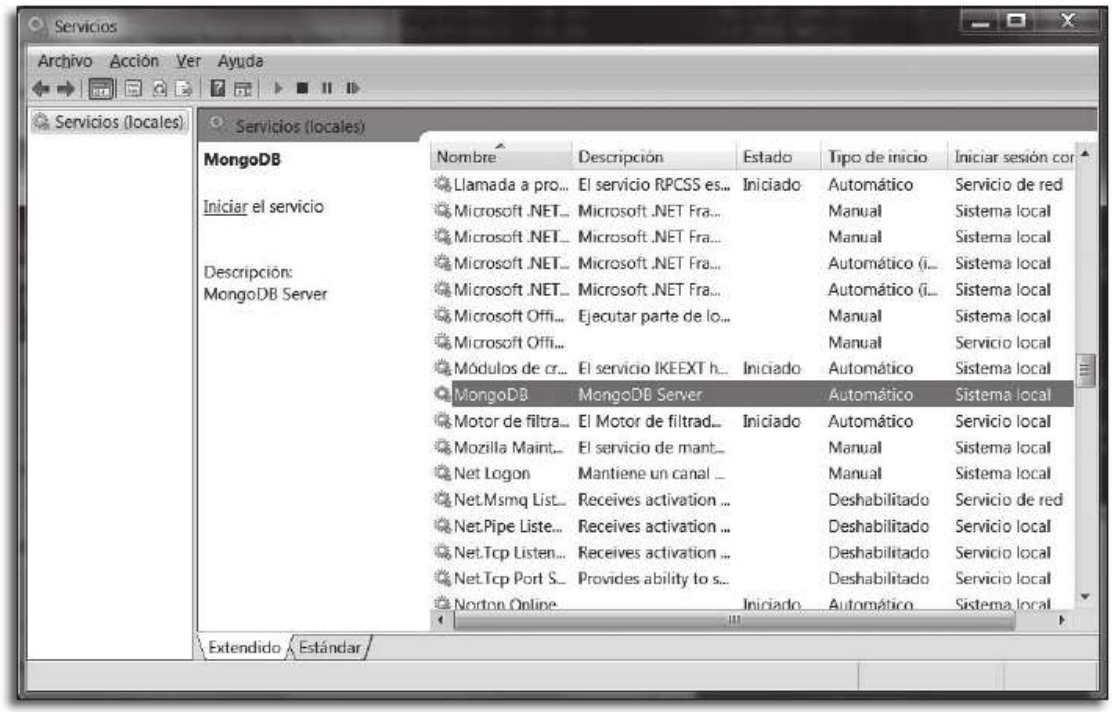
5. Finalmente crearemos el servicio ejecutando el comando siguiente:

```
C:\Program Files\mongodb\Server\3.4\bin>mongod.exe -config c:\
mongodb\data\mongodb.cfg -install
```



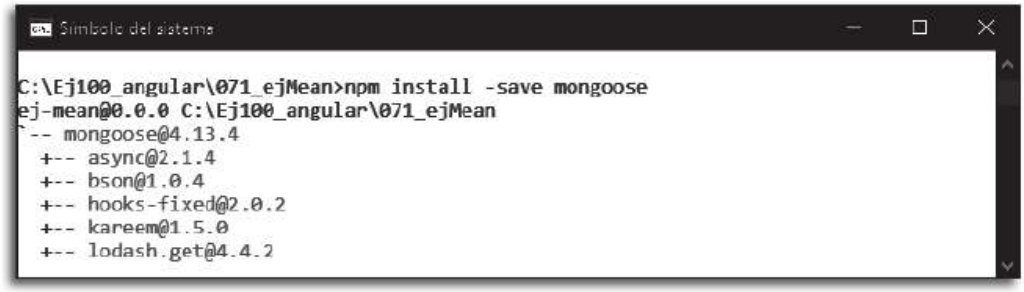
Hay que tener en cuenta que debemos hacerlo como **usuario Administrador** (botón derecho sobre el icono de "Símbolo del Sistema" [CMD] y pulsar "Ejecutar como administrador") y que mongod.exe **puede variar de ubicación según la versión** que hubiéramos instalado en el paso 1.

6. Ahora pasaremos a verificar que se haya creado el servicio accediendo a la ventana "Servicios" de Windows desde el menú Windows o ejecutando services.msc desde CMD. Para poner en marcha el servicio podemos pulsar **Iniciar** desde esta misma ventana o hacerlo reiniciando el ordenador.



7. En estos últimos apartados del ejercicio añadiremos el código necesario en nuestra aplicación MEAN para que realice la conexión al servidor de base de datos MongoDB. Primero instalaremos el módulo **mongoose**, que es la librería que usaremos para conectarnos a MongoDB:

```
C:\Ej100_angular\071_ejMean>npm install -save mongoose
```



8. Una vez terminada la instalación, abriremos nuestro proyecto con nuestro editor (**Atom**), y añadiremos el siguiente código en el archivo **server.js**:

```
var mongoose = require('mongoose');

//////////////////////////////////////////////////////
// Conexión a la base de datos MongoDB a traves de Mongoose

var dbURI = 'mongodb://localhost/db_mean';
mongoose.connect(dbURI, {useMongoClient: true});
```

```
// Configuracion de los eventos de la conexión Mongoose
mongoose.connection.on('connected', function () {
  console.log('Mongoose default connection open to ' + dbURI);
});

mongoose.connection.on('error',function (err) {
  console.log('Mongoose default connection error: ' + err);
});

mongoose.connection.on('disconnected', function () {
  console.log('Mongoose default connection disconnected');
});

// Si el proceso 'Node' termina, se cierra la conexión Mongoose
process.on('SIGINT', function() {
  mongoose.connection.close(function () {
    console.log('Mongoose default connection disconnected through app
termination');
    process.exit(0);
  });
});
////////////////////////////////////////////////////
// Creamos la aplicacion express y la configuramos
...
```

9. Con este código nuestro servidor realiza la conexión a la base de datos MongoDB y configura la gestión de eventos principales para interceptar las conexiones/desconexiones que se produzcan en la base de datos.

10. Finalmente ejecutaremos nuestra aplicación de la siguiente manera:

```
C:\Ej100_angular\071_ejMean>node server.js
```

Desde la propia ventana de CMD deberíamos ver el texto correspondiente al evento "connected", que indicaría que la conexión se ha realizado correctamente.

073 MEAN: Creación de la API Restful (parte I)

En este ejercicio construiremos la **RESTful API** de nuestra aplicación, implementando los casos **CRUD** (Create/Read/Update/Delete) para crear/ver/editar/borrar las "tareas" de nuestra base de datos. En la siguiente tabla se muestran los 5 endpoints que vamos a implementar y que definirán nuestra API:

HTTP	URL	Descripción
GET	/api/tareas	Devuelve todas las tareas
POST	/api/tareas	Crea una tarea
GET	/api/tareas/:tareaId	Devuelve una tarea
PUT	/api/tareas/:tareaId	Modifica una tarea
DELETE	/api/tareas/:tareaId	Borra una tarea

Para construir la API seguiremos el patrón de diseño **MVC** (Modelo-Vista-Controlador). MVC separa datos y lógica de negocio de una aplicación, respecto su interfaz de usuario. Para ello, propone la construcción de estos tres componentes:

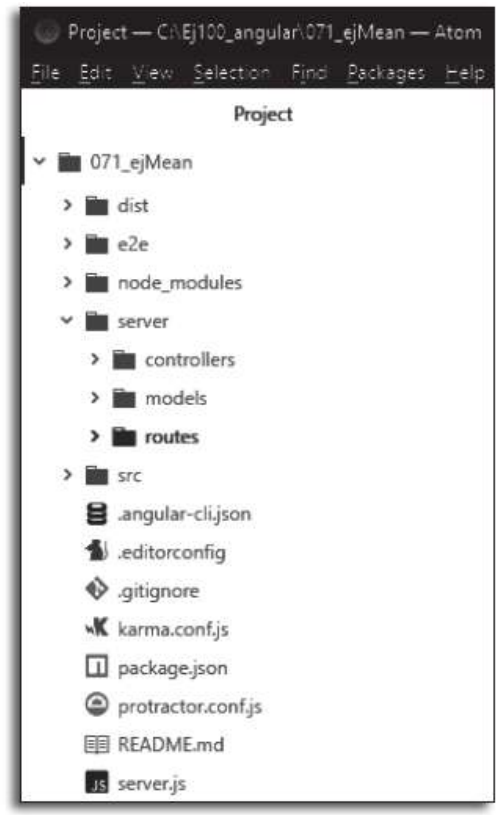
- **Modelo:** representa los datos y lógica de negocio de la aplicación. En nuestro caso, el modelo quedará representado por una entidad "Tarea" con las propiedades y métodos necesarios.
- **Vista:** elemento con el que interactúa el usuario. En nuestro caso, al ser una API, el usuario va a ser otro componente software y la vista van a ser los endpoints disponibles y los datos que devolvemos (archivos JSON).
- **Controlador:** hace de intermediario entre la vista y el modelo: pide datos al modelo para devolvérselos a la vista, y realiza acciones sobre el modelo cuyo origen se produce en la vista.

Importante

Es importante que siempre usemos patrones de diseño, como el MVC (Modelo-Vista-Controlador) en nuestros desarrollos. Nos ayudan a estructurar los proyectos facilitando su desarrollo y mantenimiento.

Vamos a empezar con el ejercicio:

1. Abriremos el proyecto 071_ejMean con el editor **Atom** y, en la raíz de nuestro proyecto, crearemos los directorios **"server"**, **"server/controllers"**, **"server/models"** y **"server/routes"**.



2. A continuación, crearemos las rutas o endpoints de nuestra API. Para hacerlo, primero crearemos el fichero **tarea.js** en la carpeta **routes**, y luego le añadiremos el siguiente código:

server/routes/tarea.js

```
module.exports = function(app){

    var tareaCtrl = require('../controllers/tarea');

    app.route('/api/tareas')
      .get(tareaCtrl.list_all_tareas)
      .post(tareaCtrl.create_tarea);

    app.route('/api/tareas/:tareaId')
      .get(tareaCtrl.read_tarea)
      .put(tareaCtrl.update_tarea)
      .delete(tareaCtrl.delete_tarea);
}
```

3. Seguidamente crearemos el controlador al que hacíamos referencia en el código anterior. Crearemos el fichero **tarea.js** en la carpeta **controller**, y luego le añadiremos el siguiente código:

server/controller/tarea.js

```
exports.list_all_tareas = function(req, res) {res.send('<Lista
    de tareas>');};
exports.create_tarea = function(req, res) {res.send('Tarea
    creada');};
exports.read_tarea = function(req, res) {res.send('<Tarea>');};
exports.update_tarea = function(req, res) {res.send('Tarea
    modificada');};
exports.delete_tarea = function(req, res) {res.send('Tarea
    eliminada');};
```

De momento, las funciones de nuestro controlador únicamente devuelven un texto indicando la operativa realizada. En el siguiente capítulo completaremos este código.

4. Finalmente pondremos la carga de las rutas de la API en el código del servidor. Para ello, simplemente añadiremos el requerimiento de **"server/routes/tarea.js"** en **server.js**:

server.js

```
//Cfg. de las rutas
app.get('/api', (req, res) => {
  res.send('La API funciona');
});

require('./server/routes/tarea')(app);

app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, 'dist/index.html'));
});
```

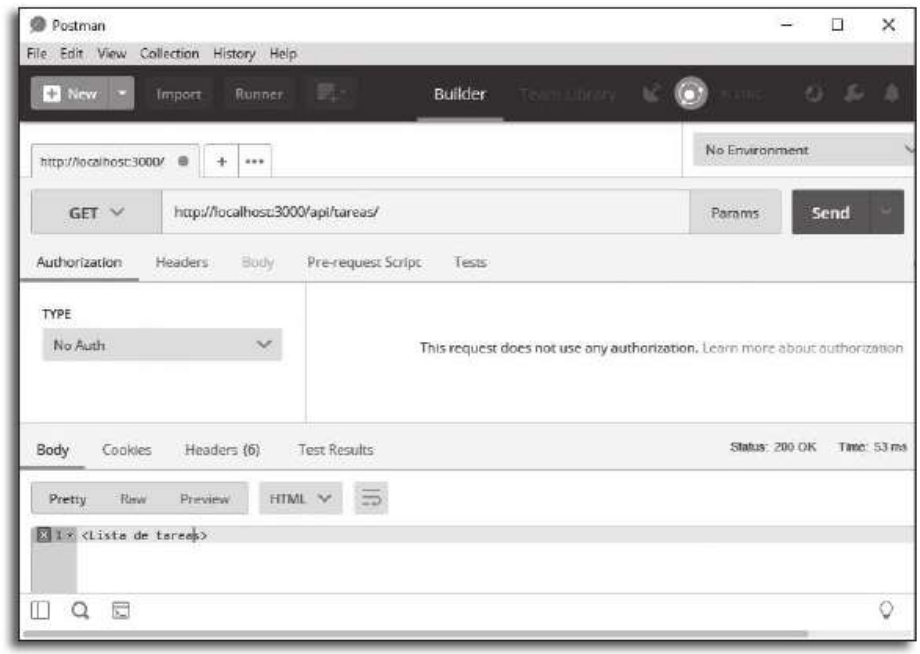
Es importante mantener el orden de líneas indicado, ya que el servidor resuelve las rutas de arriba abajo. Si pusiéramos la referencia "…/tarea" por debajo de la ruta "app.get('*',…", ninguna petición HTTP llegaría a nuestra API.

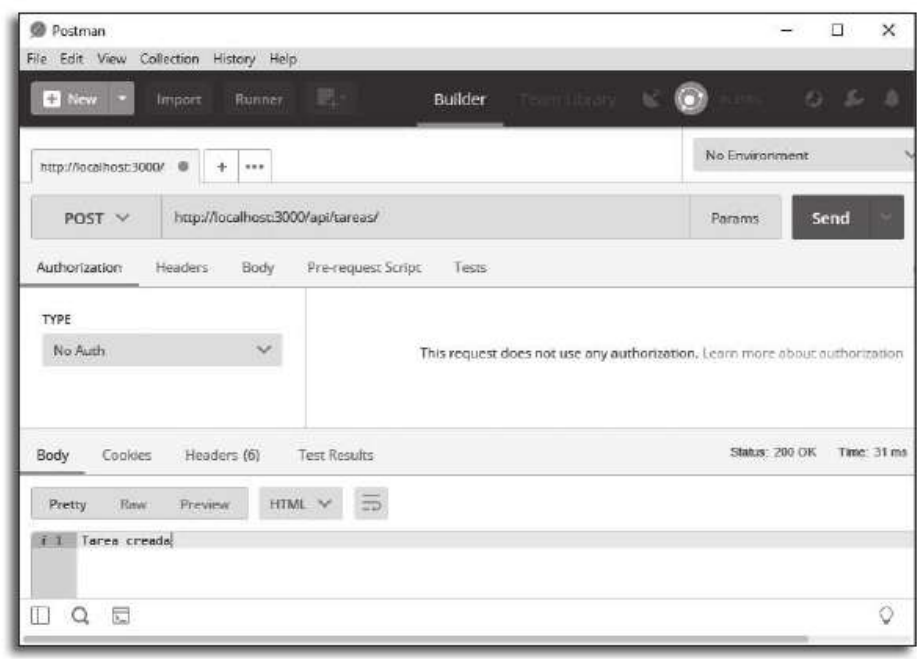
5. Una vez introducido todo el código, vamos a poner en marcha la aplicación para probar la API:

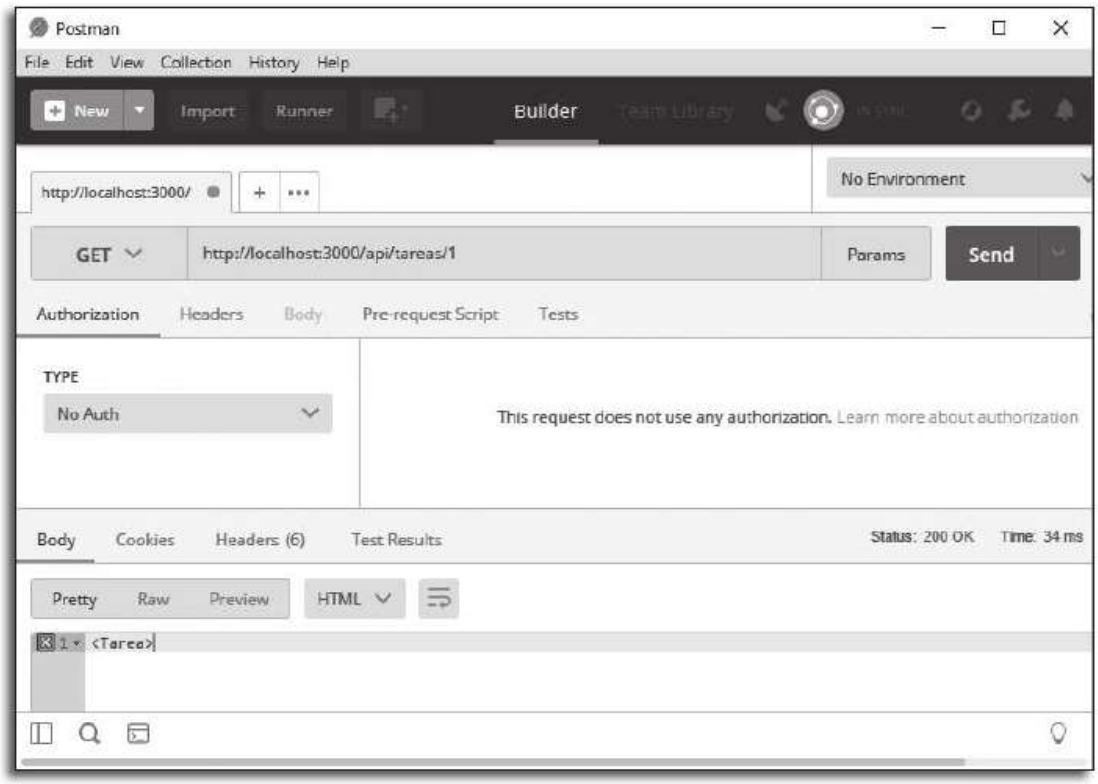
```
C:\Ej100_angular\071_ejMean>node server.js
```

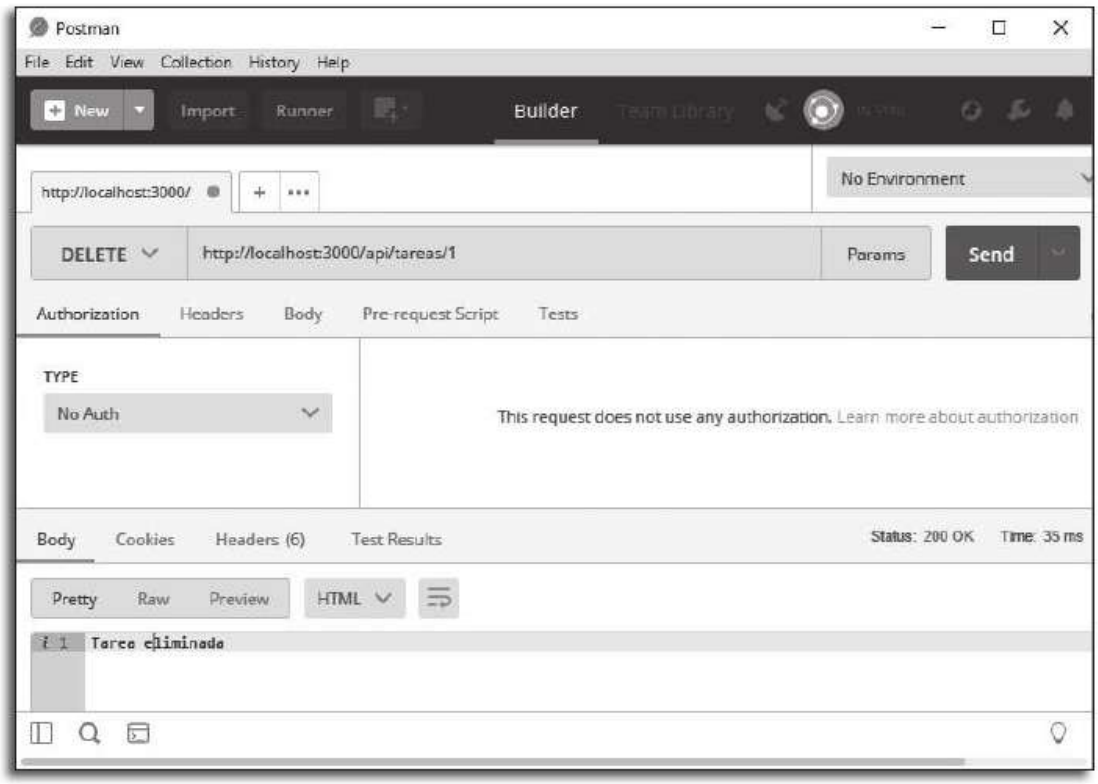
6. Las peticiones HTTP de prueba las lanzaremos a través del software **Postman**. Postman funciona como una extensión de Google Chrome permitiendo construir y lanzar peticiones HTTP de forma muy sencilla. Con este software, podremos probar nuestra API sin tener que esperar al desarrollo de la aplicación Angular. Postman lo podemos descargar de forma gratuita desde www.getpostman.com, y su instalación es muy fácil e intuitiva.

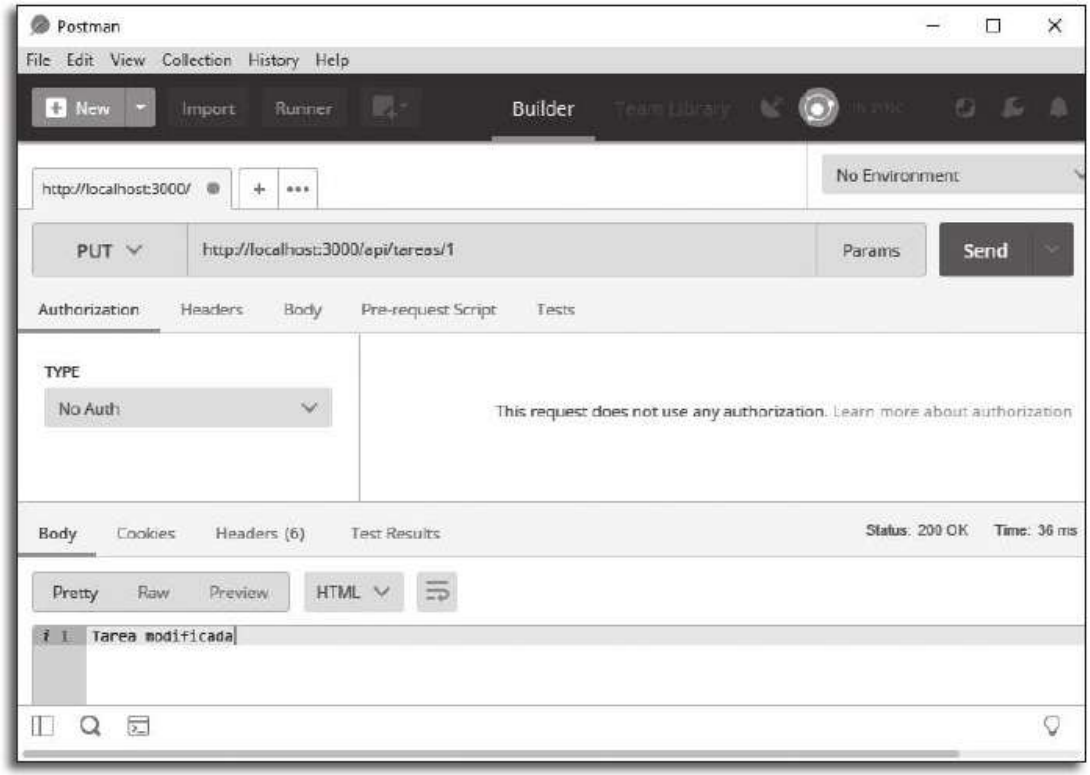
7. A través de Postman lanzaremos peticiones HTTP para cada uno de los endpoint definidos, configurando método (get,...) y URL según el caso. Vea distintos ejemplos:






Postman
File Edit View Collection History Help
New
Import
Runner
Builder
http://localhost:3000/
No Environment
GET
http://localhost:3000/api/tareas/1
Params
Send
Authorization
Headers
Body
Pre-request Script
Tests
TYPE
No Auth
This request does not use any authorization. Learn more about authorization
Body
Cookies
Headers (6)
Test Results
Status: 200 OK
Time: 34 ms
Pretty
Raw
Preview
HTML
1 <Tarea>


Postman
File Edit View Collection History Help
New
Import
Runner
Builder
http://localhost:3000/
No Environment
DELETE
http://localhost:3000/api/tareas/1
Params
Send
Authorization
Headers
Body
Pre-request Script
Tests
TYPE
No Auth
This request does not use any authorization. Learn more about authorization
Body
Cookies
Headers (6)
Test Results
Status: 200 OK
Time: 35 ms
Pretty
Raw
Preview
HTML
1 Tarea eliminada


Postman
File Edit View Collection History Help
New
Import
Runner
Builder
Team Library
http://localhost:3000/
No Environment
PUT
http://localhost:3000/api/tareas/1
Params
Send
Authorization
Headers
Body
Pre-request Script
Tests
TYPE
No Auth
This request does not use any authorization. Learn more about authorization
Body
Cookies
Headers (6)
Test Results
Status: 200 OK
Time: 36 ms
Pretty
Raw
Preview
HTML
1
Tarea modificada

074 MEAN: Creación de la API Restful (parte II)

En este ejercicio terminaremos la **RESTful API** de nuestra aplicación. El primer paso será crear el **modelo de datos "Tarea"** con los siguientes campos: titulo, creada (fecha creación) y estado ('Por hacer', 'En progreso', 'Hecha'). Este modelo de datos lo crearemos usando Mongoose, de esta manera, el modelo dispondrá de forma automática de todos los métodos necesarios para trabajar con la base de datos MongoDB a la que nos habíamos conectado. Una vez hecho esto, completaremos el **controlador "tarea"**.

Importante

Las funciones de Mongoose pueden devolver todo tipo de avisos y errores ("id de registro inexistente", etc.). Es importante gestionarlos e informar al usuario de ellos.

Vamos a empezar con el ejercicio:

1. Primero de todo abriremos el proyecto 071_ejMean con el editor Atom, y crearemos el modelo de datos "Tarea". Para ello, crearemos el fichero **tarea.js** en la carpeta **models** y le añadiremos el siguiente código:

server/models/tarea.js

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var tareaSchema = new Schema({
  titulo: {
    type: String,
    Required: 'El campo titulo es obligatorio.'
  },
  fecha: {
    type: Date,
    default: Date.now
  },
  estado: {
    type: [{
      type: String,
      enum: ['Por hacer', 'En progreso', 'Hecha']
    }],
    default: ['Por hacer']
  }
});

module.exports = mongoose.model('Tarea', tareaSchema);
```

2. Seguidamente modificaremos las funciones del **controlador "tarea"** sustituyendo el envío de texto informativo por la operativa real sobre la base de datos.

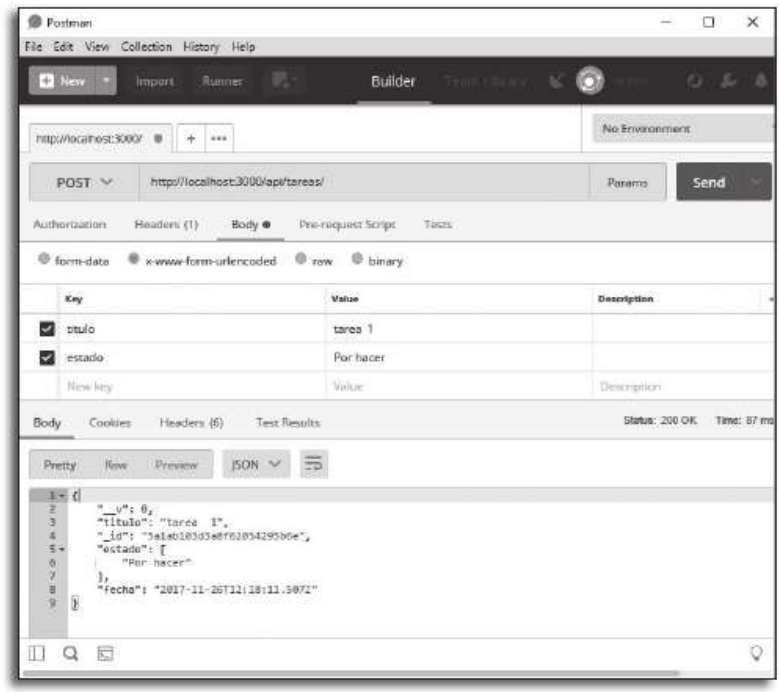
server/controllers/tarea.js

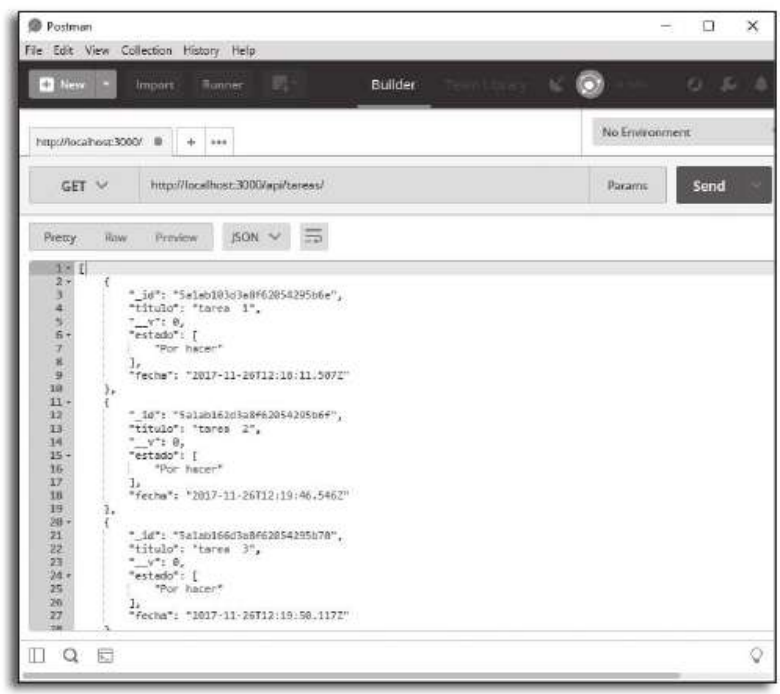
```
var mongoose = require('mongoose');
var Tarea = require('../models/tarea');

exports.list_all_tareas = function(req, res) {
  Tarea.find({}, function(err, tarea) {
    if (err) res.send(err);
    res.json(tarea);
  });
};

exports.create_tarea = function(req, res) {
  var new_tarea = new Tarea(req.body);
  new_tarea.save(function(err, tarea) {
    if (err) res.send(err);
    res.json(tarea);
  });
};

exports.read_tarea = function(req, res) {
  Tarea.findById(req.params.tareaId, function(err, tarea) {
    if (err) res.send(err);
    res.json(tarea);
  });
};

exports.update_tarea = function(req, res) {
  Tarea.findOneAndUpdate({_id: req.params.tareaId}, req.body,
{new: true}, function(err, tarea) {
    if (err) res.send(err);
    res.json(tarea);
  });
};

exports.delete_tarea = function(req, res) {
  Tarea.remove({_id: req.params.tareaId}, function(err, tarea) {
    if (err) res.send(err);
    res.json({ message: 'Tarea eliminada correctamente' });
  });
};
```

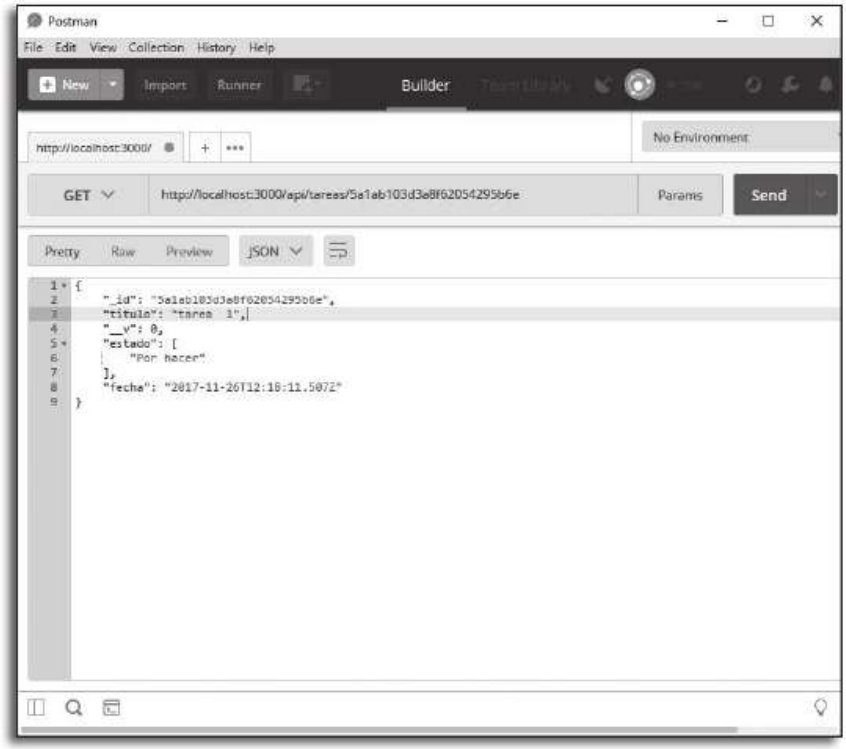
Como se puede ver en el código, el objeto creado a partir del modelo "Tarea" ya dispone de todos los métodos necesarios para trabajar con la base de datos MongoDB y se usan según convenga.

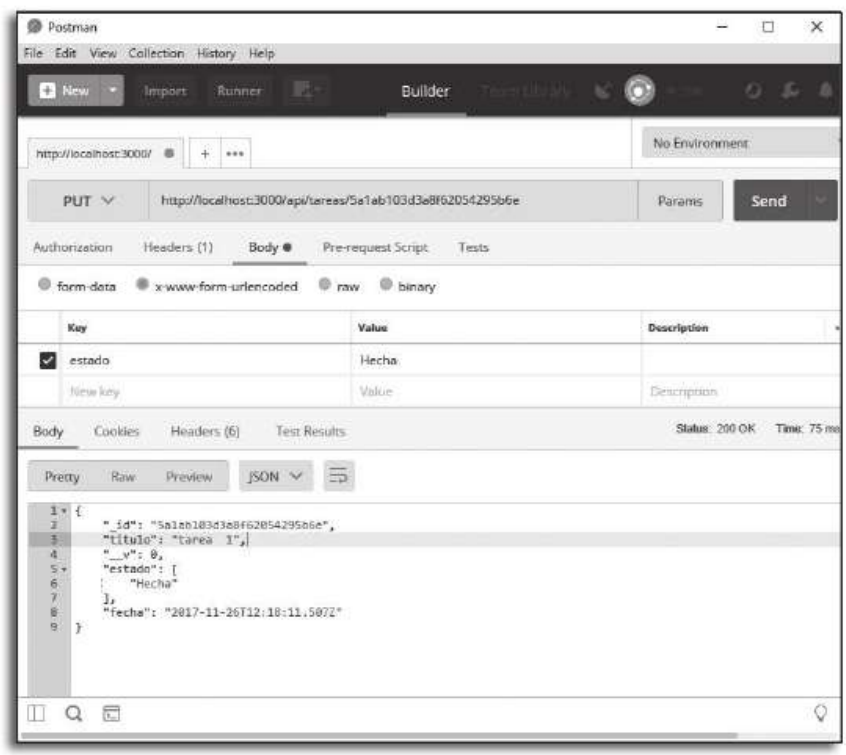
3. Con esto ya habremos terminado nuestra API. Para probarla ejecutaremos nuestro servidor y haremos las pruebas con Postman.

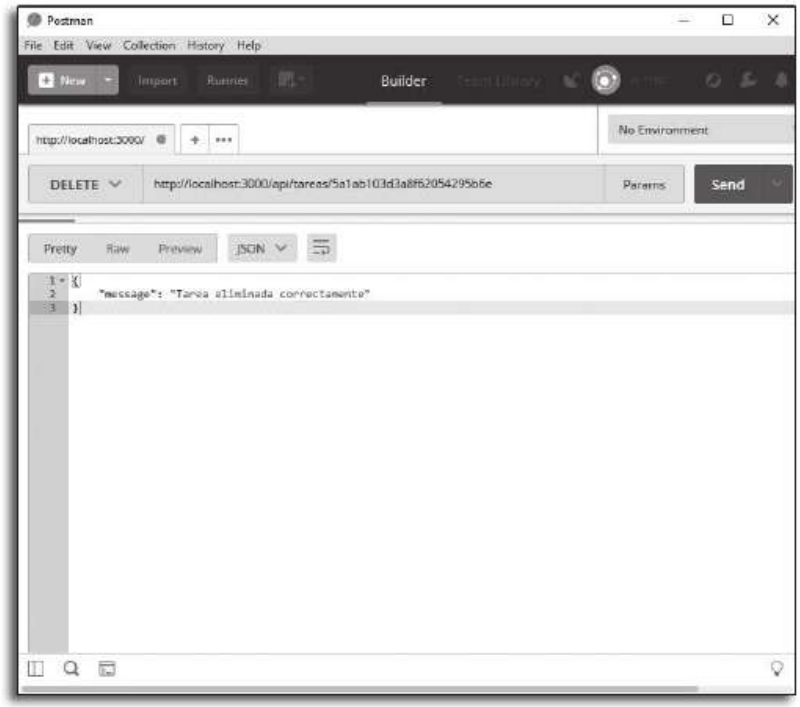
```
C:\Ej100_angular\071_ejMean>node server.js
```






Postman
File Edit View Collection History Help
New Import Runner Builder
No Environment
GET
http://localhost:3000/api/tareas/5a1ab103d3e8f62054295b6e
Params
Send
Pretty Raw Preview JSON
{
"_id": "5a1ab103d3e8f62054295b6e",
"titulo": "tarea 1",
"__v": 0,
"estado": [
"Por hacer"
],
"fecha": "2017-11-26T12:18:11.507Z"
}


Postman
File Edit View Collection History Help
New Import Runner Builder
No Environment
PUT
http://localhost:3000/api/tareas/5a1ab103d3e8f62054295b6e
Params
Send
Authorization Headers (1) Body Pre-request Script Tests
form-data x-www-form-urlencoded raw binary
Key Value Description
estado Hecha
New key Value Description
Body Cookies Headers (6) Test Results
Status: 200 OK Time: 75 ms
Pretty Raw Preview JSON
{
"_id": "5a1ab103d3e8f62054295b6e",
"titulo": "tarea 1",
"__v": 0,
"estado": [
"Hecha"
],
"fecha": "2017-11-26T12:18:11.507Z"
}


Postman
File Edit View Collection History Help
New
Import
Runner
Builder
No Environment
DELETE
http://localhost:3000/api/tareas/5a1ab103d3a8f62054295b6e
Params
Send
Pretty
Raw
Preview
JSON
{
"message": "Tarea eliminada correctamente"
}

075 MEAN: Desarrollo de componentes y rutas de la aplicación Angular

Una vez terminado el backend de la aplicación, ahora trabajaremos en su frontend. Desarrollaremos la aplicación Angular que habíamos creado en el ejercicio 071 de esta serie. La aplicación mostrará una vista principal con las tareas de la base de datos, y una de secundaria para modificarlas y para crear nuevas tareas.

Vamos a empezar con el ejercicio:

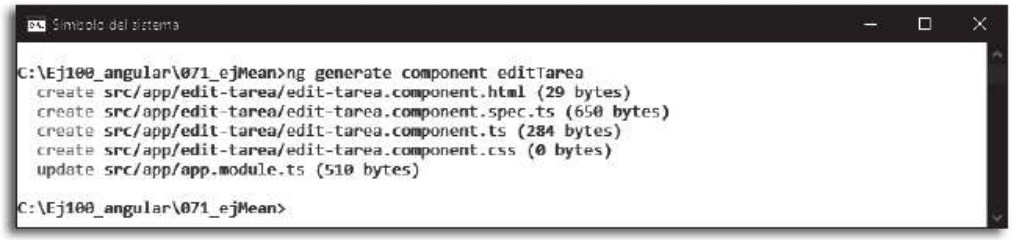
1. Primero de todo abriremos el proyecto **071_ejMean** con el editor Atom.

2. A continuación, pondremos en marcha la aplicación Angular. En estos primeros ejercicios no necesitaremos la API de servicios, por lo que de momento no pondremos en marcha la aplicación Express.

```
C:\Ej100_angular\071_ejMean>ng serve
```

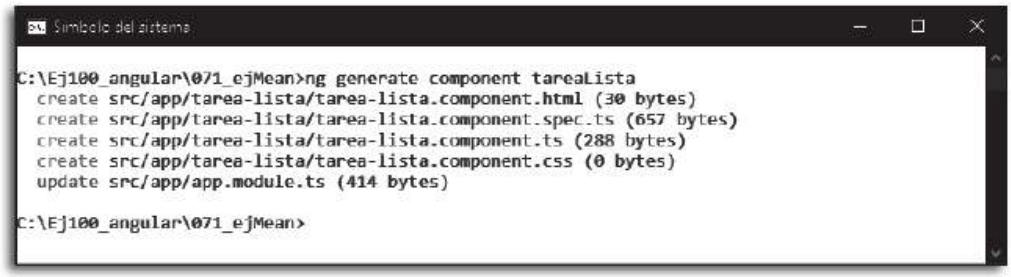
3. Seguidamente añadiremos los links de **Bootstrap** (https://v4-alpha.getbootstrap.com) en **index.html** para poder hacer uso de sus hojas de estilo. Consulte el ejercicio 087 para obtener más información.

4. Desde la línea de comandos crearemos los componentes **tareaLista** y **editTarea**. El primer componente será la vista principal de la aplicación y será donde visualizaremos la lista de tareas de la base de datos. Mientras que el segundo componente será la vista que utilizaremos para crear y modificar tareas.



Importante

Es importante que organicemos adecuadamente el código en carpetas para facilitar su gestión. Si no lo hacemos correctamente, esta gestión puede complicarse en aplicaciones grandes. Consulte https://angular.io/guide/styleguide#folders-by-feature-structure para obtener más información.




```
C:\Ej100_angular\071_ejMean>ng generate component tareaLista
C:\Ej100_angular\071_ejMean>ng generate component editTarea
```

5. Una vez creados los componentes, pasaremos a realizar la configuración del servicio **Router** para poder pasar de una vista a otra a través del URL. El primer paso será realizar la importación del servicio en "app.module.ts":

src/app/app.module.ts

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

6. A continuación, realizaremos la configuración de rutas en el mismo fichero. **/tareas** será nuestra ruta principal, y nos llevará al componente tareaLista. Para la modificación y creación de tareas tendremos las rutas "tareas/:id/edit" (":id" será el identificador de tarea) y "tareas/new", respectivamente, que nos llevarán al componente editTarea.

src/app/app.module.ts

```
…
const appRoutes: Routes= [
  { path:'tareas', component:TareaListaComponent},
  { path:'tareas/:id/edit', component:EditTareaComponent },
  { path:'tareas/new', component:EditTareaComponent },
  { path:'**', redirectTo:'/tareas', pathMatch:'full'} ];

 @NgModule({
   declarations: […],
   imports: […,RouterModule.forRoot(appRoutes)],
…
```

7. Finalmente pondremos la etiqueta "router-outlet" en el template del componente principal para indicar al servicio Router donde debe realizar las visualizaciones de los componentes.

src/app/app.component.html

```
<div class="container">
    <h3>Gestor de Tareas</h3>
    <router-outlet></router-outlet>
</div>
```

8. Desde el navegador puede realizar las pruebas para validar el código. Para el URL "localhost:4200/", será redireccionado a "localhost:4200/tareas", para el URL "localhost:4200/tareas" visualizara el componente tareaLista, para "localhost:4200/tareas/:id/edit" y "localhost:4200/tareas/new" visualizará el componente editTarea, y para cualquier otra ruta será redireccionado a "localhost:4200/tareas".

9. Antes de empezar a desarrollar los componentes, crearemos varios elementos que estos necesitarán: modelo de datos, servicio de acceso a la API del backend, etc. Para empezar, primero crearemos la carpeta shared dentro de **"src/app".** En ella pondremos todos estos elementos compartidos por los componentes de la aplicación. La carpeta la crearemos pulsando botón derecho del ratón encima de la carpeta **src/app**, seleccionando "New Folder", y definiendo la nueva carpeta como **shared**.

10. El primer elemento compartido que crearemos será el **modelo de datos "tarea"**, que dará forma a la entidad gestionada por la aplicación, o sea, una tarea. Pulsaremos el botón derecho del ratón encima de la carpeta **src/app/shared**, seleccionaremos "New File" y definiremos el nuevo fichero como **tarea.model.ts**. El modelo de datos "tarea" lo definiremos con los mismos campos y tipos que habíamos utilizado en el backend:

src/app/shared/tarea.model.ts

```
type TareaEstados = "Por hacer" | "En progreso" | "Hecha";
export var TareaEstadosSelect = [{value: "Por hacer"}, {value:
"En progreso"}, {value: "Hecha"}];

export class TareaModel {
    constructor(
      public _id: string,
      public titulo: string,
      public fecha: Date,
      public estado: TareaEstados
  ) {}
}
```

076 MEAN: Desarrollo de la operativa "Lectura de tareas"

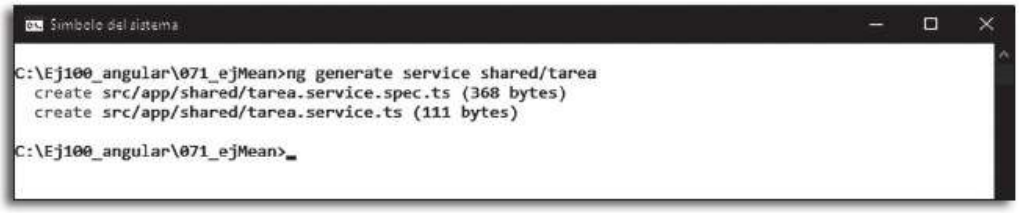
Continuado el desarrollo del frontend de nuestro "Gestor de Tareas", en este ejercicio crearemos el servicio de acceso a la API con una primera función para obtener las tareas de la base de datos, y desarrollaremos el componente **tareaLista** para que las muestre.

Vamos a empezar con el ejercicio:

1. Primero de todo abriremos el proyecto **071_ejMean** con el editor Atom.

2. Desde la línea de comandos crearemos el **servicio "tarea"** que nos dará acceso a la API del backend. Este servicio lo crearemos dentro de la carpeta **shared**.

```
C:\Ej100_angular\071_ejMean>ng generate service shared/tarea
```



3. Para poder usar el **servicio "tarea" (TareaService)**, deberemos **registrar su proveedor**. En este caso lo haremos a nivel de módulo para que todos sus componentes puedan usarlo.

   Por otra parte, "TareaService" utilizará el servicio de Angular **HttpClient** para realizar las peticiones HTTP a la API del backend. Por tanto, aprovecharemos la modificación en "app.module.ts" para añadir también la importación del **módulo HttpClientModule.**

src/app/app.module.ts

```
import {HttpClientModule} from '@angular/common/http';
import { TareaService } from './shared/tarea.service';
…
@NgModule({
  declarations: […], imports: […, HttpClientModule],
providers: [TareaService],…})
export class AppModule { }
```

4. Seguidamente crearemos la primera función de TareaService: **"getAllTareas()"**. Esta función se utilizará para obtener todas las tareas de la base de datos.

src/app/shared/tarea.service.ts

```
…
@Injectable()
export class TareaService {
  constructor(private http: HttpClient) { }
  getAllTareas() {
    return this.http.get<TareaModel[]>('http://localhost:3000/
api/tareas');
  }
}
```

Fíjese que "getAllTareas()" devuelve un **observable** obtenido a partir de la ejecución de "HttpClient.get()" sobre el endpoint "get – '/api/tareas' – 'Devuelve todas las tareas'" de nuestra API. La petición HTTP hacia la API se hará efectiva cuando se realice la **subscripción** sobre ese observable. Para más información consulte el capítulo 054.

Importante

La pipe async nos ahorra la suscripción y liberación de memoria (unsubscribe) explícitas que suelen hacerse al gestionar observables.

5. Finalmente inyectaremos el servicio "tarea" al componente "tareaLista", y modificaremos su código para que obtenga las tareas a través del servicio y las muestre por su template.

src/app/tarea-lista/tarea-lista.component.ts

```
…
@Component({…})
export class TareaListaComponent implements OnInit {
  tareas: Observable<TareaModel[]>;
  constructor(private tareaService: TareaService) { }
  ngOnInit() {
    this.tareas = this.tareaService.getAllTareas();
  }
}
```

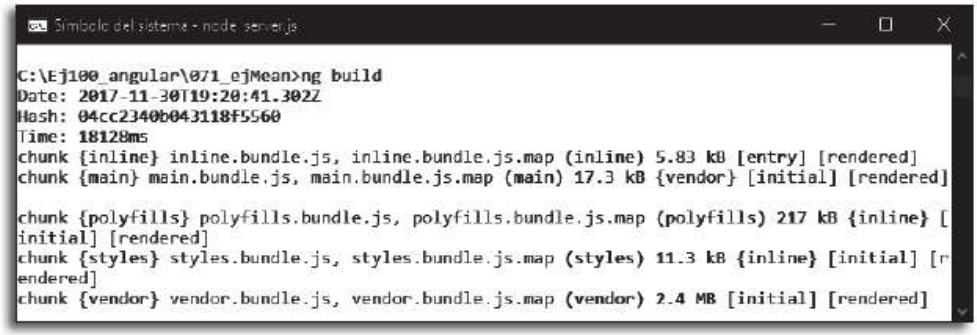
src/app/tarea-lista/tarea-lista.component.html

```
<div class="container">
    <div class="row">
       <div class="col-sm-12">
            <form class="form-inline">
              <fieldset class="form-group col-sm-11"></fieldset>
              <fieldset class="form-group col-sm-1"><a class="btn
btn-primary">+</a></fieldset>
            </form>
            <br>
            <table class="table table-striped">
                <thead>
                    <tr class="row">
                        <th class="col-sm-6">Título</th>
                         …
                    </tr>
                </thead>
                <tbody>
                    <tr *ngFor="let tarea of tareas | async"
class="row">
                      <td class="col-sm-6"><i>{{tarea.titulo}}</
i></td>
                       …
                    </tr>
                </tbody>
            </table>
       </div>
    </div>
</div>
```

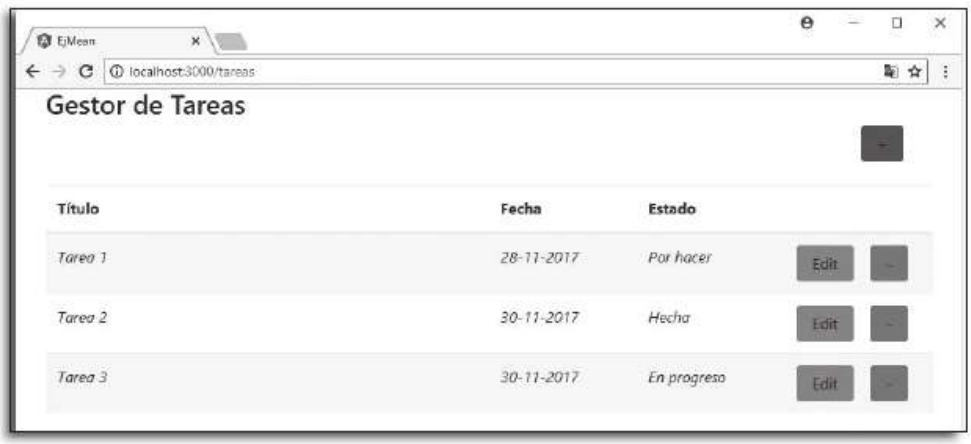
Fíjese que la subscripción al observable obtenido en "getAllTareas()" se realiza con la **pipe async**.

6. Para probar el código necesitaremos que la API esté en marcha. Por tanto, primero compilaremos la aplicación Angular mediante "ng build", y seguidamente ejecutaremos la aplicación Express con "node server.js".

7. Seguidamente, sitúese en http://localhost:3000 para que la aplicación Express le envíe la aplicación Angular al navegador. Enseguida debería ver las tareas que habíamos creado con Postman anteriormente.


Símbolo del sistema - node server.js
C:\Ej100_angular\071_ejMean>ng build
Date: 2017-11-30T19:20:41.302Z
Hash: 04cc2340b043118f5560
Time: 18128ms
chunk {inline} inline.bundle.js, inline.bundle.js.map (inline) 5.83 kB [entry] [rendered]
chunk {main} main.bundle.js, main.bundle.js.map (main) 17.3 kB {vendor} [initial] [rendered]
chunk {polyfills} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 217 kB {inline} [
initial] [rendered]
chunk {styles} styles.bundle.js, styles.bundle.js.map (styles) 11.3 kB {inline} [initial] [r
endered]
chunk {vendor} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.4 MB [initial] [rendered]


Símbolo del sistema - node server.js
C:\Ej100_angular\071_ejMean>node server.js
API running on localhost:3000
Mongoose default connection open to mongodb://localhost/db_mean


EjMean
localhost:3000/tareas
Gestor de Tareas
Título
Fecha
Estado
Tarea 1
28-11-2017
Por hacer
Edit
Tarea 2
30-11-2017
Hecha
Edit
Tarea 3
30-11-2017
En progreso
Edit

077 MEAN: Desarrollo de las operativas "creación, modificación y eliminación de tareas" (parte I)

Continuando el desarrollo del frontend de nuestro "Gestor de Tareas", en este ejercicio trabajaremos con la implementación de las funcionalidades para crear, modificar y borrar tareas. Las acabaremos en el siguiente y último ejercicio de la serie.

1. Primero abriremos el proyecto **071_ejMean** con el editor Atom.

2. Abriremos el template del componente **tareaLista** y configuraremos los botones de crear y modificar tareas. Esta configuración consistirá en añadir los links a las rutas "tareas/new" y "tareas/:id/edit" (":id": identificador de tarea), respectivamente. Las dos rutas nos llevaran al componente **editTarea.** Vea un ejemplo al pulsar la edición de una tarea.

Importante

Suele ser habitual la gestión de observables al llamar a servicios. Apóyese en los operadores RxJs para gestionar de forma efectiva todos estos observables, tal como hemos hecho aquí para cargar la tarea del componente editTarea.

src/app/tarea-lista/tarea-lista.component.html

```
…
<fieldset class="form-group col-sm-1"><a class="btn btn-primary"
[routerLink]="['/tareas/new']">+</a></fieldset>
…
<td class="col-sm-1"><a class="btn btn-info" [routerLink]="['/
tareas',tarea._id,'edit']">Edit</a></td>…
```

3. Antes de implementar el componente editTarea, vamos a añadir en el **servicio "tarea"** las distintas funciones para leer, crear y modificar una tarea de la base de datos. Cada una de ellas lanzará la petición HTTP que corresponda en cada caso.

src/app/shared/tarea.service.ts

```
…
  getTarea(id: string){
    return this.http.get<TareaModel>('http://localhost:3000/
api/tareas/' + id);
  }
  addTarea(tarea: TareaModel){
    return this.http.post<TareaModel>('http://localhost:3000/
api/tareas', { titulo: tarea.titulo, fecha: tarea.fecha,
estado: tarea.estado});
  }
  updateTarea(tarea: TareaModel){
    return this.http.put<TareaModel>('http://localhost:3000/
api/tareas/' + tarea._id, { titulo: tarea.titulo, fecha: tarea.
fecha, estado: tarea.estado});
  }
…
```

4. También necesitaremos importar **FormsModule** en nuestra aplicación, ya que en el template editTarea implementaremos un formulario para la entrada de datos de la tarea que se vaya a crear o modificar.

src/app/app.module.ts

```
import { FormsModule }    from '@angular/forms';
…
@NgModule({
  declarations: […], imports: […, FormsModule], …})
export class AppModule { }
```

5. Ahora ya podemos empezar el desarrollo del componente **editTarea**. Primero crearemos una variable tarea que contendrá la tarea que estemos modificando o creando. En la función ngOnInit() inicializaremos su valor. Si el URL contiene el parámetro id. de tarea, la inicializaremos con la tarea que nos devuelva el backend para esa id.; en caso contrario, la inicializaremos

con una nueva. Estas llamadas devuelven observables y se gestionan mediante operadores de la librería RxJs.

src/app/edit-tarea/edit-tarea.component.ts

```
…
@Component({…})
export class EditTareaComponent implements OnInit {
  tarea: TareaModel;
  tareaEstadosSelect = [];

  constructor(private route: ActivatedRoute, private router:
Router, private tareaService: TareaService) { }
  ngOnInit() {
    this.tareaEstadosSelect = TareaEstadosSelect;

    this.route.paramMap
      .map(params => params.get('id'))
      .switchMap(id => {
          if (id) return this.tareaService.getTarea(id);
          else return Observable.of(new TareaModel(null,"",new
Date(), "Por hacer"));
      })
      .subscribe(tarea => {this.tarea = tarea; console.
log(tarea);}, error => {console.log(error)});
  }
}
```

6. Finalmente, crearemos la función **onSubmit()** para guardar los datos. Si la tarea tiene identificador, se llamará a la función de modificación del servicio y, en caso contrario, a la de creación. Posteriormente, vincularemos onSubmit() al formulario del template del componente.

src/app/edit-tarea/edit-tarea.component.ts

```
…
  onSubmit(){
    if (this.tarea._id)
      this.tareaService.updateTarea(this.tarea)
        .subscribe(data => {console.log(data); this.router.
navigate(['/tareas']);},error=>console.log(error));
    else
      this.tareaService.addTarea(this.tarea)
        .subscribe(data => {console.log(data); this.router.
navigate(['/tareas']);},error=>console.log(error));
  }
}
```

078 MEAN: Desarrollo de las operativas "creación, modificación y eliminación de tareas" (parte II)

En este ejercicio acabaremos el "Gestor de Tareas" que hemos estado desarrollando durante esta serie de ejercicios.

1. Primero abriremos el proyecto **071_ejMean** con el editor Atom.

2. Seguidamente abriremos el **template** del componente **editTarea,** y le añadiremos el código para que visualice un formulario con los distintos campos de la tarea que tenga cargada (mire el anterior ejercicio para obtener más información).

Importante

A modo de práctica le proponemos añadir nuevos campos asociados a las tareas. También puede practicar con la librería HttpClient añadiendo un interceptor de mensajes para monitorizar el trafico de la aplicación con su backend.

src/app/edit-tarea/edit-tarea.component.html

```
    <div class="row">
        <div class="col-sm-12">
            <form (ngSubmit)="onSubmit()" #tareaForm="ngForm">
                <fieldset class="form-group">
                    <div class="col-sm-12">
                        <label class="control-label" for="titulo">Titulo</
label>
                        <input type="text" class="form-control" required
                         [(ngModel)]="tarea.titulo" name="titulo">
                    </div>
                </fieldset>
                <fieldset class="form-group">
                    <div class="col-sm-3">
                        <label class="control-label" for="fecha">Fecha</label>
                        <input type="date" class="form-control" required
                         [ngModel]="tarea.fecha | date: 'yyyy-MM-dd'"
                         (ngModelChange)="tarea.fecha=$event" name="fecha"/>
                    </div>
```

```
            </fieldset>
              <fieldset class="form-group">
                  <div class="col-sm-3">
                      <label class="control-label" for="estado">Estado</
label>
                      <select class="form-control" required
                       [(ngModel)]="tarea.estado" name="estado">
                          <option *ngFor="let estado of tareaEstadosSelect"
                          [value]="estado.value">{{estado.value}}</option>
                      </select>
                  </div>
              </fieldset>
              <fieldset class="form-group">
                  <div class="col-sm-offset-2 col-sm-10">
                      <button type="submit" class="btn btn-primary"
                       [disabled]="!tareaForm.form.valid">Submit</button>
                  </div>
              </fieldset>
          </form>
      </div>
  </div>
```

También hemos configurado el botón para guardar los datos. Este botón solamente estará activo si los datos son válidos, y al pulsarlo se llamará la función onSubmit() del componente.

3. En el formulario hemos indicado que todos los campos son obligatorios ("required"). Para indicar al usuario este requerimiento, vamos tocar la hoja de estilo del componente para que los campos obligatorios vacíos queden en rojo y si no en verde.

src/app/edit-tarea/edit-tarea.component.css

```
.ng-valid[required], .ng-valid.required  { border-left: 5px
solid #42A948; /* green */}
.ng-invalid:not(form)  { border-left: 5px solid #a94442; /* red
*/}
```

4. Finalmente implementaremos la funcionalidad para **borrar tareas**. Primero añadiremos la función en el **servicio tarea**:

src/app/shared/tarea.service.ts

```
deleteTarea(id: string){
  return this.http.delete<string>('http://localhost:3000/api/
tareas/' + id);
}
```

5. Y por último codificaremos el botón borrar del template de **tareaLista** para que llame a la función **deleteTarea(id)** que implementaremos en el componente.

src/app/tarea-lista/tarea-lista.component.ts

```
….
  deleteTarea(id:string) {
    this.tareaService.deleteTarea(id)
      .subscribe(data=>console.log(data),error=>console.
log(error));
    this.tareas = this.tareaService.getAllTareas();
  }
…
```

src/app/tarea-lista/tarea-lista.component.html

```
<td class="col-sm-1"><a class="btn btn-danger"
(click)="deleteTarea(tarea._id)">-</a></td>
```

6. Para probar el código recuerde ejecutar la aplicación Express con su API. Por tanto, primero compilaremos la aplicación Angular mediante "ng build", y seguidamente ejecutaremos la aplicación Express con "node server.js". Luego sitúese en http://localhost:3000 para probar la aplicación.

7. Pruebe las distintas funcionalidades implementadas. Por ejemplo, puede crear una nueva tarea, visualizarla en el listado, y editarla para cambiar su estado.


EjMean
localhost:3000/tareas/new
Gestor de Tareas
Crear Tarea
Título
Tarea 5
Fecha
15/11/2017
noviembre de 2017
lu ma mi ju vi sá do
30 31 1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 1 2 3


EjMean
localhost:3000/tareas/new
Gestor de Tareas
Crear Tarea
Título
Tarea 5
Fecha
15/11/2017
Estado
Por hacer
Por hacer
En progreso
Hecha


EjMean
localhost:3000/tareas
Gestor de Tareas
Título
Fecha
Estado
Tarea 1
28-11-2017
Por hacer
Edit
Tarea 2
30-11-2017
Hecha
Edit
Tarea 3
30-11-2017
En progreso
Edit
Tarea 5
15-11-2017
Por hacer
Edit


EjMean
localhost:3000/tareas/5a2068286...
Gestor de Tareas
Modificar Tarea
Título
Tarea 5
Fecha
15/11/2017
Estado
Por hacer
Por hacer
En progreso
Hecha

079 CSS: Introducción (parte 1)

Queda fuera del alcance de este libro la realización de un curso de **CSS**, pero hemos considerado oportuno incluir un par de ejercicios para familiarizarnos con este lenguaje prácticamente imprescindible en la programación web de hoy en día.

CSS es un lenguaje de estilo que define la presentación de los documentos **HTML** y las siglas que le dan nombre son el acrónimo de **CascadingStyle Sheets (hojas de estilo en cascada)**.

Importante

Use **CSS** para dar estilo a sus páginas **HTML** y defina todo en un **archivo externo** para facilitar el mantenimiento del mismo y de las páginas que lo utilicen.

Entre los diversos beneficios obtenidos por el uso de **CSS**, se halla el de poder concentrar todas las definiciones previstas para los distintos elementos que componen nuestras páginas en un solo archivo de estilos de forma que podemos variar el aspecto de cientos de documentos muy rápidamente solo modificando dicho archivo de estilos.

La sintaxis básica es: **selector { propiedad: valor },** donde:

- **selector:** elemento HTML al que se aplica la propiedad (p. ej., **<p>**),
- **propiedad:** propiedad que queremos modificar (p. ej., **'color'**),
- **valor:** valor de la propiedad (p. ej., **'red'**).

Las formas de aplicar **CSS** a un documento son las siguientes:

- **Directamente** a una etiqueta.
- Usando la etiqueta **style** y definirla en el apartado head de nuestra página.
- Usando un **fichero de estilos** con la extensión **css** y referenciándolo en nuestra página. Este es el método más recomendado, ya que es una de las características que más potencia da al lenguaje.

En el siguiente ejercicio vamos a probar algunas de las propiedades relacionadas con **colores**, **fuentes** y **textos**.

COLORES Y FONDOS	
color	Permite describir el **color** de **primer plano** de un elemento.
background-color	Permite describir el **color** de **fondo** de los elementos.
background-image	Permite insertar una **imagen** de **fondo**
background-repeat	Permite indicar como se ha de repetir la imagen (**horizontal**, **vertical**, **ambas** o **ninguna**).
background-attachment	Permite especificar si la **imagen** está **fija** o se **desplaza** con el elemento contenedor.
background-position	Permite **posicionar** la **imagen** de **fondo** en cualquier lugar de la pantalla.
background	Permite **comprimir** varias **propiedades** en **una** sola **instrucción**.

FUENTES	
font-family	Permiten especificar una **lista** de **fuentes** que se usarán para **mostrar** un **elemento**. Si la primera fuente no existe en el equipo, se usa la siguiente y así sucesivamente.
font-variant	Permite **elegir** entre las variantes **normal** o **small-caps**.
font-weight	Permite indicar **intensidad** de la '**negrita**' de la fuente.
font-size	Permite indicar el **tamaño** de las **fuentes** (en **píxeles** y **porcentajes**)
font	Permite incluir todas las **propiedades relativas** a fuentes en una **única** sentencia.

TEXTO	
text-indent	Permite aplica **sangría** a la **primera línea** del **párrafo**.
text-align	Permite **alinear** el **texto horizontalmente**.
text-decoration	Permite **añadir efectos** al texto (**underline**, **overline**, **line-through**)
letter-spacing	Permite especificar **espaciado entre** cada uno de los **caracteres** de texto.

1. En primer lugar, nos situaremos en la carpeta **C:\Ej100_angular** y crearemos una carpeta denominada 079_CSS_Intro1 para desarrollar este ejercicio.

2. Dentro de la misma, crearemos dos archivos: **MiPagina.html** y **estilos.css**.

3. **MiPagina.html** contendrá inicialmente lo siguiente:

```
<html>
<head>
    <title>Mi pagina</title>
    <link rel="stylesheet" type="text/css" href="estilos.
   css" />
</head>
<body>
    <h1>Pagina de pruebas</h1>
</body>
</html>
```

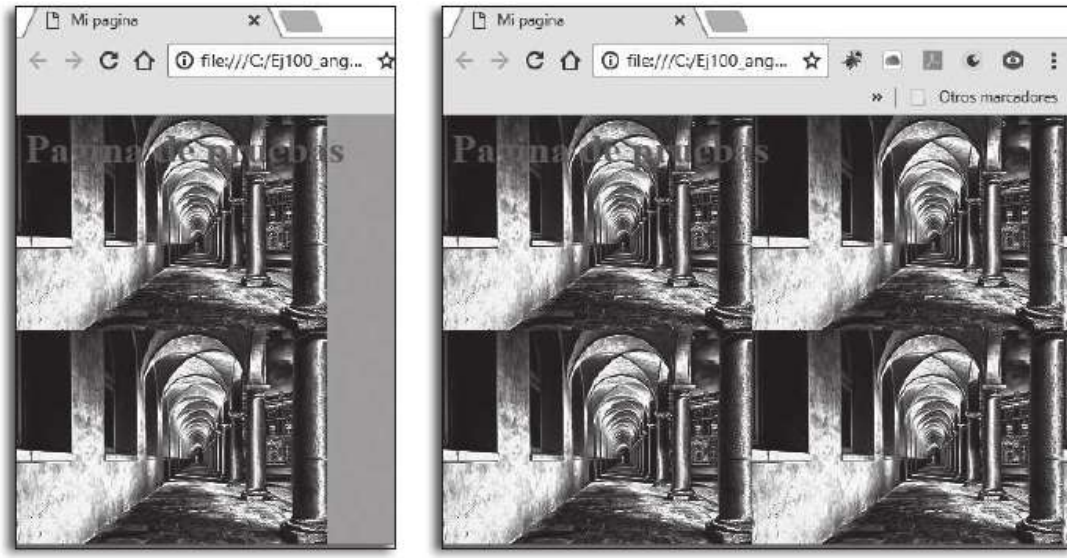
El fichero de estilos contendrá una definición de color para **<h1>** y otra de color de fondo para el **<body>** de la siguiente manera:

```
h1 {
    color: #2E64FE;
}
body {
    background-color: #58FAAC;
}
```

Si visualizamos la página en nuestro navegador veremos cómo, efectivamente, nuestro **<h1>** se muestra en color azul sobre un fondo verde .

4. Ahora, en la misma carpeta insertaremos una imagen cualquiera (p. ej., **img1.jpg**) la cual usaremos como imagen de fondo para **<body>** de la siguiente manera:

```
body {
    background-color: #58FAAC;
    background-image: url("img1.jpg");
    background-repeat: no-repeat;
}
```



Haga diversas pruebas cambiando **no-repeat** por **repeat-x**, **repeat-y** **repeat**.

5. Añadimos a **<body>** las siguientes instrucciones para colocar la imagen en la **esquina inferior derecha** y hacer que siempre se muestre en esa posición, aunque en la ventana se produzca un **scroll**:

```
    background-position: right bottom;
    background-attachment: fixed;
```

A continuación, añadiremos algunas propiedades relacionadas con las fuentes, incluyendo los siguientes estilos para **<h2>** y **<h3>** en **estilos.css**:

```
h2 {
    color: red;
    font-variant: small-caps;
    font-family: arial, verdana, sans-serif;
    font-weight: bold;
}
h3 {
    color: yellow;
    font-style: italic;
    font-size: 120%;
}
```

En la página **HTML** incluya las etiquetas **<h2>** y **<h3>** detrás de **</h1>**:

```
<body>
    <h1>Pagina de pruebas</h1>
    <h2>Segundo titulo</h2>
    <h3>Tercer titulo</h3>
</body>
```

Guarde los archivos y refresque la página en el navegador.

6. Por último, vamos a probar algunas propiedades relacionadas con el texto. Para ello, incluiremos los siguientes estilos al final del archivo **estilos.css**:

```
p {
    text-decoration: underline;
    letter-spacing: 6px;
    text-transform: uppercase;
}
```

En la página **HTML** incluimos un párrafo detrás de **</h3>** con los siguiente:

```
<p>Es un parrafo</p>
```

Gracias al estilo, observaremos que el texto del párrafo aparece subrayado, con una separación de 6 píxeles entre cada carácter y todo en mayúsculas.


Mi pagina
file:///C:/Ej100_ang...
Otros marcadores
Pagina de pruebas
SEGUNDO TITULO
Tercer titulo
ES UN PARRAFO

080 CSS: Introducción (parte 2)

En este segundo ejercicio dedicado a CSS, vamos a ver algunos ejemplos más, relacionados con la forma de mostrar enlaces, identificar elementos, crear cajas, listas, usar bordes, posicionamiento de elementos, etc.

A la hora de definir estilos en un archivo **.css**, podemos hacerlo referenciando los selectores convencionales o bien referirnos a clases (**class**), pseudo-clases (o **estado** de elementos), identificadores (**id**) o una combinación de todos ellos.

Por ejemplo, para asignar un color de fuente de color rojo, podríamos hacerlo a:

Importante

Puede aplicar diversas clases a un mismo elemento para combinar estilos y también puede realizar definiciones aplicables a diversos elementos y combinaciones entre elementos, estados e identificadores.

Elemento	Comentario	Ejemplo
selector	<h1>, <p>, ,, etc.	`span {color: blue;}`
class	Un nombre de clase añadida a un selector mediante class="miClase"	`.miClase { color: blue;}`
a:visited	Un link que ya ha sido visitado	`a:visited {color: red;}`
#miId	Un identificador determinado	`#miId {color: red;}`
selector:estado.clase	Un selector con un estado y una clase	`div:hover.impares {color: red;}`
selector:estado.id	Un selector con un estado y un identificador	`div:hover#dos {color: red;}`
Selector1, selector2, ...	Un selector u otro	`span, h1 {color: red;}`

1. En primer lugar, nos situaremos en la carpeta **C:\Ej100_angular** y crearemos la carpeta denominada **080_CSS_Intro2** para desarrollar este ejercicio. Dentro de la misma, crearemos dos archivos: **MiPagina.html** y **estilos.css**.

2. **MiPagina.html** contendrá un par de enlaces y un párrafo que incluye una etiqueta **span**. **span** básicamente es un contenedor genérico que nos permitirá asignar estilos a partes de una página.

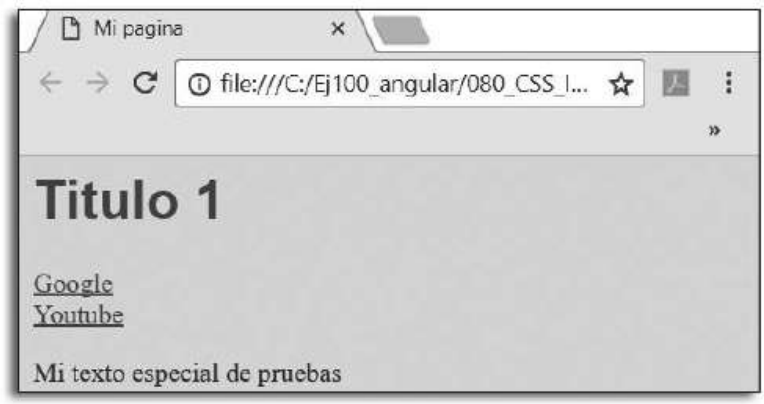
```
<html>
<head>
  <title>Mi pagina</title>
  <link rel="stylesheet" type="text/css" href="estilos.css" />
</head>
<body>
  <h1>Titulo 1</h1>
  <a href='http://www.google.com'>Google</a><br>
  <a href='https://www.youtube.com/'>Youtube</a><br>
  <p>Mi texto <span>especial</span> de pruebas</p>
</body>
</html>
```

3. **estilos.css** contendrá una definición de color para **<h1>** y otra de color de fondo para el **<body>**.

```
body { background-color: #CEF6F5; }
h1 {font-family: arial, verdana, sans-serif;}
h1 {color: blue;}
```

4. Podemos observar que **<h1>** contiene diversas definiciones y no todas tienen por qué estar agrupadas (hay 2 líneas referidas a **<h1>**). Veremos que esto facilita el hecho de que una misma definición de estilo pueda aplicarse a diferentes tags o elementos.

5. Guardamos los archivos y visualizamos la página en el navegador para ver cómo queda.

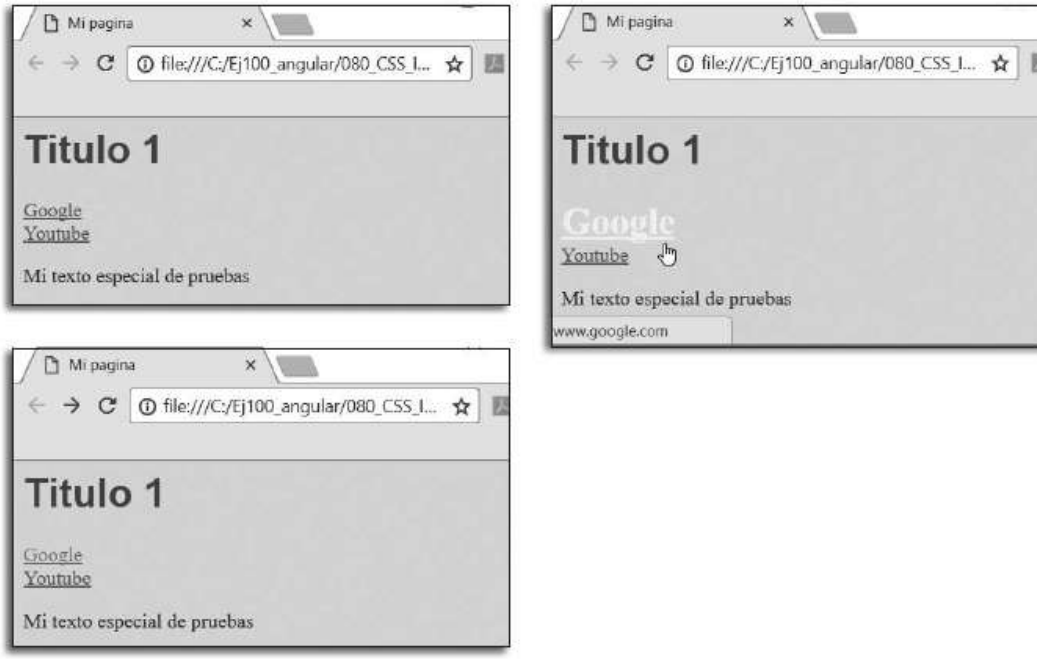


6. A continuación, vamos a utilizar el concepto de **pseudo-clase** que, en definitiva, se trata de una palabra clave que añadimos a los selectores para

poder especificar el estado de un elemento. Por ejemplo, podríamos distinguir entre los enlaces no visitados **(link)**, los visitados **(visited)** o detectar cuando pasamos el puntero del ratón sobre un enlace **(hover)**. En nuestro caso, añadiremos al final de nuestro fichero **estilos.css** una definición para cada estado.

```
a:link { color: #04B404; }
a:visited { color: red; }
a:hover { color: yellow;
  font-weight: bold;
  font-size: 30px;
}
```

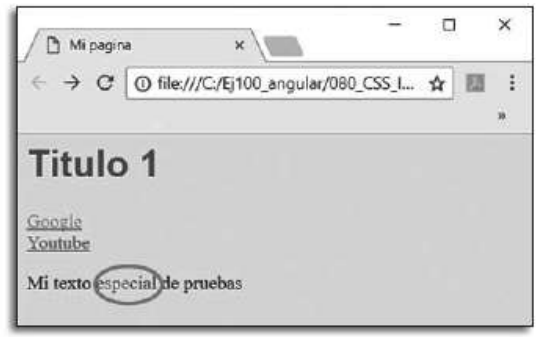
7. Si guardamos la página y la visualizamos en el navegador, vemos que los links no visitados **(link)** aparecen en **verde**, si pasamos el puntero del ratón sobre el link de Google **(hover)** el texto cambia de tamaño y de color y si pulsamos sobre el link de **Google** y regresamos de nuevo a la página, ahora veremos que el link visitado **(visited)** aparece en rojo.



8. Ahora, añadiremos una definición para mostrar en color **azul** el texto contenido la apertura y cierre de **span**. Para ello, podemos hacer una definición exclusiva para **span** o bien aprovechar la que ya hicimos para **h1** simplemente añadiendo la palabra **span** al inicio de la definición y una coma antes de **h1**.

```
h1 {font-family: arial, verdana, sans-serif;}
span, h1 {color: blue;}
a:link { color: #04B404; }
```

9. Podemos comprobar cómo queda en la página.



10. A continuación, vamos a añadir otro párrafo (**<p>**) y 3 ** ** sobre los que añadiremos algunos estilos.

```
  <p>Mi texto <span>especial</span> de pruebas</p>
  <p>Mi segundo parrafo <span>tambien especial</span> de pruebas</p>
  <div>Div 1</div>
  <div>Div 2</div>
  <div>Div 3</div>
</body>
```

11. En primer lugar, haremos una definición genérica para los div incluyendo en **estilos.css** algunos estilos.

```
div{
  margin: 5px;
  padding: 20px;
  border-width: 4;
  border-style: double;
  border-color: #0A5B05;
  font-size: 15px;
}
```

12. Guarde los archivos y compruebe cómo queda en el navegador. Recuerde que **margin** define el espacio alrededor del elemento y **padding** define el espacio entre el borde y el elemento. Compruebe también el ancho, estilo y color del borde además del tamaño de la fuente.

13. En **css** haremos una definición para una clase denominada especial (**.especial**) y otra definición para el identificador (**#dos**).

```
.especial{
  background-color: #63E31E;
  border-color: black;
  font-weight: bold;
}
#dos{
  color: green;
  background-color: #F2F5A9;
  position:absolute;
  bottom: 50px;
  right: 50px;
}
```

14. Añadimos las clases y el identificador a nuestro párrafo nuevo y a los div impares dentro del archivo HTML.

```
  <p>Mi texto <span>especial</span> de pruebas</p>
  <p class="especial">Mi segundo parrafo <span>tambien especial</span> de pruebas</p>
  <div class="especial">Div 1</div>
  <div id="dos">Div 2</div>
  <div class="especial">Div 3</div>
</body>
```

15. Veamos en el navegador cómo queda. Observe que, al redimensionar la página, el **div** con identificador dos se reposiciona dentro de la página.

16. Por último, añadiremos un par de definiciones para ver cómo se pueden combinar elementos, estados e identificadores.

17. En un caso, estamos definiendo que, al pasar el ratón por encima de un ** ** que posea la clase **especial** o al pasar el ratón sobre el título **<h1>**, cambie el aspecto del bloque. Por otra, el aspecto cambiará al pasar sobre el bloque cuyo identificador es **dos**.

```
div:hover.especial, h1:hover {
  color: blue;
  background-color: #2E9AFE;
  border-color: black;
  font-weight: bold;
}
div:hover#dos{
  color: blue;
  background-color: #F29D1B;
  font-weight: bold;
}
```

081 HTML

Es el lenguaje utilizado en Internet para fabricar páginas web. Tal y como sus propias siglas indican, se trata de un lenguaje "de marcado" **(HyperText Markup Language)** basado en una serie de etiquetas interpretadas por los navegadores para dar formato a las páginas. Además de texto, permite definir imágenes, vídeos, sonidos, juegos, etc. Para su edición, puede utilizarse cualquier editor de texto (p. ej., **Bloc de notas**) y los archivos que se generen han de tener generalmente la extensión **html** (aunque pueden usar otras como **htm**, etc.). Las etiquetas utilizadas están rodeadas por los símbolos < , > y representan un elemento. Los elementos a su vez poseen dos propiedades fundamentales: sus **atributos** y su **contenido**.

Importante

HTML es un lenguaje muy extendido y útil ya que, gracias al mismo, puede representarse el mismo contenido en cualquier plataforma y sistema que posea un navegador. Le invitamos a que profundice en el lenguaje para contar con más recursos a la hora de diseñar sus aplicaciones.

En general, todos los elementos tienen una etiqueta (o **tag**) de apertura y otra de cierre que es la misma, pero anteponiendo al nombre del elemento, una barra ('/'). A continuación, mostramos uno de los elementos más comunes como es el párrafo:

```
<p>Ejemplo de párrafo</p>
```

Un ejemplo de estructura básica de un documento **HTML** donde pudiéramos incluir el párrafo de ejemplo anterior está compuesto por el siguiente código:

```
<!DOCTYPE html>
<html>
  <head>
      <title>Título de la página</title>
  </head>
  <body>
      <p>Ejemplo de párrafo</p>
  </body>
</html>
```

Podemos observar que el documento está encerrado entre los tags **<html>** y en su interior contiene esencialmente un **<head>** y un **<body>**. **<head>** define la cabecera del documento y puede contener otros elementos relacionados con el estilo de la página o metadatos que aportan información sobre el autor u otros. **<body>** es la parte donde se define realmente el contenido y se muestra en los navegadores. Dentro de este apartado es donde se incluyen las etiquetas con las que fabricaremos nuestras páginas web. En nuestro navegador solo veremos el párrafo y el título de la página.

No podemos relacionar todas las etiquetas que pueden utilizarse, ya que nos extenderíamos mucho más allá de lo que pretendemos en este ejercicio, pero de forma muy esquemática relacionamos a continuación las etiquetas más comunes:

Etiqueta	**Comentario**
<html>	Define el documento de tipo HTML.
<title>	Especifica el título del documento.
<style>	Permite indicar información de estilo.
<body>	Contiene el cuerpo del documento.
<h1><h2><h3><h4><h5>	Encabezados (nivel de 1 a 5).
<p>	Párrafo.
<hr>	Incluye línea de separación.
<pre>	Texto preformateado que mantiene el texto tal y como está en el archivo.
	Lista no ordenada.
	Elemento de una lista.
	Contenedor para bloques de texto.
<a>	Define hipervínculos.
<i>	Muestra texto en itálica.

<b>	Muestra texto en negrita (bold).
 	Introduce un salto de línea.
<img>	Inserta una imagen.
<object>	Permite insertar aplicaciones.
<table>	Define una tabla.
<tr>	Define una fila de la tabla.
<td>	Define una columna dentro de una tabla.
<form>	Define un formulario (vea capítulo dedicado a forms dentro de este libro).
<script>	Permite insertar programas.

En el siguiente ejercicio, vamos a elaborar una página **HTML** con algunos ejemplos de las etiquetas más importantes usadas en este lenguaje.

1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos la carpeta **081_HTML** tecleando lo siguiente:

```
C:\Ej100_angular>mkdir 081_HTML
```

   Dentro de la carpeta, crearemos un archivo llamado **MiPagina.html** con el siguiente contenido:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <table border=1 style="background-color:#A9F5F2">
        <th>Etiqueta</th>
        <th>Ejemplo</th>
        <tr>
            <td>&lt;p&gt;</td>
            <td><p>Ejemplo de párrafo</p></td>
        </tr>
    </table>
</body>
</html>
```

En la página, hemos creado una tabla para ir depositando los ejemplos que vayamos insertando (**<**) y (**>**) son los caracteres de '<' y '>', respectivamente, que han de indicarse así para evitar que **HTML** los interprete como caracteres especiales del lenguaje. Guarde la página y ábrala en el navegador.

2. A partir de aquí, pruebe de incorporar más etiquetas para hacer prácticas. Por ejemplo, trate de incorporar una fila para cada una de las siguientes etiquetas **<h1>**, **<pre>**, **<ul>** y **<li>**, **<a>**, **<i>** y **<b>** e **<img>**. Para el ejemplo de **img**, tendrá que copiar una imagen en la carpeta que ha creado para este ejercicio y llamarla **img1.jpg**.

3. Compruebe cómo queda la página con los ejemplos anteriores.

```
        <td><p>Ejemplo de párrafo</p></td>
    </tr>
    <tr>
        <td>&lt;h1&gt;</td>
        <td><h1>Título 1</h1></td>
    </tr>
</table>
```

```
        <td><h1>Título 1</h1></td>
    </tr>
    <tr>
        <td>&lt;pre&gt;</td>
        <td><pre>Texto   pre formatea d   o</pre></td>
    </tr>
</table>
```

```
    <tr>
        <td>&lt;ul&gt;, &lt;li&gt;</td>
        <td>
          <ul>
            <li>Elemento 1</li>
            <li>Elemento 2</li>
            <li>Elemento 3</li>
          </ul>
        </td>
    </tr>
</table>
```

```
    <tr>
        <td>&lt;a&gt;</td>
        <td><a href="http://www.google.com">Google</a></td>
    </tr>
</table>
```

```
    </tr>
    <tr>
        <td>&lt;i&gt;, &lt;b&gt;</td>
        <td><i>Itálica</i><br><b>Negrita</b></td>
    </tr>
</table>
```

```
    </tr>
    <tr>
        <td>&lt;img&gt;</td>
        <td><img src="img1.jpg" alt="Mi Imagen" width=150 height=120></td>
    </tr>
</table>
```

082 JSON

JSON (JavaScript Object Notation) es un formato de intercambio de datos bastante ligero que puede ser tratado por casi todos los lenguajes de programación. Es una alternativa a **XML** y existen multitud de utilidades que convierten ambos formatos entre sí. Se define dentro del estándar de lenguaje de programación **ECMAScript (ECMA)**.

Importante

JSON es un formato cada vez más extendido y que permite aligerar el intercambio de información entre multitud de lenguajes de programación y entornos.

Básicamente, **JSON** posee dos estructuras:

- Una **colección de pares** de **nombre/valor**: objeto (o registro, estructura, etc.).
- Una **lista ordenada de valores**: vectores o listas.

Puede representar los siguientes tipos de datos primitivos: **cadenas de texto**, **números**, **booleanos** y valores **nulos**. También es posible definir dos tipos de datos estructurados: **objetos** y **arreglos**.

Cada pareja **atributo/valor** se define de la siguiente manera:

```
"nombre": "Juanto"
```

En este caso, el atributo es "**nombre**" y el valor "**Juanto**". Los atributos siempre se encierran entre comillas ("). Los valores también excepto los numéricos que van sin comillas. Los objetos se encierran entre llaves de la siguiente manera:

```
{
    "codigo": 1,
    "nombre": "Juanto"
}
```

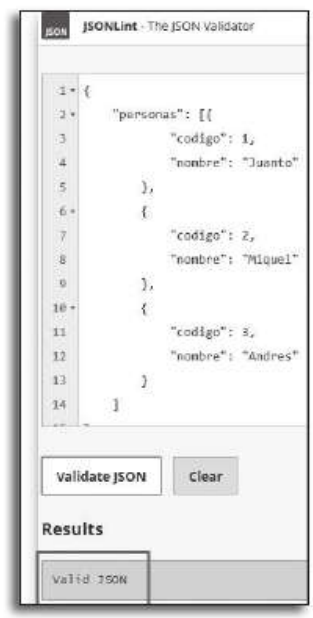
Vemos que cada pareja atributo/valor se separa de la anterior por una coma (,). Los arreglos se encierran entre corchetes **[]** y separan sus valores entre comas. Por ejemplo, si definimos el atributo "**aficiones**" con varios valores, podemos usar un arreglo de la siguiente forma:

```
"aficiones": ["Leer", "Escribir", "Musica"]
```

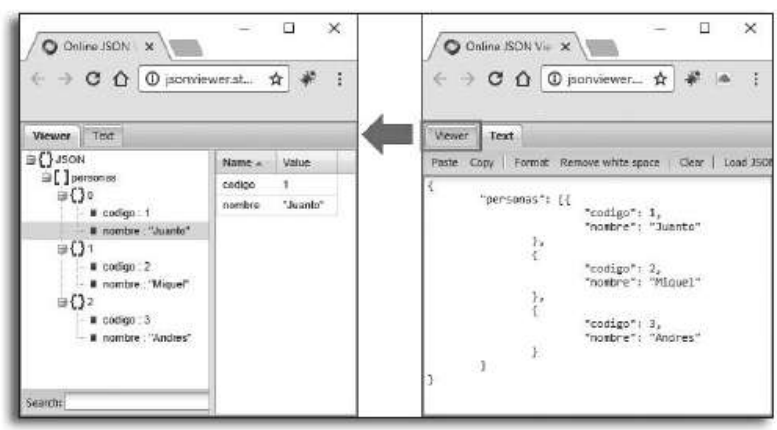
Los valores pueden ser valores simples o también objetos que a su vez pueden contener objetos y así sucesivamente. Por ejemplo, podríamos definir el objeto personas como un conjunto de objetos de la siguiente manera:

```
{
    "personas":
    [
        { "codigo": 1, "nombre": "Juanto" },
        { "codigo": 2, "nombre": "Miquel" },
        { "codigo": 3, "nombre": "Andres" }
    ]
}
```

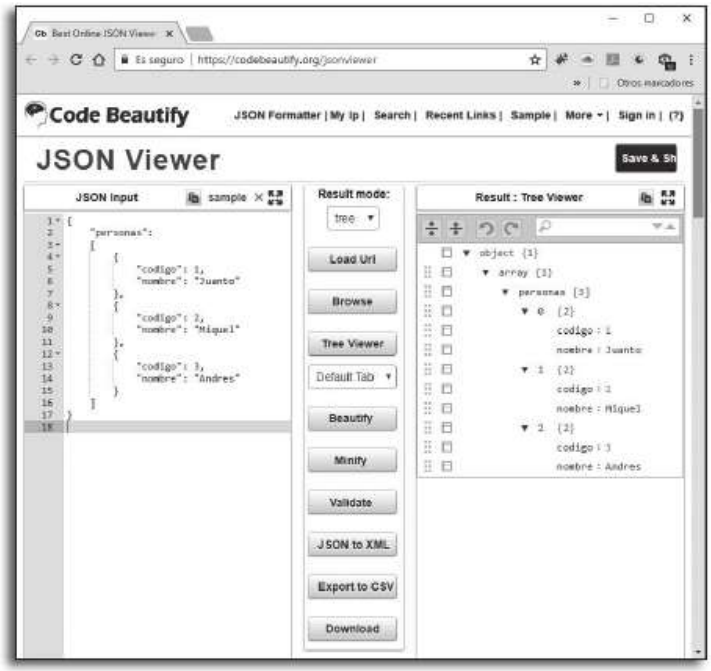
Existen muchas utilidades que nos permiten tratar archivos **JSON** validando su formato, convirtiéndolo a **XML**, exportándolo a otros formatos como **CSV**, normalizando su indentación, etc. Por ejemplo, puede usar un validador en línea llamado **JSONLint** en el siguiente URL: https://jsonlint.com/. Si acude a dicha página e introduce el código anterior en la caja de texto preparada a tal efecto y pulsa sobre el botón **Validate JSON** podrá comprobar si la estructura es correcta o no.

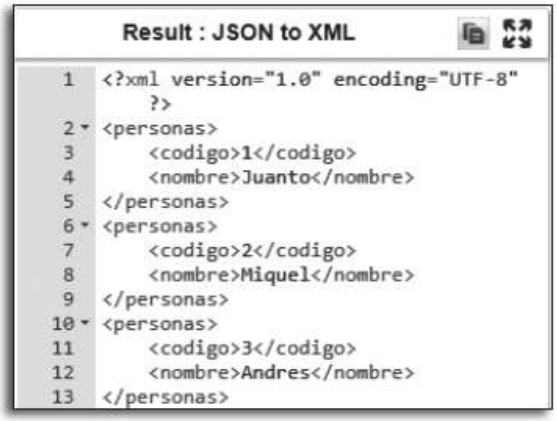


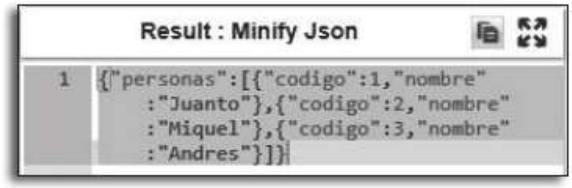
Podemos utilizar también visores para inspeccionar los datos y navegar por ellos como, por ejemplo: http://jsonviewer.stack.hu/. Si accede a esta página y copia el contenido del ejemplo anterior, puede pulsar en la pestaña **Viewer** para ver la jerarquía de la información y navegar por ella.



Por último, mostraremos otra utilidad muy completa **(Code Beautify)** que permite diferentes acciones que pueden resultar muy interesantes como, por ejemplo, validar, normalizar, convertir, exportar, editar código **JSON**, etc. Puede encontrar esta utilidad en https://codebeautify.org/jsonviewer. Por favor, introduzca el código anterior en esta utilidad y pruebe alguna de sus funcionalidades como, por ejemplo, la de **Tree Viewer** para ver la estructura de forma jerárquica y poder navegar por la misma, **Minify** para reducir el tamaño al máximo o **JSON to XML** para convertir un archivo **JSON** a **XML**.







En el siguiente ejercicio, vamos a fabricar una pequeña aplicación que simplemente nos permita mostrar en el navegador y también en su consola cómo podemos crear, acceder y mostrar alguno de los valores de una estructura **JSON**.

1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos el proyecto tecleando **ng new intJSON**.

2. Seguidamente, renombraremos el proyecto ubicándonos en **Ej100_angular**, y escribiendo lo siguiente:

```
C:\Ej100_angular>rename intJSON 082_intJSON
```

A continuación, abrimos el proyecto con nuestro editor y modificamos la clase **AppComponent** del fichero **app.component.ts** para cambiar el título por defecto y añadir algunas definiciones de objetos de JSON.

```
export class AppComponent {
  title = '082 intJSON';
  persona = {
              "codigo":1,
              "nombre":"Juan",
              "aficiones":["Leer","Escribir","Musica"]
            }
  personas = [
              { "codigo": 1, "nombre": "Juanto" },
              { "codigo": 2, "nombre": "Miquel" },
              { "codigo": 3, "nombre": "Andres" }
             ]
}
```

3. Podemos ver que hemos creado un objeto persona con 3 atributos (**codigo**, **nombre** y **aficiones**) y en el caso de aficiones, hemos creado un arreglo con las diferentes aficiones que posee esa persona. Podemos ver también que se ha creado un array de personas que incluye tres objetos persona cada uno de los cuales solo tiene **codigo** y **nombre**.

4. A continuación, vamos a definir en nuestro archivo **styles.css** algunos estilos para mejorar la presentación.

```
.verde { color: green; padding: 2px; margin: 2px;
         font-weight: bold; font-size: 20px; }
.divEjemplo { padding: 2px; margin: 2px;
              border: 2px solid blue;
              background-color: #CEF6F5; }
.divPersonas { padding: 2px; margin: 2px;
               border: 2px solid blue;
               background-color: #A9E2F3; }
.divPerson { padding: 2px; margin: 2px;
             border: 1px solid green;
             background-color: #81BEF7; }
```

5. Ahora, modificaremos el archivo **app.component.html** para modificar el contenido que posee por defecto y mostrar los objetos que contienen **JSON**. Tras el título, insertaremos un primer bloque para mostrar el objeto persona en formato **JSON** y también, cada uno de sus atributos y valores.

6. Seguidamente y detrás del bloque anterior, añadimos el objeto array de personas también en formato **JSON** y luego, cada una de las personas en formato **JSON**.

```
<div>
  <h1>{{title}}</h1>
  <div class="divEjemplo">
    <span class="verde ">persona:</span><br>
    <div class="divPerson">{{ persona | json }}</div>
    <div class="divPerson">Codigo: <b>{{ persona.codigo }}</b></div>
    <div class="divPerson">Nombre: <b>{{ persona.nombre }}</b></div>
    <div class="divPerson">Aficiones: <b>{{ persona.aficiones }}</b></div>
  </div>
</div>
```

```
    <div class="divPerson">Aficiones: <b>{{ persona.aficiones }}</b></div>
  </div>
  <div class="divEjemplo">
    <span class="verde">personas:</span>
    <div class="divPerson">{{ personas | json }}</div>
    <span class="verde">persona 1:</span>
    <div class="divPerson">{{ personas[0] | json }}</div>
    <span class="verde">persona 2:</span>
    <div class="divPerson">{{ personas[1] | json }}</div>
    <span class="verde">persona 3:</span>
    <div class="divPerson">{{ personas[2] | json }}</div>
  </div>
</div>
```

7. En el último bloque que añadiremos a continuación del anterior, y mediante ***ngFor**, recorremos el array de **personas** y mostramos por separado cada una de las mismas con sus **atributos** y **valores**.

```
    <div class="divPerson">{{ personas[2] | json }}</div>
  </div>
  <div class="divPersonas container" *ngFor="let persona of personas; let i = index">
      <span class="verde">persona {{ i + 1 }}:</span>
      <div class="divPerson container">
          Codigo:
          <b>{{ persona.codigo }}</b>
          <br> Nombre:
          <b>{{ persona.nombre }}</b>
      </div>
  </div>
</div>
```

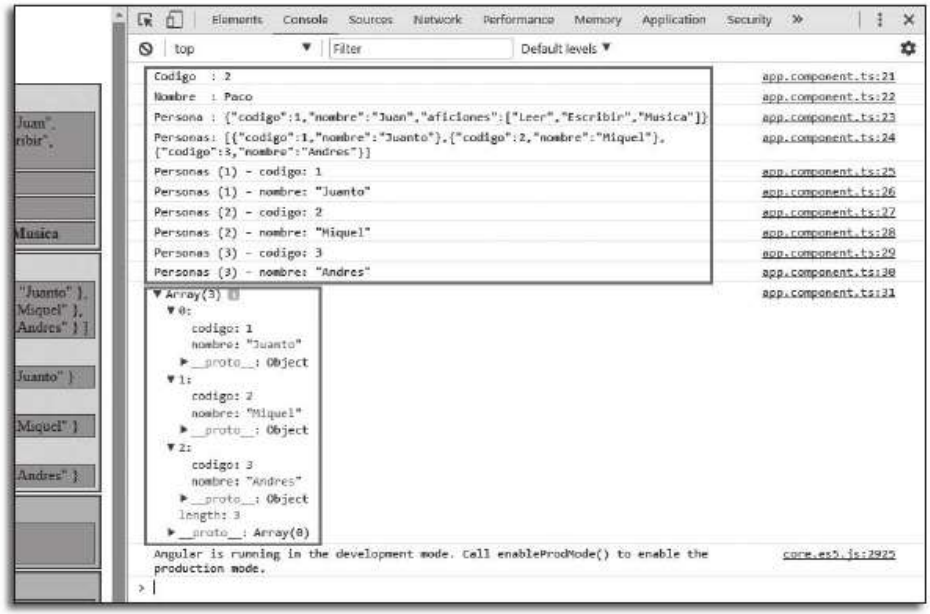
8. Guardamos todo y arrancamos la aplicación en el navegador para ver la información de cada bloque.

9. En **app.component.ts** podemos añadir el método **ngOnInit()** para mostrar por la consola la información de los objetos de forma parecida a como lo hemos hecho en **HTML**. Lo añadiremos detrás de la definición de **personas** y antes del cierre de la clase **AppComponent**. Definiremos la función con un conjunto de llamadas a **console.log** para mostrar la información de diversas formas. Observe que en **ngOnInit()** hemos usado **JSON.parse** para crear un nuevo objeto, y que

usamos también **JSON.stringify** para elaborar un **string** con el contenido del objeto tratado.

```
  { "codigo": 3, "nombre": "Andres" }
]
  ngOnInit(){
    var nuevoObjeto = JSON.parse('{"codigo":"2","nombre":"Paco"}');
    console.log("Codigo  : " + nuevoObjeto.codigo);
    console.log("Nombre  : " + nuevoObjeto.nombre);
    console.log("Persona : " + JSON.stringify(this.persona));
    console.log("Personas: " + JSON.stringify(this.personas));
    console.log("Personas (1) - codigo: " + JSON.stringify(this.personas[0].codigo));
    console.log("Personas (1) - nombre: " + JSON.stringify(this.personas[0].nombre));
    console.log("Personas (2) - codigo: " + JSON.stringify(this.personas[1].codigo));
    console.log("Personas (2) - nombre: " + JSON.stringify(this.personas[1].nombre));
    console.log("Personas (3) - codigo: " + JSON.stringify(this.personas[2].codigo));
    console.log("Personas (3) - nombre: " + JSON.stringify(this.personas[2].nombre));
    console.log(this.personas);
  }
}
```

10. Abra el navegador (p. ej., en **Google Chrome** con **CTRL+ MAYUS + I**) y analice las salidas por la consola. Vea cómo al mostrar el objeto personas al final de todo es posible desplegarlo pulsando sobre las flechas que lo acompañan para ver su estructura.



083 Google: Herramientas de desarrollador

Las herramientas para desarrolladores nos permiten analizar diferentes aspectos de nuestra aplicación web como son el rendimiento, su composición, el uso de memoria, etc. Por ejemplo, **Google Chrome (CTRL+MAYUS+I)**, **Internet Explorer (F12)**, **Mozilla Firefox (CTRL+MAYUS+I**) ofrecen un menú similar para realizar las mismas acciones.

A continuación, tomando como ejemplo **Google Chrome**, comentamos brevemente cada una de estas opciones.

Importante

Utilice las herramientas de desarrolladores para depurar programas y visualizar mensajes por la **Console**, por ejemplo, para mostrar fácilmente el contenido de una variable o el paso por un método.

Opción	Comentario
	Permite seleccionar un elemento de la página **(CTRL+MAYUS+C)**.
	Simula la visualización de nuestra aplicación en un dispositivo **(CTRL+MAYUS+M)**.
Elements	Permite analizar el estado del elemento **HTML** o **CSS** seleccionado y, además, es posible editarlo. Para deshacer cambios basta con pulsar **CTRL+Z**.
Console	Permite ver los mensajes de error y advertencias y también es muy útil si usamos console.log("**texto**") en nuestros métodos para realizar alguna traza.
Sources	Permite abrir cualquier componente de la web ofreciendo una estructura jerárquica y en árbol de los mismos y también permite depurar.
Network	Muestra el panel **Network** donde visualiza información sobre el rendimiento y actividad de red.
Performance	Muestra el panel **TimeLine** para analizar la actividad en tiempo de ejecución.

Memory	Muestra el panel **Profiles** y permite analizar el tiempo de ejecución y uso de memoria.
Application	Administra almacenamiento y BBDD.
Security	Muestra el panel **Security**, que permite depurar problemas con nuestro certificado y otros. Analiza la seguridad general de una página.
Audits	Muestra el panel **Audit**, que analiza la página cuando se carga y ofrece sugerencias para mejorar su rendimiento.
Augury	Permite depurar aplicaciones y visualizar la aplicación mediante herramientas gráficas.

En el siguiente ejercicio, mostraremos algunas de las funcionalidades para que podamos ver cómo acceder y manejar las opciones del menú de herramientas de **Google Chrome**.

Podríamos utilizar cualquier página web, pero, en nuestro caso, nos basaremos en la aplicación realizada en el ejercicio **028 dirNgStyle** al que añadiremos un bloque de salidas por **Console**.

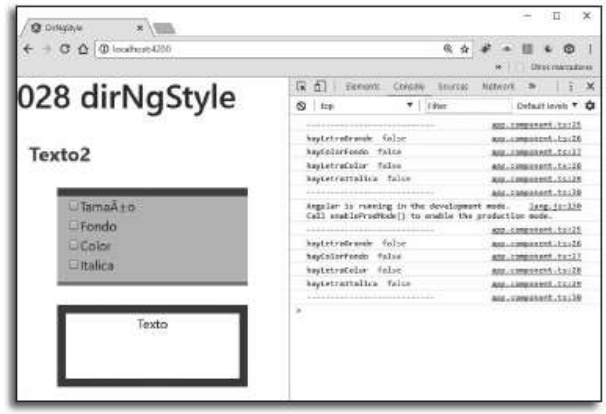
1. Abrimos el ejercicio con nuestro editor (p. ej., **Atom**) y añadimos el siguiente bloque en **app.component.ts** detrás de la clase **AppComponent**:

```
  defineEstilos(){
...
    return styles;
  }
  ngAfterContentChecked(){
    console.log("--------------------------- ");
    console.log("hayLetraGrande ",this.hayLetraGrande);
    console.log("hayColorFondo ",this.hayColorFondo);
    console.log("hayLetraColor ",this.hayLetraColor);
    console.log("hayLetraItalica ",this.hayLetraItalica);
    console.log("--------------------------- ");
  }
```

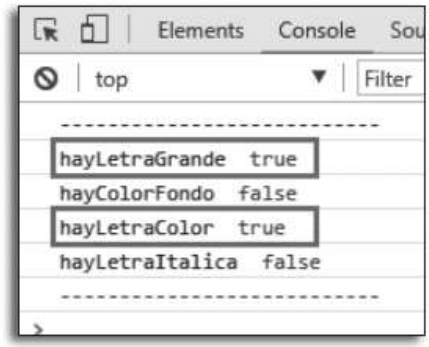
   Seguidamente nos ubicamos en **C:\Ej100_angular\028_dirNgStyle** y arrancamos la aplicación tecleando **ng serve**.

2. Abrimos el navegador y tecleamos el URL http://localhost:4200 para acceder a nuestra aplicación.

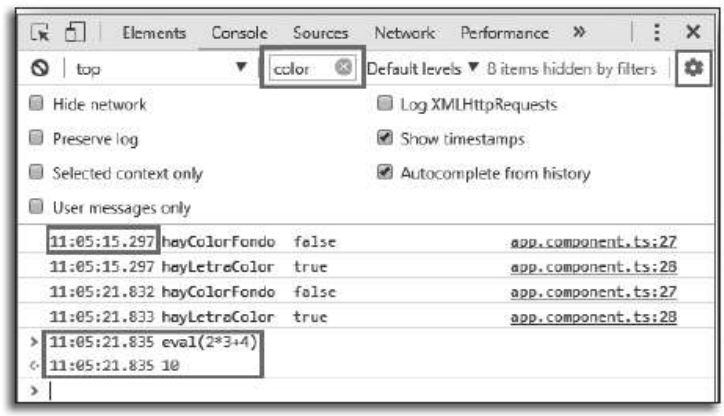
3. Accedemos a las herramientas de desarrolladores pulsando **CTRL+MAYUS+I**.



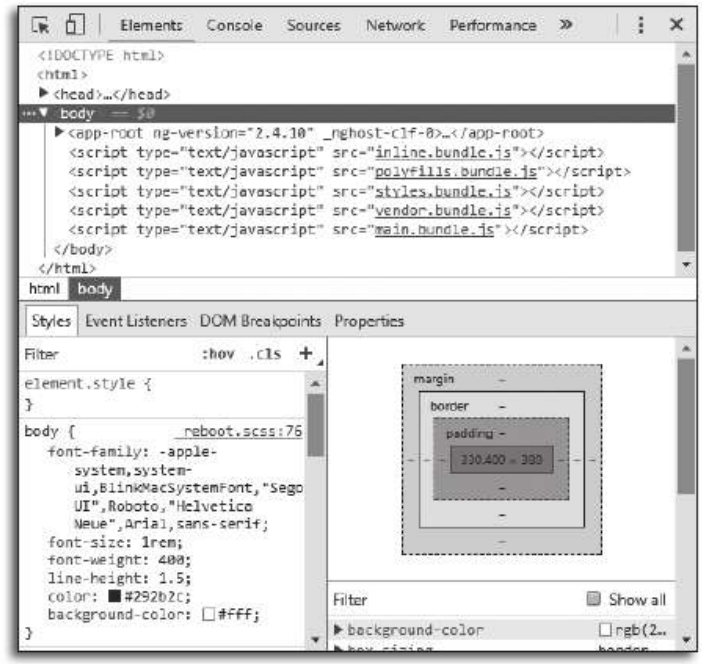
4. Si pulsamos sobre los checks **Tamaño** y **Color**, veremos que las variables **hayLetraGrande** y **hayLetraColor** pasan a valer **true**.



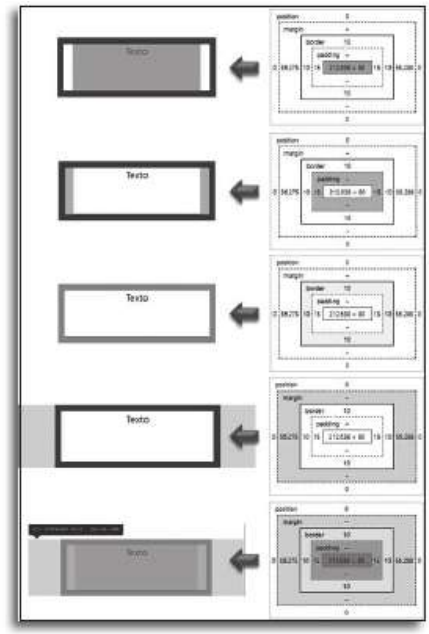
5. Podemos jugar un poco sobre la pestaña de **Console** pulsando sobre el icono ( ) y limpiando el contenido de la misma. Introducimos un filtro (p. ej., color) para mostrar solo aquellos mensajes que contengan dicho texto. Pulsamos sobre el botón **Console settings** ( ) para mostrar más opciones y marcar **Show timestamps** para añadir la hora a cada mensaje. Podemos realizar un sencillo cálculo mediante la función **eval** (p. ej., **eval(2*3+4)**) en el prompt (>) de la **Console**.



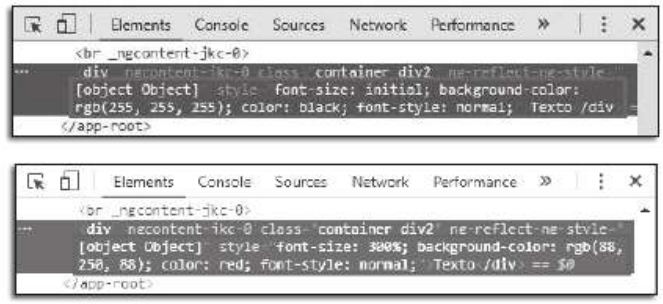
6. A continuación, seleccionaremos la opción **Elements** y observaremos que muestra el código fuente de la página además de información sobre sus **estilos**, **eventos**, **breakpoints** y **propiedades**.



7. Tecleamos **CTRL+MAYUS+C** (o pulsamos sobre el icono) y seleccionamos el bloque inferior que contiene '**Texto**'. En la parte inferior derecha de la ventana, veremos que hay una serie de rectángulos contenidos entre sí que al pasar el puntero del ratón por encima de los mismos van resaltando en la página el aspecto asociado a cada rectángulo (**position**, **margin**, **border**, **padding** y **contenido**).



8. Podemos observar también que en la parte superior izquierda aparece destacada la etiqueta div, en la que se muestra entre otras propiedades: el estilo, el color de fondo, el tamaño, etc. Si marcamos los checks **Tamaño**, **Fondo** y **Color**, observaremos cómo cambian los valores dentro de cada propiedad correspondiente.



9. Para acabar, podríamos simular la visualización de nuestra aplicación en un dispositivo móvil pulsando **CTRL+MAYUS+M** o bien pulsando sobre el botón ().

084 Control de versiones Git: Instalación, configuración y uso

Git es un sistema de control de versiones libre y de código abierto. Como todo sistema de control de versiones, Git registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de manera que se puedan recuperar versiones específicas cuando sea necesario. Además, Git es fácil de usar, rápido, eficiente y el editor Atom le da soporte. Por todo ello hemos pensado que sería interesante incluir este capítulo en el libro para ver sus conceptos de uso básicos.

Importante

Git es un sistema de control de versiones distribuido, libre, de código abierto, fácil de usar, rápido, eficiente y con soporte para la mayoría de editores como Atom o VisualCode de Microsoft.

Los fundamentos básicos de Git son los siguientes:

- Es un sistema de control de versiones **distribuido**. Los clientes replican completamente el repositorio de versiones de un servidor central. A posteriori esto permite a los clientes realizar casi todas las operaciones de forma local evitando problemas de red. Además, se mejora la seguridad en el sentido de que si el servidor central falla, siempre es posible recuperar el repositorio a partir de cualquier cliente.
- Un **proyecto Git** tiene tres partes principales: el **directorio Git** con el repositorio de versiones y otros elementos de Git, **el directorio de trabajo** con una copia de una versión del proyecto sacada del directorio **Git** para su uso y modificación, y el área **de preparación**, un archivo del directorio **Git** con los ficheros modificados preparados para la próxima confirmación (**commit**).
- Los ficheros del directorio de trabajo pueden estar **bajo seguimiento (tracked)** o **sin seguimiento (untracked)** por parte de Git, como sería el caso de los ficheros nuevos acabados de crear. Los ficheros bajo seguimiento (tracked) pueden estar **sin modificar**, **modificados** o **preparados (staged)**. El flujo general de trabajo consistiría en modificar archivos del directorio de trabajo, prepararlos para que se incluyan en la próxima confirmación (commit) y finalmente realizar la confirmación (commit) para guardar los cambios en el repositorio.

Para empezar a usar Git, el primer paso será **instalarlo**. Para ello nos desplazaremos https://git-for-windows.github.io/, y descargaremos el instalador para Windows. Luego lo ejecutaremos seleccionando todas las opciones por defecto. Una vez

instalado, tendremos tanto la versión de línea de comandos como la interfaz gráfica de usuario estándar.

Seguidamente pasaremos a **configurar un nombre de usuario y email**. Esta información es importante porque Git la utiliza y vincula a las confirmaciones (commit) que realicemos. Configúrelo indicando sus propios datos:

```
C:\>git config --global user.name "Miquel"

C:\>git config --global user.email miquel@example.com
```

Ejecutando "git config –list" puede confirmar la configuración.

Vamos a seguir practicando con Git en un proyecto Angular:

1. Nos ubicaremos en **ej100_angular**, y crearemos el proyecto tecleando **ng new testGit**. Seguidamente lo renombraremos para que haga referencia a nuestro número de ejercicio ejecutando **rename testGit 084_testGit**, y finalmente lo abriremos con **Atom**.

2. Atom da soporte Git para la mayoría de operativas más comunes, pero para las más especializadas es probable que necesitemos usar la versión Git de línea de comandos. Para visualizar la ventana Git de Atom, seleccionaremos la opción "Packages/GitHub/Toggle Git Tab" del menú.

3. Para usar Git en el nuevo proyecto, primero deberemos **crear su repositorio de versiones**. Esto lo haremos desde la ventana Git de Atom seleccionando la opción **"Create repository"** e indicando que lo cree en C:\Ej100_angular\084_testGit. Con esta acción crearemos el subdirectorio **Git** (".git") con el repositorio.

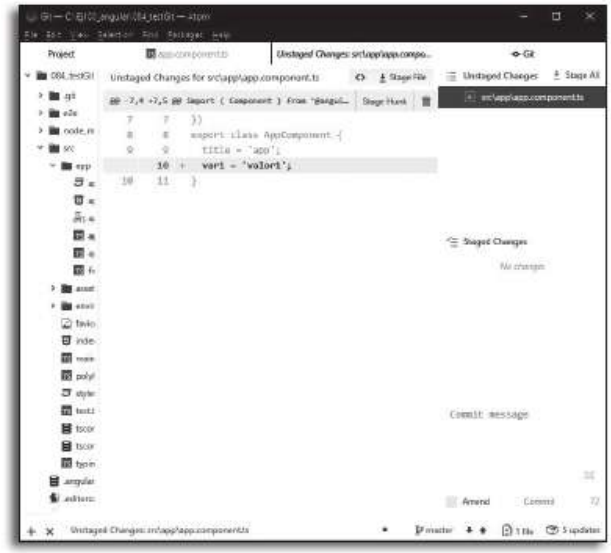
4. En este momento, todos los ficheros del proyecto están **sin seguimiento (untracked)**. Aparecen todos en "Unstaged Changes" con un icono verde. Sin embargo, si se fija bien, realmente no aparecen todos. Por ejemplo, no aparecen los ficheros descargados con npm (**directorio/node_modules**)

para los que no tiene sentido realizar ningún seguimiento. Esto es así gracias a las excepciones indicadas en el archivo **.gitignore** que se creó de forma automática al crear el proyecto con "ng new…".

5. Para empezar a controlar sus versiones, primero deberemos ponerlos bajo seguimiento (tracked). Para ello tendremos que hacer una confirmación inicial con todos ellos: pulsaremos la opción **"Stage All"**, pondremos nombre al commit, por ejemplo, "version inicial", y pulsaremos **"Commit"**.

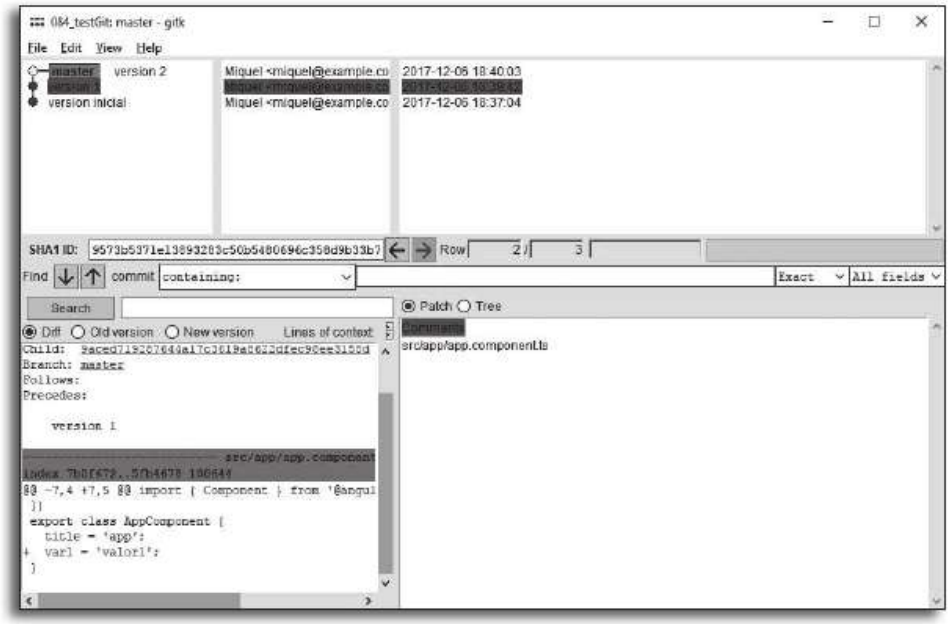
6. Ahora los ficheros están **bajo seguimiento (tracked) de Git**. Si realiza una modificación, por ejemplo, en el fichero **src/app/app.component.ts**, este fichero pasará a estado modificado y aparecerá en "Unstaged Changes" con un icono amarillo distinto al que veíamos en el punto 4. A partir de aquí, puede realizar **todo tipo de acciones** desde esta misma ventana:

   a. Para un fichero modificado, puede hacer clic para ver los cambios respecto su última confirmación, puede hacer doble clic para ponerlo en estado preparado (staged), o también puede deshacer los cambios desde el menú contextual (clic derecho) del fichero.



   b. Puede deshacer la preparación (staged) de un fichero con un clic.
   c. Puede crear una nueva confirmación (**commit**) con los ficheros preparados (**staged**).

La mejor forma de ver el **histórico de confirmaciones realizadas** es hacerlo a través de la aplicación "Git GUI", o sea, la versión Git con interfaz gráfica incluida en la instalación. Para ello ejecute "Git GUI", abra el repositorio **C:\Ej100_angular\084_testGit**, y seleccione la opción "Repository/Visualize master's History". Tendrá una vista similar al de la imagen.



Por último, comentar que podemos trabajar con **repositorios remotos** para facilitar la colaboración en un proyecto Git. Veamos, brevemente, cómo podemos crear uno en **GitHub** (servicio host para repositorios Git):

1. Primero nos situaremos en https://github.com/ y crearemos una cuenta para poder usarlo.

2. Seguidamente **creamos un repositorio** con el mismo nombre que nuestro proyecto y nos apuntaremos el URL que nos devuelva la operación.

3. A continuación, añadiremos el repositorio remoto, que llamaremos "origin" en nuestro proyecto de la siguiente forma. Hay que introducir el URL que habíamos obtenido en el punto anterior.

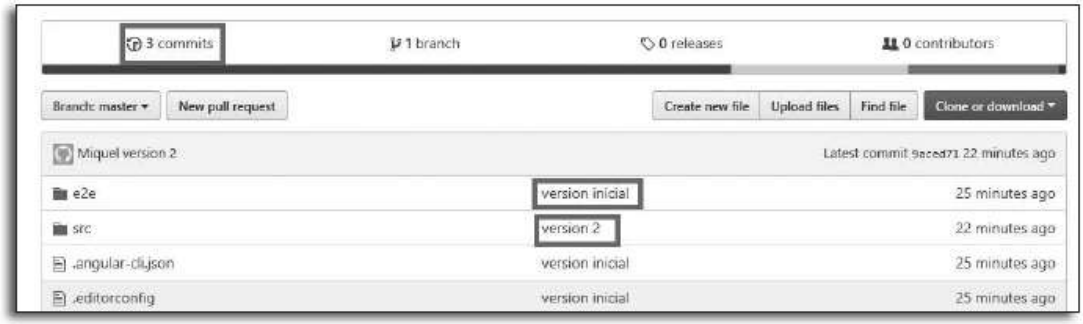
```
C:\Ej100_angular\084_testGit>git remote add origin ht-
    tps://.../084_testGit.git
```

   Y por último enviaremos nuestro repositorio local al repositorio remoto con el siguiente comando:

```
C:\Ej100_angular\084_testGit>git push -u origin master
```

   Desde GitHub puede comprobar que se ha subido todo el repositorio local. En la vista principal puede ver los ficheros con su id. de versión (nombre del

commit donde se realizó su último cambio), y también puede observar la opción que nos permitiría ver el histórico de confirmaciones.

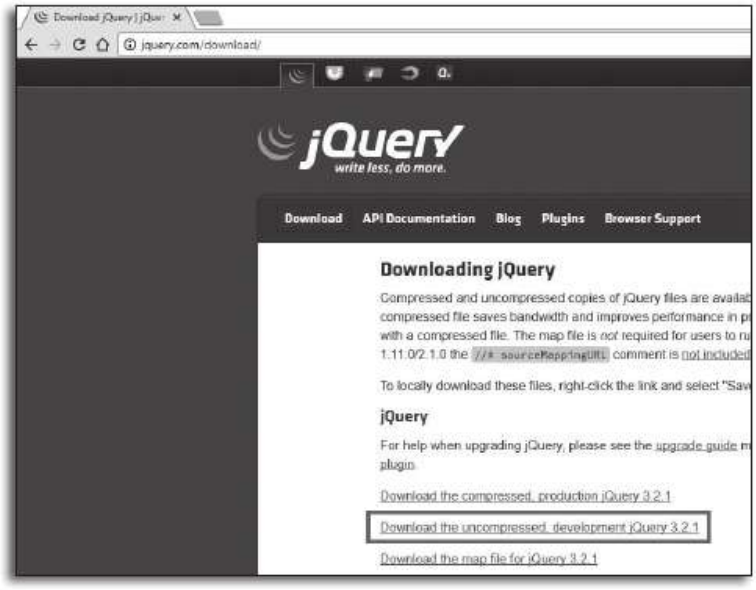


085 jQuery: Parte I

jQuery es un framework **JavaScript** que simplifica la programación de páginas web. La biblioteca que aporta simplifica parte de la problemática de trabajar con diferentes navegadores y sus particularidades. La declaración de intenciones de **jQuery** es **"escribir menos para hacer más" ("write less, do more")**. La primera versión de **jQuery** fue desarrollada en 2006 por **John Resig** y la última versión puede obtenerse desde la web de **jQuery** ubicada en la siguiente dirección: http://jquery.com/download/.

Importante

jQuery resulta de gran ayuda para manipular el DOM y usado en combinación con otras herramientas facilita enormemente la programación en web.



A modo de resumen, los temas que trata **jQuery** son:

- **DOM:** estructura que contiene los diferentes objetos existentes en un documento cargado en el navegador (***document objet model***) o, lo que es igual, **modelo de objetos del documento**.
- **Selector:** expresión mediante la cual podemos seleccionar uno o varios elementos que cumplen con un determinado criterio (**filtros**: **contains**, **ends with**, **starts**, etc., **genéricos**: .**class**, **element**, #**id**-).
- **Atributo:** de **elementos** (**attr**(), **prop**(), **val**(), **removeAttr**(), **removeProp**(), etc.), de **clase** (**addClass**(), **removeClass**() y **hasClass**()).

- **Eventos:** suceso que se produce en un sistema como consecuencia de una acción interna o externa al mismo y que este es capaz de detectar y controlar (p .ej. **click**, **focus**, **keydown**, etc.).
- **Efectos:** básicos (**hide**(), **show**() y **toggle**()), personalizados (**animate**(), **delay**(), **finish**(), **stop**(), etc.).
- **Formularios:** obtener o establecer el valor de un control de un formulario.

jQuery es un tema extenso y aquí solo pretendemos presentarlo y realizar algún ejercicio. Invitamos al lector a profundizar un poro más consultando el libro de esta misma colección denominado: Aprender jQuery con 100 ejercicios prácticos. Descargaremos la librería de la siguiente página: http://jquery.com/download/. Podemos descargar la librería con el texto "**Download the uncompressed, development jQuery x.y.z**". "**z.y.z**" será el valor de la versión que dependerá del momento en el que el lector se la descargue.

1. Nos ubicaremos en **Ej100_angular** y crearemos la carpeta **085_jQuery_I** tecleando lo siguiente:

```
C:\Ej100_angular>mkdir 085_jQuery_I
```

   En la carpeta, depositaremos la librería descargada de jQuery y la renombraremos como jquery.js.

2. En la carpeta, crearemos una página HTML llamada Hola_Mundo.html con el siguiente contenido:

```
<html>

<head>

    <title>085 jQuery (I)</title>

    <script type="text/javascript" src="jquery.js"></script>

    <script type="text/javascript">

        $(document).ready(function () {

            $("body").css("color", " blue");

        });

    </script>

</head>

<body>Hola Mundo</body>

</html>
```

   La librería no contiene ninguna ruta ya que esta se halla en la misma carpeta que nuestra página.

3. La función principal de jQuery ($() o función jQuery) utiliza el método ready() para determinar cuándo se ha cargado completamente el DOM y, entonces, cambiar el color del body a azul seleccionando el body y aplicando css. Guarde la página y obsérvela en el navegador.

4. Ahora en el body, añadiremos un párrafo y dos div (id="miId" y class="miClase") para usarlos con css.

5. Ahora, la función jQuery queda de la siguiente manera:

```
<script type="text/javascript">

    $(document).ready(function () {

        $("p").css("color", "blue");

        $("#miId").css("color", "red");

        $(".miClase").css("color", "green");

    });

</script>
```

Y el body contendrá:

```
<body>
    <div id="bloque">
        Hola Mundo
        <p>Esto es un parrafo</p>
        <div id="miId">Esto es un bloque div con identificador</
    div>
        <div class="miClase">Esto es un bloque div con clase</
    div>
    </div>
</body>
```

Guardemos la página y veamos cómo queda en el navegador.

6. Por último, añadimos dos botones para poner/quitar un fondo y un marco usando la clase **toggleClass**.

7. Para ello, modificamos la función jQuery para que contenga lo siguiente:

```
        $(document).ready(function () {
            $("p").css("color", "blue");
            $("#miId").css("color", "red");
            $(".miClase").css("color", "green");
            $("#bt_fondo").click(function () {
                $("#bloque").toggleClass("fondo");
            });
            $("#bt_marco").click(function () {
                $("#bloque").toggleClass("marco");
            });
        });
```

Detrás de la etiqueta </script> y antes del cierre de </head> añadiremos este bloque para el estilo:

```
<style>
  .fondo { background: #58FAF4; }
  .marco { color: blue; border-style: solid; border-width:
  3px; }
</style>
```

Y el body, para que quede así:

```
<body>
    <div id="bloque">
        Hola Mundo
        <p>Esto es un parrafo</p>
        <div id="miId">Esto es un bloque div con identificador</
    div>
        <div class="miClase">Esto es un bloque div con clase</
    div>
        <br>
        <input type="button" value="Fondo" id="bt_fondo" />
        <input type="button" value="Marco" id="bt_marco" />
    </div>
</body>
```

Guardamos la página y la recargamos en el navegador. Probamos pulsando el botón Fondo, pulsando el botón Marco y pulsando ambos botones.

086 jQuery: Parte II

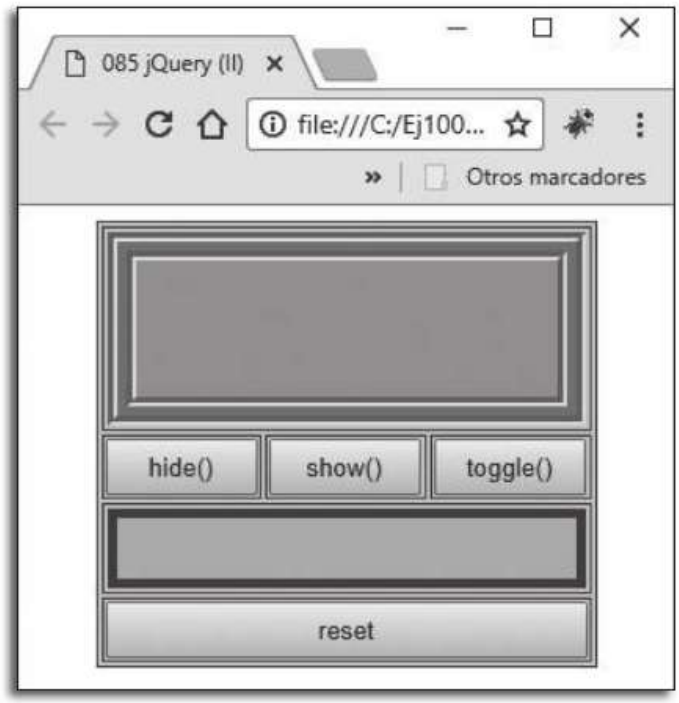
En el siguiente ejercicio, vamos a realizar una prueba con los métodos básicos **hide**(), **show**(), **toggle**(), **fadeIn**() y **fadeOut**() y, posteriormente, añadiremos alguna animación con los efectos **animate**(), **delay**() y **slideToggle**().

1. Nos ubicamos en **Ej100_angular** y creamos la carpeta **085_jQuery_II** tecleando lo siguiente:

```
C:\Ej100_angular>mkdir 085_jQuery_II
```

   En la carpeta, depositaremos la librería descargada de **jQuery** del ejercicio anterior.

2. En la carpeta, crearemos una página **HTML** llamada **086_jQuery_II.html** con el siguiente contenido:



```
<!DOCTYPE html>
<html>
<head>
 <title>085 jQuery (II)</title>
 <script type = "text/javascript" src = "jquery.js"> </script>
 <script type = "text/javascript">
  $(document).ready(function(){
    $("#b_hide").click(function(){ $("#d1").hide(); resaltar("hide"); });
    $("#b_show").click(function(){ $("#d1").show(); resaltar("show"); });
    $("#b_toggle").click(function(){$("#d1").toggle();resaltar("toggle");});
    $("#b_reset").click(function(){location.reload();});
    function resaltar(texto){
      $("#res").html(texto).fadeOut(1000,'linear',function(){
        $(this).html("").fadeIn();
      });
    }
  });
 </script>
 <style>
  div {width: 240px;height:90px;background-color:#00BFFF; margin:1px; padding:1px;
    border-style:groove;border-width:5px; font-size:18px;}
  div#d1  {width: 220px;height:70px;background-color:#40FF00; margin:4px; paddin-
    g:1px; border-style:groove;border-width:5px; font-size:18px;}
  div#msg {width: 240px;height:30px;background-color:#80FF00; margin:1px; paddin-
    g:1px; border-style:solid; border-width:5px; font-size:25px;border-color:#-
    0000FF;text-align:center;}
  .cb     {width: 80px; height:30px;}
  .cbr    {width: 254px; height:30px;}
 </style>
</head>
<body>
 <center>
  <table border=1 bgcolor="#A9F5F2">
    <tr><td colspan=3><div><div id="d1"></div></div></td></tr>
    <tr>
      <td><input type="button" value="hide()" id="b_hide" class="cb"/></td>
      <td><input type="button" value="show()" id="b_show" class="cb"/></td>
      <td><input type="button" value="toggle()" id="b_toggle" class="cb"/></td>
    </tr>
    <tr><td colspan=3><div id="msg"><font color="red"><span id="res"/></font></
    div></td></tr>
    <tr><td colspan=3><input type="button" value="reset" id="b_reset" class="c-
    br"/></td></tr>
  </table>
 </center>
</body>
</html>
```

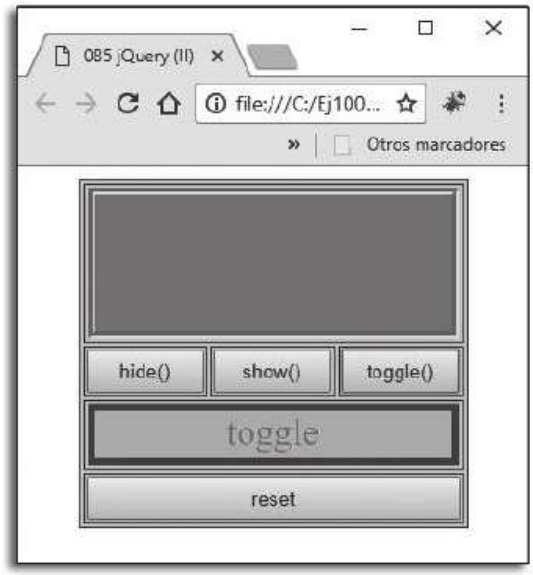
Guarde el archivo y ábralo en el navegador. Pulse sobre el botón **hide**() y observe que en la parte inferior aparece el texto **hide** brevemente y luego se desvanece al mismo tiempo que el bloque superior en verde se oculta.

3. Pulse sobre el botón **show**() y observe que, además de mostrar el texto en la parte inferior, el bloque verde vuelve a aparecer.

Importante

jQuery dispone, entre otras muchas funcionalidades, de una gran cantidad de efectos que dan vistosidad a nuestras páginas y aplicaciones web.

4. Pulse sobre el botón **toggle**() y vea como el bloque aparece y desaparece alternativamente, de forma que si se está mostrando se oculta y si está oculto se visualiza.



5. El botón **reset** vuelve a recargar la página y la deja en su estado inicial.

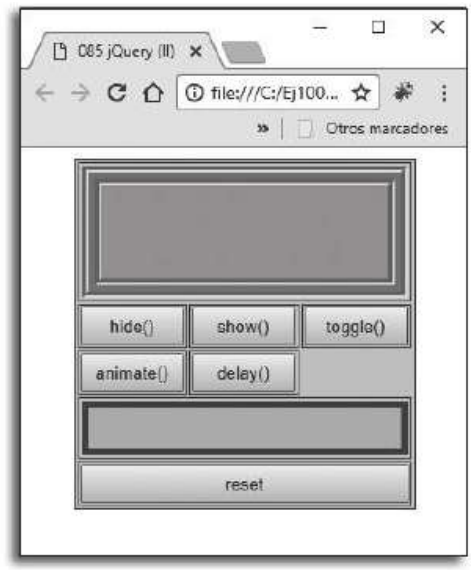
6. A continuación, vamos a añadir una animación y, para ello, añadiremos dos botones más en una fila nueva detrás de los ya existentes con el siguiente código:

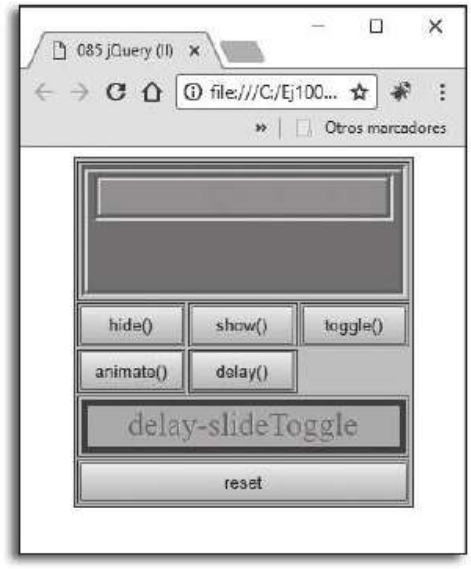
```
… <td><input type="button" value="toggle()" id="b_toggle"
    class="cb" /></td>
</tr>
<tr>
    <td><input type="button" value="animate()" id="b_animate"
    class="cb" /></td>
    <td><input type="button" value="delay()" id="b_delay"
    class="cb" /></td>
</tr>
```

También añadimos el siguiente código justo después de **$(document).ready(function(){**:

```
var vel1 = 4000; var vel2 = 100; var vd1 = false;
$("#b_animate").click(function () {
  resaltar("animate");
  vd1 = !vd1;
  $("#d1").animate({height: "toggle",width: "toggle"}, vel1);
  $("#d2").animate({height: "toggle",width: "toggle"}, vel1);
});
$("#b_delay").click(function () {
  resaltar("delay-slideToggle");
  if (!vd1) {
 $("#d1").delay(vel2).slideToggle(function (){ $("#d2").de-
  lay(vel2).slideToggle(); });
  } else {
 $("#d2").delay(vel2).slideToggle(function () { $("#d1").de-
  lay(vel2).slideToggle();});
  }
});
```

Guarde la página y compruebe su funcionamiento en el navegador. Pulse sobre el botón **animate**() para ver que el bloque superior se va haciendo pequeño progresivamente hasta desaparecer y, a continuación, vuelva a pulsarlo para ver como aparece de nuevo lentamente. Por último, pulse sobre **delay**() para ver que desaparece verticalmente con un cierto retardo.

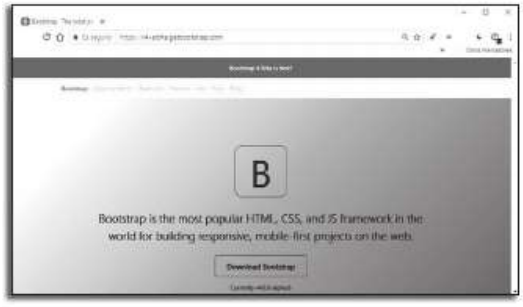


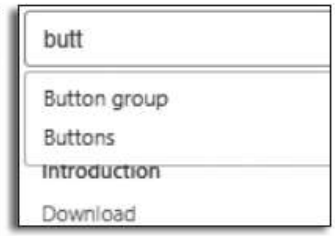


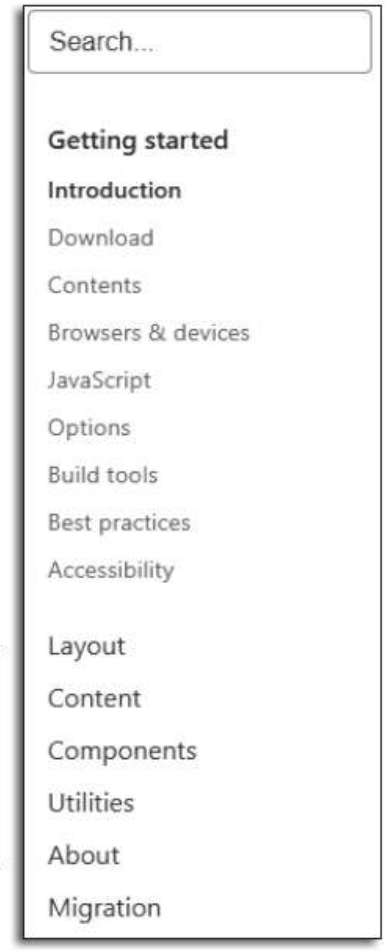
087 Bootstrap: Introducción

Bootstrap es un framework que facilita el desarrollo web facilitando la adaptación de las pantallas de nuestra aplicación a los diferentes tamaños de dispositivos. Combina **HTML**, **CSS** y **JavaScript** y su curva de aprendizaje es muy asequible. Existe mucha información en Internet, aunque sugerimos visitar el siguiente link https://v4-alpha.getbootstrap.com/, ya que nos permite realizar diversas acciones como, por ejemplo, descargar la librería para trabajar en local, consultar la documentación del framework, ver muchos ejemplos que podemos copiar y adaptar a nuestros desarrollos, etc.

Al acceder a esta página, hallamos un menú desde el que podemos acceder a diversas opciones entre las que destacamos **Documentation**. Si accedemos a **Documentation**, veremos una introducción explicando qué es **Bootstrap** y unas indicaciones sobre cómo podemos empezar a utilizarlo rápidamente, entre otras opciones. Al margen de esto, en la parte de la derecha disponemos de una caja de texto **'Search'**, que nos permitirá localizar fácilmente el recurso que buscamos según vamos tecleando su nombre. Bajo dicha caja de texto, disponemos de una serie de links organizados en forma de columnas que nos permitirá acceder a los temas de una forma organizada en función de lo que necesitemos. Sin lugar a dudas, todas las opciones que nos ofrece el menú de la página de **Bootstrap** son interesantes, pero nosotros vamos a destacar las siguientes:

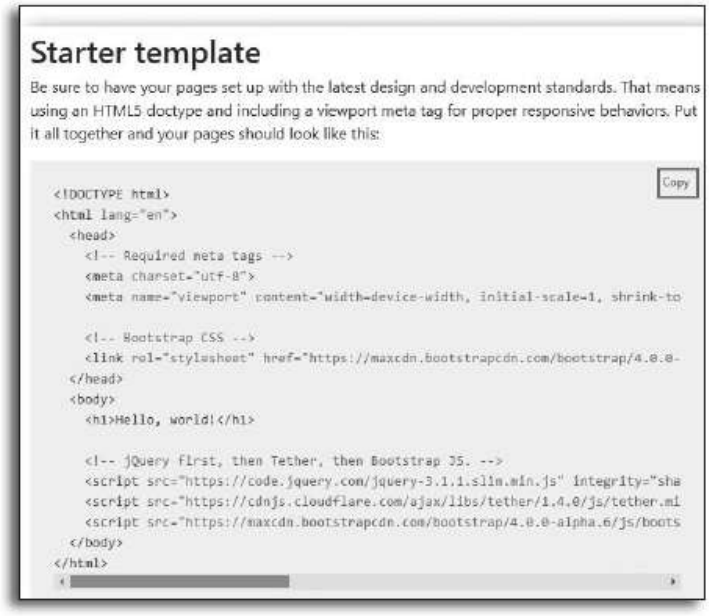






Opción	Comentario
Layout	Permite definir estilos globales y estructurar nuestras páginas. (**Overview**, **Grid**, **Media object** y **Responsive utilities**)
Contents	Uso de contenedores y diseño basado en anchos mínimos para que nuestra aplicación se ajuste según cambie el tamaño de la vista. (**Reboot**, **Typography**, **Code**, **Images**, **Tables** y **Figures**)
Components	Ofrece un conjunto de componentes que podemos copiar y pegar en nuestro código para que, con unos pequeños ajustes, puedan utilizarse fácilmente. (**Alerts**, **Badge**, **Breadcrumb**, **Buttons**, **Button group**, **Card**, **Carousel**, **Collapse**, **Dropdowns**, **Forms**, **Input group**, **Jumbotron**, **List group**, **Modal**, **Navs**, **Navbar**, **Pagination**, **Popovers**, **Progress**, **Scrollspy y Tooltips**)
Utilities	Conjunto de clases que pretenden facilitar tareas frecuentes como el uso de colores, alineaciones, posicionamiento, espaciados, bordes, etc. (**Borders**, **Clearfix**, **Close icon**, **Colors**, **Flexbox**, **Display property**, **Image replacement**, **Invisible content**, **Position**, **Responsive helpers**, **Screenreaders**, **Sizing**, **Spacing**, **Typography** y **Vertical align**)

A continuación, vamos a crear una pequeña aplicación sobre la que copiaremos el template que nos encontramos al acceder a **Documentation**, concretamente en el apartado **Starter template**. Si pulsamos sobre **Copy**, copiaremos todo el texto que se propone como template.



A continuación, vamos a crear una aplicación en la que mostraremos cómo empezar a usar **Bootstrap** rápidamente y de forma sencilla.

1. En primer lugar, nos ubicaremos en **Ej100_angular** y teclearemos **ng new btIntro** para crear nuestro nuevo proyecto.

2. Seguidamente, renombraremos el proyecto escribiendo lo siguiente:

Importante

Utilice Bootstrap para facilitar muchas de las tareas rutinarias que un desarrollo puede comportar y para aprovechar las mejoras que se incorporan periódicamente a este framework.

```
C:\Ej100_angular>rename btIntro 087_btIntro
```

A continuación, nos ubicaremos en **087_btIntro** y arrancaremos la aplicación para comprobar cómo funciona en nuestro navegador. Para ello, tecleamos lo siguiente:

```
C:\Ej100_angular>cd 087_btIntro

C:\Ej100_angular\087_btIntro>ng serve
```

Ahora, abrimos nuestro proyecto y editamos el archivo **src/app/app.component.html**. En el mismo pegaremos el contenido de la plantilla que nos ofrece el apartado **Starter template** comentado antes.

3. Insertaremos un **div** para encerrar nuestro "**<h1>Hello, world!</h1>**" y que quede de la siguiente manera:

```
<div class="container-fluid">

  <h1>Hello, world!</h1>

</div>
```

La clase **container-fluid** permite que el contenedor ocupe todo el ancho del visor. Guardamos el archivo y acudimos al navegador a ver el aspecto de la página.

4. Seguidamente, vamos a **Utilities->Color** de la página de Bootstrap y copiamos la siguiente sentencia para pegarla a continuación de "**<h1>Hello, world!</h1>**":

```
<p class="text-success">Duis mollis, est non commodo luc-
   tus, nisi erat porttitor ligula.</p>
```

Modificamos el texto "**Duis ...lígula**" para que ponga un texto más personalizado como por ejemplo **"Mi texto de pruebas"**, guardamos el archivo y vemos el resultado en el navegador. La clase **text-success** es la encargada de la apariencia del texto.

5. A continuación, creamos la carpeta **images** bajo **src (src/images)** y copiamos una imagen cualquiera que podemos llamar **img1.jpg**.

6. Vamos a la página de **Bootstrap** y en **Utilities->Borders** localizamos **Border-radius** para copiar lo siguiente y pegarlo a continuación del párrafo insertado anteriormente:

```
<img src="..." alt="..." class="rounded-circle">
```

Luego, adaptamos lo copiado para que quede de la siguiente manera:

```
<img src="images/img1.jpg" alt="Mi imagen" class="roun-
    ded-circle img-fluid w-50">
```

La clase **rounded-circle** produce un redondeo sobre la imagen, **img-fluid** crea una escala de imagen acorde a cada dispositivo y **w-50** indica que la imagen ha de ocupar el **50%** del tamaño de la ventana. Pruebe de cambiar el tamaño de la ventana y observe que la imagen se redimensiona para ocupar el **50%** del ancho de la misma.

088 Bootstrap: Layout. El sistema Grid

En el apartado **Layout** de la documentación Bootstrap, encontrará todo lo relacionado con las herramientas de de diseño que nos ofrece el framework. Entre estas, cabe destacar el Sistema "Grid" de Bootstrap.

El **sistema de "Grid"** de Bootstrap nos permitirá diseñar y alinear contenidos, mediante contenedores, filas y columnas. Vamos a ver todo su potencial a través de varios ejercicios.

Importante

Mediante el sistema Grid de 12 Columnas de Bootstrap podremos estructurar el contenido Web de nuestras aplicaciones. Una de sus principales ventajas es la facilidad con la que nos permite la creación de distintos diseños para distintos tipos de pantalla (móvil, ordenador, etc.).

1. Nos ubicaremos en **ej100_angular** y crearemos el proyecto tecleando **ng new btLayout**. Luego, lo renombraremos para que haga referencia a nuestro número de ejercicio ejecutando **rename btLayout 088_btLayout** y, finalmente, pondremos en marcha la aplicación ejecutando **ng serve** desde dentro la carpeta 088_btLayout. A continuación abriremos el proyecto con **Atom.**

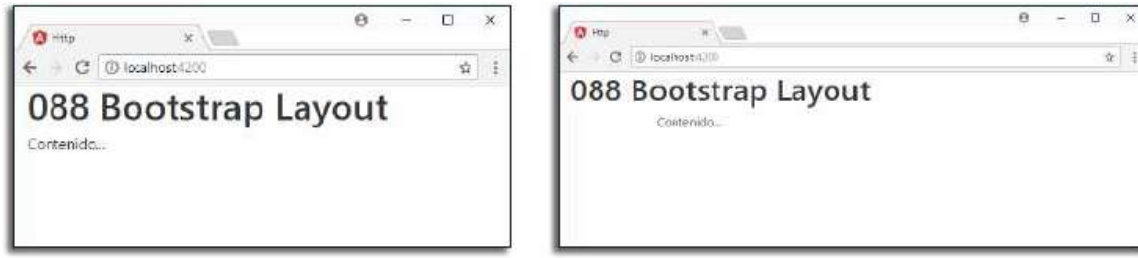
2. Seguidamente añadiremos los links de **Bootstrap** (https://v4-alpha.getbootstrap.com) en index.html para poder hacer uso de sus hojas de estilo. Consulte "087 Bootstrap ..." para más información.

3. Ahora editaremos **src/app/app.component.html** y añadiremos el siguiente código:

```
<div class="container-fluid">
    <h1>088 Bootstrap Layout</h1>
</div>
<div class="container">
    <p>Contenido...</p>
</div>
```

Con las clases de Bootstrap **.container-fluid** y **.container** estamos definiendo un **contenedor** de tamaño variable y otro de tamaño fijo. Observe el resultado en el navegador. Si modifica el tamaño del navegador podrá ob-

servar cómo el primer contenedor se adapta a ello mientras que el segundo mantiene el tamaño fijo.



4. Dentro los contenedores, definiremos nuestras **filas y columnas** mediante las clases **.row** y **.col-sm**. El sistema "Grid" de Bootstrap nos permite definir un máximo de **12 columnas de tamaño 1.** A partir de aquí, podemos realizar las combinaciones que queramos: una fila con una columna de tamaño 2, otra de tamaño 4 y otra de tamaño 6, una fila con una columna de tamaño 9 y otra de tamaño 3, etc.

   Vamos a crear una par de filas con distintas configuraciones de columnas. Para ello, sustituye el segundo contenedor que habíamos creado en **src/app/app.component.html,** por el siguiente:

```
<div class="container">
    <div class="row">
        <div class="col-sm-2">col-sm-2</div>
        <div class="col-sm-4">col-sm-4</div>
        <div class="col-sm-6">col-sm-6</div>
    </div>
    <div class="row">
        <div class="col-sm-9">col-sm-9</div>
        <div class="col-sm-3">col-sm-3</div>
    </div>
</div>
```

5. Para facilitar la comprensión de este ejercicio, crearemos un borde para cada una de las "celdas" creadas a partir de la definición de filas y columnas. Para ello, vamos a añadir la siguiente clase en nuestra hoja de estilo **src/app/app.component.css**:

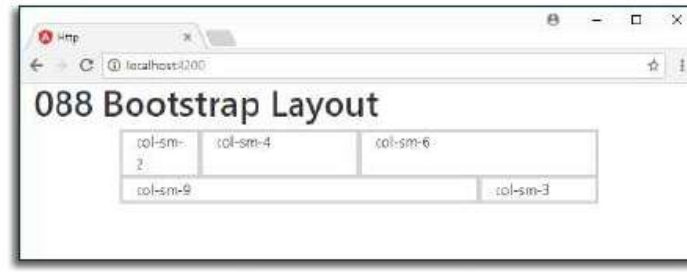
```
.celda {
    border: #cdcdcd medium solid;
}
```

   Finalmente aplicaremos la clase a cada una de las columnas que hemos implementado antes en **src/app/app.component.html:**

```
<div class="container">
    <div class="row">
      <div class="col-sm-2 celda">col-sm-2</div>
…
```

6. Observe en el navegador el resultado de este código.

7. Con Bootrsap tenemos 4 tipos de columnas. Cada una de ellas se utiliza para mostrar en distintos dispositivos con pantallas de resolución distinta: **col-xs** (para pantallas de resolución menor a 768px), **col-sm** (para pantallas de resolución mayor o igual a 768px), **col-md** (para pantallas de resolución mayor o igual a 992px) y **col-lg** (para pantallas de resolución mayor o igual a 1200px). Estos tipos de columna se pueden usar a la vez, cosa que nos permitirá hacer diseños para todo tipo de pantallas al mismo tiempo.



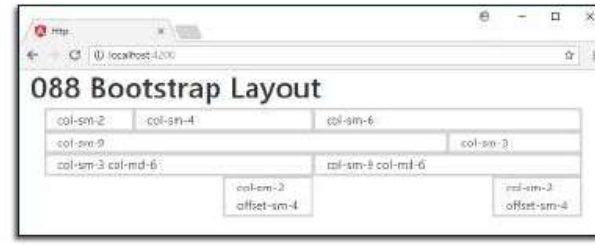
Veamos un ejemplo. Añada la siguiente nueva fila en **src/app/app.component.html**:

```
<div class="container">
..
    <div class="row">
        <div class="col-sm-4 col-md-6 celda">col-sm-3 col-md-6</div>
        <div class="col-sm-8 col-md-6 celda">col-sm-9 col-md-6</div>
    </div>
</div>
```

8. Observe en el navegador el resultado. Fíjese cómo, en la nueva fila, se aplica una u otra configuración según el tamaño del navegador.

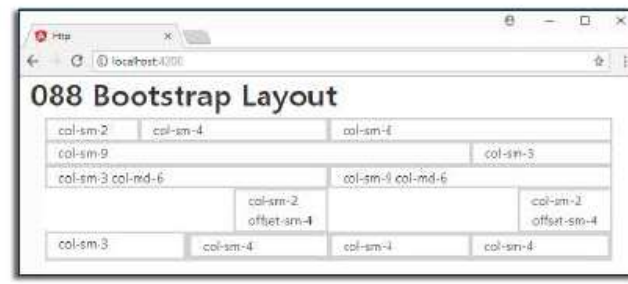
9. De forma parecida, también tenemos otros tipos de clases que se aplican según el tamaño de pantalla. Estas son las de ocultar o mostrar columnas: **.hidden-xs**, **.hidden-sm**, etc., **visible-xs**, **.visible-xm,** etc.

10. Usando la clase **.offset-xx-tamaño,** podremos **desplazar columnas**. Cuando desplazamos una columna, lo que hacemos realmente es crear una columna vacía a la izquierda de la columna desplazada. Veamos un ejemplo. Añada la siguiente nueva fila en **src/app/app.component.html** y observe el resultado en el navegador:




```
    <div class="row">
        <div class="col-sm-2 offset-sm-4 celda">col-sm-2
offset-sm-4</div>
        <div class="col-sm-2 offset-sm-4 celda">col-sm-2
offset-sm-4</div>
    </div>
```

11. Para terminar, vamos a ver cómo podemos **anidar columnas**. Para ello, simplemente crearemos nuevas filas (rows) dentro de columnas. Añada la siguiente nueva fila en **src/app/app.component.html** y observe el resultado en el navegador:




```
    <div class="row">
        <div class="col-sm-3 celda">col-sm-3</div>
        <div class="col-sm-9 celda">
            <div class="row">
                <div class="col-sm-4 celda">col-sm-4</div>
                <div class="col-sm-4 celda">col-sm-4</div>
                <div class="col-sm-4 celda">col-sm-4</div>
            </div>
        </div>
    </div>
```

089 Bootstrap: Tables

Dentro del apartado **Content**, podemos encontrar un tema dedicado a las tablas el cual nos permitirá añadir un aspecto y funcionalidad a las mismas simplemente añadiendo alguna de las clases que se ofrecen. Algunas clases afectan a la tabla, otras a los encabezados, otras a las filas, etc.

Importante

Cree tablas aprovechando la plantilla que ofrece Bootstrap y preocúpese solo por rellenar su contenido.

A continuación, fabricaremos una aplicación para probar algunos ejemplos y ver su resultado en el navegador.

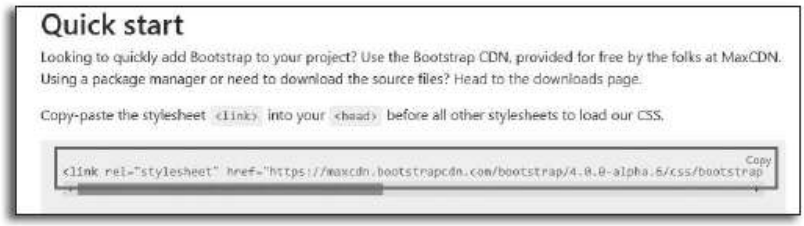
1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos el proyecto **btTables** tecleando **ng new btTables**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename btTables 089_btTables
```

   Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso usamos **Atom**) y abrimos el archivo **app.component.ts** para modificar el título (**title**) y poner, por ejemplo, "**089 Bootstrap Tables**".

3. Después accedemos a la página de **Bootstrap** y copiamos el contenido que se halla en **Documentation->Quick start,** que empieza por:

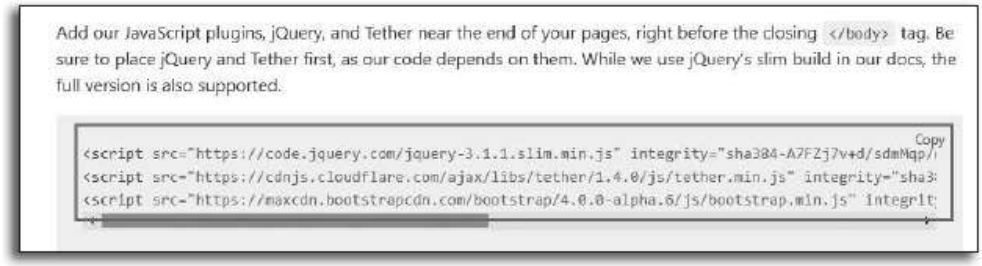



```
<link  rel="stylesheet"  href="https://maxcdn.bootstrapcdn.
com/bootstrap/4.0.0- ...
```

Dicho contenido lo pegamos en la primera línea del archivo **app.component.html** dejando que el resto del archivo contenga lo siguiente:

```
<div>
    <h1>
        {{title}}
    </h1>
</div>
```

Luego, en la misma página de **Bootstrap**, veremos que tras la sentencia anterior, se hallan 3 líneas que hacen referencia a **jquery**, **tether** y **bootstrap,** que empiezan por:




```
<script src="https://code.jquery.com/jquery...
```

Pulsamos el botón **Copy** y las pegamos al final de nuestro archivo **app.component.html**.

4. A continuación, nos ubicaremos en **089_btTables** y arrancaremos la aplicación con **ng serve** para comprobar cómo funciona en nuestro navegador. Para ello, teclearemos lo siguiente:

```
C:\Ej100_angular>cd 089_btTables

C:\Ej100_angular\088_btTables>ng serve
```

A partir de aquí, en la página de **Bootstrap**, situados en **Documentation**, teclearemos en la caja de texto **Search**, la palabra *tables* y seleccionaremos de la lista resultante precisamente **Tables**.

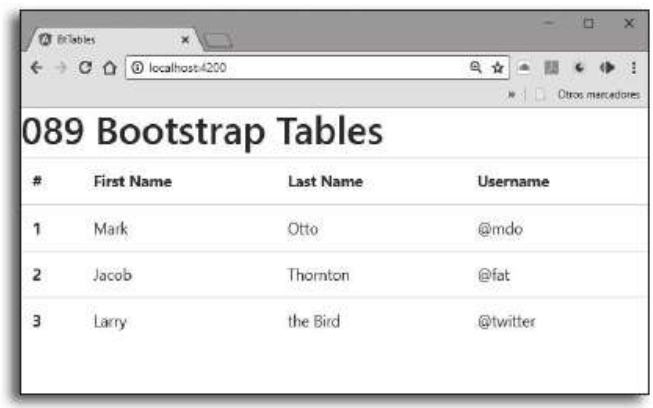
5. Además de revisar el contenido de esta página, podemos copiar el primer ejemplo pulsando sobre el botón **Copy** que hay debajo del primer ejemplo, y pegarlo en **app.component.html,** justo debajo de nuestro **</h1>** que cierra el título (**title**). Lo primero que destacamos es que al tag **table**, le asignamos la clase "**table**".

app.component.html

```
<link rel="stylesheet" href

<h1>
  {{title}}
</h1>
<table class="table">
  <thead>
    <tr>
      <th>#</th>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Username</th>
```

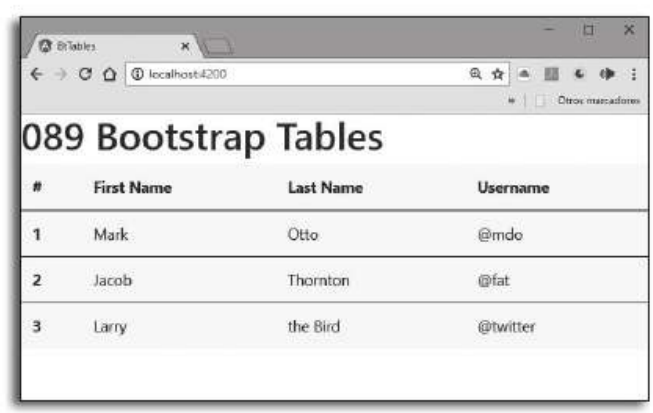
6. Si consultamos el navegador donde está corriendo la aplicación, veremos el resultado de esta acción.

7. A continuación, veremos el siguiente ejemplo propuesto por **Bootstrap** que simplemente muestra en vídeo inverso el contenido de la tabla añadiendo la clase **table-inverse** a la clase ya existente y dejando el tag de table con el siguiente contenido:




```
<table class="table table-inverse">
```

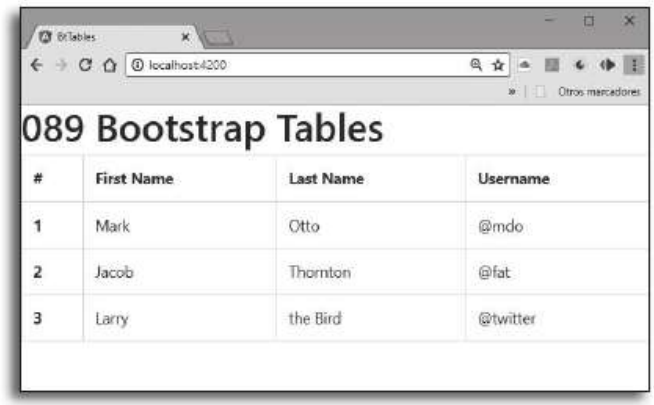
8. Observe el resultado en el navegador.



9. Sustituyamos a continuación la clase **table-inverse** por **table-bordered** para que quede de la siguiente manera:

```
<table class="table table-bordered">
```

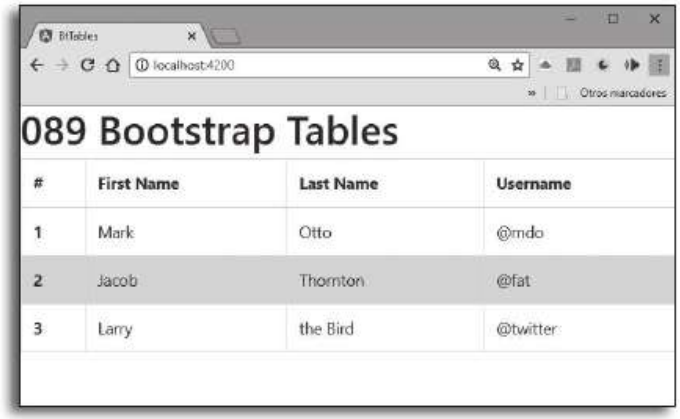
Observe el resultado en el navegador.



10. Añada la clase **table-hover** para que la fila cambie de color al pasar el puntero del ratón sobre la misma y deje el siguiente contenido:

```
<table class="table table-bordered table-hover">
```

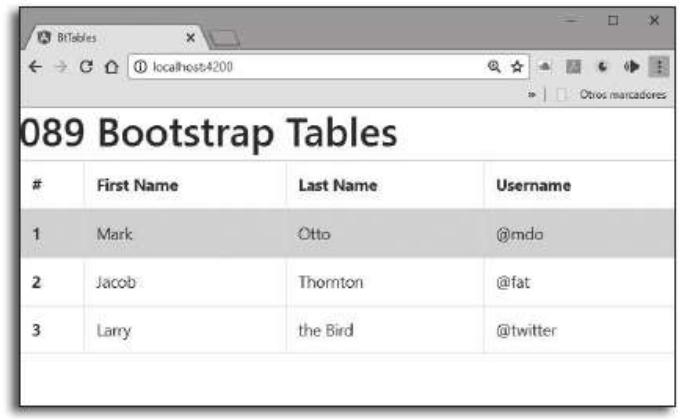
Observe el resultado en el navegador.



11. Por último, pruebe las "**Contextual clases**" añadiendo a la primera fila del **tbody** la clase **table-success,** para que quede de la siguiente manera:

```
<tbody>
  <tr class="table-success">
```

Compruebe el resultado en el navegador.



12. Pruebe con el resto de clases ofrecidas por la página para experimentar sobre las diferentes posibilidades que ofrece este tema.

090 Bootstrap: Alerts

En este ejercicio, presentaremos las alertas que podemos utilizar con **Bootstrap** de forma sencilla y practicaremos con algunas de las mismas. Al igual que comentamos en ejercicios anteriores y que se repetirá en los ejercicios relativos a **Bootstrap**, nos hemos de conectar a la página https://v4-alpha.getbootstrap.com/ y seleccionar la opción de menú **Documentation**. En dicha página, teclearemos en la caja de texto **Search,** el término **Alerts,** para seleccionar la opción propuesta y trasladarnos a la página en la que se explica su cometido y se ofrecen diferentes recursos para su uso.

Vamos a crear un proyecto donde poder probar este tema. Para ello, haremos lo siguiente:

1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos el proyecto **btAlerts** tecleando **ng new btAlerts**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename btAlerts 090_btAlerts
```

   Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso usamos **Atom**) y abrimos el archivo **app.component.ts** para modificar el título (**title**) y poner, por ejemplo, "**090 Bootstrap Alerts**".

3. De la misma forma que explicamos en ejercicios anteriores, modificaremos nuestro archivo **app.component.html** para insertar al principio la línea que hace referencia a la hoja de estilos de **Bootstrap**:

```
<link  rel="stylesheet"  href="https://maxcdn.bootstrapcdn.
com/bootstrap/4.0.0- ...
```

4. Tras esta línea de referencia, comprobaremos que el resto de la página contiene lo siguiente:

Importante

Use las alertas cuando quiera mostrar una información temporalmente o un aviso.

```
<div>
    <h1>
        {{title}}
    </h1>
</div>
```

Y al final del mismo archivo, comprobaremos las referencias a **jquery**, **tether** y **bootstrap**:

```
<script src="https://code.jquery.com/jquery ...
```

Arrancamos la aplicación con **ng serve** y observamos cómo se ve nuestra aplicación en el navegador.

5. A continuación, acudimos a la página de **Bootstrap** y copiamos el primer ejemplo que contiene lo siguiente:

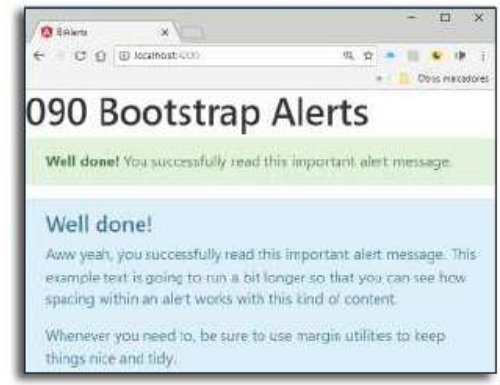
```
<div class="alert alert-success" role="alert">
  <strong>Well done!</strong> You successfully read
   this important alert message.
</div>
```

Lo pegamos debajo del **</h1>** que cierra nuestro título, guardamos el archivo y vemos el resultado en el navegador.

6. Seguidamente, probamos el ejemplo que muestra una cabecera y un párrafo. En la página de **Bootstrap**, en **Aditional content** veremos lo siguiente:

```
<div class="alert alert-success" role="alert">
  <h4 class="alert-heading">Well done!</h4>
   <p>Aww yeah, you successfully read this important alert
    message. This example text is going to run a bit longer
    so that you
    can see how spacing within an alert works with this kind
    of content.</p>
  <p class="mb-0">Whenever you need to, be sure to use mar-
    gin utilities to keep things nice and tidy.</p>
</div>
```

Lo copiamos (**Copy**) y lo pegamos a continuación de la alerta anterior en nuestra página **HTML** y modificamos **alert-success** por **alert-info** y vemos cómo queda en el navegador.

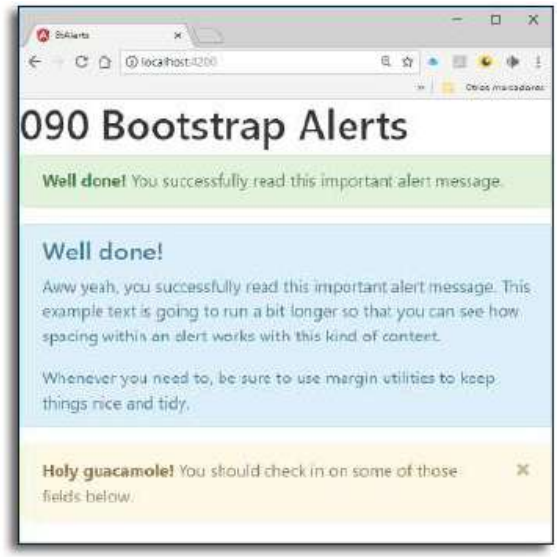


7. Si seguimos avanzando en la página, encontramos el apartado **Dismissing** donde veremos que la alerta puede cerrarse. Podemos copiar (**Copy**) el ejemplo y pegarlo en nuestra página **HTML** a continuación de la alerta anterior:

```
<div class="alert alert-warning alert-dismissible fade show"
    role="alert">
   <button type="button" class="close" data-dismiss="alert"
    aria-label="Close">
     <span aria-hidden="true">&times;</span>
  </button>
   <strong>Holy guacamole!</strong> You should check in on
    some of those fields below.
</div>
```

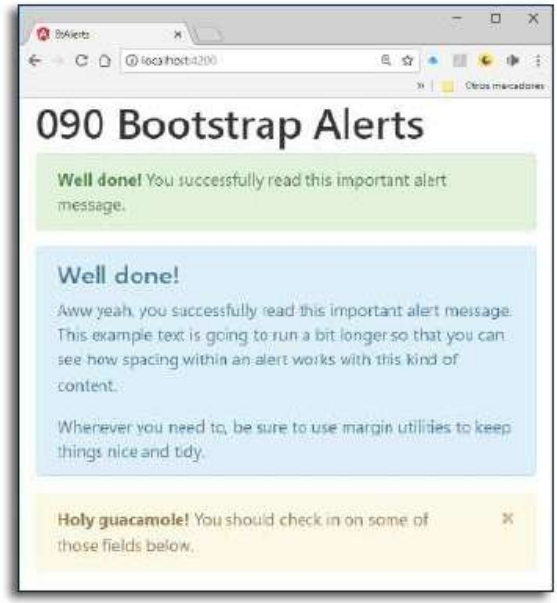
Guardamos el archivo y vemos cómo queda en el navegador. Si pulsa sobre la cruz que aparece en la esquina superior derecha verá cómo se cierra la alerta.

8. Ahora, para que se vea mejor y quede mejor organizado, encerraremos nuestro título y nuestras alertas en un **DIV** que use la clase "**container**":




```
<div class="container">
  <h1>
...
</div>
```

Veamos cómo se muestra en el navegador.



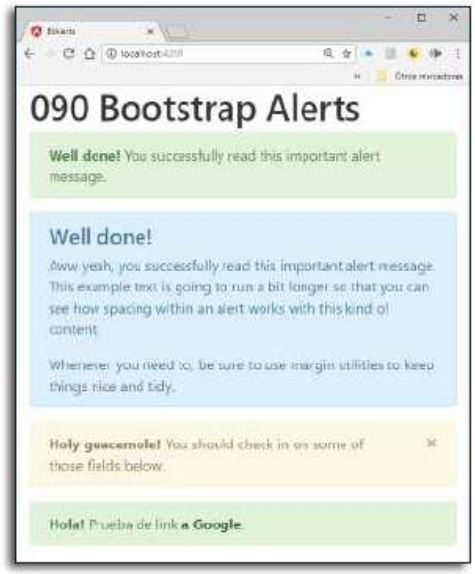
9. Por último, podemos añadir una alerta con un link localizando en la página el apartado **Link color** y copiando el primer ejemplo que posee el siguiente contenido:

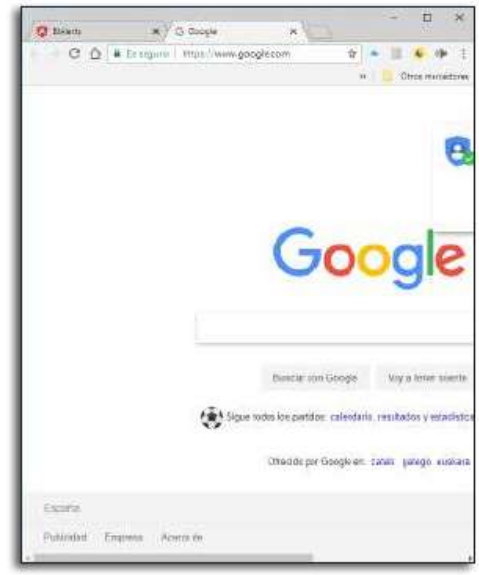
```
<div class="alert alert-success" role="alert">
   <strong>Well done!</strong> You successfully read <a
   href="#" class="alert-link">this important alert mes-
   sage</a>.
</div>
```

Lo pegamos detrás de la última alerta y modificamos el contenido para que contenga lo siguiente:

```
<div class="alert alert-success" role="alert">
   <strong>Hola!</strong> Prueba de link <a href="http://
   www.google.com" target="_blank" class="alert-link">a
   Google</a>.
</div>
```

Guardamos el archivo y acudimos al navegador para ver la nueva alerta (06) y observaremos que al pulsar sobre el link, se crea una nueva pestaña en la que se muestra la página del buscador de Google.





091 Bootstrap: Buttons y ButtonGroups

En este capítulo, abordaremos los detalles relacionados con los Buttons y con los ButtonGroups que, aunque tengan un nombre parecido, realmente son controles diferentes.

Importante

Use las clases asociadas a los botones para darle un aspecto más agradable a los mismos y también para cambiar la visualización de tags como <a>, Checkbox, etc.

Buttons

Los estilos aportados para los botones nos permiten personalizar el aspecto de diferentes elementos en nuestros formularios y diálogos. Además de cambiar la forma de visualizar los botones en si mismos, podemos cambiar el aspecto de elementos como, por ejemplo, los de tipo **<a>**, **<input>**, **Checkbox** y **Radio buttons**.

Podemos encontrar información relativa a los **buttons** en la página de **Bootstrap** https://v4-alpha.getbootstrap.com/, tecleando la palabra **Buttons** en la caja de texto **Search,** existente en **Documentation**.

Como siempre, en esta página encontraremos toda la información asociada a este tipo de elementos además de ejemplos y plantillas que nos agilizaran los desarrollos.

Veremos que, por defecto, vienen una serie de estilos predefinidos que son: **Primary**, **Secondary**, **Success**, **Info**, **Warning**, **Danger** y **Link**.

Si seguimos curioseando la página, veremos aspectos como:

- **Outline buttons**: elimina las imágenes y colores de fondo que puedan traer los botones.
- **Sizes**: Para crear botones con diferentes tamaños (**large**, **small**, **block**).
- **State**: Para crear botones que estén habilitados o deshabilitados o que funcionen como un interruptor ON/OFF (**Active**, **disabled**, **toggle**).
- Etc.

Ejercicio para Buttons

En el siguiente ejercicio, crearemos una aplicación en la que simplemente mostraremos unos cuantos botones con diferentes estilos a los que no daremos funciona-

lidad, excepto al que lleva asociado un link, que lo usaremos para navegar a otra página.

1. En primer lugar, nos ubicaremos en **Ej100_angular** y crearemos el proyecto **btButtonsB** tecleando **ng new btButtonsB**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename btButtonsB 091_btButtonsB
```

   Ahora, abrimos nuestro proyecto con nuestro editor (en nuestro caso usamos **Atom**) y abrimos el archivo **app.component.ts** para modificar el título (**title**) y poner por ejemplo "**091 Bootstrap ButtonsB**".

3. De la misma forma que explicamos en ejercicios anteriores, modificaremos nuestro archivo **app.component.html** para insertar al principio la línea que hace referencia a la hoja de estilos de **Bootstrap**:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
com/bootstrap/4.0.0- ...
```

   El resto del archivo contendrá lo siguiente:

```
<h1>
  {{title}}
</h1>
```

   Y al final del mismo archivo, las referencias a **jquery**, **tether** y **bootstrap**:

```
<script src="https://code.jquery.com/jquery ...
```

   Añadiremos un elemento **DIV** con la clase "**container**" alrededor de nuestro **<h1>** que muestra el **título** para que quede de la siguiente manera:

```
<div class="container">
  <h1>
    {{title}}
  </h1>
</div>
```

Arrancamos nuestra aplicación (con **ng serve** desde **C:\Ej100_angular>091_btButtonsB**) y observamos cómo se ve nuestra aplicación en el navegador.

4. Seguidamente, desde la página de **Bootstrap** dedicada a los botones, copiaremos el primer ejemplo correspondiente al estilo '**Primary',** que posee el siguiente contenido:

```
<button type="button" class="btn btn-primary">Primary</
button>
```

Lo pegamos a continuación de **</h1>** asociado a **title**. Para organizar mejor los bloques, encerraremos este botón en un **DIV** al que le añadimos la clase "**row** " para indicar que se trata de una fila y **m-2** para que añada un margen alrededor del botón (por favor, consulte el apartado **Utilities->Spacing** para más detalles). Queda de la siguiente manera:

```
<div class="row m-2">
  <button type="button" class="btn btn-success">Primary</
   button>
</div>
```

Guardamos el archivo y lo visualizamos en el navegador.

5. Luego, localizamos en la página de **Bootstrap** el apartado **Button tags** y copiamos el ejemplo correspondiente al tag **<a>** que usa las clases "**btn ...**" y que concretamente contiene lo siguiente:

```
<a class="btn btn-primary" href="#" role="button">Link</a>
```

Volvemos a nuestra página **HTML** y añadimos otro bloque **DIV** idéntico al del ejemplo anterior para colocar lo que hemos copiado, pero modificaremos la propiedad **href** para que apunte a "**http://www.google.com**" quedando de la siguiente manera:

```
<div class="row m-2">
   <a class="btn btn-primary" href="http://ww.google.com"
  role="button">Link</a>
 </div>
```

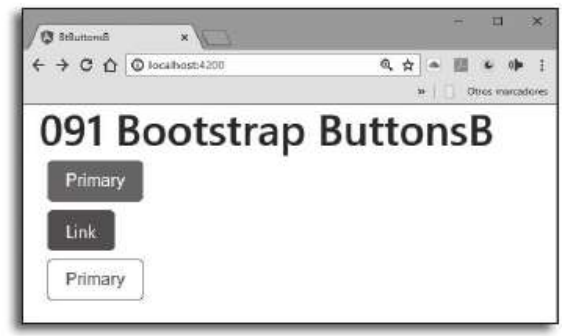
Guarde el archivo y observe cómo se ve en el navegador. Haga click sobre el botón **Link** y compruebe que navega hasta la página de **Google**.

6. Seguidamente, probaremos un ejemplo del apartado **Outline buttons**. Localícelo en la página y copie el ejemplo que posee el siguiente contenido:

```
<button type="button" class="btn btn-outline-primary">Pri-
mary</button>
```

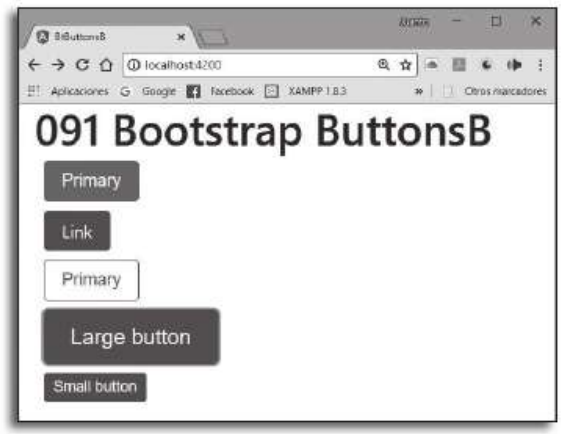
Inserte un **DIV** como los comentados anteriormente y dentro del mismo, pegue el contenido copiado. Guarde y compruebe el resultado en el navegador.

7. Por último, haremos una prueba con un botón grande y otro pequeño. Para ello, localice el apartado **Sizes** en la página de **Bootstrap** y copie los primeros ejemplos de **Large button** y **Small button**. Luego, cree un **DIV** para cada uno de ellos en nuestro archivo **HTML** de forma que al final contenga lo siguiente:



```
<div class="row m-2">
  <button type="button" class="btn btn-primary btn-lg">Lar-
    ge button</button>
</div>
<div class="row m-2">
  <button type="button" class="btn btn-primary btn-sm">S-
    mall button</button>
</div>
```

Guarde el archivo y compruebe cómo queda en el navegador.



8. Le animamos a que pruebe el resto de ejemplos de la página para comprobar el resultado ya que, experimentar, es una muy buena forma de aprender.

ButtonGroup

Los **Button group** son elementos que permiten la creación de grupos de botones en una sola línea (o varias) y que pueden modificar el aspecto también de **checkbox** y **radio-buttons**. Podemos encontrar la información relativa a los **button group** en la página de **Bootstrap** https://v4-alpha.getbootstrap.com/, tecleando en la caja de texto **Search** existente en **Documentation**, el texto **button group**. Como de costumbre, en la página hallaremos ejemplos, plantillas y, además, una explicación sobre posibles aspectos a tener en cuenta como, por ejemplo:

Importante

Use las clases asociadas a **button group** no solo para botones sino también para **checkbox** y **radio buttons** si lo desea.

- **Buttons toolbar**: permite combinar grupos de botones en una barra de herramientas de botones
- **Sizing**: permite aplicar un tamaño estándar (**btn-group-lg**, **btn-group-sm**) a un grupo de botones en lugar de ir uno a uno.

- **Nesting**: es posible incluir grupos de botones anidados.
- **Vertical variation**: permite mostrar los botones de un grupo verticalmente.

En el siguiente ejercicio veremos algunas de las funcionalidades expuestas en la página para que el lector tenga una pequeña experiencia con el tema. En primer lugar, crearemos un sencillo ejemplo de 3 **radio buttons** que devuelven 3 valores diferentes y que no usan de momento, ninguna clase especial de los **button group**. Posteriormente copiaremos un bloque de código de la página de **Bootstrap** y la adaptaremos para hacer la misma acción que la comentada en el ejemplo clásico.

1. Nos ubicamos en **Ej100_angular** y creamos el proyecto **btBtGroupBG** tecleando **ng new btBtGroupBG**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename btBtGroupBG 091_btBtGroupBG
```

   Abrimos nuestro proyecto (por ejemplo con **Atom**) y abrimos el archivo **app.component.ts** para modificar el título (**title**) y poner por ejemplo "**091 Bootstrap Button Group**".

3. Igual que en ejercicios anteriores, modificaremos el archivo **app.component.html** para insertar al principio la línea que hace referencia a la hoja de estilos de **Bootstrap**:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
   bootstrap/4.0.0- ...
```

   El resto del archivo contendrá lo siguiente:

```
<h1>
  {{title}}
</h1>
```

   Y al final del mismo archivo, las referencias a **jquery**, **tether** y **bootstrap**:

```
<script src="https://code.jquery.com/jquery ...
```

   Añadiremos un elemento DIV con la clase "**container**" alrededor de nuestro **título** para que quede así:

```
<div class="container">
  <h1>{{title}}</h1>
</div>
```

Arrancamos la aplicación (con **ng serve** desde **C:\Ej100_angular>091_btBtGroupBG**) y observamos cómo se ve nuestra aplicación en el navegador.

4. A continuación, en nuestro archivo **app.component.html** añadiremos después de **</h1>** lo siguiente:

```
<div class="container">

  <label><input type="radio" name="grupo1" value="1"
   (click)="onClick($event)">uno</label><br>

  <label><input type="radio" name="grupo1" value="2"
   (click)="onClick($event)">dos</label><br>

  <label><input type="radio" name="grupo1" value="3"
   (click)="onClick($event)">tres</label>

</div>

Opción seleccionada: {{ res }}
```

5. Hemos incluido 3 radio **buttons** con un valor para cada uno y con una llamada al método **onClick** cuando hacemos un click. También visualizamos la variable **res** usando la interpolación (**{{}}**).

6. Ahora, en **app.component.ts** modificamos la clase **AppComponent** para que quede con lo siguiente:

```
export class AppComponent {

  title = '091 Bootstrap Button Group';

  res: string;

  onClick(event) { this.res = event.target.value; }

}
```

Al hacer un click sobre cualquier radio, se llamar al método **onClick** el cual, obtendrá el valor de la propiedad **value** y lo asignará a la variable **res** para que se muestre como la opción seleccionada.

7. Guardamos el archivo y vemos cómo funciona en el navegador haciendo **click** sobre los **radio buttons** y observando cómo cambia el valor de la opción seleccionada.

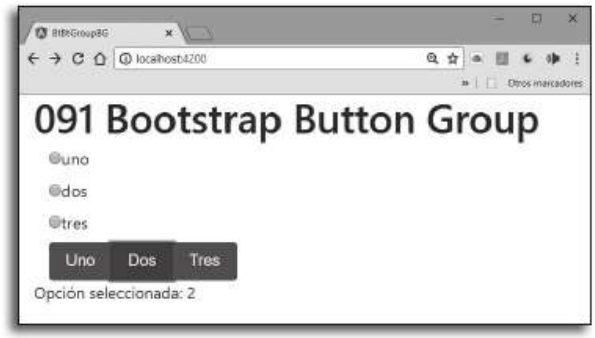
8. Ahora, en la página dedicada a los **Button group** que indicamos al principio, localizamos el ejemplo básico y copiamos (**Copy**) su contenido para pegarlo antes de "**Opción seleccionada**":

```
<div class="btn-group" role="group" aria-label="Basic
    example">
  <button type="button" class="btn btn-secondary">Left</
    button>
  <button type="button" class="btn btn-secondary">Midd-
    le</button>
  <button type="button" class="btn btn-secondary">Right</
    button>
</div>
```

Una vez copiado lo modificamos para adaptarlo al primer ejemplo y que quede de la siguiente manera:

```
<div class="container">
  <div class="btn-group" role="group" aria-label="Basic
    example">
    <button type="button" class="btn btn-primary btn-
    group" name="grupo2" value="1" (click)="onClick($even-
    t)">Uno</button>
    <button type="button" class="btn btn-primary btn-
    group" name="grupo2" value="2" (click)="onClick($even-
    t)">Dos</button>
    <button type="button" class="btn btn-primary btn-
    group" name="grupo2" value="3" (click)="onClick($even-
    t)">Tres</button>
  </div>
</div>
```

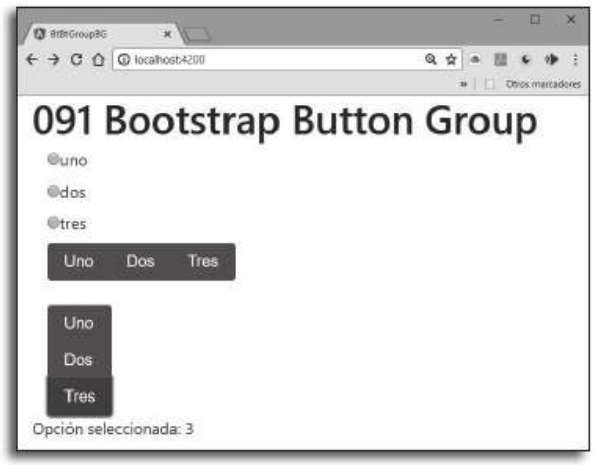
Guardamos el archivo y probamos su funcionamiento en el navegador.



9. Seguidamente, copiamos este mismo bloque y lo pegamos a continuación del ** ** que lo cierra. Añadamos también un **
** para separar los bloques **DIV** y modifiquemos la clase **btn-group** por **btn-group-vertical** para mostrar los botones verticalmente:

```
…
</div>
<br>
<div class="container">
  <div class="btn-group-vertical" role="group" aria-la-
   bel="Basic example">
    <button type="button" class="btn btn-primary btn-
   group" name="grupo3" value="1" …
…
```

Guarde el archivo pruebe su funcionamiento en el navegador pulsando sobre los botones.



10. Para acabar, probaremos cómo cambiar el tamaño de los botones. Añadiremos la clase **btn-group-sm** al primer grupo de botones que hemos copiado para que la clase del **div** quede con lo siguiente:

```
<div class="btn-group btn-group-sm"
```

Los botones se verán pequeños (**sm->small**). Guarde archivo y compruébelo en el navegador.

11. Ahora, mostremos en tamaño grande (**lg->large**) los botones que mostramos verticalmente y, para ello, añadiremos la clase **btn-group-lg** junto a la clase **btn-group-vertical** quedando así:

```
<div class="btn-group-vertical btn-group-lg"
```

Guarde el archivo y compruebe cómo se visualiza en el navegador.

092 Bootstrap: Cards

Las **Cards** son un tipo de contenedor que ofrece multitud de opciones para encabezados, pies de página, colores de fondo, etc. Una de las ventajas es que son flexibles y extensibles. Además, se construyen con muy poca cantidad de código y están fabricadas con **flexbox** (sistema de elementos flexibles que facilita el posicionamiento de dichos elementos cuando cambia el tamaño de la página).

En esta página nos encontramos diversos temas que resumimos a continuación:

- **Content types:** permite incluir una gran variedad de elementos en su interior.
- **Blocks:** facilita una sección bien espaciada dentro de una tarjeta.
- **Titles**, **text** y **links**: dentro de una tarjeta podemos encontrarnos con diferentes partes (títulos, subtítulos, textos, links, etc.) que poseen su correspondiente clase (**card-title**, **card-subtitle**, **card-text**, **card-link**, etc.) para facilitar su tratamiento.
- **Images/overlays:** podemos añadir una imagen en la parte superior o inferior de la tarjeta. Podemos usar una imagen como fondo y poner el texto encima.
- **List groups:** crea lista de contenido en una tarjeta.
- **Kitchen sink:** permite combinar diferentes tipos de contenido sobre un ancho fijo.
- **Header** and **footer:** permite añadir encabezados y pies.
- **Sizing:** podemos usar tags de grid para disponer las tarjetas en filas o columnas según necesitemos. Recordemos que podemos distribuir nuestros elementos en 12 columnas que podemos agrupar como nos interese mediante expresiones col-X, siendo X el número de columnas que pretendemos ocupar con el elemento.
- **Using utilities:** podemos definir que usen ancho variables como, por ejemplo, **w-50** (ocupa el **50%** del ancho de la página).
- **Text alignment:** es posible alinear el texto fácilmente **(izquierda**, **centro**, **derecha)**.
- **Navigation:** añade navegación al encabezado de la tarjeta.

Importante

Use **Cards** para organizar mejor su información. Es ideal para tratar fichas donde un determinado objeto de base de datos puede mostrar diferentes atributos con diferentes aspectos y ubicaciones.

- **Card styles:** diferentes tipos de fondos, bordes y colores (**Background variants**, **Outline cards**).
- **Card layout:** permite visualizar las tarjetas en series (**Card groups**, **Card decks**, o unidas entre sí, **Card columns**).

A continuación, vamos a realizar un ejercicio en el que veremos algunos ejemplos de los descritos en la página.

1. En primer lugar, nos ubicamos en **Ej100_angular** y creamos el proyecto **btCards** tecleando **ng new btCards**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename btCards 092_btCards
```

   Abrimos nuestro proyecto (por ejemplo, con **Atom**) y abrimos el archivo **app.component.ts** para modificar el título **(title)** y poner por ejemplo **"092 Bootstrap Cards"**.

3. Igual que en ejercicios anteriores, modificaremos el archivo **app.component.html** para insertar al principio y al final del mismo las referencias a la hoja de estilos de **Bootstrap** (al principio) y a **jquery**, **tether** y **bootstrap** (al final):

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.
   com/bootstrap/4.0.0- ...

…

<script src="https://code.jquery.com/jquery ...
```

   Añadiremos un elemento **div** con la clase **container** alrededor de nuestro **title** para que quede así:

```
<div class="container">

  <h1>{{title}}</h1>

</div>
```

   En la página de Bootstrap dedicada a Cards (https://v4-alpha.getbootstrap.com/components/card/) localizamos el apartado **Titles**, **text**, and **links** y copiamos **(Copy)** el ejemplo que muestra:

```
<div class="card">
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
    <h6 class="card-subtitle mb-2 text-muted">Card sub-
   title</h6>
    <p class="card-text">Some quick example text to build
   on the card title and make up the bulk of the card's
   content.</p>
    <a href="#" class="card-link">Card link</a>
    <a href="#" class="card-link">Another link</a>
  </div>
</div>
```

Lo pegamos detrás del tag **</h1>** que cierra el título, guardamos el archivo y arrancamos la aplicación (con **ng serve** desde **C:\Ej100_angular>092_btCards**) para ver cómo se ve nuestra aplicación en el navegador.

4. Modifiquemos el tamaño de nuestra pantalla y comprobemos cómo la tarjeta se adapta a cada tamaño.

5. Para el siguiente ejemplo, necesitaremos disponer de una imagen, la cual, al igual que explicamos en el ejercicio 087, almacenaremos en una carpeta creada en nuestro proyecto y denominada src/images. La imagen la llamaremos **img1.jpg**.

6. El siguiente ejemplo es de los más completos ya que mezcla diferentes funcionalidades en uno solo. Dicho ejemplo lo fabricaremos localizando el apartado **Kitchen sink** y copiando **(Copy)** su contenido para pegarlo a continuación del cierre del tag div de la tarjeta anterior:

```
…
</div>
<div class="card" style="width: 20rem;">
  <img class="card-img-top" src="..." alt="Card image
   cap">
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some quick example text to build
   on the card title and make up the bulk of the card's
   content.</p>
  </div>
```

```
  <ul class="list-group list-group-flush">
    <li class="list-group-item">Cras justo odio</li>
    <li class="list-group-item">Dapibus ac facilisis in</
   li>
    <li class="list-group-item">Vestibulum at eros</li>
  </ul>
  <div class="card-block">
    <a href="#" class="card-link">Card link</a>
    <a href="#" class="card-link">Another link</a>
  </div>
</div>
```

Añadimos un **
** entre ambas tarjetas y modificamos el **src** del código copiado para que quede de la siguiente manera:

```
<img class="card-img-top" src="images/img1.jpg" alt="Card
   image cap">
```

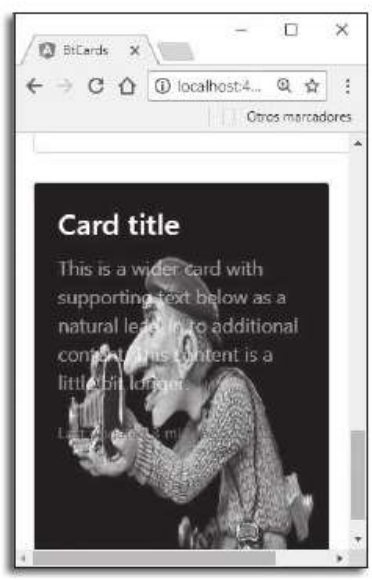
Guardamos y vemos el archivo en el navegador.

7. Por último, localice el ejemplo asociado al apartado **Image overlays**, copie su contenido **(Copy)** y péguelo a continuación del div correspondiente al ejemplo anterior. Inserte un **
** entre **div** para dejar una fila de separación.

8. Modifique el **src** para que apunte a **images/img1.jpg**.

9. Guarde el archivo y visualícelo en el navegador.

10. Cambie el navegador de tamaño (el ancho) y compruebe la adaptación de esta tarjeta.



11. Le invitamos a que pruebe más ejemplos para comprobar las diversas funcionalidades expuestas en la página de **Cards** de Bootstrap.

093 Bootstrap: Instalación local

Otra forma de trabajar con las librerías de **Bootstrap** es descargándolas sobre nuestro propio puesto de trabajo y referenciándolas desde nuestro proyecto. Esto aporta alguna ventaja como puede ser poder trabajar sin conexión y un aumento de la eficiencia. Así pues, en los próximos ejercicios, usaremos la instalación local para probar una nueva forma de utilizar **Bootstrap**. Para conseguir esta instalación, descargaremos las librerías de **Bootstrap**, **jQuery** y **Tether** y las incluiremos en nuestro proyecto. Veamos los detalles de cada una de las descargas:

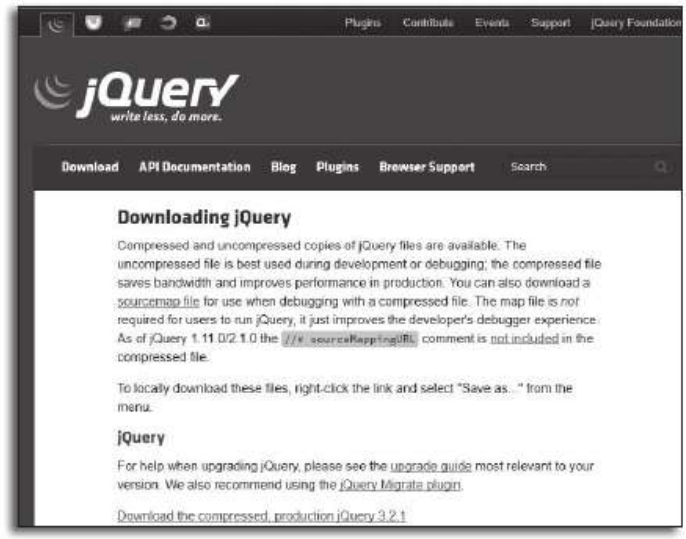
Importante

Instale localmente las librerías para poder trabajar con conexión a **Internet** y para agilizar la carga de las páginas como alternativa al uso de referencias **CDN**.

- **Descarga de bootstrap:** en la página https://v4-alpha.getbootstrap.com/getting-started/download/ descargaremos **Bootstrap CSS and JS** pulsando sobre el botón **"Download Bootstrap"**. Dependiendo del momento en el que nos descarguemos el archivo, su nombre (o versión) puede variar, pero, en definitiva, nos ofrecerá descargar un archivo similar al siguiente: **bootstrap-4.0.0-alpha.6-dist.zip**. Podemos descargarlo en un directorio temporal para poder extraerlo más fácilmente. Una vez descargado, lo extraemos y comprobamos que en su interior hay dos carpetas (**css** y **js**), cada una de ellas con distintas librerías.

- **Descarga de jQuery:** acudiremos a la siguiente página: https://jquery.com/download/ y localizaremos la opción: **Download the compressed, production jQuery 3.2.1 slim build** (https://code.jquery.com/jquery-

3.2.1.slim.min.js). Esta es la versión comprimida que ocupa menos espacio y que será suficiente para conseguir nuestro propósito.



- **Descarga de Tether:** acudiremos a la página: http://tether.io/ y pulsaremos sobre el botón: **Download ZIP (v1.3.3)** para descargar un fichero con un nombre similar al siguiente: **tether-1.3.3.zip**. En nuestro temporal, lo descomprimimos y más tarde extraeremos lo necesario.

Una vez que ya disponemos de todas las librerías que vamos a necesitar, crearemos un proyecto para adjuntarlas y usarlas.

1. En primer lugar, nos ubicamos en **Ej100_angular** y creamos el proyecto **btInsLocal** tecleando **ng new btInsLocal**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename btInsLocal 093_btInsLocal
```

Abrimos nuestro proyecto (por ejemplo, con **Atom**) y, bajo la carpeta **src/assets**, crearemos las carpetas sobre las que copiaremos, respectivamente, las librerías indicadas a continuación (1):

Carpeta	Librería
bootstrap4/css	bootstrap.min.css
bootstrap4/js	bootstrap.min.js
jquery	jquery.slim.min.js
tether	tether.min.js

A continuación, hemos de modificar el archivo **angular-cli.json** para hacer referencia a estas librerías. Para ello, una vez abierto este archivo, localizamos el apartado que contiene lo siguiente:

```
"styles": [
  "styles.css"
],
"scripts": [],
```

Y lo modificamos por este contenido:

```
"styles": [
  "styles.css",
  "assets/bootstrap4/css/bootstrap.min.css"
],
  "scripts": [
    "assets/jquery/jquery-3.2.1.slim.min.js",
    "assets/tether/tether.min.js",
    "assets/bootstrap4/js/bootstrap.min.js"
  ],
```

Guardamos el archivo y abrimos el archivo **app-component.ts** para modificar el contenido de **title** y poner **"093 Bootstrap instalación local"**.

3. Desde el directorio **C:\Ej100_angular>093_btInsLocal** arrancaremos la aplicación tecleando **ng serve** y comprobaremos en el navegador cómo se visualiza la misma sin ningún error.

4. Para comprobar que la instalación funciona correctamente, podemos acudir a la página de **Bootstrap Documentation** y, en **Search**, buscar **Colors** para acceder a la página https://v4-alpha.getbootstrap.com/utilities/colors/. Una vez aquí, localizamos el ejemplo que hay antes del apartado **Dealing with specificity** y copiamos las tres primeras líneas del mismo. Una vez copiadas, las pegaremos al final del contenido de nuestro archivo app.component.html justo debajo del cierre de **</h1>** de nuestro título.

5. Para que se visualice mejor, envolveremos todo el contenido de la página en un **div** con la clase **container**, de forma que todo quede de la siguiente manera:

```
<div class="container">
  <h1>{{title}}</h1>
  <div class="bg-primary text-white">Nullam id dolor id
   nibh ultricies vehicula ut id elit.</div>
  <div class="bg-success text-white">Duis mollis, est non
   commodo luctus, nisi erat porttitor ligula.</div>
  <div class="bg-info text-white">Maecenas sed diam eget
   risus varius blandit sit amet non magna.</div>
</div>
```

Guardamos el archivo y vemos cómo se visualiza en el navegador.

6. Hagamos alguna prueba más buscando el tema **Typography** (por favor, use **Search**) y, dentro de la página, localice **Text transform** para copiar el ejemplo propuesto y pegar a continuación de lo anterior (antes del cierre del último ** **) lo siguiente:

```
<div class="container my-2" style="border-style: so-
 lid;">
  <p class="text-lowercase">Lowercased text.</p>
  <p class="text-uppercase">Uppercased text.</p>
  <p class="text-capitalize">CapiTaliZed text.</p>
</div>
```

Además de añadir un contenedor, un borde y un margen sobre el eje y (**my-2**), veremos la conversión a **minúsculas**, **mayúsculas** y "**capitalización**" del texto.

7. Por último, localice el apartado **Font weight and italics** y copie su ejemplo **(Copy)** para pegarlo y envolverlo con un **div** (con la clase **container** y otro borde diferente) a continuación del bloque anterior para que quede de la siguiente manera:

```
<div class="container" style="border-style: double;">
  <p class="font-weight-bold">Bold text.</p>
  <p class="font-weight-normal">Normal weight text.</p>
  <p class="font-italic">Italic text.</p>
</div>
```

Guarde el archivo y observe cómo se ve en el navegador.

8. Compruebe cómo se adapta el texto a los diferentes tamaños de pantalla.

094 Bootstrap: Carousel

El carrusel es un elemento de **Bootstrap** que nos permite realizar presentaciones muy vistosas a través de la visualización de imágenes, texto y otros elementos de forma cíclica. Como de costumbre, en la página de **Bootstrap** dedicada a este tema (https://v4-alpha.getbootstrap.com/components/carousel/) encontraremos diversos ejemplos listos para ser copiados y pegados en nuestra página para poder adaptarlos posteriormente a nuestras necesidades con poco esfuerzo.

Si analizamos la página nos encontraremos los siguientes temas:

- **Slides only:** muestra solo diapositivas (una detrás de otra).
- **With controls:** añade controles para desplazarnos por las diapositivas anterior y posterior.
- **With indicators:** con elementos que nos permiten desplazarnos directamente a una diapositiva en concreto.
- **With captions:** permite añadir subtítulos que pueden mostrarse o no en función del tamaño de la vista.
- **Usage:** opciones que permiten determinar la posición del carrusel, la velocidad de transición de una diapositiva a otra, si ha de reaccionar a eventos de teclado, etc.

A continuación, vamos a realizar un ejercicio en el que veremos algunos ejemplos de los descritos en la página.

1. En primer lugar, nos ubicamos en **Ej100_angular** y creamos el proyecto **btCarousel** tecleando **ng new btCarousel**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename btCarousel 094_btCarousel
```

   Abrimos nuestro proyecto (por ejemplo, con **Atom**) y abrimos el archivo **app.component.ts** para modificar el título **(title)** y poner por ejemplo **"094 Bootstrap Carousel"**.

3. Tal y como explicamos en el ejercicio anterior, instalaremos las librerías de forma local descargándolas e incorporándolas en el proyecto bajo la carpeta **assets**. Si ha realizado el ejercicio anterior, puede copiar la carpeta **assets** y pegarla bajo **src** de este mismo ejercicio. No olvide modificar el archivo **angular-cli.json** para hacer referencia a estas librerías tal y como se explicó.

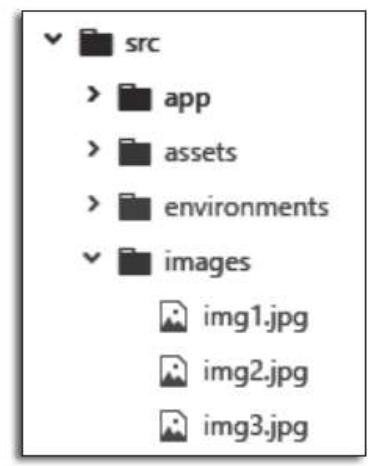
Importante

Cree presentaciones vistosas copiando y pegando el código ofrecido por Bootstrap y acelere la fabricación de vistas dejando que las clases de Bootstrap hagan el trabajo.

4. A continuación, en nuestro archivo **app.component.html** añadiremos un elemento **div** con la clase **container** alrededor de nuestro **title** para que quede así:

```
<div class="container-fluid">
  <h1>{{title}}</h1>
</div>
```

Para este ejercicio, necesitaremos tres imágenes (p. ej., **img1.jpg**, **img2.jpg** e **img3.jpg**) que depositaremos en una nueva carpeta llamada **images** que crearemos bajo **src**. Procure que sean imágenes de un cierto tamaño para que al mostrarlas en pantalla grande se vean bien.



5. En la página de **Bootstrap/Carousel** (https://v4-alpha.getbootstrap.com/components/carousel/) localizamos el ejemplo que muestra bajo el apartado **With controls** y copiamos su contenido para pegarlo a continuación del cierre del tag **</h1>** que cierra nuestro **title**:

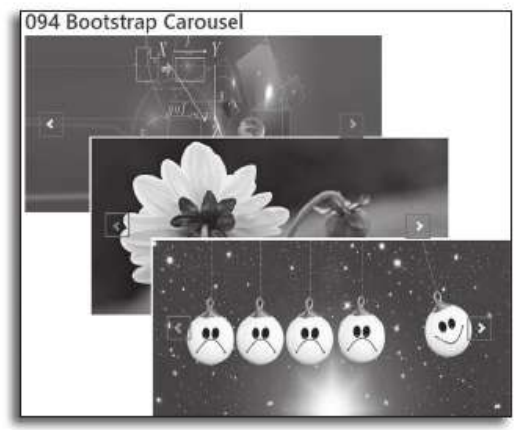
```
…
<h1>{{title}}</h1>
<div id=”carouselExampleControls”class=”carousel slide”-
   data-ride=”carousel”>
  <div class=”carousel-inner”role=”listbox”>
    <div class=”carousel-item active”>
      <img class=”d-block img-fluid”src=”...”alt=”First
   slide”>
    </div>
    <div class=”carousel-item”>
      <img class=”d-block img-fluid”src=”...”alt=”Second
   slide”>
    </div>
    <div class=”carousel-item”>
      <img class=”d-block img-fluid”src=”...”alt=”Third
   slide”>
    </div>
 </div>
  <a class=”carousel-control-prev”href=”#carouselExam-
   pleControls”role=”button”data-slide=”prev”>
    <span class=”carousel-control-prev-icon”aria-hid-
   den=”true”></span>
    <span class=”sr-only”>Previous</span>
  </a>
  <a class=”carousel-control-next”href=”#carouselExam-
   pleControls”role=”button”data-slide=”next”>
    <span class=”carousel-control-next-icon”aria-hid-
   den=”true”></span>
    <span class=”sr-only”>Next</span>
  </a>
</div>
…
```

Ahora, modificaremos las propiedades **src** de cada imagen para que apunten a la ubicación de nuestras imágenes y añadiremos la clase **w-100** a las clases del tag **img** de la siguiente manera:

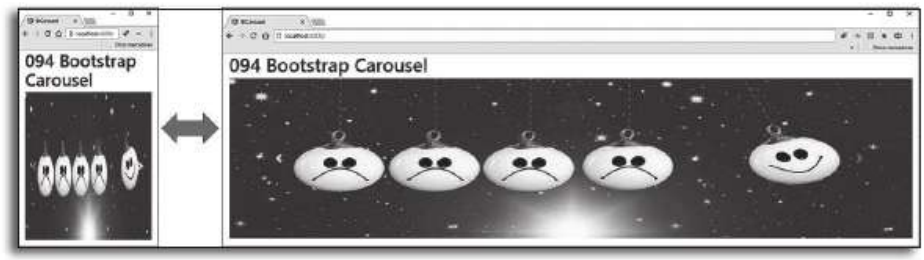
```
<img class="d-block img-fluid w-100" src="images/img1.jpg"
    alt="First slide">
```

Para las imágenes 2 y 3 realizaremos la misma operación. Guardaremos el archivo y arrancaremos la aplicación ubicándonos en **C:\Ej100_angular\094_btCarousel** y tecleando **ng serve**. Observe el resultado en el navegador accediendo a el **URL** http://localhost:4200/.

6. Pulse sobre las flechas laterales para provocar la visualización de las imágenes. Si no las pulsa, estas se irán visualizando cada cierto tiempo.



7. Varíe el tamaño de la página para comprobar cómo la visualización se adapta en cada momento.



8. A continuación, vamos a probar otro ejemplo similar, pero algo más completo. Para ello, localizaremos un poco más abajo, en la misma página de **Bootstrap**,

el apartado **With indicators** y copiaremos **(Copy)** su contenido para pegarlo en nuestra página **app.component.html** a continuación del ** ** que cierra nuestro carrusel anterior:

```
…
</a>
</div>
<div id="carouselExampleIndicators" class="carousel sli-
   de" data-ride="carousel">
  <ol class="carousel-indicators">
    <li data-target="#carouselExampleIndicators" da-
   ta-slide-to="0" class="active"></li>
    <li data-target="#carouselExampleIndicators" da-
   ta-slide-to="1"></li>
    <li data-target="#carouselExampleIndicators" da-
   ta-slide-to="2"></li>
  </ol>
  <div class="carousel-inner" role="listbox">
    <div class="carousel-item active">
      <img class="d-block img-fluid" src="..." alt="First
   slide">
    </div>
    <div class="carousel-item">
      <img class="d-block img-fluid" src="..." alt="Second
   slide">
    </div>
    <div class="carousel-item">
      <img class="d-block img-fluid" src="..." alt="Third
   slide">
    </div>
  </div>
  <a class="carousel-control-prev" href="#carouselExam-
   pleIndicators" role="button" data-slide="prev">
    <span class="carousel-control-prev-icon" aria-hid-
   den="true"></span>
    <span class="sr-only">Previous</span
```

```
  </a>
  <a class="carousel-control-next" href="#carouselExam-
    pleIndicators" role="button" data-slide="next">
     <span class="carousel-control-next-icon" aria-hid-
    den="true"></span>
     <span class="sr-only">Next</span>
  </a>
</div>
…
```

Al igual que en el caso anterior, modificaremos los tags correspondientes a las imágenes para que apunten a las ubicaciones de nuestro proyecto y contengan la clase **w-100**:

```
<img class="d-block img-fluid w-100" src="images/img1.jpg"
    alt="First slide">
```

Para separar un carrusel de otro, añadiremos la clase **my-2** para incluir un margen en el eje **y** (verticalmente) y un borde al bloque. La primera línea de nuestro carrusel quedará así:

```
<div id="carouselExampleIndicators" class="carousel sli-
    de my-2" data-ride="carousel" style="border-style:
    double;">
```

Seguidamente, añadiremos subtítulos a cada imagen y, localizando el apartado **With captions**, copiaremos el bloque que hay justo debajo de la imagen que contiene la clase **carousel-caption**:

```
<img src="..." alt="...">
<div class="carousel-caption d-none d-md-block">
  <h3>...</h3>
  <p>...</p>
</div>
```

Pegaremos dicho bloque bajo el tag de cada una de nuestras imágenes y añadiremos un texto para los tags **<h3>** y **<p>**. Por ejemplo, el bloque asociado a la **imagen 1** quedaría así:

```
<img class="d-block img-fluid w-100" src="images/img1.jpg"
   alt="First slide">
<div class="carousel-caption d-none d-md-block">
  <h3>Imagen 1</h3>
  <p>Esta es una prueba de la imagen 1</p>
</div>
```

Copiemos dicho bloque sobre el resto de imágenes modificando **Imagen 1** por la que corresponda, guardemos el archivo y visualicemos cómo queda en el navegador. Compruebe cómo pulsando sobre los indicadores existentes a pie de página, se muestra la imagen asociada a cada uno de los mismos.

095 Bootstrap: Collapse

Mediante el complemento **Collapse**, podemos mostrar u ocultar determinadas partes de nuestra página facilitando la optimización del espacio y proporcionando un efecto visual muy agradable. La idea es que podamos colapsar o expandir un bloque como si fuera un acordeón. En el siguiente ejercicio, vamos a poner en marcha uno de los ejemplos expuestos en la página de **Bootstrap** dedicado a este tema (https://v4-alpha.getbootstrap.com/components/collapse/) y lo personalizaremos un poco para poder jugar con él.

1. En primer lugar, nos ubicamos en **Ej100_angular** y creamos el proyecto **btCollapse** tecleando **ng new btCollapse**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename btCollapse 095_btCollapse
```

   Abrimos nuestro proyecto (por ejemplo, con **Atom**) y abrimos el archivo **app.component.ts** para modificar el título **(title)** y poner por ejemplo **"095 Bootstrap Collapse"**.

3. Tal y como explicamos en el ejercicio anterior, instalaremos las librerías de forma local descargándolas e incorporándolas en el proyecto bajo la carpeta **assets** y también modificaremos el archivo **angular-cli.json** para hacer referencia a estas librerías. Por favor, revise el ejercicio 093 si tiene alguna duda.

4. A continuación, en nuestro archivo **app.component.html** añadiremos un elemento **div** con la clase **container** alrededor de nuestro **title** para que quede así:

```
<div class="container-fluid">
  <h1>{{title}}</h1>
</div>
```

En la página de **Bootstrap/Collapse** localizamos el primer ejemplo y copiamos su contenido para pegarlo a continuación del cierre del tag **</h1>** que cierra nuestro **title**:

```
…
<p>
  <a class="btn btn-primary" data-toggle="collapse" hre-
   f="#collapseExample" aria-expanded="false" aria-con-
   trols="collapseExample">
    Link with href
  </a>
  <button class="btn btn-primary" type="button" data-tog-
   gle="collapse" data-target="#collapseExample" aria-ex-
   panded="false"
    aria-controls="collapseExample">
    Button with data-target
  </button>
</p>
<div class="collapse" id="collapseExample">
  <div class="card card-block">
    Anim pariatur cliche reprehenderit, enim eiusmod high
   life accusamus terry richardson ad squid. Nihil anim
   keffiyeh helvetica, craft beer labore wes anderson cred
   nesciunt sapiente ea proident.
  </div>
</div>
…
```

Guardaremos el archivo y arrancaremos la aplicación ubicándonos en C:\Ej100_angular\095_btCollapse y tecleando **ng serve**. Observe el resultado en el navegador accediendo al URL http://localhost:4200/.

Importante

Organice su contenido en poco espacio aplicando **Collapse** y haciendo que la información se muestre cuando sea requerida.

5. Pulsemos ambos botones para comprobar cómo se muestra y oculta el bloque asociado.

6. Ahora, modificaremos un poco el código copiado para personalizarlo y simplificarlo. Básicamente, lo que haremos es rebautizar el id del **div** asociado a la clase **collapse** y las referencias al mismo, tanto en el tag **<a>** como **<button>**. El código quedará de la siguiente manera:

```
<p>
  <a class="btn btn-primary" data-toggle="collapse" hre-
   f="#uno" aria-expanded="false" aria-controls="uno">U-
   no Con Ref</a>
  <button class="btn btn-primary" type="button" data-to-
   ggle="collapse" data-target="#uno" aria-expanded="-
   false" aria-controls="uno">
      Uno con boton
    </button>
</p>
<div class="collapse" id="uno">
  <div class="card card-block">Mi bloque uno</div>
</div>
```

Guardamos el archivo y comprobamos de nuevo cómo queda en el navegador. Observe que el bloque se muestra y oculta alternativamente sea cual sea el botón que pulsemos.

7. Para dar un poco más de juego al ejemplo, introduciremos un bloque dentro de otro para anidar los **acordeones**.

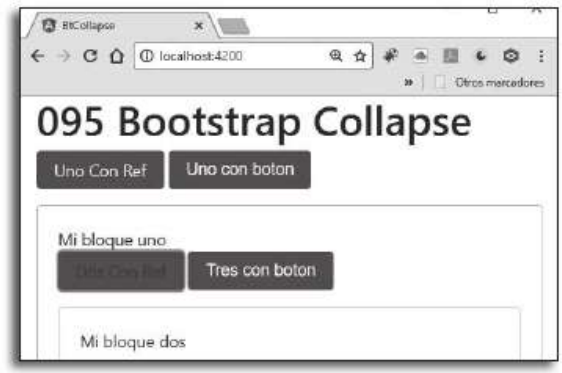
8. En este caso, copiaremos debajo del texto **"Mi bloque uno"**, un bloque similar al copiado y adaptado inicialmente, el cual empieza en el tag **<p>** y acaba en el tag ** **. La idea, en este caso, es que el botón copiado relacionado con el tag **<a>** apunte a un bloque denominado **"dos"**, y el botón tipo **"button"** apunte a un div llamado **"tres"**. En definitiva, el bloque que contenía el texto **"Mi bloque uno"** quedará de la siguiente manera:

```
<div class="collapse" id="uno">
  <div class="card card-block">
    Mi bloque uno
    <p>
      <a class="btn btn-primary" data-toggle="collapse"
   href="#dos" aria-expanded="false" aria-controls="-
   dos">Dos Con Ref</a>
      <button class="btn btn-primary" type="button" da-
   ta-toggle="collapse" data-target="#tres" aria-expande-
   d="false" aria-controls="tres">
          Tres con boton
        </button>
    </p>
    <div class="collapse" id="dos">
      <div class="card card-block">Mi bloque dos</div>
    </div>
    <div class="collapse" id="tres">
      <div class="card card-block">Mi bloque tres</div>
    </div>
  </div>
</div>
```

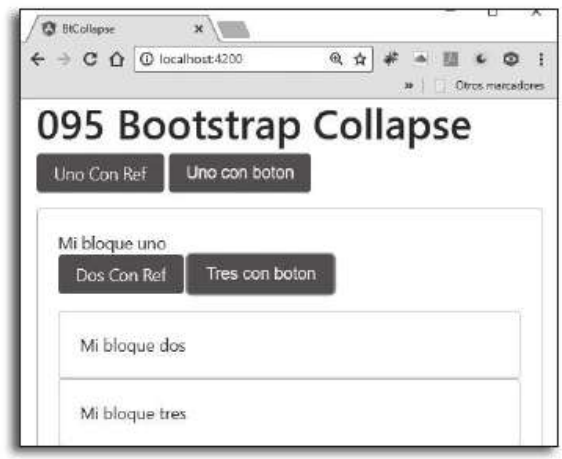
Guardamos el archivo y lo visualizamos en nuestro navegador.

9. Si pulsamos cualquiera de los botones **"Uno Con Ref"** o **"Uno con Boton"**, aparece el bloque con el texto **"Mi bloque uno"** con los dos botones incluidos en el mismo.

10. Si pulsamos el botón **"Dos Con Ref"**, veremos el bloque **"dos"** asociado al mismo.



11. Si pulsamos el botón **"Tres con boton"**, veremos el bloque **"tres"** asociado al mismo.



12. A partir de aquí, le invitamos a que pruebe cualquier combinación de pulsación de botones para ver cómo se comporta la visualización en función de su anidamiento.

096 Bootstrap: Dropdowns

Los **Dropdowns** son elementos que se despliegan de forma interactiva y que permiten mostrar listas de elementos como, por ejemplo, vínculos mediante los cuales se pueden realizar selecciones o acciones. Si analizamos la página de Bootstrap dedicada a Dropdowns (https://v4-alpha.getbootstrap.com/components/dropdowns/), veremos que hace referencia a diferentes temas relacionados con:

- **Diferentes tipos de botones**: sencillos, separando la flecha del elemento en un botón aparte.
- **Sizing:** para botones pequeños (**small**) y grandes (**large**).
- **Dropup variation**: para que el despliegue de la lista se haga hacia arriba.
- **Menu items:** ofrece el uso de botones en lugar de links para las opciones.
- **Menu alignment:** permite alinear el despliegue de opciones a la derecha.
- **Menu headers:** inserción de cabeceras en la lista.
- **Menu dividers:** inclusión de divisores en la lista.
- **Disabled menu items:** deshabilita ítems de una lista.

Importante

Como alternativa a la organización de opciones, puede disponer de listas desplegables a las que puede asociar a links o botones y realizar acciones de forma sencilla.

En el siguiente ejercicio, adaptaremos uno de los ejemplos de dicha página para seleccionar un color de la lista y aplicarlo a un bloque que muestra el resultado seleccionado.

1. Ubicados en **Ej100_angular**, creamos el proyecto **btDropDown** tecleando **ng new btDropDown**.

2. Ahora, renombraremos el proyecto desde el directorio **Ej100_angular**, escribiendo lo siguiente:

```
C:\Ej100_angular>rename btDropDown 096_btDropDown
```

   Abrimos el proyecto y en **app.component.ts** modificamos **title** con **"096 Bootstrap btDropDown"**.

3. Instalaremos las librerías localmente. Por favor, revise el ejercicio 093.

4. En **app.component.html** añadimos un **div** con la clase **container** alrededor de nuestro **title** así:

```
<div class="container-fluid"><h1>{{title}}</h1></div>
```

Localizamos el primer ejemplo de Dropdowns, y lo pegamos en **app.component.html** detrás de **</h1>**:

```
<div class="container-fluid">
  <h1>{{title}}</h1>
  <div class="dropdown">
    <button class="btn btn-secondary dropdown-toggle"
   type="button" id="dropdownMenuButton" data-toggle="-
   dropdown" aria-haspopup="true"
    aria-expanded="false">
      Dropdown button
    </button>
    <div class="dropdown-menu" aria-labelledby="dropdown-
   MenuButton">
      <a class="dropdown-item" href="#">Action</a>
      <a class="dropdown-item" href="#">Another action</
   a>
      <a class="dropdown-item" href="#">Something else
   here</a>
    </div>
  </div>
```

Guardamos el archivo y arrancaremos la aplicación desde **C:\Ej100_angular\096_btDropDown** tecleando **ng serve**. Observe el resultado en el navegador en el **URL** http://localhost:4200/.

5. Pulse sobre la lista para desplegarla y compruebe que funciona perfectamente.

6. A continuación, crearemos un bloque de tipo **"row"** y dentro del mismo, dos bloques de tipo **"col"** donde reubicaremos nuestro elemento recién copiado y un nuevo div para mostrar el resultado de la selección, respectivamente, la cual, implementaremos a continuación. Las columnas ocuparán cinco columnas de la rejilla de **Bootstrap** (12 columnas), y así dejamos **margins** y **paddings** (**m-1** y **p-1**) entre bloques.

7. Ahora, el código de nuestro **app.component.html** quedará de la siguiente manera:

```
<div class="container">
  <h1>{{title}}</h1>
  <div class="row ">
    <div class="col col-5 borde m-1 p-1">
      <div class="dropdown">
        <button class="btn btn-secondary dropdown-tog-
   gle" type="button" id="menu1" data-toggle="dropdown"
   aria-haspopup="true"
          aria-expanded="false">Dropdown</button>
        <div class="dropdown-menu" aria-labelledby="me-
   nu1">
          <button class="dropdown-item" type="button"
   (click)="onClick('Azul')">Azul</button>
          <button class="dropdown-item" type="button"
   (click)="onClick('Verde')">Verde</button>
          <button class="dropdown-item" type="button"
   (click)="onClick('Amarillo')">Amarillo</button>
        </div>
      </div>
    </div>
    <div class="col col-5 m-1 p-1 borde text-center"
   [ngStyle]="{'background-color': color1, 'font-size':
   '130%'}">Opcion: <b>{{ res1 }}</b></div>
  </div>
</div>
```

En el fichero **styles.css** incluiremos la definición de la clase **borde** con el siguiente contenido:

```
.borde{ border: 3px solid blue; }
```

En el archivo **app.component.ts**, la clase **AppComponent** quedará con el siguiente contenido:

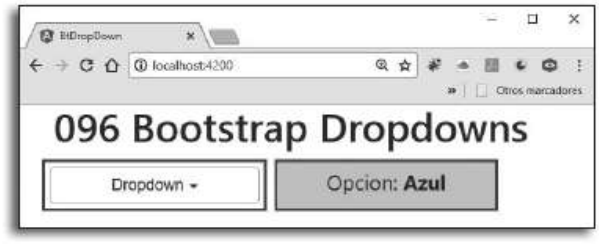
```
export class AppComponent {
  title = '096 Bootstrap Dropdowns';
  res1:string = "Azul";
  color1:string = '#A9F5F2';
  onClick(par:string){
    this.res1=par;
    switch(this.res1) {
      case "Verde": this.color1 = "#A9F5BC"; break;
      case "Amarillo": this.color1 = "#FFFF00"; break;
      default: this.color1 = "#A9F5F2"; break;
    }
  }
}
```

Guardemos el archivo y veamos cómo queda en el navegador.

8. Añadimos la clase **w-100** al botón que define el dropdown con **id="menu1"** para que quede así:

```
<button class="btn btn-secondary dropdown-toggle w-100"
```

Guardamos el archivo y vemos que el botón del Dropdown ocupará todo el bloque.



9. Probemos desplegar el botón y seleccionar Verde o Amarillo.

10. Sustituyamos la clase **btn-secondary** del primer botón para poner **btn-success** para ver cómo queda.

11. Por último, añadiremos la clase **btn-sm** al primer botón para ver cómo el botón se muestra más pequeño. El código asociado al botón queda de la siguiente manera:


```
<button class="btn btn-success dropdown-toggle w-100
   btn-sm" type="button" id="menu1" data-toggle="drop-
   down" aria-haspopup="true"
  aria-expanded="false">Dropdown</button>
```

097 Bootstrap: Forms

Bootstrap ofrece una gran cantidad de controles y clases para la fabricación de formularios con elementos de diferentes estilos y tamaños. Puede consultar la página en el apartado **Forms** de **Bootstrap** (https://v4-alpha.getbootstrap.com/components/forms/). Por defecto, se aplican estilos a todos los controles y estos se apilan verticalmente, ya que **Bootstrap** establece **display: block** y **width: 100%** a casi todos los componentes del formulario. No obstante, podemos usar la clase **form-inline** para colocar las etiquetas a la izquierda de los campos y hacer que los controles se muestren en la medida de lo posible de forma horizontal.

Los clásicos controles soportados son: **Input**, **Textarea**, **Checkbox**, **Radio**, **Select** y **Controles estáticos**.

La etiqueta y el input asociados a un campo se suelen agrupar en un **div** al que se le añade la clase **form-group** para optimizar mejor el espacio y obtener una mayor organización. En los controles de tipo input, podemos usar **placeholder** para mostrar un texto explicativo sobre lo que se solicita. En los controles **input**, **textarea** y **select**, podemos añadir la clase **form-control** para que el ancho se establezca a **width: 100%.**

Importante

Construya sus formularios por bloques y aproveche las funcionalidades que le ofrecen las clases de **Bootstrap** para ahorrarse tiempo y problemas.

Los campos de tipo input pueden ser: de texto **(text**, **password)**, de fecha **(datetime**, **datetime-local**, **date**, **month**, **time**, **week)**, de número **(number)** y otros (**email**, **url**, **search**, **tel**, y **color**).

A continuación, fabricaremos un formulario sencillo con algunos controles para mostrar algunas de las funcionalidades de las clases que **Bootstrap** ha preparado.

1. Nos ubicamos en **Ej100_angular** y creamos el proyecto **btForms** tecleando **ng new btForms**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename btForms 097_btForms
```

Abrimos nuestro proyecto (por ejemplo, con **Atom**) y abrimos el archivo **app.component.ts** para modificar el título **(title)** y poner por ejemplo **"097 Bootstrap Forms"**.

3. Tal y como ya hemos explicado en ejercicios anteriores, instalaremos las librerías de forma local descargándolas e incorporándolas en el proyecto. Por favor, revise el ejercicio 093 si tiene alguna duda.

4. A continuación, en nuestro archivo **app.component.html** añadiremos un elemento **div** con la clase **container** alrededor de nuestro **title** para que quede así:

```
<div class="container">
  <h1>{{title}}</h1>
</div>
```

A continuación del tag **</h1>** anterior, insertaremos un formulario vacío solo con el botón **submit**:

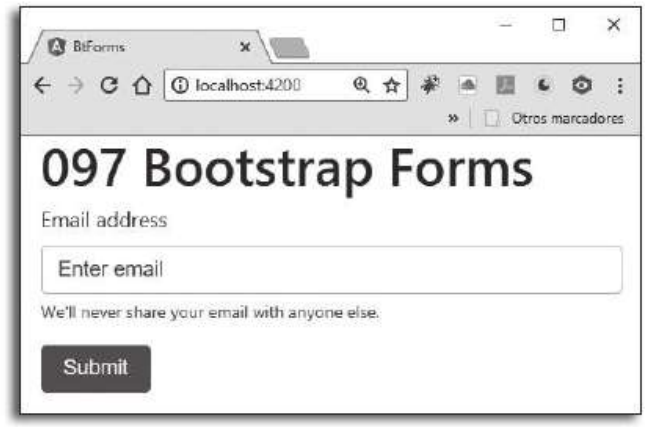
```
…</h1>
<form>
  <button type="submit" class="btn btn-primary">Submit</but-
 ton>
</form>
```

Guarde el archivo y arranque la aplicación ubicándose en **C:\Ej100_angular\097_btForms** y tecleando **ng serve**. Observe el resultado en el navegador (http://localhost:4200/).

5. Ahora, desde de la página de **Bootstrap** dedicado a **Forms**, localizamos el tema **Form controls** y copiamos del primer ejemplo el grupo que hace referencia al **exampleInputEmail1**:

```
<div class="form-group">
  <label for="exampleInputEmail1">Email address</label>
  <input type="email" class="form-control" id="exam-
   pleInputEmail1" aria-describedby="emailHelp" placehol-
   der="Enter email">
  <small id="emailHelp" class="form-text text-mute-
   d">We'll never share your email with anyone else.</
   small>
</div>
```

Lo pegamos antes del botón **submit**. Guardamos el archivo y vemos cómo queda en el navegador.

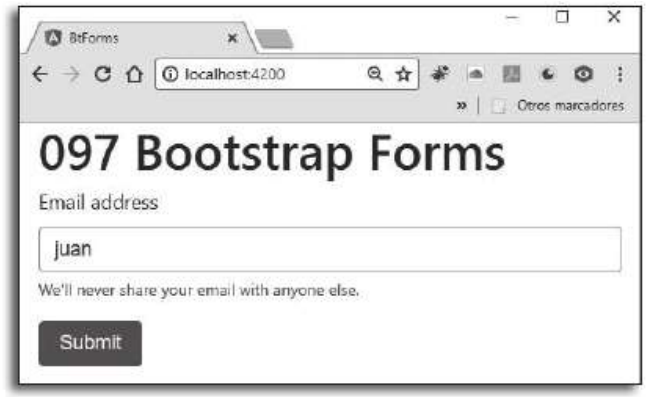


6. Si añadimos la clase **form-inline** al tag **<form>** veremos cómo los elementos se disponen horizontalmente en la pantalla:


```
<form class="form-inline">
```

Eliminamos esta clase para dejarlo como estaba antes y dejar que los controles se apilen verticalmente.

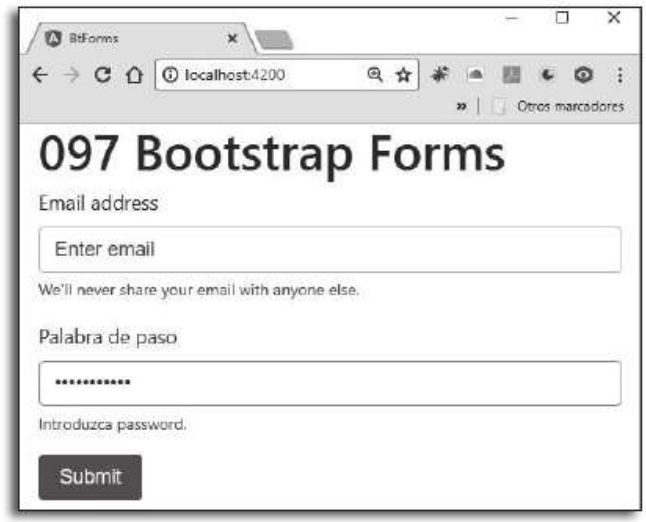
7. Observe como al hacer clic sobre el e-mail, el borde del mismo cambia de color y, al empezar a escribir, desaparece el **placeholder**.



8. Ahora copiamos el bloque anterior y lo pegamos debajo del mismo para cambiar algunos valores y convertirlo en un input de **password**. Ponemos lo siguiente:

```
<div class="form-group">
  <label for="password">Palabra de paso</label>
  <input type="password" class="form-control" id="example-
   password" aria-describedby="password">
  <small id="password" class="form-text text-muted">Introduzca
   password.</small>
</div>
```

Guardemos el archivo, visualicémoslo en el navegador e introduzcamos algo en el control para comprobar cómo oculta los caracteres introducidos.



9. En el ejemplo de **Bootstrap**, buscamos la parte correspondiente al **Select1** (sencillo) y la pegamos a continuación del bloque anterior:

```
<div class="form-group">
  <label for="exampleSelect1">Example select</label>
  <select class="form-control form-control-sm" id="exam-
   pleSelect1">
          <option>1</option>
          <option>2</option>
          <option>3</option>
          <option>4</option>
          <option>5</option>
        </select>
</div>
```

Guardamos el archivo, y en el navegador desplegamos la lista para ver cómo funciona.

10. Ahora, en la página de **Bootstrap**, dentro del apartado **Textual inputs**, localizaremos el div que contiene el ejemplo **example-date-input** y pegaremos su contenido a continuación del bloque anterior:

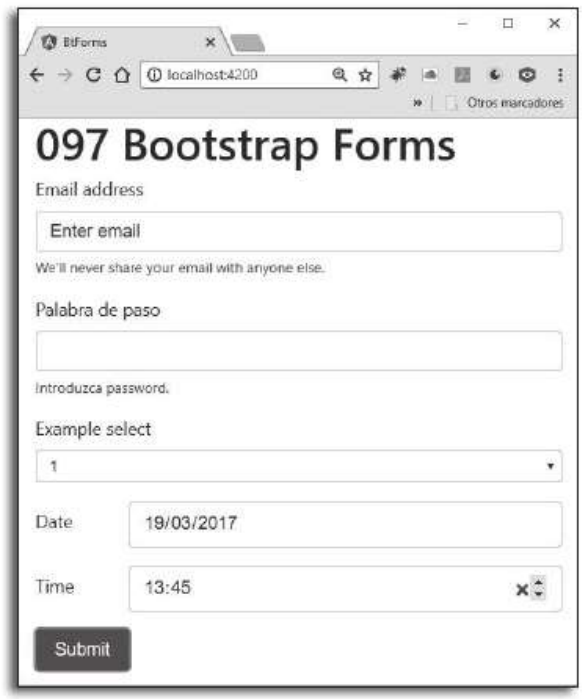
```
<div class="form-group row">
  <label for="example-date-input" class="col-2
   col-form-label">Date</label>
  <div class="col-10">
    <input class="form-control" type="date" va-
   lue="2011-08-19" id="example-date-input">
  </div>
</div>
```

Guardamos la página y en el navegador, desplegaremos el control pulsando sobre la flecha.

11. En el mismo apartado **Textual inputs**, localizaremos el div que contiene el ejemplo **example-time-input** y pegaremos su contenido a continuación del bloque anterior:

```
<div class="form-group row">
  <label for="example-time-input" class="col-2
   col-form-label">Time</label>
  <div class="col-10">
    <input class="form-control" type="time" va-
   lue="13:45:00" id="example-time-input">
  </div>
</div>
```

Guarde el archivo y pruebe en el navegador con el nuevo control para ver su comportamiento.



098 Bootstrap: List group

Los **List group** son elementos que permiten visualizar información mostrándola en forma de lista. En la página se analizan diferentes funcionalidades relativas a las listas como por ejemplo: **Active items** (para indicar la selección activa), **Disabled items** (para deshabilitar ítems), **Links and buttons** (para crear listas a partir de links o de botones), **Contextual classes** (permite diseñar un fondo y un color con el estado), **With badges** (permite añadir **insignias** para mostrar conteos de actividad; p. ej., mensajes no leídos, etc.), **Custom content** (permite realizar listas personalizadas añadiendo prácticamente cualquier tag **<HTML>**).

Importante

Las listas son un buen recurso para mostrar información o enumerar un grupo de acciones a realizar. Use Bootstrap para fabricar listas apoyándose no solo en elementos de listas clásicos sino también en links y botones.

En el siguiente ejercicio crearemos un par de listas a las que les añadiremos algunas funcionalidades propuestas en la página de **Bootstrap** dedicada a **List group** (https://v4-alpha.getbootstrap.com/components/list-group/).

1. Nos ubicamos en **Ej100_angular** y creamos el proyecto **btListGroup** tecleando **ng new btListGroup**.

2. Renombraremos el proyecto para que haga referencia a nuestro número de ejercicio tecleando:

```
C:\Ej100_angular>rename btListGroup 098_btListGroup
```

   Abrimos el proyecto y **app.component.ts** modificando **title** con **"098 Bootstrap List group".**

3. Instalamos las librerías de forma local descargándolas (revise el capítulo 093).

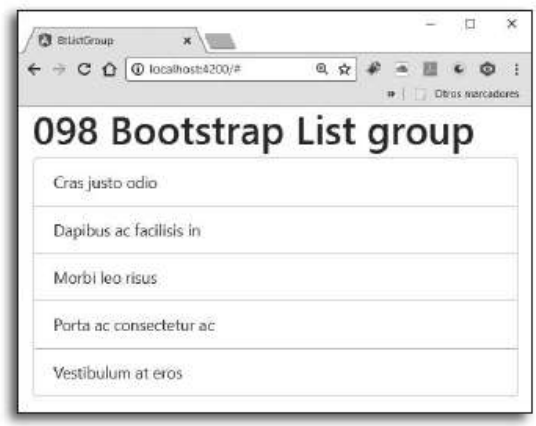
4. En **app.component.html** añadiremos un **div** con la clase **container** alrededor de **title**:

```
<div class="container"><h1>{{title}}</h1></div>
```

Ahora, copiamos el primer ejemplo de **Bootstrap** y lo pegamos detrás del tag **</h1>** que cierra **title**:

```
  …</h1>
  <ul class="list-group">
     <li class="list-group-item">Cras justo odio</li>
…</ul>
```

Guarde el archivo y arranque la aplicación ubicándose en **C:\Ej100_angular\098_btListGroup** y tecleando **ng serve**. Observe el resultado en el navegador (http://localhost:4200/).



5. Localice el apartado **Links and buttons** y copie (**Copy**) el primer ejemplo que empieza así:

```
<div class="list-group">
  <a href="#" class="list-group-item active">
    Cras justo odio  …
```

Lo modificaremos para que, al seleccionar un elemento de la lista, se muestre el resultado en un bloque a pie de pantalla y se cambie el color del mismo. Es similar al ejercicio **096_Bootstrap_Dropdowns**.

6. Una vez modificado, el bloque quedará con el siguiente contenido:

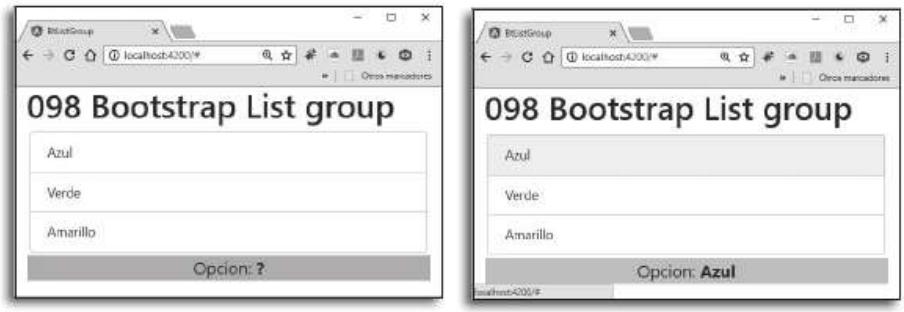
```
<div class="list-group p-1">
  <a href="#" class="list-group-item list-group-item-
   action justify-content-between" (click)="onClic-
   k('Azul')">Azul</a>
```

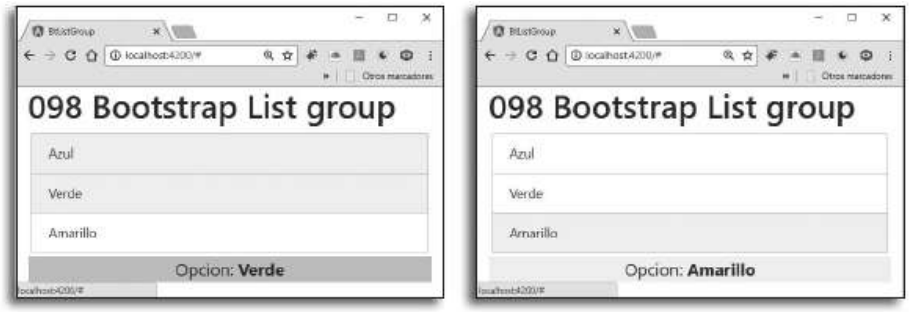
```
 <a href="#" class="list-group-item list-group-item-ac-
   tion justify-content-between" (click)="onClick('Ver-
   de')">Verde</a>
  <a href="#" class="list-group-item list-group-item-ac-
   tion justify-content-between" (click)="onClick('Amari-
   llo')">Amarillo</a>
</div>
<div class="col text-center" [ngStyle]="{'background-co-
   lor': color1, 'font-size': '130%'}">Opcion: <b>{{ res1
   }}</b></div>
```

Seguidamente, modificaremos **app.component.ts** para que la clase **AppComponent** quede así:

```
export class AppComponent {
  title = '098 Bootstrap List group';
  res1: string = "?";
  color1: string = '#D8D8D8';
  onClick(par: string) {
    this.res1 = par;
    switch (this.res1) {
      case "Verde": this.color1 = "#A9F5BC"; break;
      case "Amarillo": this.color1 = "#FFFF00"; break;
      case "Azul": this.color1 = "#A9F5F2"; break;
    }
  }
}
```

Guarde la aplicación y en el navegador y pulse sobre la lista para ver el resultado.





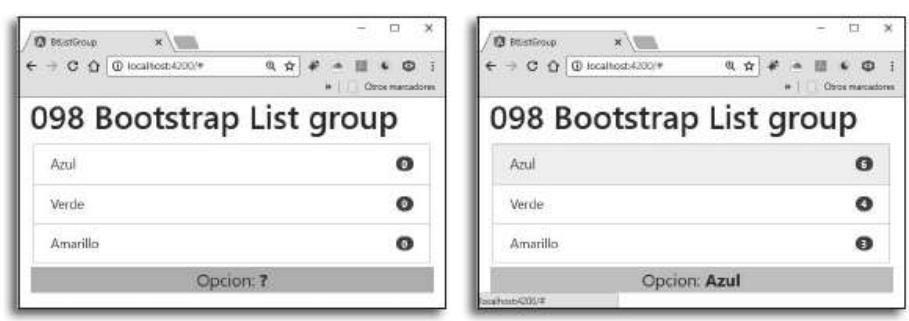
7. Por último, observaremos el ejemplo descrito en el apartado **With badges** y añadiremos un **span** con las clases **badge** al lado de cada descripción de elemento de lista, quedando de la siguiente manera:

```
<div class="list-group p-1">
  <a href="#" class="list-group-item list-group-item-
   action justify-content-between" (click)="onClic-
   k('Azul')">Azul
      <span class="badge badge-default badge-pill">{{
   num1 }}</span></a>
  <a href="#" class="list-group-item list-group-item-ac-
   tion justify-content-between" (click)="onClick('Ver-
   de')">Verde
      <span class="badge badge-default badge-pill">{{
   num2 }}</span></a>
  <a href="#" class="list-group-item list-group-item-ac-
   tion justify-content-between" (click)="onClick('Ama-
   rillo')">Amarillo
      <span class="badge badge-default badge-pill">{{
   num3 }}</span></a>
</div>
<div class="col text-center" [ngStyle]="{'background-co-
   lor': color1, 'font-size': '130%'}">Opcion: <b>{{
   res1 }}</b></div>
```

A la clase **Appcomponent** del archivo **app.component.ts** hay que añadirle ciertas variables para contar los **clics** cada vez que se haga clic sobre un elemento y modificar el método **onClick()** así:

```
export class AppComponent {
  title = '098 Bootstrap List group';
  res1: string = "?";
  color1: string = '#D8D8D8';
  num1: number = 0;
  num2: number = 0;
  num3: number = 0;
  onClick(par: string) {
    this.res1 = par;
    switch (this.res1) {
      case "Verde": this.color1 = "#A9F5BC"; this.num2 +=
   1; break;
      case "Amarillo": this.color1 = "#FFFF00"; this.num3
   += 1; break;
      case "Azul": this.color1 = "#A9F5F2"; this.num1 +=
   1; break;
    }
  }
}
```

Guarde el archivo y visualícelo en el navegador. Pruebe haciendo clic sobre cualquier elemento y vea cómo se incrementan los contadores.



099 Bootstrap: Navbar

Navbar es un elemento contenedor que se muestra en forma de barra y que permite incluir encabezados y opciones que ayudan a la navegación dentro de nuestra aplicación y que ofrece comportamientos **responsive**.

En la página (https://v4-alpha.getbootstrap.com/components/navbar/) que **Bootstrap** dedica a este tema, podemos encontrar explicaciones sobre:

- **How it Works:** contenido fluido por defecto, fácil alineamiento, comportamiento **responsive**, etc.
- **Supported content:** soporta subcomponentes para mostrar la marca o nombre de proyecto, navegación, colapso y expansión de información, controles de formulario con **form-inline**, cadenas de texto, etc.
- **Nav:** los elementos se muestran horizontalmente y alineados siempre que sea posible.
- **Forms:** permite aplicar controles de formulario con **form-inline**.
- **Text:** pueden contener texto.
- **Color schemes:** permite aplicar temas (fondos y colores para textos).
- **Containers:** se puede incluir un **navbar** en un **container** para centrarlo mejor.
- **Placement:** permite ubicar el **navbar** en diferentes posiciones de la página (**top**, **bottom**, **sticky**).
- **Responsive behaviors:** permite mostrar u ocultar elementos asociados a un botón cuando el tamaño de la pantalla cambia.
- **Toggler:** permite posicionar el botón a la derecha o a la izquierda.
- **External content:** permite ocultar información desplegando o colapsando información.

En el siguiente ejercicio implementaremos algunos ejemplos de los que se muestran en la página de **Bootstrap**.

1. En primer lugar, nos ubicamos en **Ej100_angular** y creamos el proyecto **btNavbar** tecleando **ng new btNavbar**.

2. Seguidamente, renombraremos el proyecto para que haga referencia a nuestro número de ejercicio. Para ello, estando ubicados en el directorio **Ej100_angular**, escribiremos lo siguiente:

```
C:\Ej100_angular>rename btNavbar 099_btNavbar
```

Abrimos nuestro proyecto (por ejemplo, con **Atom**) y abrimos el archivo **app.component.ts** para modificar el título **(title)** y poner por ejemplo **"099 Bootstrap btNavbar"**.

3. Tal y como explicamos en el ejercicio anterior, instalaremos las librerías de forma local descargándolas e incorporándolas en el proyecto bajo la carpeta **assets** y también modificaremos el archivo **angular-cli.json** para hacer referencia a estas librerías. Por favor, revise el ejercicio 093 si tiene alguna duda.

Importante

Use **Navbar** para crear, por ejemplo, una barra de opciones para su aplicación que se adapte a cualquier dispositivo y ofrezca las opciones en forma de menú.

4. A continuación, en nuestro archivo **app.component.html** añadiremos un elemento **div** con la clase **container** alrededor de nuestro **title** para que quede así:

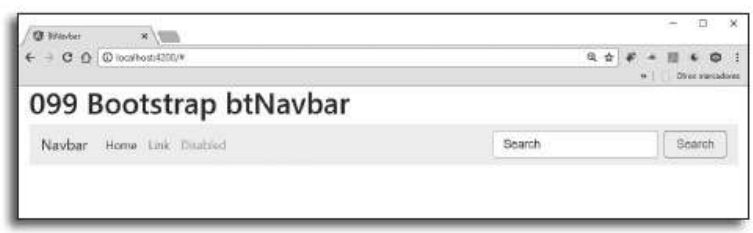
```
<div class="container-fluid">
  <h1>{{title}}</h1>
</div>
```

En la página de **Bootstra /Navbar** localizamos el primer ejemplo asociado a **Supported content** y copiamos su contenido para pegarlo a continuación del cierre del tag **</h1>** que cierra nuestro **title**:

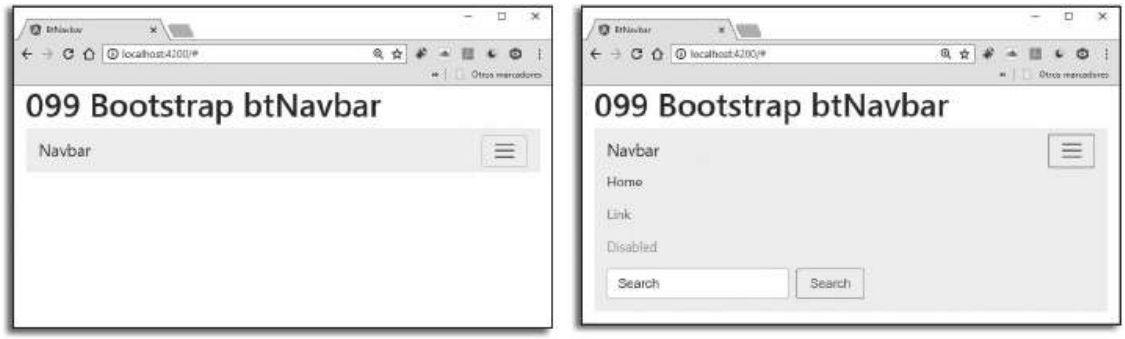
```
<div class="container-fluid">
    <h1>{{title}}</h1>
<nav class="navbar navbar-toggleable-md navbar-light
   bg-faded">
  <button class="navbar-toggler navbar-toggler-right"
   type="button" data-toggle="collapse" data-target="#-
   navbarSupportedContent"
    aria-controls="navbarSupportedContent" aria-expan-
   ded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <a class="navbar-brand" href="#">Navbar</a>
...
```

```
<div class="collapse navbar-collapse" id="navbarSuppor-
   tedContent">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item active">
        <a class="nav-link" href="#">Home
          <span class="sr-only">(current)</span>
        </a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Link</a>
      </li>
      <li class="nav-item">
        <a class="nav-link disabled" href="#">Disa-
   bled</a>
      </li>
    </ul>
    <form class="form-inline my-2 my-lg-0">
      <input class="form-control mr-sm-2" type="text"
   placeholder="Search">
      <button class="btn btn-outline-success my-2 my-
   sm-0" type="submit">Search</button>
    </form>
  </div>
</nav>
```

Guardaremos el archivo y arrancaremos la aplicación ubicándonos en **C:\Ej100_angular\099_btNavbar** y tecleando **ng serve**. Observe el resultado en el navegador accediendo a el URL http://localhost:4200/.



5. Ahora, reduzca el ancho de la página para comprobar cómo se ocultan las opciones y aparece un botón el cual, al pulsarlo, muestra las opciones verticalmente.



6. A continuación, localizaremos el apartado **Color schemes** y modificaremos el tag **<nav>** para que en lugar de la clase **navbar-light** ponga **navbar-inverse bg-inverse** y, tras guardar el archivo, comprobamos el resultado en el navegador.

7. Ahora, por probar otro tema, sustituiremos **navbar-inverse bg-inverse** por **navbar-inverse bg-primary** y veremos cómo queda.

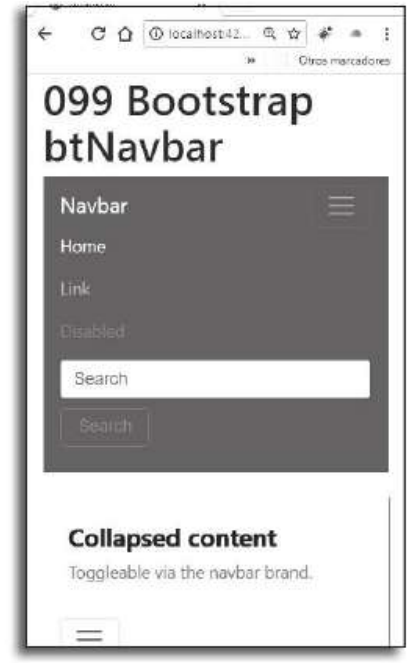
8. Por último, localizaremos el apartado **External content** y copiaremos el ejemplo propuesto a continuación del bloque anterior insertando un **
** entre ambos bloques:

```
...
</div>
<br>
</nav>
   <div class="pos-f-t">
     <div class="collapse" id="navbarToggleExternalCon-
   tent">
       <div class="bg-inverse p-4">
         <h4 class="text-white">Collapsed content</h4>
         <span class="text-muted">Toggleable via the nav-
   bar brand.</span>
       </div>
     </div>
     <nav class="navbar navbar-inverse bg-inverse">
       <button class="navbar-toggler" type="button" da-
   ta-toggle="collapse" data-target="#navbarToggleExtern-
   alContent" aria-controls="navbarToggleExternalContent"
      aria-expanded="false" aria-label="Toggle naviga-
   tion">
       <span class="navbar-toggler-icon"></span>
    </button>
  </nav>
</div>
...
```

Guardamos el archivo y vemos cómo se muestra en el navegador.

9. Ahora, comprobamos cómo al pulsar sobre el botón, se muestra el contenido "oculto".

10. Para comprobar cómo se adaptan los elementos a la pantalla cuando esta cambia de tamaño, reduzca al máximo y despliegue todos los bloques para ver cómo se muestra.



100 Bootstrap: Progress

El componente **Progress** permite mostrar el avance de un proceso visualizando una barra de progreso. Para añadir un valor a la barra, añadimos dicho valor al **div** al que le hemos asociado la clase **progress-bar**. Podemos variar el alto de la barra con la propiedad **height** mediante **"style"** (p. ej., **style="height: 1px;"**). Podemos definir el **background** mediante las clases **bg-success**, **bg-info**, y mostrar varias barras simultáneamente solapadas entre sí y aplicando una animación mediante la clase **progress-bar-animated**. Veamos algún ejemplo de la página de **Bootstrap** (https://v4-alpha.getbootstrap.com/components/progress/).

Importante

La barra de progreso ofrece información sobre el avance de una tarea de forma gráfica y simple.

1. Nos ubicamos en **Ej100_angular**, y creamos el proyecto **btProgress** tecleando **ng new btProgress**.

2. Ahora, renombraremos el proyecto desde el directorio **Ej100_angular**, escribiendo lo siguiente:

```
C:\Ej100_angular>rename btProgress 100_btProgress
```

   Abrimos el proyecto y en **app.component.ts** modificamos **title** con **"100 Bootstrap Progress"**.

3. Instalaremos las librerías localmente. Por favor, revise el ejercicio 093.

4. En **app.component.html** añadimos un **div** con la clase **container** alrededor de nuestro **title** así:

```
<div class="container"><h1>{{title}}</h1></div>
```

   Localizamos el primer ejemplo de **Progress**, y copiamos el bloque que muestra el **25%** para pegarlo en **app.component.html** detrás de **</h1>** para que el archivo quede de la siguiente manera:

```
<div class="container"><h1>{{title}}</h1>
    <div class="progress">
        <div class="progress-bar" role="progressbar" style="wi-
    dth: 25%" aria-valuenow="25" aria-valuemin="0" aria-value-
    max="100"></div>
    </div>
</div>
```

Guardamos el archivo y arrancamos la aplicación desde **C:\Ej100_angular\100_btProgress** tecleando **ng serve**. Observe el resultado en el navegador en el **URL** http://localhost:4200/.

5. A continuación, modificamos **app.component.ts** para que la clase **AppComponent** quede así:

```
export class AppComponent {
  title = '100 Bootstrap Progress';
  valorN: number = 20;
  valorT: string = "20%";
  onChange($event) {
    this.valorN = parseInt($event.target.value);
    this.valorT = this.valorN + "%";
  }
}
```

El evento para el input lo añadiremos a nuestra página así:

```
<div class="container-fluid">

    <h1>{{title}}</h1>

    <div class="progress m-2">

        <div class="progress-bar" role="progressbar" [ngS-
    tyle]="{'width': valorT}" aria-valuenow="25" aria-value-
    min="0" aria-valuemax="100"></div>

    </div>

    <br><input (change)="onChange($event)">

</div>
```

Hemos añadido un **[ngStyle]** para que el ancho (**width**) sea variable según el valor que se introduzca en el input y un evento **"onChange()"** para poder redibujar la barra de progreso.

6. Guardamos el archivo y probamos en el navegador con los valores 30 y 80.

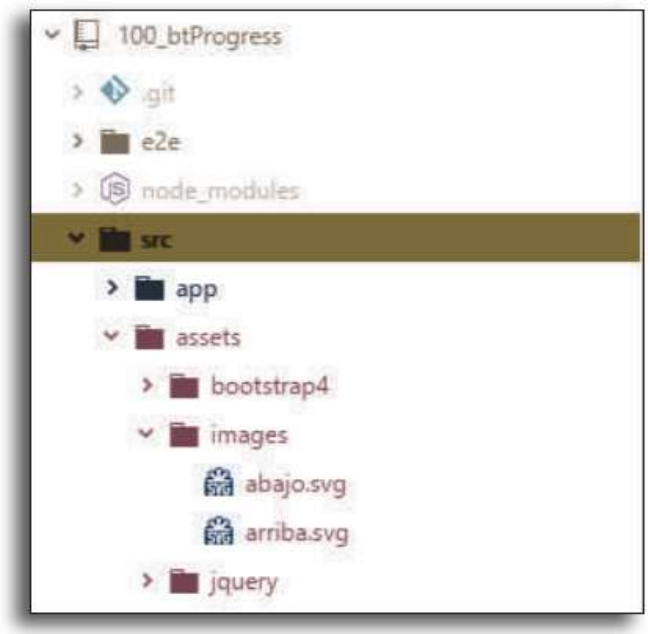
7. Añadimos el contenido de la variable **valor** modificando el **div** que contiene la clase **progress-bar**:

```
<div class="progress-bar" role="progressbar" [ngStyle]="{'wid-
    th': valorT}" aria-valuenow="25" aria-valuemin="0"

    aria-valuemax="100">{{ valorT }}</div>
```

Guardamos el archivo y vemos cómo queda en el navegador con el valor **50**.

8. Ahora añadiremos un par de botones para modificar el tamaño de la barra de progreso. Escogemos dos imágenes para **flecha arriba** y **abajo** para aumentar y disminuir, respectivamente, dicho tamaño.

9. Creamos la carpeta **src/images** y en la misma ponemos las imágenes (p. ej., **arriba.svg** y **abajo.svg**).



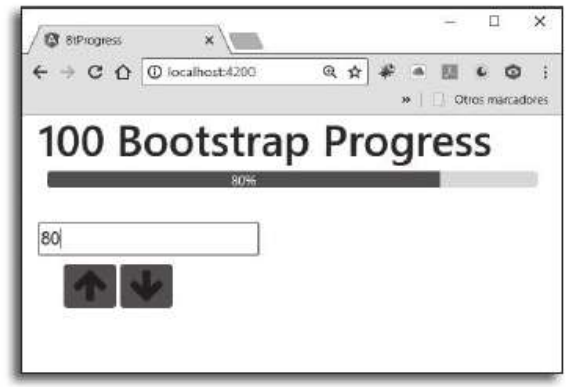
10. Seguidamente, incluimos las imágenes con el siguiente codigo detrás del tag **input** anterior:

```
<div class="container m-2">
    <button class="btn btn-primary btn-sm" type="button"
   name="button" (click)="onClick('+')" (keyup.down)="on-
   Click('+')">
      <img src="assets/images/arriba.svg" alt="" width="30"
   height=" 30"></button>
    <button class="btn btn-primary btn-sm" type="button" na-
   me="button" (click)="onClick('-')">
      <img src="assets/images/abajo.svg" alt="" width="30"
   height=" 30"></button>
</div>
```

Modificaremos la clase **AppComponent** en **app.component.ts** para que quede así:

```
export class AppComponent {
  title = '100 Bootstrap Progress';
  valor: string = "20";
  valorN: number = 20;
  valorT: string = "20%";
  incr: number = 1;
  onChange($event) {
    this.valorN = parseInt($event.target.value);
    this.valorT = this.valorN + "%";
  }
  onClick(action: string) {
    console.log("action " + action);
    if (action == "+") { this.incr = 1; } else { this.incr =
    -1; }
    this.valorN = parseInt(this.valor) + this.incr;
    if (this.valorN < 0) this.valorN = 0;
    if (this.valorN > 100) this.valorN = 100;
    this.valor = this.valorN.toString();
    this.valorT = this.valorN + "%";
  }
}
```

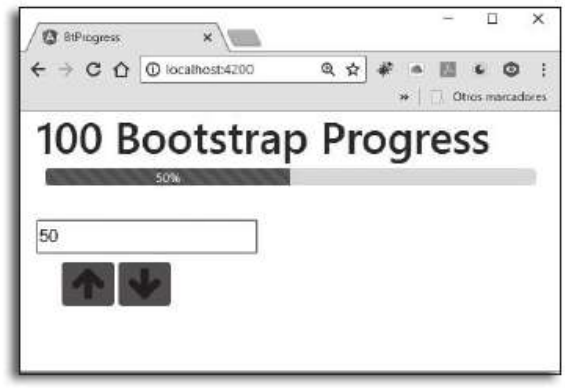
Probamos las flechas y vemos como la barra aumenta o disminuye según pulsemos un botón u otro.



11. Ahora, añadimos la clase **progress-bar-striped** al **div** que posee la clase **progress-bar** así:

```
<div class="progress-bar progress-bar-striped" role="pro-
    gressbar" [ngStyle]="{'width': valorT}" aria-valuenow="25"
    aria-valuemin="0"
     aria-valuemax="100">
```

Guarde el archivo y observe cóomo queda.



12. Por último, añadimos la clase **progress-bar-animated** y vemos cómo queda.