

Full Stack Developer

Angular

Quedan rigurosamente prohibidas, sin la autorización escrita de los titulares del Copyright, bajo las sanciones establecidas en las leyes, la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares de ella mediante alquiler o préstamo públicos. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, www.cedro.org) si necesita fotocopiar o escanear algún fragmento de esta obra.

INICIATIVA Y COORDINACIÓN

DEUSTO FORMACIÓN

COLABORADORES

Realización:

ISCA Training & Consulting S.L.

Elaboración de contenidos:

Pedro Jiménez Castela

Desarrollador senior Full Stack JavaScript, TypeScript, Angular, React, NodeJS, MongoDB y JS Testing (Jest y Cypress).

Instructor en formación oficial MongoDB y NodeJS.

Instructor del CFTIC de la Comunidad de Madrid.

Certificado como: MongoDB DBA n.º 582-916-826 y MongoDB DEV n.º 560-512-224.

Supervisión técnica y pedagógica:

MG AGNESI TRAINING

Coordinación editorial:

MG AGNESI TRAINING

© MG AGNESI TRAINING, S.L.

Barcelona (España), 2022

Primera edición: septiembre 2022

ISBN: 978-84-19142-31-3 (Obra completa)

ISBN: 978-84-19142-43-6 (Angular)

Depósito Legal: B 3409-2022

Impreso por:

SERVINFORM

Avenida de los Premios Nobel, 37

Polígono Casablanca

28850 Torrejón de Ardoz (Madrid)

Printed in Spain

Impreso en España

Esquema de contenido

1. Introducción a Angular

- 1.1. El *framework* Angular
- 1.2. Instalación de Angular CLI

2. *Binding*, componentes y plantillas

- 2.1. Interpolación en Angular
- 2.2. *Property binding*
- 2.3. *Event binding* en Angular

3. Directivas y *pipes* en Angular

- 3.1. Directivas en Angular
- 3.3. *Pipes* en Angular

4. Comunicación entre componentes y servicios en Angular

- 4.1. Comunicación entre componentes
- 4.2. Servicios en Angular

5. Routing y formularios en Angular

- 5.1. *Routing* en Angular
- 5.2. Formularios reactivos en Angular

6. Comunicaciones HTTP y despliegue a producción

- 6.1. Comunicaciones HTTP en Angular
- 6.2. Asincronía y navegación programática
- 6.3. Despliegue a producción de aplicaciones Angular

Introducción

El modelo SPA (siglas de *single page application*, “aplicación de página única”) es uno de los más utilizados en la actualidad para todo tipo de aplicaciones web y existen varios *frameworks* dedicados a este tipo de desarrollos. Angular, mantenido por Google, es uno de los más utilizados.

En este módulo vamos a aprender cómo implementar SPA mediante proyectos Angular, comenzando por el aprendizaje de la instalación de su herramienta de línea de comandos y la generación de proyectos y su estructura.

Continuaremos aprendiendo a utilizar componentes, técnicas de *binding*, directivas y *pipes* para, seguidamente, introducirnos en la comunicación de los diferentes componentes y el uso de servicios como la centralización de datos.

También aprenderemos a utilizar el sistema de navegación, la librería de formularios reactivos, el empleo de peticiones a API externas y, finalmente, el proceso de despliegue a producción de las aplicaciones.

1. Introducción a Angular

Comenzamos en esta primera unidad relacionada con el *framework* Angular, la otra gran alternativa en el desarrollo de aplicaciones *single page application* (SPA) junto a React, todo un entorno de trabajo imprescindible para los desarrolladores full stack.

En primer lugar, describiremos las principales características de Angular y sus usos, así como la instalación de Angular CLI, la herramienta de línea de comandos para el desarrollo con este *framework*.

Posteriormente, generaremos un primer proyecto con Angular CLI para aprender su estructura de archivo y comenzar a desarrollar componentes para aplicaciones con esta potente solución.

1.1. El *framework* Angular

Angular es un framework mantenido por la compañía Google para aplicaciones *single page application* cuya principal característica es el uso de módulos y componentes en los que se emplea TypeScript como lenguaje para el desarrollo de su lógica.

El resultado de los proyectos Angular una vez compilados y minificados es un conjunto de archivos JavaScript, HTML y CSS que son distribuidos en servicios web al navegador para que sean ejecutados en este siguiendo la arquitectura *client-side rendering*.

Gracias fundamentalmente al uso de librerías propias y de terceros ampliamente testadas por la comunidad y de gran fiabilidad, Angular incorpora una serie de ventajas en su utilización para proyectos *frontend*, entre las que destacan las siguientes:

- **Formularios reactivos:** a través de la librería propia de Angular Reactive Forms Module, se proporcionan funcionalidades complejas para la mayoría de los casos de uso de formularios en las aplicaciones complejas.
- **Módulo routing:** de manera nativa, Angular incorpora el módulo necesario para implementar el enruteado o la navegación mediante rutas para implementar SPA en las aplicaciones.



Para saber más

En la primera versión de Angular, conocida como Angular JS, no se utilizaba TypeScript y la estructura de enlace entre vista y componente era diferente.

Desde la versión dos, Angular usa TypeScript de manera obligatoria para que el código sea más robusto.

- **Peticiones HTTP:** Angular dispone de un módulo propio para las peticiones HTTP denominado `HttpModule`, con una gran cantidad de soluciones para realizar peticiones desde la aplicación a API Rest.
- **Reactividad:** Angular incorpora en sus proyectos la librería RxJS para aportar reactividad mediante los objetos observables en la comunicación de datos y estados entre componentes de la aplicación.

Otra de las grandes ventajas de Angular es la gran documentación del *framework* disponible en su sitio web (Figura 1.1), que no solo incluye los conceptos e implementaciones básicos, sino que profundiza en casos de uso complejos para resolver cualquier tipo de requisito de las aplicaciones.



Figura 1.1

Sitio web de la documentación oficial del *framework* Angular.

En cuanto a productividad, Angular proporciona desde la versión dos la herramienta de línea de comandos Angular CLI, que permite el desarrollo rápido de los principales elementos que se incluyen en los proyectos Angular, así como la configuración y proceso de build de los mismos, como aprenderemos a continuación.

1.2. Instalación de Angular CLI

La instalación de la herramienta Angular CLI se puede realizar de manera global para todo el equipo, de manera que podrá utilizarse en cualquier ubicación para trabajar con proyectos Angular. Se utiliza el gestor npm y simplemente hay que abrir una consola o terminal y teclear el siguiente comando:

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

Una vez llevado a cabo el proceso, podemos comprobar su correcta instalación con el siguiente comando:

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

Ahora ya podemos generar un proyecto Angular. Para ello, completamos en cualquier ubicación de nuestro equipo el siguiente comando, que incluye el nombre del directorio donde se generarán todos los archivos:

```
ng new practica1
```

Al completar este comando, Angular CLI nos consulta si queremos instalar Routing; pulsamos Intro para confirmar. A continuación, nos pide seleccionar los estilos, y de nuevo pulsamos Intro para confirmar CSS. En ese momento, comienza la generación del proyecto.

Una vez finalizado el proceso, abrimos el proyecto con Visual Studio Code y podemos comprobar toda su estructura de archivos (Figura 1.2), entre los que destaca su módulo raíz, el archivo app.module.ts y su componente raíz, compuesto por los archivos .html, .css y .ts, que comparten el nombre app.component.

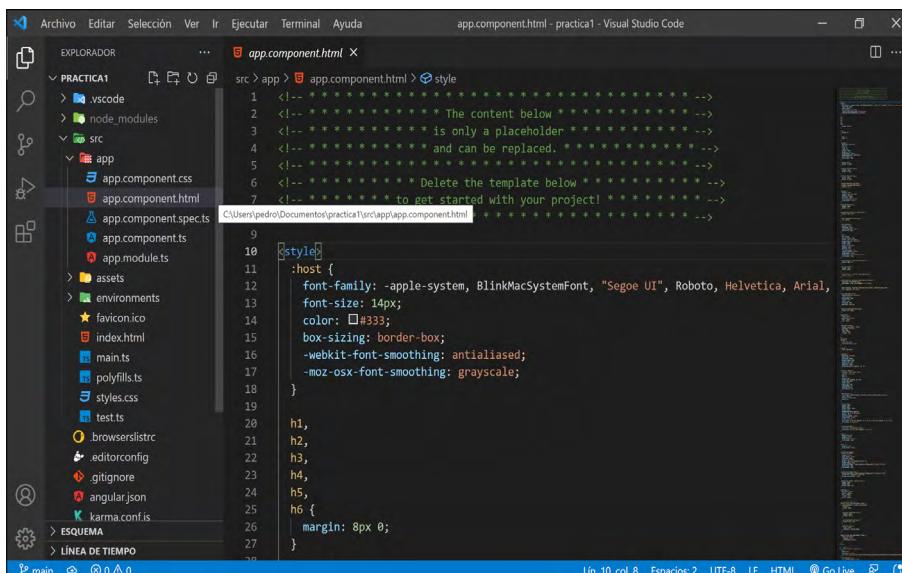


Figura 1.2

Estructura de archivos de un proyecto Angular.

Para comprobar la plantilla de este primer componente raíz que Angular CLI crea por defecto, podemos modificar todo el código del archivo app.component.html con la siguiente línea:

```
<h1>¡Hola Mundo!</h1>
```

A continuación, vamos a compilar y levantar un servidor con nuestra aplicación en desarrollo, para lo cual, en la terminal integrada de Visual Studio Code, usamos el siguiente comando de Angular CLI:

```
ng serve -o
```

Figura 1.3

Renderizado de un proyecto Angular en el navegador en fase de desarrollo.

Una vez que se produce la compilación, comprobamos el resultado en el navegador en la dirección <http://localhost:4200> (Figura 1.3).



Como la mayoría de *frameworks* JavaScript, Angular basa su empleo en el uso de componentes, y para generarlos usamos Angular CLI. Desde otra sesión de la terminal, vamos a completar el siguiente comando para generar un componente *footer* en nuestro proyecto:

```
ng generate component footer
```

Una vez creado, accedemos al directorio con su nombre y, en el archivo *footer.component.ts*, escribimos en su clase la declaración de una propiedad, variable que estará disponible solo en ese componente. Completamos el siguiente código:

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

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

export class FooterComponent implements OnInit {

  company: string = 'ACME S.A.';
```

```
constructor() { }

ngOnInit(): void {
}

}
```

Para usar esa propiedad, en la plantilla del archivo footer.component.html escribimos el siguiente código:

```
<footer>
  <p>&copy; {{company}}</p>
</footer>
```

Y a continuación, para renderizar este componente en el componente raíz, modificamos el código del archivo app.component.html añadiendo su etiqueta de la siguiente forma:

```
<h1>¡Hola Mundo!</h1>
<app-footer></app-footer>
```

También podemos añadir estilos CSS globalmente, como veremos más adelante, o en el propio componente. Por ejemplo, vamos a añadir los estilos en el archivo footer.component.css de la siguiente forma:

```
footer {
  width: 100%;
  position: fixed;
  bottom: 0;
  left: 0;
  padding: 1rem;
  background-color: #0fd3d3;
}

footer p {
  text-align: center;
  color: white;
}
```

De esta manera, al compilar automáticamente el proyecto, Angular nos mostrará el nuevo componente con los datos procedentes del archivo TypeScript, la estructura de la plantilla HTML y los estilos CSS (Figura 1.4).

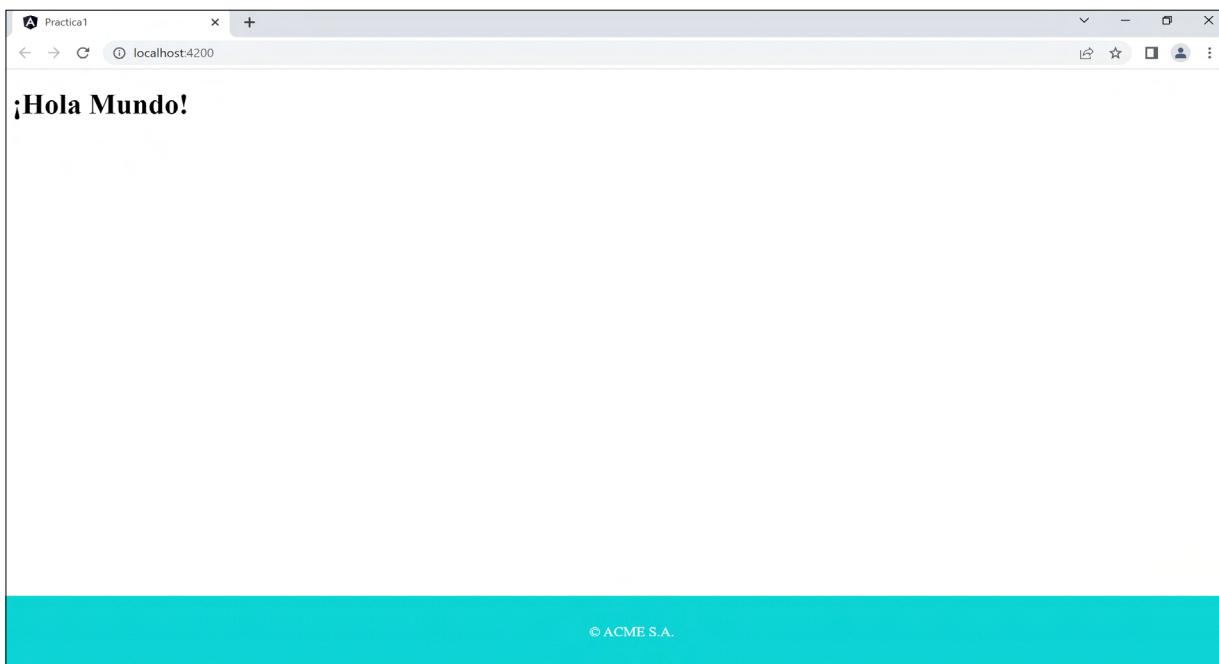


Figura 1.4 Mediante la implementación de la plantilla, los estilos y la lógica de los tres archivos de cada componente, así como de test unitario en el caso de usarlos, Angular compondrá las interfaces de la aplicación
Renderizado de un componente Angular en el navegador a partir del código de sus tres archivos.



Resumen

- Angular es un *framework* mantenido por la compañía Google para aplicaciones *single page application* (SPA) que emplea TypeScript como lenguaje para el desarrollo de la lógica de cada componente.
- Angular proporciona la herramienta de línea de comandos Angular CLI, que permite el desarrollo rápido de los principales elementos que se incluyen en los proyectos, así como su compilado final.
- Los componentes Angular están compuestos de cuatro archivos: HTML para la plantilla, CSS para su estilo, TypeScript para la lógica y un cuarto opcional también TypeScript por si se requiere realizar test unitarios.

2. Binding, componentes y plantillas

Continuamos con el aprendizaje del *framework* Angular, profundizando en la distribución de archivos de sus componentes en cuanto a la separación de lógica y plantilla de vista mediante las técnicas de *binding*.

En primer lugar, ampliaremos la técnica de interpolación, por la cual se pueden introducir dentro de los archivos HTML expresiones que rendericen el contenido de forma lógica.

Posteriormente, veremos las otras dos grandes técnicas de *binding*: *property binding* para el uso de propiedades o atributos, y *event binding* para asociar eventos en los elementos HTML con métodos del componente lógico TypeScript.

2.1. Interpolación en Angular

En los entornos JavaScript denominamos interpolación a la incorporación dentro de un elemento de una expresión cuya evaluación devuelve un tipo de dato renderizable. Esta denominación se usa para Angular, que además utiliza la sintaxis de dobles llaves.

Anteriormente ya pudimos comprobar que, si introducimos en los archivos HTML del componente el nombre de una propiedad de la clase entre dobles llaves, se renderizaba su valor en el elemento. Es importante tener en cuenta que la interpolación en Angular resuelve una expresión llamada template expression y debe cumplir las siguientes características:

- Angular la resuelve y devuelve un string.
- No modifica el estado.
- No soporta el uso de operadores de asignación, incrementales, lógicos, “new”, “typeof”, “instanceof” o expresiones encadenadas con punto y coma.

Se trata, por tanto, de introducir en HTML expresiones sencillas y realizar la complejidad de la lógica en el archivo TypeScript. Vamos a am-



Para saber más

Aunque pueden parecer demasiadas limitaciones en las *template expressions*, no suponen ningún problema, ya que en el archivo TypeScript del componente podemos realizar toda la lógica que necesitemos.

pliar los ejemplos que ya conocemos en nuestro proyecto practica1, que abrimos con Visual Studio Code. Abrimos también la terminal y creamos un nuevo componente con el siguiente comando:

```
ng generate component interpolation
```

Ahora vamos a crear en el archivo interpolation.component.ts un objeto en la clase con propiedades de un actor y un método para devolver su edad. Para ello, modificamos el código de la siguiente forma:

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

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

  actor: any = {
    apellidos: 'De Niro',
    nombre: 'Robert',
    fechaNacimiento: new Date(1943, 7, 17)
  }

  constructor() { }

  ngOnInit(): void {
  }

  getEdad(): number {
    const edad = (new Date().getTime() - this.actor.fechaNacimiento.getTime()) / (365 * 24 * 60 * 60 * 1000);
    return Math.ceil(edad);
  }
}
```

Como podemos observar, toda la lógica se encuentra en el archivo TypeScript, pero podemos utilizar sus resultados en la plantilla gracias a la interpolación. Para ello, abrimos el archivo interpolation.component.html y escribimos el siguiente código:

```
<div class="card">
  <h1>Ejemplo de interpolation</h1>
  <p>Nombre: {{actor.nombre}}</p>
  <p>Apellidos: {{actor.apellidos}}</p>
  <p>Edad: {{getEdad()}}</p>
</div>
```

Como vemos, es muy sencillo añadir las expresiones a la interpolación para conseguir el resultado esperado. Ahora vamos a añadir unos estilos para que puedan ser utilizados en cualquier componente; para ello, abrimos el archivo styles.css del directorio src y añadimos el siguiente bloque:

```
@import url('https://fonts.googleapis.com/css?family=Lato&display=swap');

* {
  padding: 0;
  margin: 0;
  box-sizing: border-box;
}

body {
  font-family: 'Lato', sans-serif;
  color: #718096;
  background-color: #fafafa;
}

h1 {
  font-size: 24px;
  color: #008489;
  margin-bottom: 12px;
}

.card {
  max-width: 760px;
  padding: 1.5rem;
  margin: 1rem auto;
  border: rgba(0,0,0,0.08);
  box-shadow: 0 2px 4px 0 rgba(0,0,0,0.08);
  background-color: white;
}
```

Y finalmente vamos a añadir la etiqueta del nuevo componente en el componente raíz, para lo cual modificamos app.component.html de la siguiente forma:

```
<app-interpolation></app-interpolation>
```

Para levantar el servidor con el proyecto, utilizamos una vez más el comando por defecto:

```
ng serve -o
```

Y comprobamos en el navegador los resultados de nuestro componente (Figura 2.1).

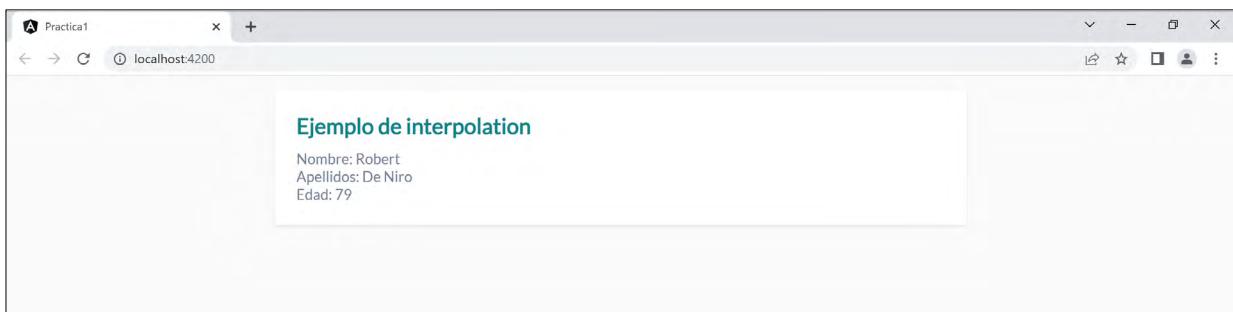


Figura 2.1

Renderizado de un componente Angular en el navegador mediante interpolación.

2.2. **Property binding**

Como su nombre indica, la técnica *property binding* enlaza los datos de la lógica del archivo TypeScript del componente con la plantilla HTML a través de propiedades del DOM que son introducidas como atributos en los elementos. La manera sintáctica en la que se implementa es rodeando los atributos con corchetes, y añadiendo como valor una *template expression*.

El valor que devuelva la expresión será el asignado a esa propiedad del DOM; de esta manera, se podrá trabajar dinámicamente sobre cada elemento. Para comprobarlo de manera práctica, vamos a crear en nuestro proyecto practica1 un nuevo componente con el siguiente comando:

```
ng generate component propertyBinding
```

Una vez generado, vamos al componente TypeScript en el archivo property-binding-component.ts y modificamos su código de la siguiente manera:

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

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

  isDisabled: boolean = true;

  constructor() { }
```

```

ngOnInit(): void {
  setTimeout(() => {
    this.disabled = false;
  }, 5000)
}

}

```

Podemos observar que hemos creado una propiedad “disabled” que comienza en “true”; mediante un “setTimeout”, cuando pasen cinco segundos pasará a “false”. Aprovechamos el hook “ngOnInit()” de Angular para invocar el “setTimeout”, ya que se ejecuta automáticamente cuando el DOM está renderizado al inicio del ciclo de vida del componente.

Ahora vamos a usar esa propiedad booleana con la técnica de *property binding*, para lo cual escribimos el siguiente código en el archivo property-binding-component.html:

```

<div class="card">
  <h1>Ejemplo de property binding</h1>
  <p>Lorem ipsum dolor, sit amet consectetur adipisicing elit.  

      Nihil ex quae magnam rem, qui pariatur blanditiis ut  

      quisquam inventore a!</p>
  <button [disabled]="disabled">Aceptar</button>
</div>

```

A continuación, completamos nuestro archivo de estilos styles.css con los siguientes bloques para los botones:

```

button {
  font-size: 16px;
  padding: 10px;
  margin: 20px 0 20px 5px;
  color: white;
  background-color: #008489;
  border: #008489 solid 1px;
  cursor: pointer;
  transition: background-color 400ms ease-in-out;
}

button:disabled {
  background-color: lightgray;
  cursor: not-allowed;
}

```

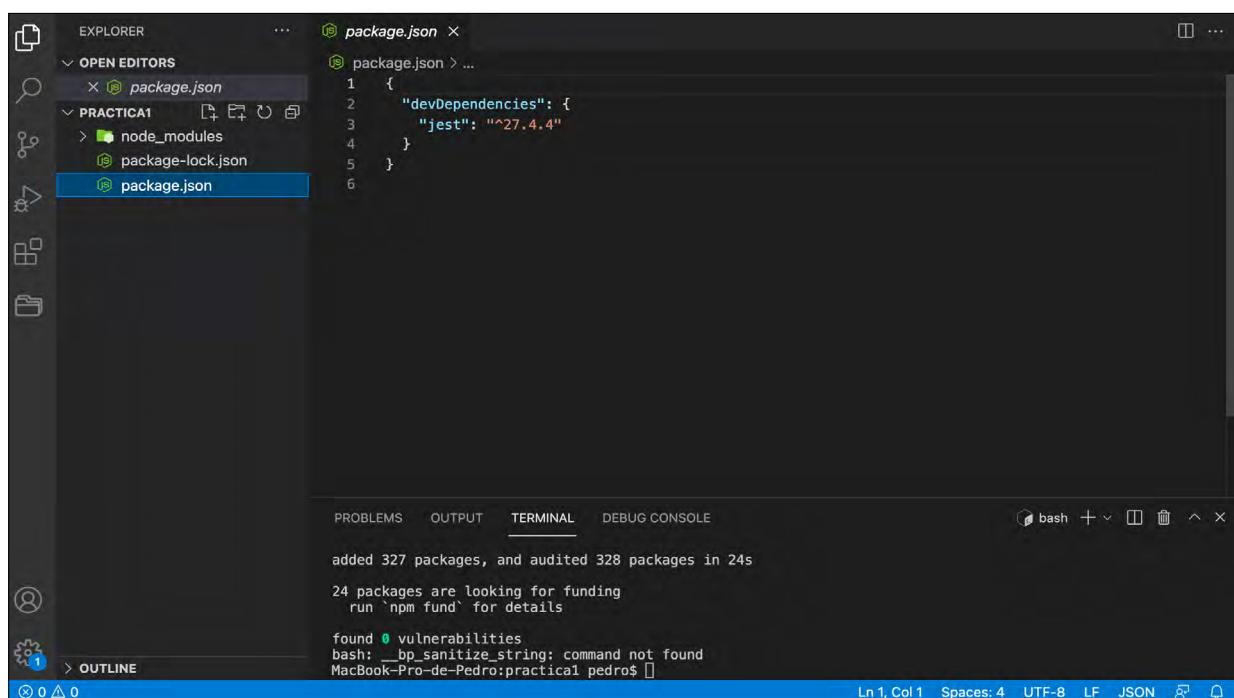
Y finalmente añadimos la etiqueta de este componente en el componente raíz modificando de nuevo app.component.html de la siguiente forma:

```
<app-interpolation></app-interpolation>
<app-property-binding></app-property-binding>
```

Así, nuestro navegador renderiza el nuevo componente y podemos comprobar que, a los cinco segundos, cambia el estado del botón (Figura 2.2).

Figura 2.2

Renderizado de un componente Angular en el navegador mediante *property binding*.



2.3. Event binding en Angular

La técnica *event binding* utiliza eventos del DOM para ejecutar las expresiones que se le asocien. Para ello, emplea la sintaxis de paréntesis, con los que rodea el evento en el elemento HTML y a los cuales se pasa normalmente la invocación de un método de la clase.

Para practicar esta técnica, vamos a crear en nuestro proyecto `practica1` un nuevo componente con el siguiente comando:

```
ng generate component eventBinding
```

Una vez creado, vamos al componente TypeScript en el archivo event-binding-component.ts y modificamos su código de la siguiente manera:

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

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

  timestamp: string = '';

  constructor() { }

  ngOnInit(): void {
  }

  setTimestamp(): void {
    const dd = ('0' + new Date().getDate()).slice(-2);
    const MM = ('0' + (new Date().getMonth() + 1)).slice(-2);
    const yy = ('0' + new Date().getFullYear()).slice(-2);
    const hh = ('0' + new Date().getHours()).slice(-2);
    const mm = ('0' + new Date().getMinutes()).slice(-2);
    const ss = ('0' + new Date().getSeconds()).slice(-2);
    const date = dd + '/' + MM + '/' + yy;
    const time = hh + ':' + mm + ':' + ss;
    this.timestamp = `Aceptado el ${date} a las ${time}`;
  }
}
```

Podemos ver que disponemos de un método que va a *setear* una marca de tiempo en una variable, así que vamos a asignar ese método al evento “clic” en la plantilla. Abrimos el archivo event-binding-component.html y escribimos el siguiente código:

```
<div class="card">
  <h1>Ejemplo de event binding</h1>
  <button (click)="setTimestamp()">Aceptar</button>
  <p>{{timestamp}}</p>
</div>
```

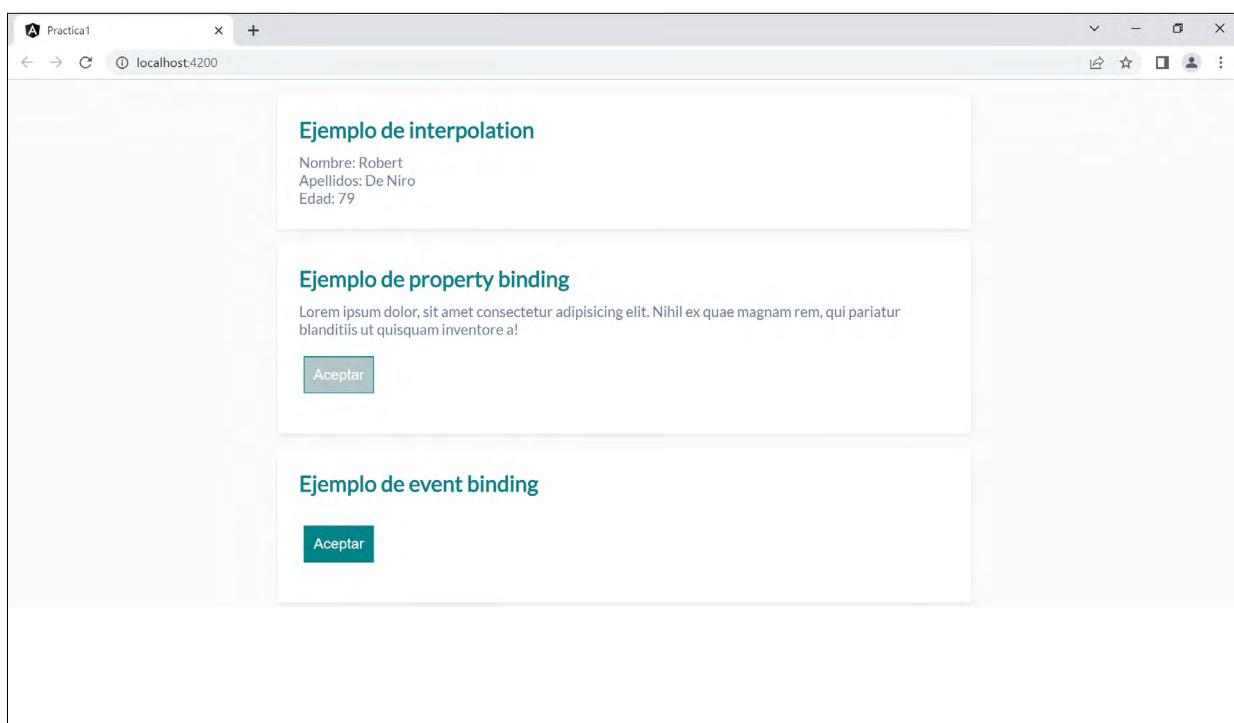
Aquí podemos ver que el evento “clic” se introduce entre paréntesis en esta técnica. A continuación, vamos a añadir la etiqueta del nuevo componente en app.component.html:

```
<app-interpolation></app-interpolation>
<app-property-binding></app-property-binding>
<app-event-binding></app-event-binding>
```

Y ya podemos comprobar la lógica de este componente en el navegador (Figura 2.3).

Figura 2.3

Renderizado de un componente Angular en el navegador mediante *event binding*.



Hay una cuarta técnica de *binding* en Angular, *two way binding*, usada en los formularios, como veremos más adelante; pero, en el resto de las situaciones, las tres técnicas estudiadas se utilizarán frecuentemente.



Resumen

- Las técnicas de *binding* permiten el enlace entre la lógica desarrollada en los archivos TypeScript del componente y la plantilla desarrollada en los archivos HTML.
- Las *template expressions* se usan en Angular para devolver un *string* que es renderizado en el caso de la interpolación, o asignado como valor a un atributo de propiedad del DOM en el caso del *property binding*, o a un evento en el *event binding*.
- Sintácticamente, las técnicas de *binding* utilizan dobles llaves en la interpolación, corchetes en *property binding* y paréntesis en *event binding*.

3. Directivas y *pipes* en Angular

En esta unidad de aprendizaje del *framework* Angular estudiaremos dos elementos esenciales para trabajar en sus componentes, las directivas y los *pipes*.

En primer lugar, aprenderemos de forma práctica qué son las directivas, cuál es su sintaxis y qué casos de uso principales permite su implementación en las plantillas de los componentes Angular.

Posteriormente, aprenderemos el uso de *pipes* en Angular, un mecanismo de formateo de datos en las plantillas de los componentes incorporado en el *core* de herramientas y utilidades de este *framework*.

3.1. Directivas en Angular

Las directivas son un mecanismo o elemento de Angular que permite implementar soluciones lógicas habituales en la programación de componentes, como, por ejemplo, que un elemento se visualice condicionalmente o que, a partir de un set de datos, se genere una lista en los elementos de la plantilla HTML.

Entre las directivas de Angular se distinguen dos tipos principales:

- **Directivas estructurales:** modifican el DOM y en su sintaxis son precedidas por el signo de asterisco.
- **Directivas de atributo:** no modifican el DOM, pero implementan valores a atributos de forma dinámica de forma parecida al *property binding*.

Entre las directivas estructurales, una de las más utilizadas es “*ngIf”, que permite renderizar un elemento de forma condicional. Para comprobar de manera práctica su uso, vamos a generar un nuevo proyecto Angular. Para ello, completamos de nuevo en cualquier ubicación de nuestro equipo el siguiente comando:

```
ng new practica2
```



Para saber más

Incluso la etiqueta de los componentes Angular es a su vez una directiva, ya que permite renderizar en un elemento HTML el contenido y la lógica desarrollados en cada archivo TypeScript, HTML y CSS.

Una vez más, Angular CLI nos consulta si queremos instalar Routing, y pulsamos Intro para confirmar; seguidamente, nos pide seleccionar los estilos, y de nuevo pulsamos Intro para confirmar CSS como hoja de estilos.

Una vez finalizado el proceso, abrimos el proyecto con Visual Studio Code y creamos un nuevo componente, para lo cual escribimos en la terminal el siguiente comando:

```
ng generate componente directivaNgIf
```

En este nuevo componente vamos a usar un paquete de Angular para poder utilizar formularios. Para ello, debemos importarlo en el módulo declarado en el archivo app.module.ts. Los módulos permiten agrupar los diferentes componentes para posteriormente extraer su lógica y su presentación en la compilación; por tanto, es donde se implementan las dependencias de estos.

Para instalar este paquete de formularios (llamado también módulo) con nombre FormsModule, vamos a modificar el archivo app.module.ts de la siguiente forma:

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

...
imports: [
  BrowserModule,
  AppRoutingModule,
  FormsModule
],
...
```

Ahora vamos a crear en la clase de nuestro componente lógico, en el archivo TypeScript directiva-ng-if.component.ts, una simple propiedad para almacenar de un valor que provendrá de un *input* de formulario. Modificamos su código de la siguiente forma:

```
...
export class DirectivaNgIfComponent implements OnInit {
  points: any = null;
```

```
constructor() { }

ngOnInit(): void {
}

}
```

Y en la plantilla, vamos a escribir un input para asociar a esa propiedad mediante “ngModel”, que usamos gracias a la importación del módulo de formularios, y vamos a añadir en un elemento la directiva “*ngIf”. A esta le pasamos una expresión condicional, de manera que si retorna “true” renderizará el elemento y, en caso contrario, lo eliminará. Escribimos el siguiente código en el archivo directiva-ng-if.component.html:

```
<div class="card">
  <h1>Ejemplo de directiva *ngIf</h1>
  <p>Introduzca la puntuación</p>
  <input type="number" [(ngModel)]="points">
  <p *ngIf="points">La puntuación obtenida es {{points}}</p>
</div>
```

A continuación, añadimos en el archivo styles.css el código CSS de la practica1, al cual añadimos el siguiente bloque para los elementos *input*:

```
...
input {
  width: 100%;
  font-size: 16px;
  padding: 10px;
  margin: 0 12px 12px 0;
  border: none;
  box-shadow: inset 0 2px 4px 0 rgba(0,0,0,0.08);
  color: #6a6a6a;
  font-family: 'Lato', sans-serif;
}
```

Por último, sustituimos en el componente raíz todo el código por la etiqueta de nuestro componente, modificando todo el archivo app.component.html con el siguiente código:

```
<app-directiva-ng-if></app-directiva-ng-if>
```

Ahora ya podemos compilar el proyecto y comprobar en el navegador (Figura 3.1) que se visualizará el texto con los puntos solo cuando el usuario complete el *input*.



Figura 3.1

Componente Angular con la directiva “*ngIf” en el navegador.

Otra de las directivas Angular más utilizadas es “*ngFor”, que permite la generación de elementos por iteración de una propiedad de la clase del componente. Para su implementación, vamos a crear un nuevo componente en nuestro proyecto con el siguiente comando:

```
ng generate componente directivaNgFor
```

Una vez generado, vamos a añadir en la clase del componente un set de datos, modificando el archivo directiva-ng-for.component.ts de la siguiente forma:

```
...
export class DirectivaNgForComponent implements OnInit {

  customers: Array<any> = [
    {name: 'Orange', cif: 'C12345678'},
    {name: 'Iberdrola', cif: 'B12345678'},
    {name: 'BBVA', cif: 'A12345678'},
  ]

  constructor() { }

  ngOnInit(): void {
  }

}
```

Y en la plantilla, en el archivo directiva-ng-for.component.html, podemos usar “ngFor” para generar un elemento “tr” por cada objeto del array “customers” y además disponer de una variable “customer”; por

eso la declaramos en singular con cada objeto para poder renderizar sus valores por interpolación. Modificamos su código de la siguiente forma:

```
<div class="card">
  <h1>Ejemplo de directiva *ngFor</h1>
  <table>
    <tr>
      <th>Nombre</th>
      <th>CIF</th>
    </tr>
    <tr *ngFor="let customer of customers">
      <td>{{customer.name}}</td>
      <td>{{customer.cif}}</td>
    </tr>
  </table>
</div>
```

Añadimos un bloque de estilos para tablas en el archivo styles.css:

```
...
table {
  width: 100%;
  margin-bottom: 24px;
}

th {
  padding: 10px 20px;
  background-color: #008489;
  color: white;
  text-align: left;
}

td {
  padding: 10px 20px;
  border: 1px solid #008489;
}
```

Y modificamos app.component.html para añadir su etiqueta:

```
<app-directiva-ng-if></app-directiva-ng-if>
<app-directiva-ng-for></app-directiva-ng-for>
```

Ahora podemos comprobar el resultado en el navegador (Figura 3.2) y observar que por cada objeto se genera un elemento con sus datos,

consiguiendo iterar resultados a partir de un set de datos que pueden ser dinámicos.

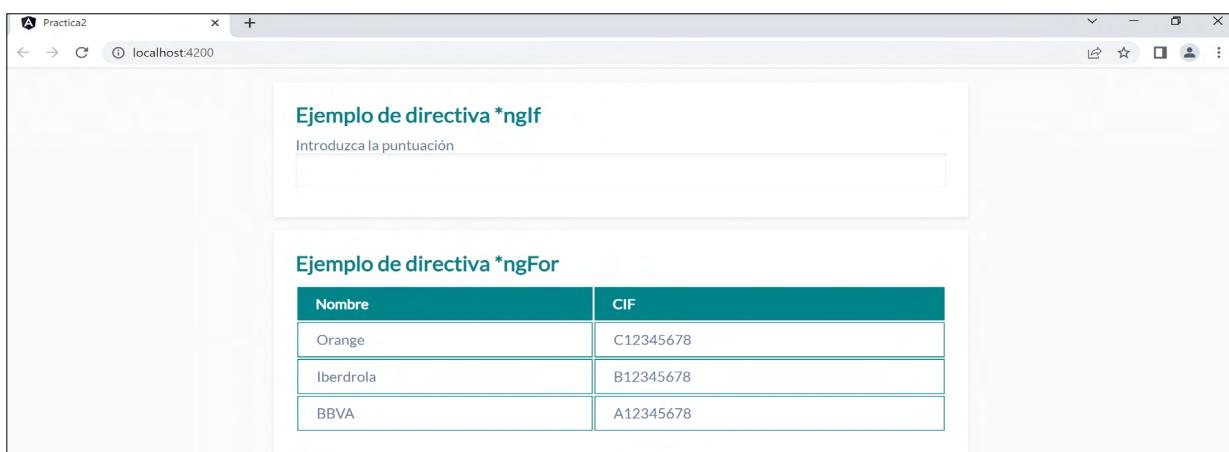


Figura 3.2
Componente Angular con la directiva “*ngFor” en el navegador.

Entre las directivas de atributo, una de las más usadas es “ngClass”, que permite implementar clases CSS de forma dinámica en los elementos, a partir del resultado de la evaluación de una expresión. Para comprobar su uso de manera práctica, vamos a crear un nuevo componente en nuestro proyecto con el siguiente comando:

```
ng generate componente directivaNgClass
```

Una vez generado, en la clase del componente, vamos a repetir una propiedad de puntos en la directiva-`ng-class`.component.ts de la siguiente forma, complementada con un método que devolverá un *string* “success” si es igual o mayor que 5 y “fail” si es menor:

```
...
export class DirectivaNgClassComponent implements OnInit {

  points: any = null;

  constructor() { }

  ngOnInit(): void {
  }

  getClass(): string {
    return this.points >= 5 ? 'success' : 'fail';
  }
}
```

Y en el archivo de plantilla (directiva-ng-class.component.html), repetimos el ejemplo de introducción de datos, pero en este caso con la directiva “ngClass”, que aplicará al texto una clase CSS en función del valor de la propiedad “points” mediante la invocación del método “getClass”. Modificamos su código de la siguiente forma:

```
<div class="card">
  <h1>Ejemplo de directiva ngClass</h1>
  <p>Introduzca la puntuación</p>
  <input type="number" [(ngModel)]= "points">
  <p *ngIf="points" [ngClass]= "getClass()">
    La puntuación obtenida es {{points}}
  </p>
</div>
```

El siguiente paso es añadir esas clases CSS, así que en el archivo styles.css añadimos los siguientes bloques:

```
...
.success {
  color: white;
  background-color: green;
}

.fail {
  color: white;
  background-color: red;
}
```

Finalmente, añadimos la etiqueta del componente en el archivo app.component.html de la siguiente forma:

```
<app-directiva-ng-if></app-directiva-ng-if>
<app-directiva-ng-for></app-directiva-ng-for>
<app-directiva-ng-class></app-directiva-ng-class>
```

Y comprobamos en el navegador (Figura 3.3) que, según el valor introducido en el *input*, se aplica dinámicamente una u otra clase CSS, requisito común en muchos componentes de interfaz de usuario de nuestras aplicaciones.

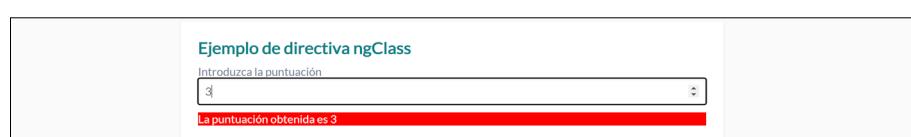


Figura 3.3

Componente Angular con la directiva “ngClass” en el navegador.

Las directivas son elementos esenciales del *framework* Angular e imprescindibles en las aplicaciones. Además, pueden ser desarrolladas a medida para ciertos casos de uso no cubiertos por las proporcionadas por defecto.

3.3. Pipes en Angular

El pipe es un mecanismo que permite modificar de forma sencilla y reutilizable el formato del resultado de las expresiones que se introducen en la interpolación en Angular, y el *framework* incorpora una colección disponible sin ningún tipo de importación en todos los componentes. Recibe este nombre porque su sintaxis usa el signo *pipe* (“|”, llamado en castellano raya vertical o pleca).

Para comprobar su empleo y su sintaxis, vamos a crear un nuevo componente en nuestro proyecto con el siguiente comando:

```
ng generate componente ejemplosPipes
```

En la clase del componente, en el archivo ejemplos-pipes.component.ts, vamos a añadir dos propiedades con diferentes tipos de datos, “string” y “date”.

```
...
export class EjemplosPipesComponent implements OnInit {

  message: string = 'villa de madrid';
  dob: Date = new Date(1980, 0, 1);

  constructor() { }

  ngOnInit(): void {
  }

}
```

Y es en la plantilla donde vamos a implementar los *pipes* a continuación de la expresión de la interpolación. Escribimos el siguiente código en ejemplos-pipes.component.html:

```
<div class="card">
  <h1>Ejemplo de Pipes</h1>
  <table>
```

```

<tr>
  <th>Valor sin pipe</th>
  <th>Valor con pipe</th>
</tr>
<tr>
  <td>{{message}}</td>
  <td>{{message | uppercase}}</td>
</tr>
<tr>
  <td>{{dob}}</td>
  <td>{{dob | date: 'dd/MM/yyyy'}}</td>
</tr>
</table>
</div>

```

Disponemos de dos tipos de *pipes*, como podemos ver en estos ejemplos: los que no reciben opciones como “uppercase”, que simplemente transforma el *string* en mayúsculas, y los que reciben un *string* de opciones, como “date”, al que podemos pasarle el formato de día, mes y año como especificación del formato de fecha. En la tabla declarada, disponemos de los valores de ambas propiedades con y sin *pipe*, para comprobar la transformación que realiza cada uno.

A continuación, añadimos este nuevo componente con los ejemplos de *pipe* en nuestro componente raíz app.component.html:

```

<app-directiva-ng-if></app-directiva-ng-if>
<app-directiva-ng-for></app-directiva-ng-for>
<app-directiva-ng-class></app-directiva-ng-class>
<app-ejemplos-pipes></app-ejemplos-pipes>

```

Ahora podemos supervisar el componente en el navegador (Figura 3.4), comprobando que los *pipes* modifican el formato de una manera muy sencilla y ágil.

| Ejemplo de Pipes | |
|---|-----------------|
| Valor sin pipe | Valor con pipe |
| villa de madrid | VILLA DE MADRID |
| Tue Jan 01 1980 00:00:00 GMT+0100 (hora estándar de Europa central) | 01/01/1980 |

Figura 3.4
Ejemplos de *pipes* Angular implementados en componentes en el navegador.

Los *pipes* permiten resolver casos habituales de presentación de datos y, como ocurre con las directivas, pueden además ser desarrollados a medida en el caso de que no exista ninguno que cumpla nuestros requisitos.



Resumen

- Las directivas son un mecanismo o elemento de Angular que permite implementar soluciones lógicas habituales en la programación de componentes en los elementos de la plantilla HTML.
- Disponemos de directivas estructurales, que modifican el DOM y se caracterizan por ir precedidas del signo de asterisco, y de atributo, que implementan valores dinámicos a atributos de los elementos.
- Los *pipes* permiten modificar de forma sencilla y reutilizable el formato del resultado de las expresiones que se introducen en la interpolación en Angular en la plantilla de sus componentes.

4. Comunicación entre componentes y servicios en Angular

La mayoría de los *frameworks* JavaScript, y Angular no es una excepción, están basados en componentes. En esta unidad vamos a profundizar en su uso y distribución en conjunción con los servicios.

En primer lugar, aprenderemos de forma práctica la principal fórmula de comunicación de los componentes Angular, mediante el empleo de una sintaxis para el paso de datos en jerarquía padre-hijo.

Posteriormente, aprenderemos la implementación y el uso de servicios en nuestras aplicaciones, otro de los elementos clave junto a los componentes en la arquitectura de desarrollo en Angular.

4.1. Comunicación entre componentes

Además de la carga dinámica de componentes mediante navegación, que aprenderemos más adelante, la forma de implementación de componentes en nuestras aplicaciones Angular es mediante la jerarquía padre-hijo, embebiendo sus etiquetas entre ellos. Este patrón es muy sencillo de implementar, ya que, en esencia, reproduce el sistema de anidación de los elementos HTML y sus nodos del DOM.

Sin embargo, Angular proporciona además un mecanismo para que los datos utilizados en la lógica de componentes puedan ser pasados de padre a hijo y de hijo a padre, con la reutilización de código como objetivo fundamental. Para aprender de forma práctica esta sintaxis, vamos a crear un nuevo proyecto Angular. En cualquier ubicación de nuestro equipo, introduciremos de nuevo el siguiente comando:

```
ng new practica3
```

Tras confirmar en la terminal la instalación de Routing y de CSS como hoja de estilos, y una vez finalizado el proceso, abrimos el proyecto con

Visual Studio Code y creamos un nuevo componente. Para ello, escribimos en la terminal el siguiente comando:

```
ng generate component books
```

A continuación, en el archivo books.component.ts añadimos un set de datos de libros y un método que, cada vez que sea invocado, sume el total del *stock* de todos ellos. Para ello, modificaremos el código de la siguiente forma:

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

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

  books: Array<any> = [
    {title: "La Historia Interminable", author: "Michael Ende", stock: 0},
    {title: "Cien Años de Soledad", author: "G. García Márquez", stock: 0},
    {title: "La Tapadera", author: "John Grisham", stock: 0},
  ]

  constructor() { }

  ngOnInit(): void {
  }

  getStock() {
    let totalStock: number = 0;
    this.books.forEach(elem => totalStock += elem.stock);
    return totalStock;
  }
}
```

Ahora, en la plantilla, dentro del archivo books.component.html, incluimos los siguientes elementos con interpolación para renderizar unos valores relacionados con la colección de datos:

```
<div class="container">
  <div class="card">
    <h1>Total Libros {{books.length}}</h1>
    <h1>Total Stock {{getStock()}}</h1>
  </div>
</div>
```

El siguiente paso es, por una parte, añadir en styles.css, dentro de la carpeta src, el código CSS desarrollado en la práctica anterior; y, por otra, introducir nuestro componente “books” en el componente raíz, para lo cual modificamos el archivo app.component.ts con el siguiente código:

```
<app-books></app-books>
```

Iniciamos la aplicación desde la terminal con el comando “ng serve -o” y se mostrará la información en nuestro navegador (Figura 4.1).



Figura 4.1

Renderizado de un componente de aplicación Angular en el navegador.

Supongamos que, además, necesitamos introducir los valores de cada libro en una tarjeta; aquí es donde podemos reutilizar código creando un componente que luego introduciremos como hijo. Para hacerlo, completamos en la terminal:

```
ng generate component card
```

Este nuevo componente va a recibir cada objeto individual de libro desde el padre, para lo cual vamos a usar el decorador “@Input” de Angular para recibir esos valores en una propiedad de la clase hijo. Por tanto, modificamos el archivo card.component.ts de la siguiente forma:

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

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

  @Input() data: any = {};

  constructor() { }

  ngOnInit(): void {
  }

}
```

Sin embargo, está lógica estará incompleta hasta que no se reciban los valores. Por tanto, vamos a introducir este componente hijo “card” en el componente padre “books”, modificando el archivo books.component.html de la siguiente forma:

```
<div class="container">
  <div class="card">
    <h1>Total Libros {{books.length}}</h1>
    <h1>Total Stock {{getStock()}}</h1>
  </div>
  <app-card *ngFor="let book of books" [data]="book"></app-card>
</div>
```

Como podemos observar, iteramos un componente hijo “card” por cada elemento del array “books” y, además, pasamos a la propiedad “data” de “card” el valor de cada elemento del array iterado. Con ese objeto pasado que recibiremos en “data”, ahora podemos escribir la plantilla del componente hijo en el archivo card.component.html para renderizar sus valores de la siguiente forma:

```
<div class="card">
  <h1>{{data.title}}</h1>
  <p>{{data.author}}</p>
  <hr>
  <p>Stock {{data.stock}}</p>
</div>
```

Solo nos queda modificar ligeramente los estilos, para lo cual hacemos cambios en el siguiente bloque en styles.css:

```
.card {
  max-width: 540px;
  padding: 1.5rem;
  margin: 1rem auto;
  border: rgba(0,0,0,0.08);
  box-shadow: 0 2px 4px 0 rgba(0,0,0,0.08);
  background-color: white;
}

.card hr {
  margin: 1rem 0;
}
```

Al grabar, podemos visualizar en el navegador (Figura 4.2) que se produce la comunicación de valores entre padre e hijo, y se reutiliza el código del segundo para renderizar las diferentes tarjetas.

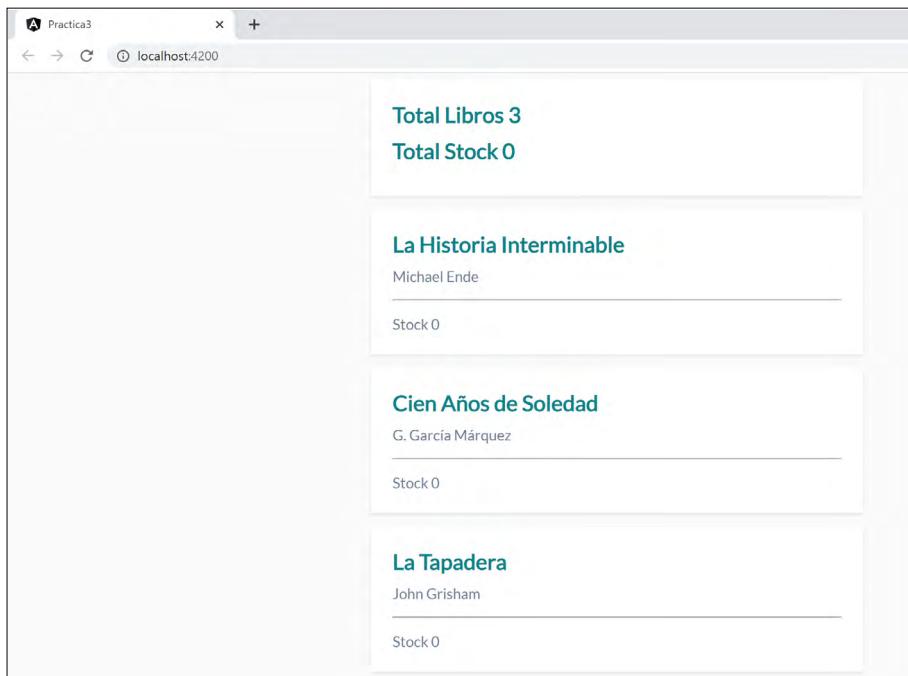


Figura 4.2

Reutilización de un componente de aplicación Angular en el navegador.

Además del flujo de datos de padre a hijo, también se puede provocar desde el hijo al padre mediante la implementación del decorador “@ Output”, el cual emite un evento con cualquier tipo de dato que es asociado a un método del padre. Por ejemplo, supongamos que necesitamos poder modificar el *stock* en cada tarjeta; crearemos la propiedad del emisor y, con un método asociado a dos botones, haremos que emita al padre un objeto con el libro y el aumento o la disminución del *stock*.

Para ello, vamos a modificar en primer lugar el archivo *card.component.ts* añadiendo el emisor y el método con el siguiente código:

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

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

  @Input() data: any = {};
```

```

@Output() emitData = new EventEmitter<any>();

constructor() { }

ngOnInit(): void {
}

updateStock(title: string, stock: number): void {
  this.emitData.emit({title, stock});
}

}

```

A continuación, en la plantilla card.component.html, vamos a añadir los botones que invocarán el método, que recibirán como argumento un objeto con el título y el incremento o decremento del *stock*:

```

<div class="card">
  <h1>{{data.title}}</h1>
  <p>{{data.author}}</p>
  <hr>
  <p>Stock {{data.stock}}</p>
  <button (click)="updateStock(data.title, 1)">+</button>
  <button (click)="updateStock(data.title, -1)">-</button>
</div>

```

Ahora, en la plantilla del padre (books.component.html), vamos a utilizar el evento emisor con las sintaxis de evento para asociarlo con un método en el que recibiremos el objeto que viene del hijo. Para ello, debemos denominar ese parámetro obligatoriamente con “\$event”. Modificamos el código de la siguiente forma:

```

...
<app-card *ngFor="let book of books" [data]="book"
  (emitData)="updateBooks($event)"></app-card>
...

```

Y, finalmente, declaramos en books.component.ts el método de actualización de los libros. Modificamos la clase de la siguiente forma:

```

export class BooksComponent implements OnInit {

  books: Array<any> = [
    {title: "La Historia Interminable", author: "Michael Ende", stock: 0},

```

```

    {title: "Cien Años de Soledad", author: "G. García Márquez", stock: 0},
    {title: "La Tapadera", author: "John Grisham", stock: 0},
]

constructor() { }

ngOnInit(): void {
}

getStock(): {
    let totalStock: number = 0;
    this.books.forEach(elem => totalStock += elem.stock);
    return totalStock;
}

updateBooks(event: any): void {
    this.books.forEach(elem => {
        if(elem.title === event.title) {
            elem.stock += event.stock;
        }
    })
}
}

```

El resultado podemos comprobarlo en el navegador (Figura 4.3): cuando utilizamos los botones para actualizar el *stock* de un libro, los valores se actualizan automáticamente tanto en el parent como en el hijo.

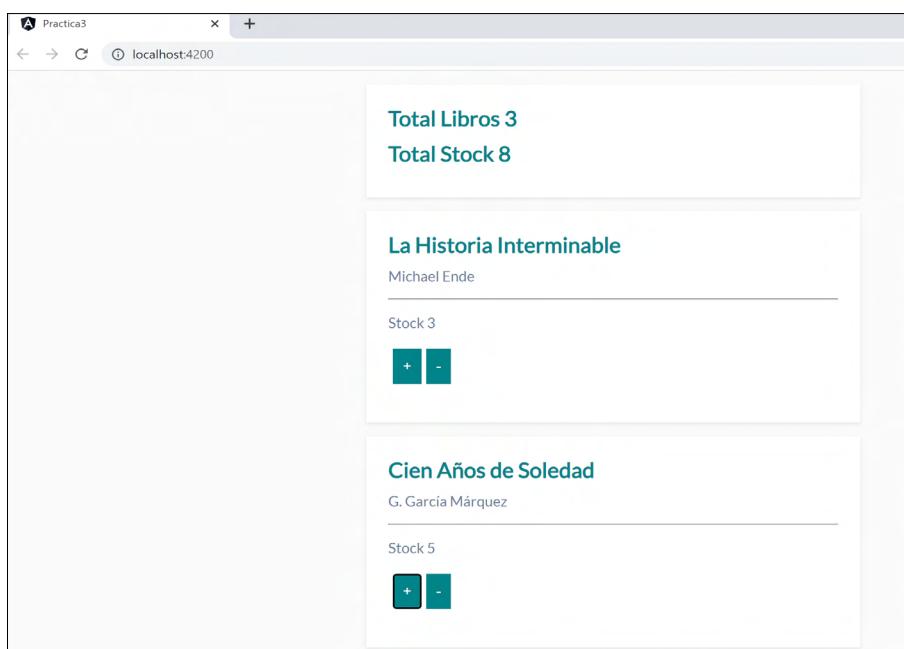


Figura 4.3
Actualización de componentes de aplicación Angular en el navegador.



Para saber más

Los servicios en Angular son declarados por defecto para la totalidad de los componentes y módulos de la aplicación, aunque se puede restringir su empleo según su modularización.

4.2. Servicios en Angular

Los servicios en Angular son unos elementos utilizados para la centralización de código usado por uno o varios componentes y, por el mismo motivo, para disponer de un contexto donde centralizar las peticiones HTTP al exterior de la aplicación para las entidades para los que son definidos.

Al contrario que en otros *frameworks*, no se trata de clases o funciones desarrolladas con totalidad, sino que parten de una implementación de Angular que los define y que, además, permite su implementación en los componentes con una determinada sintaxis.

La ventaja es que, gracias a la herramienta Angular CLI, podemos añadirlos desde la terminal. En nuestro caso, vamos a añadir un servicio para centralizar los datos de los libros escribiendo el siguiente comando en la terminal de Visual Studio Code:

```
ng generate service books
```

Los servicios se comunican con los componentes en dos direcciones a través de métodos que son invocados desde estos, así que, en nuestro caso, vamos a llevar al servicio los datos y declarar dos métodos para las dos acciones lógicas que realizamos: cargar todos los libros y actualizar su *stock*. Para ello, en el archivo del servicio, books.service.ts, escribimos el siguiente código:

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

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

  books: Array<any> = [
    {title: "La Historia Interminable", author: "Michael Ende", stock: 0},
    {title: "Cien Años de Soledad", author: "G. García Márquez", stock: 0},
    {title: "La Tapadera", author: "John Grisham", stock: 0},
  ]

  constructor() { }

  getBooks(): Array<any> {
    return this.books;
  }
}
```

```

    }

    setStockBook(title: string, value: number) {
      this.books.forEach(elem => {
        if(elem.title === title) {
          elem.stock += value;
        }
      })
    }

}

```

Ahora, para consumir este servicio, necesitamos modificar el componente padre de libros mediante su implementación en el constructor e invocar desde esa instancia los métodos expuestos. Por tanto, vamos a modificar nuestro archivo books.component.ts de la siguiente forma:

```

import { Component, OnInit } from '@angular/core';
import { BooksService } from '../books.service';

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

  books: Array<any> = [];

  constructor(private booksService: BooksService) { }

  ngOnInit(): void {
    this.books = this.booksService.getBooks();
  }

  getStock() {
    let totalStock: number = 0;
    this.books.forEach(elem => totalStock += elem.stock);
    return totalStock;
  }

  updateBooks(event: any): void {
    this.booksService.setStockBook(event.title, event.stock);
  }

}

```

Con estas modificaciones, la propiedad “booksService” declarada en el constructor permite invocar los datos en la carga del componente (por eso utilizamos el método “getBooks()” dentro del hook “ngOnInit()”) y,

posteriormente, cada vez que se actualice el *stock* invocando “setStock-Book()” desde el método del componente.

Al grabar, podemos comprobar en el navegador que se mantiene correctamente la funcionalidad de la aplicación. La ventaja del empleo de servicios es que, si otros componentes necesitan utilizar esta entidad de libros, podrán emplear los métodos declarados, que además podremos extender a medida que se implementen nuevas características y enlazar con peticiones HTTP, como veremos más adelante.



Resumen

- Una de las formas de implementación de componentes en nuestras aplicaciones Angular es mediante la jerarquía padre-hijo, embebiendo sus etiquetas entre ellos.
- El paso de datos padre-hijo y viceversa se establece en Angular mediante el uso de los decoradores “@Input” y “@Output”, con un mecanismo de paso directo por asignación y de emisor de eventos, respectivamente.
- Los servicios en Angular son elementos utilizados para la centralización de código usado por uno o varios componentes y para realizar peticiones HTTP relacionadas con las entidades para los que son definidos.

5. Routing y formularios en Angular

Llegamos en esta unidad a dos de los conceptos clave en cualquier aplicación: el sistema de navegación y el uso de formularios, funcionalidades imprescindibles para la interacción con el usuario.

En primer lugar, aprenderemos de forma práctica cómo se implementa el *routing* o navegación en Angular, a través de una de sus librerías incluidas de forma nativa en el *framework*.

Posteriormente, aprenderemos cómo desarrollar formularios en Angular con la librería reactiva, obteniendo los valores del usuario en cada cambio para implementar cualquier tipo de funcionalidad relacionada con estos elementos de la interfaz.

5.1. Routing en Angular

A diferencia de otros *frameworks* de JavaScript, Angular incorpora una librería para establecer el sistema de *routing* y, por tanto, el modelo *single page application*. Se trata de un paquete AppRoutingModule que Angular CLI implementa de acuerdo con las instrucciones al generar el proyecto.

Para probar su empleo, vamos a generar un nuevo proyecto Angular. Para ello, en cualquier ubicación de nuestro equipo, completamos de nuevo el siguiente comando:

```
ng new practica4
```

Precisamente en el primer paso de la generación, Angular CLI nos consulta si queremos instalar Routing, así que pulsamos Intro para confirmar y de nuevo confirmamos CSS como hoja de estilos.

Al finalizar el proceso, abrimos el proyecto con Visual Studio Code; vamos a crear un nuevo componente de inicio, para lo cual escribimos en la terminal el siguiente comando:

```
ng generate component home
```

Para poder comprobar la forma de navegar en Angular, creamos otro componente con el siguiente comando también en la terminal:

```
ng generate component createBook
```

La manera en la que Angular implementa el sistema de *routing* es, en primer lugar, mediante un archivo de rutas donde estas se definen, app-routing.module.ts, situado en el directorio app y en el que vamos a añadir nuestras rutas modificando su código de la siguiente forma:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { CreateBookComponent } from './create-book/create-book.component';
import { HomeComponent } from './home/home.component';

const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'create-book', component: CreateBookComponent},
  {path: '**', redirectTo: '/'}
];

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

export class AppRoutingModule {}
```

Como podemos observar, disponemos de una constante de rutas donde estas son definidas en forma de objeto con las propiedades “path” para determinar la ruta y “componente” para asociar la clase del componente que renderizará en cada una de ellas. Añadimos también el escape para rutas que no existan, las cuales son representadas por la máscara de doble asterisco y usamos la propiedad “redirectTo” para redirigir al inicio.

Una vez que disponemos de las rutas, estas cargarán sus componentes en un elemento denominado “router-outlet”, cuya etiqueta se introduce en el componente raíz. Para ello sustituimos todo el código de app.component.html por el siguiente:

```
<router-outlet></router-outlet>
```

Ahora podemos trabajar en los componentes para introducir los elementos de navegación, para lo cual se emplea la directiva “routerLink”.

En primer lugar, vamos al componente “Home” y, en su archivo home.component.html, introducimos el siguiente código con esta directiva:

```
<div class="container">
  <h1>Bienvenid@s a la aplicación</h1>
  <button routerLink="/create-book">Registrar libro</button>
</div>
```

Gracias a “routerLink”, al pulsar en el botón navegará al componente asociado a esa ruta. Para implementar también la navegación en el otro componente, modificamos su código en el archivo create-book.component.html de forma similar:

```
<div class="container">
  <h1>Nuevo libro</h1>
  <button routerLink="/">Regresar</button>
</div>
```

Figura 5.1
Sistema de navegación en aplicaciones Angular en el navegador.

A continuación, en el archivo styles.css añadimos los estilos del proyecto anterior y desde la terminal iniciamos el proyecto con el comando “ng serve -o”. Una vez en el navegador (Figura 5.1), podemos comprobar cómo llevar a cabo la navegación entre ambos componentes.



Además de la navegación con “routerLink”, el módulo o paquete de Angular para navegación permite la generación de módulos con carga *lazy loading*, implementación de parámetros de ruta, navegación programática y *guards* de protección de las rutas.

5.2. Formularios reactivos en Angular

Además de los formularios con “FormsModule” y la directiva “ngModule”, Angular incorpora desde su versión 4 un módulo de formularios reactivos con mayor control y funcionalidad. Su empleo exige importarlos y declararlos en el módulo de la aplicación, para lo cual vamos a

modificar en nuestro proyecto el archivo app.module.ts para añadirlo de la siguiente forma:

```
...
import { ReactiveFormsModule } from '@angular/forms';
...
...
imports: [
  BrowserModule,
  AppRoutingModule,
  ReactiveFormsModule
],
...
```

Una vez incorporado el módulo, podemos usar su funcionalidad en cualquier componente. El primer paso consiste en generar un objeto en la clase del componente como instancia de una clase del paquete, que nos permitirá asociarlo al formulario. En nuestro caso, vamos a añadir la declaración del objeto en el archivo create-book.component.ts y sus importaciones de la siguiente forma:

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

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

  formBook: any = {};

  constructor() { }

  ngOnInit(): void {
    this.formBook = new FormGroup({
      title: new FormControl(''),
      author: new FormControl(''),
      stock: new FormControl(0)
    })
  }
}
```

La implementación se basa en declarar en primer lugar un objeto de formulario al que se le asigna en el hook “ngOnInit()” una instancia de la

clase “FormGroup”, la cual, como segundo paso, recibe cada una de las propiedades de ese objeto con un valor consistente en otra instancia de la clase “FormControl” que recibe el valor de inicialización.

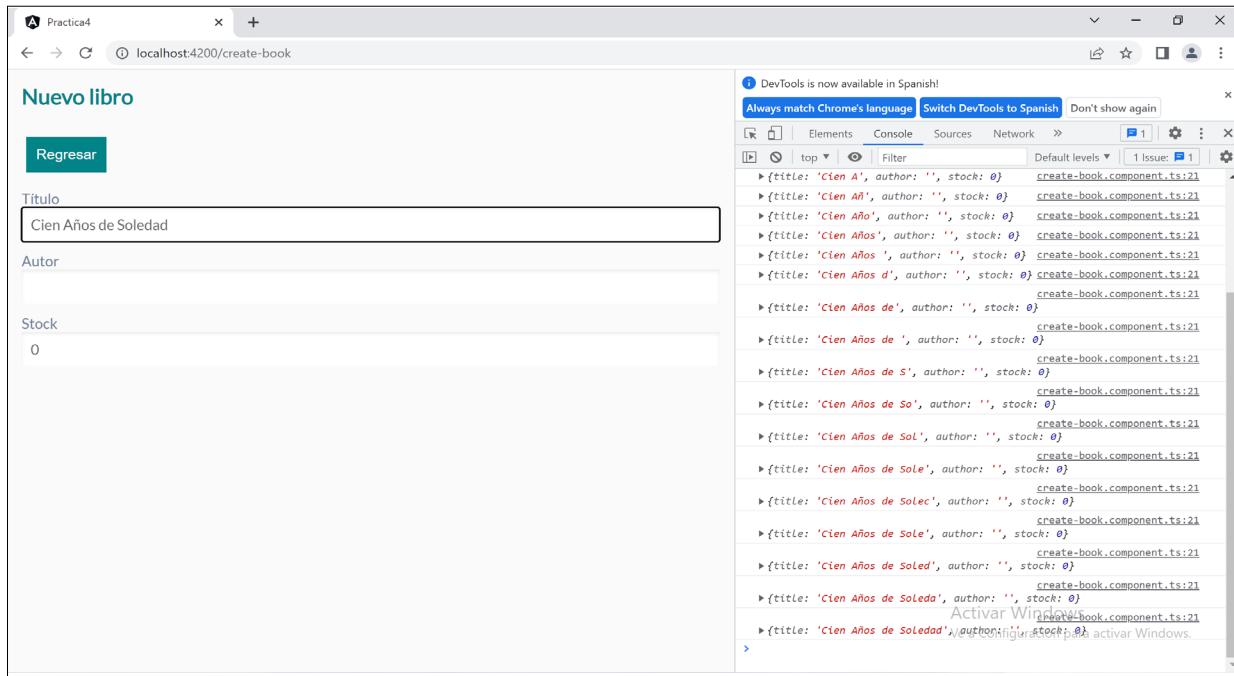
El siguiente paso consiste en añadir el formulario en la plantilla y asociarlo a este objeto. En el archivo create-book.component.html añadimos el siguiente código:

```
<div class="container">
  <h1>Nuevo libro</h1>
  <button routerLink="/">Regresar</button>
  <form [formGroup]="formBook">
    <label>Título</label>
    <input type="text" formControlName="title">
    <label>Autor</label>
    <input type="text" formControlName="author">
    <label>Stock</label>
    <input type="number" formControlName="stock">
  </form>
</div>
```

Gracias a los atributos “formGroup” y “formControlName” de la plantilla, con esto ya disponemos del enlace entre estos *inputs* y sus propiedades del objeto, y la reactividad del formulario la obtendremos gracias a la propiedad “valueChanges”, que es de tipo observable y permite ejecutar una función “callback” cada vez que cambie un valor. Vamos a implementarla en el hook “ngOnInit()” de la siguiente forma:

```
...
ngOnInit(): void {
  this.formBook = new FormGroup({
    title: new FormControl(''),
    author: new FormControl(''),
    stock: new FormControl(0)
  })
  this.formBook.valueChanges.subscribe((data: any) => console.log(data));
}
...
```

Si ahora navegamos al componente y abrimos las herramientas de desarrollador en la pestaña de consola, podemos comprobar (Figura 5.2) que cada vez que tecleamos en cualquier *input* del formulario obtenemos el valor de todo el objeto. Podremos usarlo para todo tipo de funcionalidad de validación u operaciones lógicas.



La implementación del evento “submit” es muy sencilla: emplearemos la directiva “ngSubmit” en la plantilla asociada a un método de la clase del componente. En primer lugar, modificamos el archivo create-book.component.html con el siguiente código:

```
<div class="container">
  <h1>Nuevo libro</h1>
  <button routerLink="/">Regresar</button>
  <form [FormGroup] = "formBook" (ngSubmit) = "addBook()">
    <label>Título</label>
    <input type="text" FormControlName="title">
    <label>Autor</label>
    <input type="text" FormControlName="author">
    <label>Stock</label>
    <input type="number" FormControlName="stock">
    <button type="submit">Añadir</button>
  </form>
</div>
```

Y, a continuación, añadimos el método en la clase del componente en el archivo create-book.component.ts:

```
addBook() {
  console.log(this.formBook.value);
}
```

Figura 5.2

Comprobación de reactividad en formularios Angular en el navegador.

Comprobamos la implementación en el navegador (Figura 5.3), de manera que, al pulsar Intro o en el botón “Añadir”, obtendremos en la consola el valor del formulario.

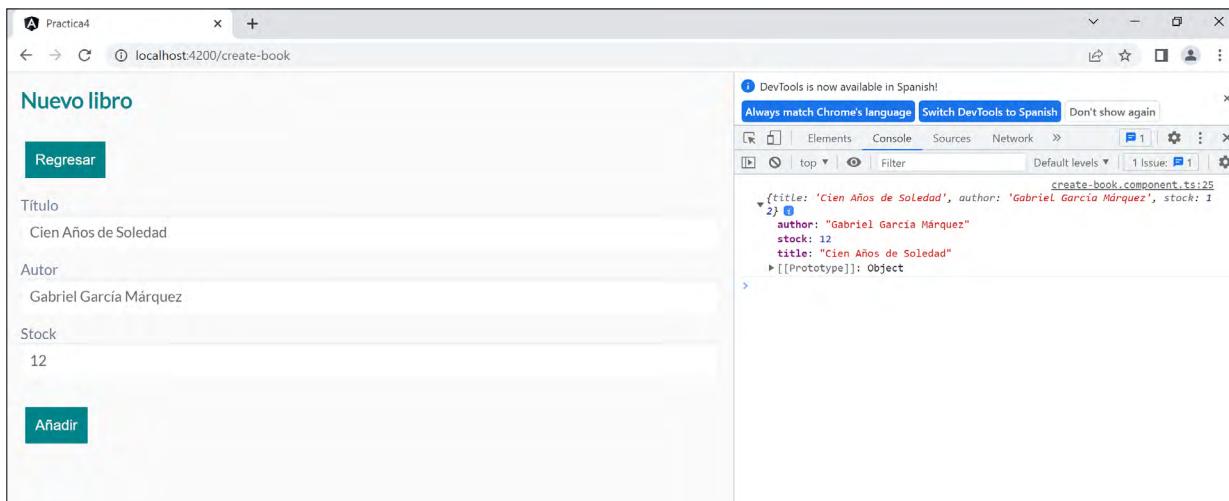


Figura 5.3

Comprobación del evento “submit” en formularios Angular en el navegador.

En este método, imprimimos por consola el valor del formulario obtenido mediante la propiedad “value”; pero podríamos emplear este valor para enviarlo a un servicio donde gestionar esa entidad, con la posibilidad añadida de extender el objeto con otras propiedades que no existan en el formulario (por ejemplo, el usuario que ha creado el registro, marcas de tiempo, etc.).



Resumen

- Angular incorpora la librería AppRoutingModule para establecer el sistema de *routing* (y, por tanto, el modelo *single page application*) utilizando un *array* de rutas enlazadas a cada componente que hay que renderizar en un componente dinámico llamado “router-outlet”.
- La librería AppRoutingModule implementa directivas como “routerLink” para navegar a cada componente y dispone de mecanismos de navegación programática, carga *lazy loading* y protección de rutas.
- Para el desarrollo de formularios reactivos, Angular incorpora la librería ReactiveFormsModule, que permite desarrollar formularios con mayor control sobre cada *input* y la obtención inmediata de los valores introducidos por el usuario.

6. Comunicaciones HTTP y despliegue a producción

Finalizamos este módulo con el aprendizaje de las comunicaciones mediante peticiones HTTP de aplicaciones Angular con API Rest, requisito esencial en las aplicaciones de página única (SPA) y con el mecanismo de despliegue.

En primer lugar, conoceremos cómo usar la librería HttpClientModule de Angular para generar peticiones con el protocolo HTTP a cualquier API Rest desde sus servicios.

Posteriormente, aprenderemos de manera práctica cómo generar el *bundle* de producción para desplegar nuestras aplicaciones desarrolladas en Angular, un sencillo proceso gracias a Angular CLI.

6.1. Comunicaciones HTTP en Angular

Angular incorpora también en el caso de las peticiones HTTP su propia librería de gestión de peticiones y su correspondiente asincronía, denominada HttpClientModule, que es necesario declarar a nivel de módulo para poder utilizarla en cualquier parte de la aplicación (normalmente, en los servicios).

Vamos a utilizar el proyecto Angular de la unidad anterior para comprobar el uso de HttpClientModule; pero, previamente, y con el fin de poder realizar peticiones con este protocolo, vamos a crear una pequeña API Rest con NodeJS y Express. De esta forma, creamos un directorio llamado apitest en cualquier ubicación de nuestro equipo, lo abrimos con Visual Studio Code y tecleamos en la terminal los siguientes comandos:



Para saber más

La asincronía de peticiones HTTP es resuelta en Angular gracias al uso de observables, ya que este framework incorpora de forma nativa el uso de RxJS y sus métodos.

```
npm init -y  
npm install express --save  
npm install nodemon --save-dev
```

A continuación, añadimos al archivo package.json el script de nodemon de la siguiente forma:

```
...
  "scripts": {
    "start": "nodemon app",
    "test": "echo \'Error: no test specified\' && exit 1"
  },
...

```

Creamos un archivo app.js en la raíz del proyecto y añadimos el siguiente código con dos métodos de petición “get” y “post”:

```
const express = require('express');
const app = express();
const port = 3000;
const cors = require('cors');

app.use(express.json());
app.use(cors());

books = [
  {title: "La Historia Interminable", author: "Michael Ende", stock: 0},
  {title: "Cien Años de Soledad", author: "G. García Márquez", stock: 0},
  {title: "La Tapadera", author: "John Grisham", stock: 0},
]

app.get('/books', (req, res) => {
  res.status(200).json({
    books
  })
})

app.post('/books', (req, res) => {
  books.push(req.body);
  res.status(200).json({
    message: 'ok'
  })
})

app.listen(port, () => {
  console.log('Servidor escuchando en http://localhost:' + port);
})
```

Para inicializar la API, tecleamos en la terminal el comando habitual:

```
npm start
```

Ahora ya podemos abrir nuestro proyecto Angular practica4 en Visual Studio Code para comprobar la librería HttpClientModule. Comenzamos

introduciendo la librería en el archivo app.module.ts, añadiendo su declaración e importación de la siguiente forma:

```
...
import { HttpClientModule } from '@angular/common/http';
...

imports: [
  BrowserModule,
  AppRoutingModule,
  ReactiveFormsModule,
  HttpClientModule
],
...
```

A continuación, vamos a generar un servicio para realizar peticiones a la API, para lo cual completamos en la terminal el siguiente comando:

```
ng generate service books
```

En primer lugar, vamos a implementar en este servicio una petición “get” para recuperar los datos de libros de la API, para lo cual generamos un método y, dentro de este, declaramos una petición “get” sobre una instancia de HttpClient que recibe la URL de la API y devuelve con el método *pipe* un observable con la respuesta de la API. Para ello, escribimos el siguiente código en books.service.ts:

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

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

  endpoint: string = 'http://localhost:3000/books';

  constructor(private http: HttpClient) { }

  getBooks() {
    return this.http.get(this.endpoint)
      .pipe((data: any) => data)
  }
}
```

Una vez que disponemos de este método en el servicio, vamos a invocarlo desde nuestro componente “Home”. Para ello, en su clase del componente declaramos un *array* de libros y en el hook “ngOnInit()” invitamos el servicio. Como se trata de un observable, utilizamos el método “subscribe()” para recibir en una función *callback* en la propiedad “next” los datos que asignaremos a la propiedad. Modificamos el código de home.component.ts de la siguiente forma:

```
import { Component, OnInit } from '@angular/core';
import { BooksService } from '../books.service';

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

  books: Array<any> = [];

  constructor(private booksService: BooksService) { }

  ngOnInit(): void {
    this.booksService.getBooks()
      .subscribe({
        next: (data: any) => this.books = data.books
      })
  }
}
```

Con esto conseguiremos los datos de la API en la propiedad “books”. Gracias a ello, podemos añadir en la plantilla en el archivo home.component.html una tabla para renderizar los datos. Modificamos su código de la siguiente forma:

```
<div class="container">
  <h1>Bienvenid@s a la aplicación</h1>
  <button routerLink="/create-book">Registrar libro</button>
  <h2>Libros</h2>
  <table>
    <tr>
      <th>Título</th>
      <th>Autor</th>
    </tr>
```

```

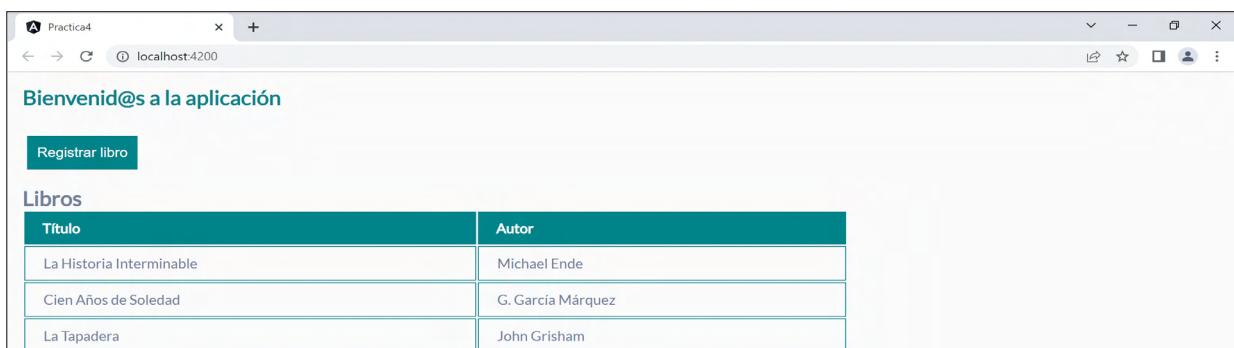
<tr *ngFor="let book of books">
  <td>{{book.title}}</td>
  <td>{{book.author}}</td>
</tr>
</table>
</div>

```

Figura 6.1

Renderizado de datos procedentes de una API Rest en el navegador.

Al comprobar en nuestro navegador la aplicación (Figura 6.1), podemos comprobar que aparecen los datos correctamente.



6.2. Asincronía y navegación programática

Además de las peticiones “get”, la librería HttpClientModule permite realizar el resto de los verbos HTTP; por ejemplo, “post”. Vamos a implementar una petición de este tipo en nuestro servicio, para lo cual ampliamos el archivo books.service.ts con el siguiente método, que recibirá un objeto “book” que será enviado en el *body* al introducirse en el método “post” como segundo argumento tras el “endpoint”.

```

sendBook(book: any) {
  return this.http.post(this.endpoint, book)
    .pipe((data: any) => data)
}

```

Como podemos observar, de nuevo encadenamos la petición con *pipe*, lo que permite retornar al método un observable que permitirá usarlo en el componente. Vamos a invocar este método en el componente para crear libros, para lo cual importamos el servicio y modificamos el método en el archivo create-book.component.ts de la siguiente forma:

...

```

constructor(private booksService: BooksService) { }

...
addBook() {
  this.booksService.sendBook(this.formBook.value)
    .subscribe({
      next: (data: any) => {
        console.log(data);
      },
      error: (err: any) => console.log(err)
    })
}
...

```

De nuevo, con el método “subscribe” manejamos la asincronía de la petición, ya que disponemos de las propiedades “next” y “error”, que ejecutarán una función *callback* cuando se reciba la respuesta de la API si es correcta en el primer caso, y si tiene un estado de error en el segundo. Con esta modificación, nuestro componente estaría listo, pero podemos aprovechar para que, en caso de respuesta correcta, el sistema navegue automáticamente al inicio. Para ello, vamos a añadir la clase “Router” del módulo de navegación y el método “navigate” con la ruta raíz. Modificamos el archivo de nuevo, y queda todo el código de la siguiente forma:

```

import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
import { Router } from '@angular/router';
import { BooksService } from '../books.service';

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

  formBook: any = {};

  constructor(private booksService: BooksService,
    private router: Router) { }

  ngOnInit(): void {
    this.formBook = new FormGroup({
      title: new FormControl(''),
      author: new FormControl(''),
      stock: new FormControl(0)
    })
  }

  addBook(): void {
    this.booksService.sendBook(this.formBook.value)
      .subscribe({
        next: (data: any) => {
          console.log(data);
          this.router.navigate(['/']);
        },
        error: (err: any) => console.log(err)
      })
  }
}

```

```

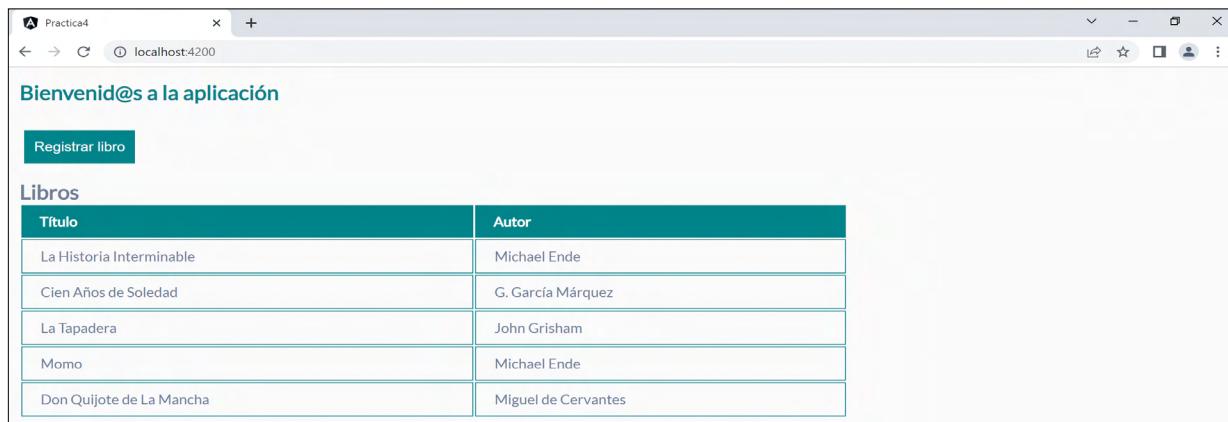
        })
    }

    addBook() {
        this.booksService.sendBook(this.formBook.value)
            .subscribe({
                next: (data: any) => {
                    console.log(data);
                    this.router.navigate(['/']);
                },
                error: (err: any) => console.log(err)
            })
    }
}

```

Figura 6.2 Empleo de API Rest en el navegador.

Con esta modificación, ya podemos comprobar cómo crear diferentes registros en nuestra aplicación en el navegador (Figura 6.2), y usar la navegación programática tras la respuesta correcta de la API.



La asincronía manejada por los métodos de la librería HttpClientModule puede ser utilizada para múltiples casos de uso. Por ejemplo, es común implementar el mapeo de respuestas erróneas con componentes de aviso al usuario, o el renderizado de spinners para mostrar que hay una petición en curso.

6.3. Despliegue a producción de aplicaciones Angular

El proceso de construcción o *build* de las aplicaciones para ser distribuidas en producción es una tarea muy sencilla en Angular: solamente es

necesario el uso del comando “build”. Para ello, en nuestro directorio de proyecto y en la terminal de Visual Studio Code, basta con completar el siguiente comando:

```
ng build
```

Con este proceso se lleva a cabo una compilación, proceso de *treeshaking* y minificado del código; con ello se extrae el código estrictamente necesario para el uso de la aplicación y se transforma en los archivos interpretables por el navegador, HTML, CSS y JavaScript. Tras unos minutos, la tarea da como resultado el paquete o bundle disponible en el directorio dist, en un directorio con el mismo nombre del proyecto (Figura 6.3).

Figura 6.3

Obtención del paquete o *bundle* de producción de una aplicación Angular.

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

- File Explorer:** Shows the project structure with a folder named "PRACTICA4" containing "src" and "dist". The "dist" folder contains files like "3rdpartylicenses.txt", "favicon.ico", "index.html", "main.e9a873cf84866e.js", "polyfills.25988913fd416ecc.js", "runtime.22e86f58c2257d87.js", and "styles.9612c9eb27865916.css".
- Code Editor:** Displays the file "create-book.component.ts" with code related to a form group and a service call.
- Terminal:** Shows the command "ng build" being run, followed by success messages: "Compiled successfully.", "Browser application bundle generation complete.", "Copying assets complete.", and "Index html generation complete.". It also displays the build statistics:

| Initial Chunk Files | Names | Raw Size | Estimated Transfer Size |
|-------------------------------|-----------|-----------|-------------------------|
| main.e9a873cf84866e.js | main | 231.23 kB | 60.19 kB |
| polyfills.25988913fd416ecc.js | polyfills | 33.02 kB | 10.59 kB |
| runtime.22e86f58c2257d87.js | runtime | 1.04 kB | 594 bytes |
| styles.9612c9eb27865916.css | styles | 1.03 kB | 379 bytes |

Initial Total | 266.33 kB | 71.64 kB

La terminal nos ofrece también información del resultado de la operación, en la que podemos observar que el tamaño de la aplicación se reduce al mínimo con el fin de mejorar el rendimiento en su descarga. El conjunto de archivos podrá ser distribuido desde cualquier servicio web convencional, como si se tratara de un sitio web estático, ya que el modelo SPA se basa precisamente en el renderizado del lado del cliente.



Resumen

- Angular incorpora su propia librería de gestión de peticiones HTTP y su correspondiente asincronía denominada HttpClientModule, que permite la incorporación de métodos en los servicios para el manejo de los datos de la aplicación.
- Los métodos de peticiones de HttpClientModule se caracterizan por utilizar el retorno de observables de la librería RxJS para el manejo de la asincronía de este proceso, mediante el uso de los métodos “pipe()” y “subscribe()”.
- El proceso de *build* en Angular extrae el código estrictamente necesario para el uso de la aplicación y se transforma en los archivos interpretables por el navegador, HTML, CSS y JavaScript, reduciendo su tamaño en un proceso de *treeshaking* y minificado.

Índice

| | |
|---|----|
| Esquema de contenido | 3 |
| Introducción | 5 |
| 1. Introducción a Angular | 7 |
| 1.1. El <i>framework</i> Angular | 7 |
| 1.2. Instalación de Angular CLI | 8 |
| Resumen | 13 |
| 2. Binding, componentes y plantillas | 14 |
| 2.1. Interpolación en Angular | 14 |
| 2.2. <i>Property binding</i> | 17 |
| 2.3. <i>Event binding</i> en Angular | 19 |
| Resumen | 22 |
| 3. Directivas y <i>pipes</i> en Angular | 23 |
| 3.1. Directivas en Angular | 23 |
| 3.3. <i>Pipes</i> en Angular | 30 |
| Resumen | 32 |
| 4. Comunicación entre componentes y servicios en Angular | 33 |
| 4.1. Comunicación entre componentes | 33 |
| 4.2. Servicios en Angular | 40 |
| Resumen | 43 |
| 5. Routing y formularios en Angular | 44 |
| 5.1. <i>Routing</i> en Angular | 44 |
| 5.2. Formularios reactivos en Angular | 46 |
| Resumen | 51 |
| 6. Comunicaciones HTTP y despliegue a producción | 52 |
| 6.1. Comunicaciones HTTP en Angular | 52 |
| 6.2. Asincronía y navegación programática | 56 |
| 6.3. Despliegue a producción de aplicaciones Angular | 58 |
| Resumen | 60 |