

Angular

Introducción

Angular es un framework para la creación de aplicaciones Web de lado de cliente. Podríamos decir que es una extensión de HTML que proporciona toda una serie de elementos y comportamientos dinámicos que hasta entonces solo podían conseguir mediante grandes cantidades de código JavaScript. Desde Angular 2, las versiones se han ido sucediendo hasta la actual Angular 7.

Angular x no es una actualización de Angular 1 o Angular JS, se trata más bien de una redefinición del framework que poco tiene que ver con aquella primera versión. entre otras diferencias:

- Angular x se basa en el uso de componentes, permitiendo la separación del código de la presentación
- Utiliza TypeScript, que es un lenguaje totalmente orientado a objetos

La creación de una aplicación Angular, se basa en definir plantillas HTML, a las que se les asocia un nombre de etiqueta, que luego puede ser aplicada en la página o páginas donde queramos utilizar dicha plantilla. Las plantillas no son meros bloques de texto HTML, contienen otra serie de elementos que proporcionan dinamismo y cierta "inteligencia" a la página. Este comportamiento dinámico es implementado mediante los **componentes**.

Configuración

Para poder trabajar con Angular, primeramente tenemos que tener instalado en nuestro equipo los siguientes paquetes software:

- Nodejs. Se deberá instalar la última versión desde la página <https://nodejs.org/en/>. Una vez instalado, podemos comprobar la versión desde línea de comandos escribiendo:

```
>node -v
```

- npm. Node Package Manager es una herramienta para la gestión de dependencias que, entre otras cosas, nos va a permitir instalar las librerías Angular que van a ser utilizadas por nuestra aplicación y transformar los componentes TypeScript (archivos .ts) en JavaScript (archivos .js). npm ya viene incorporado con Node, por lo que podemos simplemente comprobar la versión existente tecleando:

```
>npm -v
```

Para actualizar en todo momento a la última versión de npm podemos teclear:

```
>npm install npm@latest -g
```

Herramientas de desarrollo

Para crear aplicaciones Angular podemos trabajar con las siguientes herramientas:

- angular-cli. Es una herramienta de línea de comandos con la que podemos crear la estructura de un proyecto y ejecutarlo después. angular-cli se instala mediante npm:

Instalamos las herramientas CLI:

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

Y después instalamos el compilador de TypeScript:

```
>npm i -g typescript
```

Una vez instalado Angular Cli, podemos usar el siguiente comando para ver las versiones de los productos instalados:

```
>ng v
```

- Angular-IDE. Es un entorno de desarrollo basado en eclipse para crear proyectos angular de una forma más amigable. Podemos descargarlo desde la siguiente dirección:

<https://www.genuitec.com/products/angular-ide/>

Se puede también instalar un plugin en eclipse para configurarlo como angular IDE

Creación de un primer proyecto Angular

Para crear un proyecto Angular con Angular Cli simplemente escribiremos la siguiente instrucción desde línea de comandos:

```
>ng new NombreApp
```

Donde NombreApp es el nombre de la aplicación. Tras ello, veremos cómo angular comienza a crear la estructura de nuestro proyecto, algo que puede llevar unos minutos. Una vez finalizada la creación del proyecto, podemos entrar en la carpeta raíz del proyecto para ejecutarlo y ver el resultado. Primeramente, nos desplazamos a la carpeta raíz:

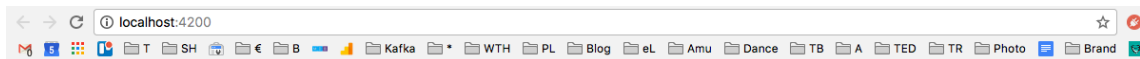
```
>cd NombreApp
```

Y allí escribimos:

```
>ng serve -o
```

Se abrirá un navegador en la dirección <http://localhost:4200/>

La app básica muestra un mensaje de bienvenida y unos enlaces en la parte inferior:



Welcome to app!



Here are some links to help you start:

- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

Si utilizamos angular IDE, crearemos el proyecto mediante la opción File->New-Angular Project. Se creará un proyecto básico igual al anterior

Estructura de un proyecto Angular

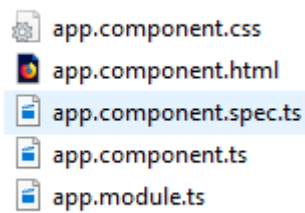
Un proyecto angular está formado por un gran número de carpetas y archivos, la gran mayoría de ellos son archivos de configuración y de librerías, de los que no tendremos que preocuparnos durante el desarrollo de nuestra aplicación.

En la siguiente imagen vemos el contenido del directorio raíz de un proyecto angular:

Nombre	Fecha de modifica...	Tipo	Tamaño
.git	16/02/2019 18:53	Carpeta de archivos	
e2e	16/02/2019 18:50	Carpeta de archivos	
node_modules	16/02/2019 18:53	Carpeta de archivos	
src	16/02/2019 18:50	Carpeta de archivos	
.editorconfig	16/02/2019 18:50	Archivo EDITORC...	1 KB
.gitignore	16/02/2019 18:50	Documento de tex...	1 KB
angular.json	16/02/2019 18:50	Archivo JSON	4 KB
package.json	16/02/2019 18:53	Archivo JSON	2 KB
package-lock.json	16/02/2019 18:53	Archivo JSON	369 KB
README.md	16/02/2019 18:50	Archivo MD	1 KB
tsconfig.json	16/02/2019 18:50	Archivo JSON	1 KB
tslint.json	16/02/2019 18:50	Archivo JSON	3 KB

En la carpeta node_modules es donde están todas las librerías de angular. La parte importante desde el punto de vista del programador está en el directorio src, más concretamente en el subdirectorio src/app, en el que encontramos el siguiente contenido:

Nombre

A file explorer window showing a list of files. The files are: app.component.css, app.component.html, app.component.spec.ts (highlighted), app.component.ts, and app.module.ts. Each file has a small icon to its left representing its type (CSS, HTML, or TypeScript).

De estos archivos los más importantes son:

- app.component.ts. Archivo de componente principal. Se trata de un archivo en código typescript donde se define el componente principal de la aplicación.
- app.module.ts. Archivo de módulo. Se trata de un archivo typescript en el que se le indica a angular todos los componentes y librerías propias que nuestro proyecto necesita para funcionar. No es necesario desde Angular 15.
- app.component.html. Archivo de plantilla. Define la estructura de la página asociada a cada componente, lo que comúnmente llamamos la vista

Seguidamente, analizaremos cada uno de estos elementos fundamentales en la creación de una aplicación angular

Componentes

El componente define el código de la aplicación y se implementa mediante una clase codificada en lenguaje TypeScript. La estructura de un componente es la siguiente:

```
@Component(  
  //características componente  
)  
  
export class ClaseComponente{  
  //implementación  
}
```

Todo componente debe estar declarado con la anotación **@Component** que establece los **metadatos asociados al componente**. Como mínimo, indicará la plantilla gestionada por el componente y el selector HTML asociado.

La clase del componente se declara con la palabra export, a fin de que pueda ser utilizado por algún otro componente o función externos. En el interior de la clase, se definirán las **propiedades y métodos del componente**, utilizando como hemos comentado anteriormente el lenguaje TypeScript

El lenguaje TypeScript es un superconjunto de JavaScript. Incluye soporte para tipos de datos y es totalmente orientado a objetos. El problema es que los navegadores actuales no soportan

TypeScript, por lo que el código deberá ser compilado a JavaScript para que pueda ser interpretado por los navegadores.

El siguiente bloque de código corresponde a la definición completa de un componente básico correspondiente a la aplicación de ejemplo desarrollada anteriormente, y cuyo contenido se encuentra en el archivo `app.component.ts`:

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

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

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

Lo primero que vemos en la línea superior es la sentencia `import`, que se utiliza para la importación de elementos externos, como directivas, funciones, etc., que necesitamos utilizar en nuestro código. Esta sentencia es necesaria para poder utilizar la anotación `@Component`, definida en la librería `core` de Angular .

Como comentamos anteriormente, mediante la anotación `@Component` establecemos los metadatos o información asociada al componente. En este caso, definimos dos propiedades:

- **selector**. Elemento HTML (atributo, etiqueta, clase de estilo, etc.) que asociaremos al componente.
- **templateUrl**. Dirección de la plantilla asociada al componente. Cuando utilicemos el elemento definido en *selector* dentro de una página, angular aplicará el bloque de etiquetas HTML definidos en el archivo `app.component.html`, además, instanciará el componente y resolverá las propiedades y llamadas a métodos definidas en la plantilla
- **styleUrls**. Lista de archivos de hojas de estilo que serán aplicadas sobre la plantilla
- **standalone**. A partir de Angular 15, los componentes personalizados se deben definir con la propiedad `standalone` al valor `true`. Esto implica, además, que deberán definirse las propiedades **imports** y **providers** que anteriormente se incluían en el `app.module`.

Si abrimos el archivo `app.component.html` veremos lo siguiente en la parte superior del bloque HTML:

```
<div style="text-align:center">
```

```
<h1>

  Welcome to {{ title }}!

</h1>
```

La expresión {{title}}, **conocida como interpolación**, hace referencia a la propiedad title del componente, que como vemos, está definida dentro de la clase AppComponent y es precisamente el valor de esta propiedad la que aparece en el mensaje de bienvenida

Si modificamos el valor de la propiedad *title* y volvemos a ejecutar el proyecto, comprobaremos como cambia el mensaje de bienvenida de la página

El archivo de módulo app.module.ts

Se trata de un componente especial, que le indica a angular los componentes que forman nuestra aplicación y como cargarlos. El contenido de este archivo es el siguiente:

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

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

Podemos apreciar las sentencias import para poder hacer uso de los elementos utilizados en el archivo, incluyendo nuestro propio componente.

La información se suministra mediante la directiva @NgModule, la cual cuenta con una serie de propiedades:

- declarations, declara los componentes que forman nuestra aplicación
- imports. Lista de componentes que deben ser importados

- providers. Lista de servicios. Más adelante se estudiarán
- bootstrap. Indica los componentes a instanciar. Como el componente tiene asociada una etiqueta, el componente se agregará al árbol DOM de la página donde aparezca dicha etiqueta.

La página index.html

En la carpeta src vemos una página llamada index.html cuyo contenido es el que se indica a continuación:

```
<!doctype html>

<html lang="en">

<head>

  <meta charset="utf-8">

  <title>MyApp2</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>
```

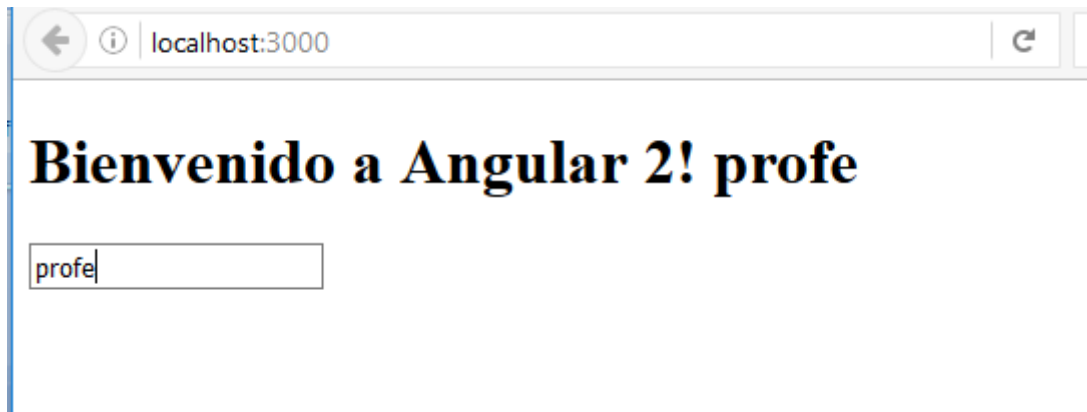
Lo destacable de esta página es la etiqueta `<app-root>`, que corresponde con el selector asociado a nuestro componente. Esta página es la página de inicio de la aplicación y es la que se carga en el navegador cuando accedemos a la aplicación mediante `http://localhost:4200`. En ese momento, **el servidor resuelve la etiqueta `<app-root>` instanciando el componente y sustituyendo dicha etiqueta por la plantilla asociada al mismo.**

Vinculación a datos

Una de las características principales que proporciona Angular es la **vinculación a datos**, que consiste en vincular la plantilla a propiedades de los componentes. Mediante este sistema, podemos recoger datos de usuario y volcarlos en propiedades del componente, así mismo, los valores de estas propiedades pueden volcarse directamente en la página, todo ello sin escribir una sola línea de código.

Para la recogida de datos de usuario y su volcado en propiedades del componente utilizaremos el atributo `[(ngModel)]` de la etiqueta `input`, al que le asignaremos el nombre de la propiedad del componente en la que queremos guardar el dato. Por su parte, utilizando la expresión de interpolación `{{propiedad}}`, volcaremos el valor de la propiedad del componente dentro de la página.

Como ejemplo, vamos a modificar el ejercicio anterior, incluyendo un campo de texto en el que el usuario pueda introducir un nombre y que según vaya escribiendo un texto en dicho campo, este texto se muestre a continuación del mensaje de saludo:



Para conseguir esto, tendremos que modificar el componente como se indica en el siguiente listado, definiendo la propiedad `nombre` en el mismo con un valor inicial:

:

```
export class AppComponent {  
    nombre='profe';  
}
```

Normalmente, las propiedades de un componente se definen de la siguiente forma:

`nombre_propiedad: tipo;`

Como vemos, en TypeScript las propiedades se definen dentro de la clase con el nombre de la propiedad, seguido de los dos puntos y el tipo. Pero si vamos a asignar un valor inicial a la propiedad, el tipo no se debe indicar, ya que este se infiere a partir del valor asignado:

`nombre_propiedad=valor_inicial`

Ahora, veamos el contenido del archivo de plantilla `app.component.html`:

```
<div style="text-align:center">  
    <h1>Bienvenido a Angular 6! {{nombre}}</h1>  
    <input [(ngModel)] = "nombre"/>  
</div>
```


Para volcar en la propiedad el texto escrito dentro de control, utilizamos dentro de la etiqueta `<input>` la expresión:

```
[(ngModel)]= 'nombre'
```

Finalmente, mediante el interpolador `{{nombre}}` mostraremos el contenido de la propiedad en cualquier parte de la página.

Si intentamos probar de nuevo el proyecto con los cambios anteriores, veremos que nos aparece un **error en esta línea** (si usamos el angular IDE aparecerá marcada en el editor):

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

Y es que, para poder utilizar la vinculación de componentes de un formulario a datos es necesario incorporar una nueva librería en el archivo de módulo, que deberá quedar ahora como se indica en el siguiente listado, mostrándote en fondo sombreado los cambios respecto a la versión anterior:

```
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 { }
```

Una vez ejecutado el proyecto y con la página cargada en el navegador, cualquier cambio que se produzca en el campo de texto se transmitirá a la propiedad y, por tanto, se verá reflejado en la parte superior de la página.

Paso de parámetros al componente

Se pueden pasar parámetros al componente desde la página, utilizando atributos de la etiqueta asociada. Por ejemplo, para pasar valores al componente a través del atributo "level", procederíamos de la siguiente forma:

```
<app-root level="5"></app-root>
```

En el componente, definimos la propiedad de la siguiente forma:

```
export class AppComponent {  
    @Input() level:string;  
}
```

Para poder utilizar el decorador @Input(), debemos importarlo desde el paquete core de Angular:

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

Podríamos haber utilizado también lo que se conoce como property binding para vincular un atributo a una propiedad del componente:

```
<app-root [level]="5"></app-root>
```

En este caso la propiedad level se definirá como number en el componente.

Utilización de clases de encapsulación

Cuando trabajamos con entidades que tienen asociados varios datos, como una persona, un empleado, etc., conviene definir una clase que encapsule estos datos. Dicha clase se puede definir dentro del propio archivo del componente.

Por ejemplo, si queremos solicitar una serie de datos al usuario a modo de ficha y queremos mostrárselos en la zona inferior de la página, podríamos definir el archivo app.component.ts de la siguiente manera:

```
import {Component} from 'angular2/core';
```

```
class Persona{  
    nombre:string;  
    apellido:string;  
    edad:number;
```

```

}

@Component({
  selector: 'ficha',
  templateUrl: './app.component.html'
})

export class AppComponent {
  persona:Persona;

  constructor(){
    this.persona=new Persona();
  }
}

```

Como vemos, hemos definido una clase Persona que encapsula los datos de una persona: nombre, apellido y edad. En el componente hemos creado una propiedad de este tipo, propiedad que inicializamos con un nuevo objeto Persona en el constructor del componente.

En cuanto al archivo de plantilla, será como se indica en el siguiente listado:

```

<div>

  <label>Nombre: </label>

  <input [(ngModel)]="persona.nombre" placeholder="Nombre"><br/>

  <label>Apellido: </label><input [(ngModel)]="persona.apellido"
placeholder="Apellido"><br/>

  <label>Edad: </label><input [(ngModel)]="persona.edad" placeholder="Edad"><br/>

</div>

<div>

  <label>Te llamas <b>{{persona.nombre}} {{persona.apellido}}</b> y tu edad es
<b>{{persona.edad}}</b></label><br/>

</div>

```

En la plantilla podemos observar el uso de la expresión *objeto.propiedad* para vincularnos a las diferentes propiedades del objeto Persona definido en la propiedad persona del componente.

Como podremos comprobar al ejecutar el proyecto, según vayamos escribiendo datos en los campos de texto irá modificándose el mensaje de información de la parte inferior de la página

Eventos

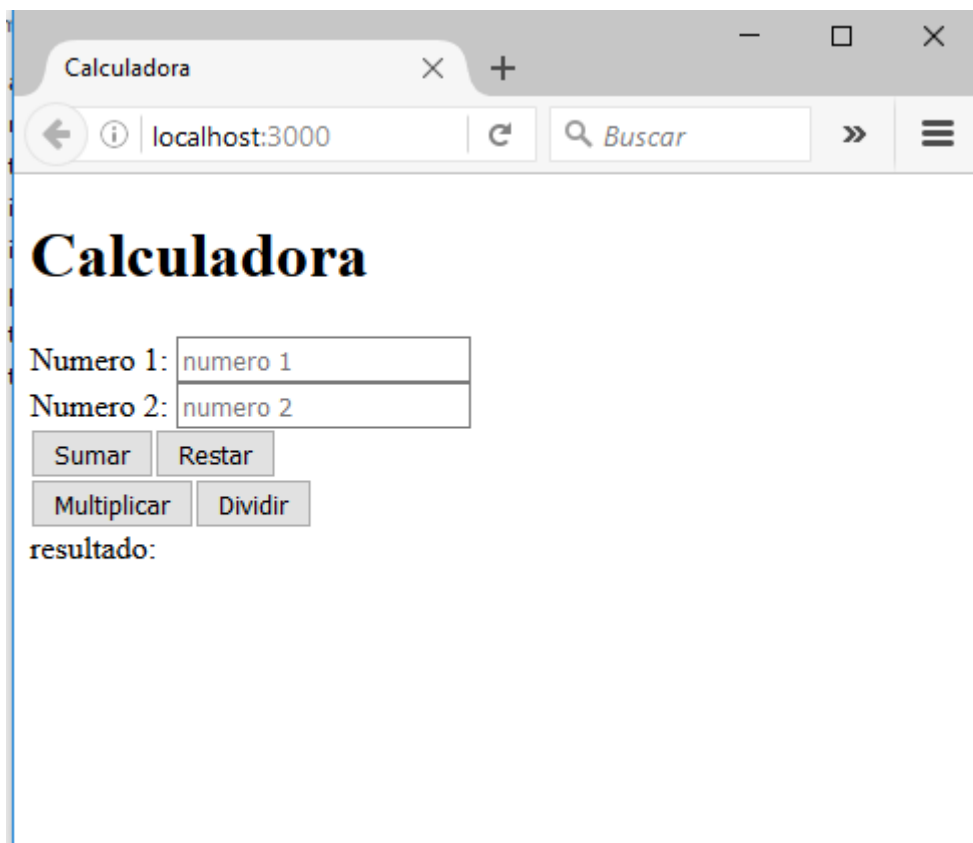
Una de las funcionalidades del código cliente de la capa Web es responder a sucesos o eventos que tienen lugar en la interfaz de usuario. En Angular 2, los métodos para responder a los eventos se definen en el componente y para asociarlos a un suceso de la interfaz, se utilizará la expresión (evento)="metodo()" dentro del elemento de la plantilla donde lo queramos capturar.

Por ejemplo, si queremos capturar el evento click de un botón y llamar al método procesar() del componente cuando se produzca dicho evento, definiríamos el botón de la siguiente manera:

```
<input type="button" (click)="procesar()" value="Pulsar">
```

Ejercicio resuelto

En el siguiente ejercicio vamos a realizar la clásica mini-calculadora, cuyo aspecto será el que se indica en la siguiente imagen:



La implementación del componente AppComponent será como se indica en el siguiente listado:

```
import {Component} from 'angular2/core';
```

```
@Component({  
  selector: 'calculator',  
  templateUrl: 'app.component.html'  
})  
  
export class AppComponent {  
  
  op1:string;  
  
  op2:string;  
  
  resultado:number;  
  
  onSumar() { this.resultado = parseInt(this.op1)+parseInt(this.op2); }  
  onRestar() { this.resultado = parseInt(this.op1)-parseInt(this.op2); }  
  onMultiplicar() { this.resultado = parseInt(this.op1)*parseInt(this.op2); }  
  onDividir() { this.resultado = parseInt(this.op1)/parseInt(this.op2); }  
}
```

El archivo de plantilla tendría el siguiente contenido:

```
<h1>Calculadora</h1>  
  
Numero 1: <input [(ngModel)]="op1" placeholder="numero 1"><br/>  
Numero 2: <input [(ngModel)]="op2" placeholder="numero 2"><br/>  
  
<input type="button" (click)="onSumar()" value="Sumar"><input type="button"  
(click)="onRestar()" value="Restar"><br/>  
  
<input type="button" (click)="onMultiplicar()" value="Multiplicar"><input type="button"  
(click)="onDividir()" value="Dividir"><br/>  
  
<div>resultado: {{resultado}}</div>
```

Como en el atributo *selector* de la anotación le hemos asignado el valor "calculator", por tanto, para utilizar el componente en la página principal index.html, el body de dicha página quedaría:

```
<body>
```

```
<calculator></calculator>
```

```
</body>
```

Referencias al objeto

Cuando capturamos eventos en una página, puede ser necesario que el **manejador de evento** **necesite una referencia al objeto sobre el que se ha producido el evento**, para ello, tenemos que pasar como parámetro el objeto **\$event** en la llamada al método manejador:

```
<div (mouseenter)="manejador($event)">
```

Desde el manejador, podemos obtener una referencia al objeto a través de la propiedad `target` del objeto `$event`:

```
manejador(evt){  
  
    evt.target.style.backgroundColor='yellow';  
  
}
```

Directivas integradas

Angular incorpora una serie de directivas que pueden ser utilizadas como atributo de cualquier etiqueta HTML para proporcionar una determinada funcionalidad. Ya hemos visto algunas directivas en ejemplos anteriores, como *ngModel* que nos permite vincular un control HTML con datos del componente, o las directivas para vincular eventos con métodos del componente, como la directiva *click*.

A parte de las ya comentadas, seguidamente analizaremos otras de las principales directivas integradas de Angular

Directiva NgStyle

Se utiliza para asignar una serie de estilos en línea. El valor de esta directiva es habitualmente una llamada a un método del componente, que devuelve la cadena de estilos a aplicar. Por ejemplo, supongamos que tenemos definido el siguiente método dentro del componente:

```
obtenerEstilos(){  
  
    let estilos={'font-size':'20px', 'color':'blue'};  
  
    return estilos;  
  
}
```

Si en la etiqueta `<div>` de la plantilla del ejemplo de la ficha quisiéramos aplicar los estilos generados por este método, podemos hacer uso de la directiva `NgStyle` tal y como se muestra en el siguiente ejemplo:

```
<div [ngStyle]="obtenerEstilos()">  
  
    <label>Te llamas <b>{{persona.nombre}} {{persona.apellido}}</b>  

```

y tu edad es {{persona.edad}}</label>

</div>

Directiva NgIf

A través de esta directiva podemos incluir o no un elemento dentro del árbol DOM de la página en función de una condición.

La utilización de esta directiva sigue el siguiente formato:

<etiqueta *ngIf="expresion">...</etiqueta>

El valor de *expresion* es evaluado como verdadero o falso. Puede ser una propiedad o una llamada a un método del componente. Si el **resultado es falso, el elemento será eliminado del árbol DOM y también todos los subelementos contenidos en el mismo**.

Observa el uso del * delante del nombre de la directiva, este prefijo se utiliza en aquellas directivas que manipulan el árbol DOM de la página.

Por ejemplo, si en el ejercicio de la ficha queremos que el texto informativo con los datos de la persona solo aparezca cuando se ha introducido un valor en el campo nombre, la forma de definir el <div> sería:

<div [ngStyle]="obtenerEstilos()" *ngIf="persona.nombre">

:

</div>

La expresión de cadena asignada a *ngIf es interpretada como **verdadera cuando es distinta de nulo o cadena vacía**.

Directiva NgFor

Se trata de una directiva repetitiva. Al incluirla en una etiqueta, esta aparecerá tantas veces como se indique en la expresión de iteración asignada a la directiva. El formato de esta expresión de iteración es:

<etiqueta *ngFor="let variable of array">...</etiqueta>

Ejercicio de ejemplo

Vamos a presentarte un ejercicio de ejemplo en el que se pondrán en juego las dos directivas últimas que hemos comentado. Se trata de una página que nos muestra la lista de nombres de personas, extraída de un array de objetos Persona. Al pulsar sobre uno de los nombres de la lista, nos aparecerá en la parte inferior un texto con los datos de dicha persona:

Lista de personas

- Luis
- Marta
- Ana
- Belén
- Marcos

Nombre: Marta

Email: n2@gmail.com

La lista de personas se definirá en un array de objetos JSON en el archivo .ts del componente.
El siguiente listado corresponde al código del archivo app.component.ts:

```
import {Component} from 'angular2/core';

export class Persona{

    nombre:string;

    dni:number;

    email:string;

}

@Component({

    selector: 'listado',

    template:`

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

        <div >

            <div>

                <ul>

                    <li *ngFor="let per of personas" (click)="seleccion(per)" >

                        {{per.nombre}}

                    </li>

                </ul>

            </div>

        </div>

    `

})
```



```

</div>\n\

<div *ngIf="seleccionado">

    <label>Nombre:</label> {{seleccionado.nombre}}<br/>

    <label>Email:</label> {{seleccionado.email}}<br/>


</div>

</div>
,

})

export class AppComponent{

    titulo="Lista de personas";

    seleccionado:Persona;

    public personas=aPersonas;

    seleccion(per){

        this.seleccionado=per;

    }

}

var aPersonas:Persona[]=[

    {"nombre":"Luis","dni":1111,"email":"n1@gmail.com"},

    {"nombre":"Marta","dni":2222,"email":"n2@gmail.com"},

    {"nombre":"Ana","dni":3333,"email":"n3@gmail.com"},

    {"nombre":"Belén","dni":4444,"email":"n4@gmail.com"},

    {"nombre":"Marcos","dni":5555,"email":"n5@gmail.com"}

];

```

Como podemos ver, utilizamos la expresión " let per of personas" para recorrer cada uno de los objetos Persona de la propiedad personas de tipo array definida en el componente. Al utilizarlo como argumento de *ngFor, se generarán tantos elementos como objetos haya en el array.

Directiva NgSwitch

Utilizamos esta directiva para mostrar diferentes elementos de un conjunto de ellos, en función del resultado de una expresión.

Dado el siguiente ejemplo:

```
<div [ngSwitch]="seleccionado.nombre">  
  
  <div *ngSwitchWhen="Ana">Vive cerca</div>  
  
  <div *ngSwitchWhen="Belén">Acaba de mudarse</div>  
  
  <div *ngSwitchWhen="Marcos">Vive ahí desde siempre</div>  
  
  <div *ngSwitchDefault>desconocidos</div>  
  
</div>
```

Se mostrará un mensaje adicional en función de la persona seleccionada. El valor de la cadena a evaluar se asigna a la directiva ngSwitch. Cada valor de comparación se asigna a *ngSwitchWhen, y si se produce coincidencia, ese elemento será el que aparezca en el árbol DOM.

La directiva *ngSwitchDefault en un elemento hace que se incluya dicho elemento en caso de no haber coincidencia.

Directiva innerHTML

se emplea para incluir un bloque HTML dentro de un elemento. El bloque html puede estar definido dentro de una propiedad del componente.

Por ejemplo, si en el componente tenemos definida la propiedad texto de esta manera:

```
texto="<b>hello</b>"
```

Para incluir ese contenido dentro de la plantilla sería:

```
<div [innerHTML]="texto"></div>
```

Creación de directivas personalizadas

Vamos a ver cómo podemos crear nuestras propias directivas para luego emplearlas como atributos de etiquetas en una página.

Una directiva se define mediante un componente, que en vez de anotarse con @Component se anota con @Directive.

Para ilustrar la creación de una directiva, vamos a desarrollar una sencilla directiva de ejemplo, cuya función será la de ocultar el elemento HTML sobre el que se aplica. La directiva la

definiremos en un archivo TypeScript al que llamaremos `oculta.directive.ts` y que tendrá que estar situado dentro de la subcarpeta `app` del proyecto. Este será el código del componente:

```
import { Directive, ElementRef } from 'angular2/core';

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

export class OcultaDirective {

  constructor(elemento: ElementRef) {

    elemento.nativeElement.style.visibility="hidden";

  }

}
```

Importamos dos elementos de `angular2`, la anotación `@Directive` y la clase `ElementRef`, que es el tipo del dato recibido en el constructor. Cuando utilizamos en una página el atributo asociado a la directiva (el nombre del atributo se indica en *selector*), Angular instancia la clase del componente y le pasa un objeto *ElementRef* al constructor.

A través del objeto *ElementRef* podemos obtener una referencia al elemento donde está aplicado el atributo para, entre otras cosas, poder acceder a las propiedades de estilo y modificar su comportamiento. En el ejemplo anterior, llamamos a la propiedad *nativeElement* de *ElementRef* para obtener el objeto etiqueta, al que le modificamos su propiedad de estilo *visibility* para ocultarlo.

La directiva será utilizada en la plantilla de otro componente, para ello, dicho componente tendrá que:

- Importar al clase de la directiva
- Indicar el nombre de la clase en el atributo `directives` de la anotación `@Component`

Por ejemplo, si quisiéramos aplicar la directiva para ocultar la etiqueta de título de nuestro componente que genera la lista de personas, así debería quedar la definición del componente:

```
import {Component} from 'angular2/core';

import { OcultaDirective } from './oculta.directive';

export class Persona{

  nombre:string;

  dni:number;

  email:string;

}

@Component({
```

selector: 'listado',

template:`

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

```
<div >
```

```
<div>\n\
```

```
<ul>
```

```
<li *ngFor="let per of personas" (click)="seleccion(per)" >
```

```
  {{per.nombre}}
```

```
</li>
```

```
</ul>
```

```
</div>\n\
```

```
<div *ngIf="seleccionado">
```

```
  <label>Nombre:</label> {{seleccionado.nombre}}<br/>
```

```
  <label>Email:</label> {{seleccionado.email}}<br/>
```

```
</div>
```

```
</div>
```

```
`;
```

```
directives:[OcultaDirective]
```

```
})
```

```
export class AppComponent{
```

```
  titulo="Lista de personas";
```

```
  seleccionado:Persona;
```

```
  public personas=aPersonas;
```

```

        seleccion(per){

            this.seleccionado=per;

        }

    }

    var aPersonas:Persona[]={

        {"nombre":"Luis","dni":1111,"email":"n1@gmail.com"},

        {"nombre":"Marta","dni":2222,"email":"n2@gmail.com"},

        {"nombre":"Ana","dni":3333,"email":"n3@gmail.com"},

        {"nombre":"Belén","dni":4444,"email":"n4@gmail.com"},

        {"nombre":"Marcos","dni":5555,"email":"n5@gmail.com"}

    ];

```

Te hemos indicado en fondo sombreado los cambios a realizar en el componente. Fíjate como en la sentencia *import* indicamos la ruta relativa del archivo .ts, en este caso, suponemos que ambos componentes están en el mismo directorio

A nivel de archivo main.ts no necesitamos hacer ningún cambio, ya que el único componente que se debe de pasar a la función *bootstrap* es el componente principal.

Datos de entrada en una directiva

En el ejemplo de directiva que te hemos presentado no se asignaba ningún valor a la misma, ya que la función a realizar no dependía de ningún valor de entrada. Sin embargo, muchas directivas requieren la asignación de un valor con el que trabajar, valor que es interpretado como una propiedad del componente.

Para definir un valor de entrada en la directiva, debemos declarar un atributo en la clase que recoja su valor, anotándolo con *@Input*:

```
@Input('nombre_directiva') variable:tipo;
```

Como siempre, lo mejor será desarrollar un ejemplo. Supongamos que queremos crear una directiva que al ser aplicada sobre una etiqueta esta cambie su color de fondo al pasar el ratón por encima de la misma. El color de fondo que se aplicará se proporcionará como valor de entrada a la directiva.

Además de lo que es la definición del valor de entrada, otra funcionalidad que habrá que desarrollar en la implementación del componente de directiva es la respuesta a los eventos sobre la etiqueta, en este caso del ratón. Dado que es una funcionalidad proporcionada por la directiva, esta debe ser implementada dentro de la misma, no en la plantilla.

Los métodos de respuesta a los eventos se implementarán en la clase de la directiva y, a través de un objeto JSON indicado en el atributo *hosts* de `@Directive`, se asocia cada evento con su método de respuesta. Veamos como quedaría el código de la nueva directiva:

```
import { Directive, ElementRef, Input } from 'angular2/core';

@Directive({

  selector: '[resalta]',

  host: {

    '(mouseenter)': 'onEnter()',

    '(mouseleave)': 'onLeave()'

  }

})

export class ResaltaDirective {

  private elemento: ElementRef;

  constructor(elemento: ElementRef) { this.elemento = elemento }

  //declara parámetro de entrada de la directiva

  @Input('resalta') fondoResalte: string;

  onEnter() { this.elemento.nativeElement.style.backgroundColor = this.fondoResalte; }

  onLeave() { this.elemento.nativeElement.style.backgroundColor = null; }

}
```

Por un lado, vemos en el atributo *host* como se vincula cada evento del ratón con el método de respuesta definido en la clase `ResaltaDirective`. Por otro lado, en esta clase observamos la definición del atributo *fondoResalte*, que gracias a la anotación `@Input` recoge el valor de entrada asignado a la directiva.

A continuación, vamos a aplicar esta directiva a los elementos `` de la plantilla del `AppComponent` que genera la lista de nombres de personas, de modo que quede resaltado cada nombre al pasar el ratón por encima:

Lista de personas

- Luis
- Marta
- Ana
- Belén
- Marcos

La definición del componente quedaría entonces:

```
import {Component} from 'angular2/core';
```

```
import { OcultaDirective } from './oculta.directive';
```

```
import {ResaltaDirective } from './resalta.directive';
```

```
export class Persona{
```

```
    nombre:string;
```

```
    dni:number;
```

```
    email:string;
```

```
}
```

```
@Component({
```

```
    selector: 'listado',
```

```
    template:`
```

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

```
    <div >
```

```
    <div>\n\
```

```
    <ul>
```

```
    <li *ngFor="let per of personas" [resalta]="color" (click)="seleccion(per)" >
```

```
        {{per.nombre}}
```

```
    </li>
```

```

    </ul>

    </div>\n\

    <div *ngIf="seleccionado">

        <label>Nombre:</label> {{seleccionado.nombre}}<br/>

        <label>Email:</label> {{seleccionado.email}}<br/>


    </div>

</div>
`
directives:[OcultaDirective,ResaltaDirective]
})

export class AppComponent{

    titulo="Lista de personas";

    color="blue";

    seleccionado:Persona;

    public personas=aPersonas;

    seleccion(per){

        this.seleccionado=per;

    }

}

var aPersonas:Persona[]=[

    {"nombre":"Luis","dni":1111,"email":"n1@gmail.com"},

    {"nombre":"Marta","dni":2222,"email":"n2@gmail.com"},

    {"nombre":"Ana","dni":3333,"email":"n3@gmail.com"},

    {"nombre":"Belén","dni":4444,"email":"n4@gmail.com"},

    {"nombre":"Marcos","dni":5555,"email":"n5@gmail.com"}

];

```


Como vemos, definimos una nueva propiedad `color` que se asigna a la directiva `resalta`. Si se hubiera querido asignar un color directamente a la directiva, en vez de hacer a través de una propiedad, se tendría que haber indicado el nombre del color entre comillas simples, por ejemplo:

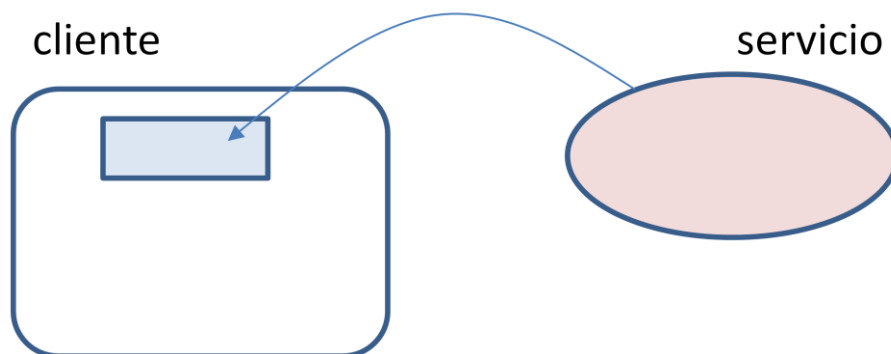
```
[resalta]="blue"
```

Observa también como, además de importar el archivo de definición de la directiva, la clase de la misma se indica en el atributo `directives` de `@Component`, junto con la directiva anterior.

Inyectables

En algunas aplicaciones, determinados componentes, a los que denominamos servicios, encapsulan la realización de tareas que deben llevarse a cabo en otros componentes de la aplicación, como puede ser por ejemplo el acceso a algún recurso externo.

La manera más sencilla de utilizar estos servicios por los componentes clientes es mediante la inyección de dependencia, que consiste en delegar en Angular la tarea de búsqueda e instanciación del servicio:



Con la inyección de dependencia, el componente cliente declara una variable del tipo del componente servicio. Angular se encargará de localizar el servicio, instanciarlo e inyectarlo en la variable cliente.

Para que un componente se pueda inyectar, debería ser declarado con la anotación `@Injectable`:

```
import { Injectable } from 'angular2/core';
```

```
@Injectable()
```

```
export class EjemploService {
```

```
:
```

```
}
```

La anotación `@Injectable` forma parte del core de Angular2.

De cara a utilizar el servicio dentro de la aplicación cliente, se deberán realizar las siguientes tareas:

- **Importar el componente.** Al igual que con cualquier otro elemento que queramos utilizar en un archivo TypeScript, será necesario importar el componente del servicio a través de la instrucción `import`
- **Declaración del proveedor.** El componente injectable es considerado un proveedor, que debe ser incorporado a través de la propiedad `providers` de la directiva `@Component`
- **Inyección del componente.** El servicio se inyecta en el constructor del componente cliente. Para ello, declararemos una variable en dicho constructor que sea del tipo del servicio.

Así sería como quedaría la definición del componente cliente del servicio `EjemploService` definido anteriormente:

```
import {Component} from 'angular2/core';  
  
import {EjemploService} from './ejemplo.service';  
  
@Component({  
  selector: 'saludo',  
  template: '<h1>Bienvenido a Angular 2!</h1>',  
  providers: [EjemploService]  
})  
  
export class AppComponent {  
  constructor(private srv:EjemploService){  
    srv.test(); //llamada a método del servicio  
  }  
}
```

El parámetro "srv" definido en el constructor como *private*, implica la inclusión implícita de un atributo privado con ese nombre en la clase del componente.

Peticiones Http

Una de las tareas más habituales que se llevan a cabo desde la capa cliente es la solicitud y/o envío de datos al servidor, para lo cual, la capa cliente tendrá que lanzar solicitudes Http al servidor.

Para realizar esta tarea, Angular 2 cuenta con el componente HttpClient localizado en la librería http. Este componente dispone de métodos para lanzar los distintos tipos de peticiones HTTP. Uno de estos métodos es get():

```
get(url: string, options?: RequestOptionsArgs): Observable<Response>
```

El método get() se utiliza para lanzar peticiones HTTP GET al servidor, recibe como parámetro la url del recurso solicitado y una lista de parámetros opcionales enviados en la petición.

La llamada a get() devuelve un objeto Observable, el cual proporciona un método llamado map() que permite transformar la respuesta en un objeto o array JSON, aunque a partir de Angular 7 ya no es necesario utilizar map(), ya que la respuesta por defecto es JSON; se puede usar un get<TipoRespuesta>(), para mapear directamente la respuesta a un objeto.

Para comprender mejor el uso de los métodos Http, vamos a crear una nueva versión del ejercicio de listado de personas, en el que los datos de las personas serán obtenidos de un recurso del servidor. Por ejemplo, supongamos que los datos de las personas se encuentran en el siguiente archivo personas.json:

```
[{"nombre":"Amalia","email":"amalia@gmail.com","dni":1111}, {"nombre":"Elena","email":"elena@gmail.com","dni":2222}, {"nombre":"Jorge","email":"jorge@gmail.com","dni":3333}]
```

Las instrucciones para el lanzamiento de la petición contra el recurso y la recuperación de la respuesta la implementaremos en un servicio que luego inyectaremos en nuestro componente principal. El código de dicho servicio, al que llamaremos PersonasService, se implementará en el archivo pesonas.service.ts:

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

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

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

//import 'rxjs/add/operator/map'

@Injectable()

export class PersonasService {

  private url="assets/personas.json";

  public resultado;

  constructor (private http: HttpClient) {
```

```

        this.resultado = this.http.get<Persona[]>(this.url);    //.map(response => response.json());
    }
}

```

Lo primero que podemos observar en el código anterior es que, además de los elementos `Injectable` y `HttpClient`, importamos una función llamada `map` de la sección de funciones `rxjs`. Esta función se expone como método de la clase `Observable`, pero debe ser importada para que se pueda utilizar.

Una vez dentro del código del componente, vemos como el objeto `Http` es inyectado en el constructor de la clase y almacenado en un atributo privado llamado `http`. En el interior del constructor, llamamos al método `get()` de dicho objeto para enviar la petición al recurso `personas.json`.

El método `get()` devuelve un objeto `Observable`, sobre el que aplicamos el método `map()` indicado anteriormente. Este método devuelve el resultado de transformar la respuesta recibida en la petición en un objeto `JSON`, utilizando la siguiente expresión `lambda`:

```
response => response.json()
```

El método `map` devuelve un nuevo objeto `Observable` que envía el array `JSON` generado por la expresión anterior. Este array `JSON` resultante es almacenado en la propiedad `resultado` del servicio.

El método `get()` devuelve un objeto `Observable` al que se suscribirá el componente.

En cuanto al componente `AppComponent` que va a hacer uso del servicio para recuperar la lista de personas, esta será su implementación:

```

import {Component,OnInit} from '@angular/core';
import {PersonasService} from './personas.service'

```

```

export class Persona{

    nombre:string;

    dni:number;

    email:string;

}

```

```

@Component({

    selector: 'listado',

    template:`

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

```

```
<div >
```

```
<div>
```

```
<ul>
```

```
<li *ngFor="let per of personas" (click)="seleccion(per)" >
```

```
  {{per.nombre}}
```

```
</li>
```

```
</ul>
```

```
</div>\n\
```

```
<div *ngIf="seleccionado">
```

```
  <label>Nombre:</label> {{seleccionado.nombre}}<br/>
```

```
  <label>Email:</label> {{seleccionado.email}}<br/>
```

```
</div>
```

```
</div>
```

```
  ,
```

```
  })
```

```
export class AppComponent implements OnInit {
```

```
  titulo="Lista de personas";
```

```
  seleccionado:Persona;
```

```
  public personas:Persona[];
```

```
  constructor(private pservice:PersonasService){}
```

```
  ngOnInit(){
```

```
    //asocia los datos recibidos a la propiedad
```

```
    //personas
```

```
    this.pservice.resultado.subscribe(data=>this.personas=data);
```

```
  }
```

```

        seleccion(per){

            this.seleccionado=per;

        }

    }
}

```

Además de la anotación `@Component`, se importan otros elementos que van a ser utilizados en la implementación del componente, como la interfaz `OnInit` que proporciona el método `ngOnInit` en el que programaremos las instrucciones de llamada al servicio, el componente de servicio `PersonasService`.

Sobre esto último, vemos como en la definición de los metadatos del componente incluimos el atributo *providers* en el que indicamos la lista de servicios a utilizar. Dado que vamos a hacer uso del servicio `Http` de Angular 2, se debe incluir en esta lista el elemento `HTTP_PROVIDERS`.

Ya en el código del componente, vemos como se inyecta el servicio `PersonasService` en el constructor del componente. El servicio es utilizado posteriormente en el método `ngOnInit()`, que es invocado por Angular durante la inicialización del componente (suceso que se produce tras la instanciación del objeto).

En `ngOnInit()` utilizamos la propiedad *resultado* del componente, que contiene el objeto `Observable` obtenido en la petición `Http`, para llamar al método `subscribe()`, al que le pasamos una implementación de la tarea a realizar, que es básicamente almacenar el dato resultante de la petición en la propiedad `personas` del componente:

```
data=>this.personas=data
```

Tras la inicialización del componente `AppComponent`, tenemos en la variable `personas` el array de objetos `Persona` obtenidos en la petición `get()` realizada al recurso `personas.json`

En cuanto al archivo `app.module`, quedará como se indica a continuación:

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';
import { PersonasService } from './personas.service';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [PersonasService],
})

```

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

El último ajuste que tendremos que hacer antes de hacer funcionar la aplicación es incluir el siguiente script dentro del archivo index.html:

```
<script src="node_modules/angular2/bundles/http.dev.js"></script>
```

Tras la instalación de las librerías y la ejecución del comando start de npm, se cargará la siguiente página en el navegador:

Una vez compilado, se abrirá el navegador y se mostrará:

Lista de personas

- Amalia
- Elena
- Jorge

Nombre: Elena

Email: elena@gmail.com

Envío de parametros en peticiones Http

-Parametros por queryString:

```
let url="http://localhost:8002/pedidos";

let params=new HttpParams();

params=params.append("usuario",usuario);

return this.http.get(url,{"params":params});
```

-Parametros form-encoded:

```
let url="http://localhost:8002/pedidos";

let params=new HttpParams();

params=params.set("idCurso",idCurso);
```

```

let heads=new HttpHeaders();

heads.set('Content-Type','application/x-www-form-urlencoded');

return this.http.post<Alumno[]>(url,params,{headers:heads});
return this.http.post<Alumno[]>(url,{params:params,headers:heads});

```

-Parametros como objeto JSON

```

let url="http://localhost:8002/alta";

let curso=new Curso();

curso.nombre="Java";

curso.precio=200;

curso.duracion=100;

return this.http.post<string>(url,curso);

```

Routing

En muchas aplicaciones, se requiere incluir enlaces en la vista de un componente que carguen otros componentes. Por ejemplo, un menú superior o lateral con enlaces a otros componentes. Para ello, utilizamos el módulo de routing, que puede incluirse durante la construcción del proyecto.

Esto hará que se cree el archivo app-routing.module.ts en la capeta app del proyecto. En este archivo se incluirá el mapeo entre las direcciones de los enlaces y los componentes asociados:

```

import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { BusquedaComponent } from './busqueda/busqueda.component';
import { AltaComponent } from './alta/alta.component';

const routes: Routes = [
  {
    path:'consulta',
    component:BusquedaComponent
  },
  {
    path:'alta',
    component:AltaComponent
  }
];

```

```

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

```



```
    exports: [RouterModule]
  })
  export class AppRoutingModule { }
```

Este archivo deberá estar también referenciado en el app.module.ts:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { AltaComponent } from './alta/alta.component';
import { BusquedaComponent } from './busqueda/busqueda.component';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';
```

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

A partir de ahí, simplemente incluiremos los enlaces en la vista del componente principal, indicando el path de cada componente asociado. La etiqueta <router-outlet> indicará donde deben mostradas las vistas de los componentes enlazados:

```
<div style="text-align:center">
```

```
<a routerLink="/alta">Alta de cursos</a>-<a routerLink="/consulta">Consultar Cursos</a>
```

```
<div>
```

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

```
</div>
```

```
</div>
```

Desde la vista de un componente, se podría utilizar el selector de otro componente si queremos incluirlo en el mismo. Usando el selector del otro componente también podemos pasar valores a sus posibles atributos.

Por ejemplo, en la vista de un componente llamado entrada:

```
<app-alta></app-alta>
```

```
<p>Esto es la vista del componente</p>
```

Distribución de una aplicación angular

Para generar los archivos de distribución de una aplicación angular que nos permita desplegar en cualquier servidor, nos desplazamos a la carpeta del proyecto y ahí escribimos el siguiente comando desde cmd:

```
>ng build --base-href=/app/
```

Donde app es la dirección que queremos asignarle a la aplicación.

Tras ello, se genera una carpeta dist con los archivos JavaScript del proyecto y el index.html. Llevamos todos estos archivos a un subcarpeta del wepapps de tomcat a la que daremos el nombre de la aplicación, y ya solo tenemos que acceder:

<http://localhost:8080/app/index.html>