Angular

Desarrollo de Aplicaciones Híbridas

Tabla de contenidos

1

Resumen

Resumen del tutorial de Angular.

Objetivos

- Conocer la arquitectura de las aplicaciones Angular
- Aprender a utilizar los componentes y directivas básicas de Angular para crear una aplicación web Angular.
- Aprender a crear y usar servicios
- Aprender a manejar las rutas de una aplicación

2

Introducción

- Angular es el framework de Google para el desarrollo de SPA (Single Page Applications). Las SPAs son un tipo de aplicación web en las que la interacción se realiza reescribiendo la página actual en lugar de cargando páginas completas desde el servidor.
- Angular ofrece una arquitectura que permite desarrollar aplicaciones de gran tamaño, mantenible y que pueda crecer en el futuro. Una aplicación pequeña se puede hacer en Javascript. Una aplicación grande no.
- Angular está orientado a componentes. Una página está compuesta por varios componentes que se comunican entre sí. Esto nos permite reutilizar componentes en múltiples páginas.
- Las aplicaciones Angular se desarrollan en TypeScript, aunque admiten otros lenguajes. TypeScript es un superconjunto tipado de JavaScript que se compila a JavaScript plano.

En la web oficial de Typestscript[1] existe un breve tutorial a TypeScript[2]. El handbook de TypeScript[3] también es un buen punto de referencia ofreciendo gran cantidad de ejemplos.

2.1. Versiones de Angular

- Angular 1 o AngularJS (2010)

[1] https://www.typescriptlang.org/index.html
[2] https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html
[3] https://www.typescriptlang.org/docs/handbook/basic-types.html

- Angular 2 (2016). Incoroporó gran cantidad de cambios respecto a AngularJS. Los proyectos de AngularJS no son válidos en Angular 2.
- Angular 4 (Marzo 2017). Incorpora algunas características nuevas respeto a Angular 2. En este caso sí hay compatibilidad.
- Angular 5 (Noviembre 2017). Nue<va versión con nuevas características manteniendo compatibilidad.

Angular y AngularJS son dos proyectos diferentes e incompatibles. Angular comprender las versiones desde Angular 2 en adelante. Actualmente, AngularJS[4] y Angular[5] siguen su desarrollo independiente.

[4] https://github.com/angular/angular.js
[5] https://github.com/angular/angular

3

Configuración del entorno

Instalaremos lo siguiente:

- Git[1].
- Node.js[2]. En Linux descargar el paquete de Node.js, moverlo a una carpeta `/opt/nodejs` y crearle un enlace simbólico en `/usr/bin/node` (`ln -s /opt/nodejs/bin/node /usr/bin/node`).
- npm[3] como Gestor de paquetes para JavaScript. npm se distribuye con Node.js.
- Consola de Angular (Angular CLI[4]) con `npm`.

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

La opción `-g` indicar que Angular CLI se instale globalmente por lo que lo podemos usar desde cualquier directorio.

1 https://git-scm.com/downloads
2 https://nodejs.org/en/
3 https://www.npmjs.com/
4 https://cli.angular.io/

Angular CLI

Angular CLI es una herramienta de interfaz de línea de comandos que permite crear proyectos, añadir archivos, y realizar tareas de desarrollo como testing, bundling y deployment.

4

Creación de nuestra primera aplicación Angular

1. Desde el directorio de trabajo, crear un proyecto nuevo

```
ng new pruebaAngular ❶
cd pruebaAngular
ng serve -o ❷
```

❶ Crea un diretorio y crea el proyecto. La operación de crear el proyecto llevará un tiempo mientras descarga los paquetes npm.

❷ Servir el proyecto (`serve`) y abrir navegador con la aplicación (`-o`). El proyecto se sirve a través de un servidor web que incorpora Angular y que ofrece *live reload*, lo que permiteque la aplicación se recargue automáticamente al hacer cambios en los archivos fuente.

2. La aplicación está disponible en `http://locahost:4200`.

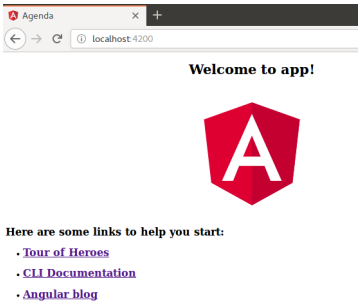


Figura 4.1. Primera aplicación Angular de ejemplo

Versiones de Node.js, npm y Angular CLI usadas en este tutorial

Para este tutorial necesitamos Node.js > 6.9.x y npm > 3.x.x.

```
$ node -v
v8.9.4
$ npm -v
3.5.2
$ ng -v


    _                      _                 ____ _     ___
   / \   _ __   __ _ _   _| | __ _ _ __     / ___| |   |_ _|
  / # \ | '_ \ / _` | | | | |/ _` | '__|   | |   | |    | |
 / ___ \| | | | (_| | |_| | | (_| | |      | |___| |___ | |
/_/   \_\_| |_|\__, |\__,_|_|\__,_|_|       \____|_____|___|
               |___/

Angular CLI: 1.7.2
Node: 8.9.4
OS: linux x64
Angular:
...
```

5

Organización de un proyecto Angular

Los componentes son los bloques básicos de las aplicaciones Angular. Muestran datos en la pantalla, están atentos a la entrada del usuario, y realizan una acción en función de esa acción.

Figura 5.1. Organización de una aplicación en componentes

Al crear el proyecto, Angular CLI ha creado el componente principal de la aplicación disponible en `src/app/app.component.ts`. Al definir un componente, definimos un `selector`, que es el luego se utilizará en la aplicación para incluir este componente. CLI define como selector para este componente `app-root`. Este valor se puede modificar.

Ejemplo 5.1. Componente principal de la aplicación Angular

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

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

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

❶ Objeto *metadata* que describe las características del componente.

❷ Selector del componente. Define una etiqueta HTML personalizada que la aplicación luego usará en los archivos HTML para incluir este componente (ver ejemplo siguiente).

❸ Plantilla externa asociado al componente escrita en HTML.

❹ Lista de hojas de estilos a aplicar al componente además de la propia de la aplicación (`src/styles.css`).

❺ Exportación de la clase para que puedan ser usadas por otros componentes

Ejemplo 5.2. Referencia a un selector

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>PruebaAngular</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> ❶
</body>
</html>
```

❶ Al incluir el selector `app-root` se incluirá en este archivo HTML su componente asociado `app-component` (ver ejemplo anterior).

También es posible incluir el código de la plantilla *inline* en lugar de en un archivo externo. El código de la plantilla irá entre *backsticks* o apóstrofes.

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

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

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

❶ Uso de `template` para definir la plantilla *inline*. No olvidar la coma del final si hay más elementos en los metadatos del componente.

5.1. Explorando el código

Podemos cambiar el texto de bienvenida modificando `src/app/app.component.html`. Ahí vemos que aparece un encabezado `<h1>` con el texto que hemos visto al abrir la aplicación

```
<h1>
  Welcome to {{ title }}!
</h1>
```

Lo cambiaremos por

```
<h1>
  Bienvenido a {{ title }}!
</h1>
```

Interpolación

`{{title}}` es lo que se conoce como sintaxis de interpolación de binding. Se trata de un mecanismo por el podemos asignar un valor a un atributo dentro de un componente. Esta interporlación de binding conecta el componente TypeScript con la plantilla HTML presentando el valor de `title` en el HTML y toma valor en el momento de renderizar la página. En el ejemplo la asignación del valor se realiza en `src/app/app.component.ts`. Esto evita manpipular el DOM, ya sea directamente o mediante jQuery.

```
<h1>
  Bienvenido a {{ title }}! ❶
</h1>
```

❶ Presentación (interpolación) del valor en `src/app/app.component.html`

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

Asignación del valor en `src/app/app.component.ts`

Cambiaremos el valor de `title` en `src/app/app.component.ts` por `mi primera aplicación Angular`.

```
export class AppComponent {
  title = 'mi primera aplicación Angular';
}
```

El estilo del componente lo podemos cambiar en `src/app/app.component.css`. Añadiremos el estilo para `<h1>`.

```
h1 {
  color: blue;
```

```
  font-size: 250%;
}
```

Tras estos cambios nuestra aplicación tiene un nuevo aspecto!!

Código del proyecto disponible en GitHub[1]

MVC en Angular

En Angular, el componente juega el rol de controlador y la plantilla representa la vista.

5.2. Archivos de configuración

- `package.json`: Fichero de configuración de dependencias.
- `tsconfig.json`: Fichero de configuración de Typescript, el lenguaje de interacción con Angular.
- `angular-cli.json`: Este fichero sólo está disponible si el proyecto se ha creado con CLI. Establece nombres de carpetas, prefijo de la aplicación y los archivos que se incluyen al crear el proyecto.
- `.editorconfig`: Parámetros de configuración para el editor respecto al proyecto (charset, tamaño del espacio de tabulación, …)
- `.gitignore`: Contiene la lista de archivos que no están sujetos a control de versiones del repositorio Git inicializado al crear el proyecto.

Angular CLI también ha incluido un archivo `README.md` con información de interés, como la creación de componentes (`ng generate component component-name`), build del proyecto (`ng build`) y ejecución de pruebas (`ng test`).

[1] https://github.com/ualmtorres/pruebaAngular

5.3. Estructura de carpetas

- `e2e`. Carpeta para pruebas
- `node_modules`. Contiene los paquetes instalados.
- `src`. Contiene el código del proyecto.

  # `app`

  # `assets`. Contiene las imágenes utilizadas en el proyecto.

  # `environment`. Detalles acerca de los entornos de producción y desarrollo.

5.4. Archivos importantes

- `src/index.html`. Es el archivo que se muestra en el navegador. `<body>` contiene `<app-root></app-root>`. Este el el selector que se usa en el archivo `src/app/app.component.ts`, el cual mostrará el archivo `src/app/app.component.html`.
- `src/app/app.module.ts`. Indica a Angular cómo construir la aplicación. También incluye los componentes que forman la aplicación.
- `src/app/app.component.ts` es el componente inicial. En nuestro caso asigna el valor `app` a la variable `title` y muestra el contenido del template `app.component.html` aplicándole el estilo `app.component.css`.
- `src/styles.css`. Estilos globales de la aplicación.
- `src/test.ts` Punto de entra a los tests unitarios.

6

Creación de componentes

Con Angular CLI también podemos añadir nuevos componentes a la aplicación (`ng generate component` *`new-component`*).

```
ng generate component heroes
```

Al crear un componente con Angular CLI ocurre lo siguiente:

1. Se modifica el archivo `src/app/app.module.ts` incluyendo el nuevo componente

```
import { HeroesComponent } from './app.component'; ❶
...
@NgModule({
  declarations: [
    ...
    HeroesComponent, ❷
    ...
  ],
  ...
  bootstrap: [AppComponent], ❸
  ...
```

❶ Importación del nuevo componente
❷ Declaración del nuevo componente
❸ Especificación del componente inicial de la aplicación

2. Se añade una carpeta a `src/app` con el nombre del nuevo componente (`heroes`). La nueva carpeta incluye los archivos TypeScript, HTML y CSS del nuevo componente:

   - `heroes.component.css`
   - `heroes.component.html`
   - `heroes.component.spec.ts`
   - `heroes.component.ts`

Ejemplo 6.1. Archivo TypeScript del componente creado

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

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

 constructor() { }

 ngOnInit() {
 }
}
```

En el archivo de la clase del componente encontramos un *decorador* `@Component` que especifica los metadatos para el componente. Angular CLI genera tres propiedades de estos metadatos:

- `selector`: Selector para el componente. Es el nombre que se usar para hacer referencia al componente desde una plantilla que incluya este componente.

El selector tiene que ser único para que pueda ser referenciado sin equívoco desde cualquier plantilla HTML. El valor predeterminado para configurar el nombre del selector es `app-`*`nombreComponente`*. En este caso el componente `heroes` da lugar al selector `app-heroes`. No obstante, este valor puede ser modificado

> y asignarle otro nombre asegurando siempre que sea único.

- `templateUrl`: Archivo HTML de la plantilla del componente
- `styleUrls`: Array de archivos de estilos CSS propios del componente

Como consideración adicional:

- La clase del componente se exporta para que otro componente pueda importarla.
- La clase incluye el constructor `constructor()` para que lo podamos personalizar.
- La clase incluye el método `ngOnInit()` para colocar en él cualquier código de inicialización necesaria para el componente.

7

Creación de una aplicación de galería de imágenes

```
ng new mygallery
```

En lugar de tener que descargar todos los paquetes cada vez que quieras crear un proyecto, puedes tener un proyecto como base actualizado para replicar cada vez que tengas que crear un nuevo proyecto. Luego bastaría con hacer unos ligeros cambios para adaptarlo al nuevo proyecto. Los cambios se tendrían que hacer en:

- `angular-cli.json`. Modificar el elemento `name` de `project`.
- `app.e2e-spec.ts`. Modificar la cadena del parámetro en el método `describe`
- `package.json`. Modificar el elemento `name`
- `README.md`. Modificar el título del documento
- `index.html`. Modificar el `<title>`

Actualización de dependencias

Para actualizar las dependencias de un proyecto tendremos que tener instalado previamente el comprobador de dependencias de Node.js. Lo instalaremos con

```
sudo npm install -g npm-check-updates
```

Después, el comando `ncu` ejecutado sobre la carpeta del proyeto a actualizar nos devolverá las dependencias a actualizar y nos indicará cómo proceder para realizar la actualización en caso de ser necesario

```
$ ncu

 @types/node   ~6.0.60  →  ~9.4.7
 jasmine-core   ~2.8.0  →  ~3.1.0
 protractor     ~5.1.2  →  ~5.3.0
 ts-node        ~4.1.0  →  ~5.0.1
 typescript     ~2.5.3  →  ~2.7.2

Run ncu with -u to upgrade package.json
```

7.1. Un primer prototipo

7.1.1. Usar Bootstrap en el proyecto

En la web de Boostrap[1] encontraremos las indicaciones para usar Boostrap en un proyecto. Aquí utilizaremos la opción Boostrap CDN y colocaremos el enlace en `index.html`

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
bootstrap/4.0.0/css/bootstrap.min.css">
```

[1] https://getbootstrap.com/

7.1.2. Crear la barra de navegación

En primer lugar creamos el componente para la barra de navegación.

```
ng generate component navbar --flat (1)
```

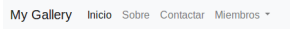
(1) El flag `--flat` evita la creación de una carpeta `navbar` para los archivos del componente. En este caso, los archivos se situarán dentro de la carpeta `app`.

Recuerda que al crear el componente con Angular CLI, además de crearse los archivos del componente, se actualiza `app.module.ts`.

```
$ ng generate component navbar --flat
  create src/app/navbar.component.css (0 bytes)
  create src/app/navbar.component.html (25 bytes)
  create src/app/navbar.component.spec.ts (628
 bytes)
  create src/app/navbar.component.ts (269 bytes)
  update src/app/app.module.ts (391 bytes) (1)
```

(1) `app.module.ts` ha sido actualizado para importar el nuevo componente y añadirlo a las declaraciones.

A continuación, incluimos en el archivo `navbar.component.html` el código para crear una barra de navegación disponible en la documentación de Boostrap para crear barras de navegación[2]. Haremos unos pequeños cambios para personalizarla y que quede de esta forma. El código está disponible en el repositorio GitHub del proyecto[3].



Por último, incluiremos el componente de la barra creada en su componente padre. Para ello, modificamos el archivo `app-component.html` para incluir el selector de la barra de navegación reemplazando su contenido por

[2] https://getbootstrap.com/docs/4.0/components/navbar/
[3] https://github.com/ualmtorres/mygallery

```
<app-navbar></app-navbar>
```

7.1.3. El componente de la galería

Crearemos un componente para la galería con Angular CLI.

```
ng generate component gallery
```

Como ya sabemos, se creará una carpeta con los archivos del componente y se actualizará `app.module.ts`

Por ahora, la galería mostrará una serie estática de imágenes. Añadiremos el código siguiente a `gallery.component.html`

```
<div class="container">
  <div class="row">
    <a href = "#">
      <div class = "col-md-3 col-sm-4 col-xs-6"><img class="img-
responsive" src="https://images.pexels.com/photos/9051/pexels-photo.jpg?
h=150" /></div>
    </a>
    <a href = "#">
      <div class = "col-md-3 col-sm-4 col-xs-6"><img class="img-
responsive" src="https://images.pexels.com/photos/23475/pexels-
photo.jpg?h=150" /></div>
    </a>
    <a href = "#">
      <div class = "col-md-3 col-sm-4 col-xs-6"><img class="img-
responsive" src="https://images.pexels.com/photos/9050/pexels-photo.jpg?
h=150" /></div>
    </a>
    <a href = "#">
      <div class = "col-md-3 col-sm-4 col-xs-6"><img class="img-
responsive" src="https://images.pexels.com/photos/754998/pexels-
photo-754998.jpeg?h=150" /></div>
    </a>
  </div>
</div>
```

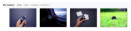
Y definiremos estos estilos en `gallery.component.css` para el componente definido

```
img {
    box-shadow: 0px 1px 6px 1px gray;
    margin-bottom: 30px;
}
img:hover {
    -webkit-filter: grayscale(1);
}
```

Por último, añadimos el tag del componente gallery `<app-gallery>` a su componente padre `app.component.html` para mostrar la galería

```
<app-navbar></app-navbar>
<app-gallery></app-gallery>
```

El resultado debería ser algo similar a este:



7.2. Mejora de la aplicación. Separación de datos y presentación

Hasta ahora, la galería de imágenes está almacenando de forma conjunta los datos de las imágenes y su presentación. En este apartado estructuraremos la galería con esta relación jerárquica de componentes.

```
gallery
|_image-list
  |_image
```

A continuación crearemos un subcomponente de `gallery` al que denominaremos `image-list`. Lo denominamos subcomponente porque lo creareamos dentro de `gallery` y no dentro de `app`.

También crearemos un subcomponente de `image-list` al que denominaremos `image`. En este caso usaremos el parámetro `--flat` para indicar a Angular CLI que no cree una carpeta aparte para el componente, sino que cree los archivos en la misma ruta desde la que se está creando.

```
cd src/app/gallery
ng generate component image-list
```

```
cd image-list
ng generate component image --flat
```

La estrucuctura de archivos de `gallery` deberá ser así:

```
gallery/
├── gallery.component.css
├── gallery.component.html
├── gallery.component.spec.ts
├── gallery.component.ts
└── image-list
    ├── image.component.css
    ├── image.component.html
    ├── image.component.spec.ts
    ├── image.component.ts
    ├── image-list.component.css
    ├── image-list.component.html
    ├── image-list.component.spec.ts
    └── image-list.component.ts
```

El haber creado los archivos del componente `image` dentro del mismo directorio que `image-list` no contraviene el que los componentes tengan luego en la presentación (HTML) la relación jerárquica `image-list` *está formado por* `image`.

El código está disponible en el repositorio GitHub del proyecto[4].

7.2.1. Creación del modelo

Pese a no ser obligatorio, es conveniente que los modelos de una aplicación estén agrupados dentro de un mismo directorio. En nuestro caso, crearemos un directorio `models` dentro del directorio `app`. Desde `models` crearemos una clase `image` con Angular CLI. Esto creará un archivo TypeScript (`image.ts`) para la clase dentro de `models`.

```
ng generate class image
```

[4] https://github.com/ualmtorres/mygallerymodel

Dentro de la clase definiremos su constructor

```
constructor(public imageURL: string, public author: string, public
 website: string) {}
```

El componente `image` corresponde al *Modelo* en el patrón MVC

7.2.2. Refactorización de `gallery.component.html`

El código de `gallery.component.html` será sustituido por el selector del componente `image-list`. Su código ahora se distribuirá entre las plantillas y las clases de los componentes `image-list` e `image.

Ejemplo 7.1. `gallery.component.html` **refactorizado**

```
<app-image-list></app-image-list>
```

7.2.3. Separación de los datos de la presentación mediante `image-list.component.ts`

Este componente está dedicado a inicializar los valores de la lista de imágenes a mostrar (URLs de las imágenes, autores, …)

```
import { Component, OnInit } from '@angular/core';
import { Image } from '../../models/image'; ❶

@Component({
  selector: 'app-image-list',
  templateUrl: './image-list.component.html',
  styleUrls: ['./image-list.component.css']
})
export class ImageListComponent implements OnInit {
  images: Image[] = [ ❷
    {'imageURL':'https://images.pexels.com/photos/9051/pexels-photo.jpg?
h=150', 'author':'Oliur Rahman', 'website':'http://photos.oliur.com'},
```

```
    {'imageURL':'https://images.pexels.com/photos/23475/pexels-
photo.jpg?h=150', 'author':'Donald Tong', 'website':''},
    {'imageURL':'https://images.pexels.com/photos/9050/pexels-photo.jpg?
h=150', 'author':'Pixabay', 'website':'http://pixabay.com'},
    {'imageURL':'https://images.pexels.com/photos/754998/pexels-
photo-754998.jpeg?h=150', 'author':'Tarun Netha Amballa', 'website':''}
  ];

  constructor() { }

  ngOnInit() {
  }

}
```

❶ Importación de a clase imagen para poder crear un array de objetos `image`

❷ Creación del array de objetos `image`. Los objetos `image` se pueden crear en JSON o con `new Image(`*`param1`*`, `*`param2`*`, …)`

7.2.4. Presentación de la lista de imágenes

Para presentar la lista de imágenes nos valdremos de la directiva `* ngFor`

Directiva * ngFor

La directiva `*ngFor` (no olvidar el asterisco) instancia una plantilla una vez por cada elemento de un iterable.

```
<ul>
<li *ngFor="let i of [1,2,3]">Elemento {{i}}</li>
</ul>
```

devuelve

- Elemento 1
- Elemento 2
- Elemento 3

La plantilla de la lista de ima#enes iterará sobre el array `images` del componente. Además, interactuará con el componente `image` para pasarle en cada iteración la imagen a presentar. Para indicar que se quiere pasar un objeto al componente `app-image`, el objeto se encerrará entre corchetes (p.e `[image]`).

Consulta la documentación oficial de Angular[5] para saber más de la interacción de componentes.

Ejemplo 7.2. `image-list.component.html`

```
<div class="container">
  <div class="row">
    <app-image *ngFor="let image of images" [image]="image"></app-
image> ❶
  </div>
</div>
```

❶ `[image]` indica una interacción con el componente `<app-image>`. En `<app-image>` se recibirá el objeto en `[image]`. `[image]` toma en cada iteración una imagen (`image`) del bucle `* ngFor`

7.2.5. El componente para la imagen

El componente para la imagen recibe de `image-list.component.html` una propiedad de entrada (`[image]`) con un decorador `@Input`. Por tanto, el componente tendrá que importar `Input` de `@angular/core`.

[5] https://angular.io/guide/component-interaction

Ejemplo 7.3. `image.component.ts`

```
import { Component, OnInit, Input } from '@angular/core'; ❶
import { Image } from '../../models/image'; ❷

@Component({
  selector: 'app-image',
  templateUrl: './image.component.html',
  styleUrls: ['./image.component.css']
})
export class ImageComponent implements OnInit {
  @Input() image: Image; ❸

  constructor() { }

  ngOnInit() {
  }

}
```

❶ Importación de Input

❷ Importación de la clase de la imagen para poder usarla

❸ Propiedad de entrada `image` enviada desde `image-list.component.html`

7.2.6. Presentación de la imagen

Ya sólo queda usar las interpolación para presentar los datos de ima imagen. Esto lo haremos accediendo a la propiedad `imageURL` de `image`.

Ejemplo 7.4. `image.component.html`

```
<a href = "#">
    <div class = "col-md-3 col-sm-4 col-xs-6"><img class="img-
responsive" src="{{image.imageURL}}" /></div>
</a>
```

Como los estilos para las imágenes estaban en `gallery.component.css` las imágenes han perdido su estilo. Basta con mover los estilos definidos a `image.component.css`.

7.3. Mejora de la aplicación. Creación de un servicio

El problema que tiene actualmente la aplicación de galería de imágenes es que el componente de galería de imágenes sabe demasiado acerca de la cómo construir la lista de imágenes. Es más, tiene encargado la construcción de la lista de imágenes a partir de sus datos.

La solución está en delegar el trabajo de crear la lista de imágenes a otro componente y crear lo que se conoce como un *servicio*. Un servicio nos va a permitir ocultar los detalles acerca de cómo recuperar datos y compartir datos entre componentes de nuestra aplicación.

Desde el directorio base del proyecto crearemos un servicio denominado `image` con Angular CLI. Esto creará los archivos TypeScript (`image.service.ts` e `image.service.spec.ts`) para el servicio dentro de `services`

```
ng generate service image --module=app ❶
```

❶ Creación del servicio `image` y actualización de `app.module.ts` con los datos del servicio `image`.

Para que este servicio pueda ser más adelante siguiendo el patrón de *Inyección de dependencias* el servicio tiene que ser importado en `app.module.ts` y registrado en la lista de `providers`.

Ejemplo 7.5. Fragmento de `app.module.ts` tras definir el servicio `image`

```
....
import { ImageService } from './image.service';
....
@NgModule({
  ....
  providers: [ImageService],
  ....
})
....
```

Ejemplo 7.6. `image.service.ts`

```
import { Injectable } from '@angular/core';
import { Image } from './models/image'; <1> Importar modelo imagen

@Injectable() ❶
export class ImageService {
  images: Image[] = [ ❷
    {'imageURL':'https://images.pexels.com/photos/9051/pexels-
photo.jpg?h=150', 'author':'Oliur Rahman', 'website':'http://
photos.oliur.com'},
    {'imageURL':'https://images.pexels.com/photos/23475/pexels-
photo.jpg?h=150', 'author':'Donald Tong', 'website':''},
    {'imageURL':'https://images.pexels.com/photos/9050/pexels-
photo.jpg?h=150', 'author':'Pixabay', 'website':'http://
pixabay.com'},
    {'imageURL':'https://images.pexels.com/photos/754998/pexels-
photo-754998.jpeg?h=150', 'author':'Tarun Netha Amballa',
 'website':''}
  ];
  constructor() { }

  getImages() { ❸
    return this.images;
  }
}
```

Importar modelo `image`

❶ El decorador `@Injectable` indica que este servicio puede tener dependencias inyectadas.

❷ Inicializar array de imágenes

❸ Creación de un método que devuelva el array de imágenes

Ejemplo 7.7. `image-list.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { Image } from '../../models/image';
import { ImageService } from '../../image.service'; ❶

@Component({
  selector: 'app-image-list',
  templateUrl: './image-list.component.html',
  styleUrls: ['./image-list.component.css']
})
export class ImageListComponent implements OnInit {
  images: Image[] = []; ❷

  constructor(private imageService: ImageService) { } ❸

  ngOnInit() {
    this.images = this.imageService.getImages(); ❹
  }

}
```

❶ Importación del componente del servicio
❷ Declaración del array de imágenes
❸ Modiifcación del constructor para incluir el servicio de imágenes
❹ Inicialización del array de imágenes con lo que devuelva el servicio

De esta forma hemos conseguido aislar el componente de lista de imágenes de los detalles de cómo obtener la lista de imágenes. Ahora, el componente sólo se limita a usar el servicio de imágenes para obtener la lista de imágenes.

7.3.1. Mostrando los detalles de una imagen

En este apartado veremos cómo implementar la funcionalidad de mostrar los detalles de una imagen al hacer clic sobre ella.

1. Añadir el evento de clic a `image-list.component.html`. Al hacer clic sobre una imagen llamaremos a un método `onSelect()` pasándole como argumento la imagen seleccionada

```
<app-image *ngFor="let image of
 images" [image]="image" (click)="onSelect(image)"></app-image>
```

2. Añadir a `image-list.component.ts` una variable de instancia de tipo `Image` denominada `selectedImage`. Esta variable representa la imagen seleccionada de la lista

```
....
export class ImageListComponent implements OnInit {
  images: Image[] = [];
  selectedImage: Image;
  ....
}
```

3. Añadir a `image-list.component.ts` el método `onSelect()` que inicialice `selectedImage` con la imagen seleccionada.

```
  onSelect(image: Image) {
    this.selectedImage = image;
  }
```

4. Añadir en la parte superior de `image-list.component.html` el selector de la imagen de detalle (`app-image-detail`) para mostrar el detalle de las imágenes en la parte superior de la lista de imágenes. Este elemento pasará al componente `ImageDetail` la imagen seleccionada mediante `[selectedImage]` que será recibida mediante `@Input()`.

```
<app-image-detail [selectedImage]="selectedImage"></app-image-detail>
```

5. Añadir a `image-detail.component.ts` la recepción de `selectedImage`.

```
import { Component, OnInit, Input } from '@angular/core'; ❶
import { Image } from '../../models/image'; ❷

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

```
export class ImageDetailComponent implements OnInit {
  @Input() selectedImage: Image; ❸

  constructor() { }

  ngOnInit() {
  }
}
```

❶ Importar `Input`

❷ Importar el modelo de la imagen para poder usarlo

❸ Variable de instancia creada a partir del valor recibido

6. Mostrar los detalles en `image-detail.component.html`

```
<div class="container image-detail"> ❶
  <div class = "row">
    <div class = "col-sm-6 col-xs-12">
      <img class="img-responsive" src="{{selectedImage.imageURL}}" /
> ❷
    </div>
    <div class = "col-sm-6 col-xs-12">
      <h1>{{selectedImage.author}}</h1> ❸
      <h2><a href =
 "{{selectedImage.website}}">{{selectedImage.website}}</a></h2>
    </div>
  </div>
</div>
```

❶ El estilo `image-detail` lo definiremos en la hoja de estilos del componente de detalle.

❷ Mostrar la imagen seleccionada

❸ Mostrar otras propiedades de la imagen seleccionada

Ejemplo 7.8. `image-detail.css`

```
.image-detail {
    margin: 20px auto;
}
```

7.4. Routing

Con routing podemos dividir una aplicación en áreas que podemos llamar páginas, e indicar a Angular qué paginas mostrar en función de la ruta especificada.

Las aplicaciones Angular son SPA (Single Page Applications). En realidad sólo existe una página aunque tengamos la sensación de estar navegando por páginas diferentes. Routing es la técnica que lo permite.

Como ejemplo, generemos dos componentes para asociarlos a los elementos de menú `Sobre` y `Contactar`. Estos serán los componentes `about` y `contact` que generaremos con Angular CLI tal y como se muestra a continuación. A cada uno le asignaremos una ruta.

```
ng generate component about
ng generate component contact
```

Las rutas se definen en `app.module.ts`.

Ejemplo 7.9. `app.module.ts`

```
...

import { Routes, RouterModule } from '@angular/router'; ❶
import { ModuleWithProviders } from '@angular/core';
...
const appRoutes: Routes = [
  {path: '', redirectTo: '/gallery', pathMatch: 'full'}, ❷
  {path: 'gallery', component: GalleryComponent}, ❸
  {path: 'about', component: AboutComponent},
  {path: 'contact', component: ContactComponent}
];

@NgModule({
  ...
  imports: [
    RouterModule.forRoot(appRoutes), ❹
    BrowserModule
  ],
  ...
})
...
```

❶ Importación de módulos necesarios para el routing

❷ Redirigir el path vacío a la galería

❸ Establecer los componentes a cargar en cada ruta

❹ Modificación de los imports del módulo

A continuación, debemos indicar dónde colocar el contenido de cada ruta. Para ello, Angular cuenta con el tag `<router-outlet>`. En nuestro caso, cambiaremos el tag en `app.component.html` para que muestre en la página principal el contenido de la ruta seleccionada.

Ejemplo 7.10. `app.component.html`

```
<app-navbar></app-navbar>
<router-outlet></router-outlet>
```

En el componente de la barra de navegación `navbar.component.html` debemos hacer varios cambios.

- Los atributos `href` serán sustituidos por `[routerLink]` para convertirlos en links Angular y que usen las rutas definidas.
- Aplicar el estilo de elemento activo del menú a la opción seleccionada:

```
      <li routerLinkActive="active"> ❶
        <a class="nav-link" [routerLink]="['/']">Inicio <span class="sr-
only">(current)</span></a> ❷
      </li>
      <li routerLinkActive="active">
        <a class="nav-link" [routerLink]="['/about']">Sobre</a> ❸
      </li>
      <li routerLinkActive="active">
          <a class="nav-link" [routerLink]="['/contact']">Contactar</
a> ❹
      </li>
```

❶ Utilizaremos `routerLinkActive` en cada link para aplicarle un estilo diferente al link activo

❷ Uso de `[routerLink]` para establecer la ruta de inicio

❸ Uso de `[routerLink]` para establecer la ruta de `/about`

❹ Uso de `[routerLink]` para establecer la ruta de `/contact`

Por último, debemos hacer un último cambio en el componente `image.component.html`, ya que si hacemos clic sobre una imagen veremos que se recarga su página de detalle, pero desaparece al instante. Para ello, haremos dos cambios:

- Eliminar el atributo `href` en el link de la imagen en `image.component.html`.

```
<a> ❶
    <div class = "col-md-3 col-sm-4 col-xs-6"><img class="img-
responsive" src="{{image.imageURL}}" /></div>
</a>
```

❶ Quitar el atributo `href` para que no se produzca una regarga de página al mostrar detalles

- Añadir a los estilos de `image.component.css` este estilo

```
a:hover {
    cursor: pointer;
}
```