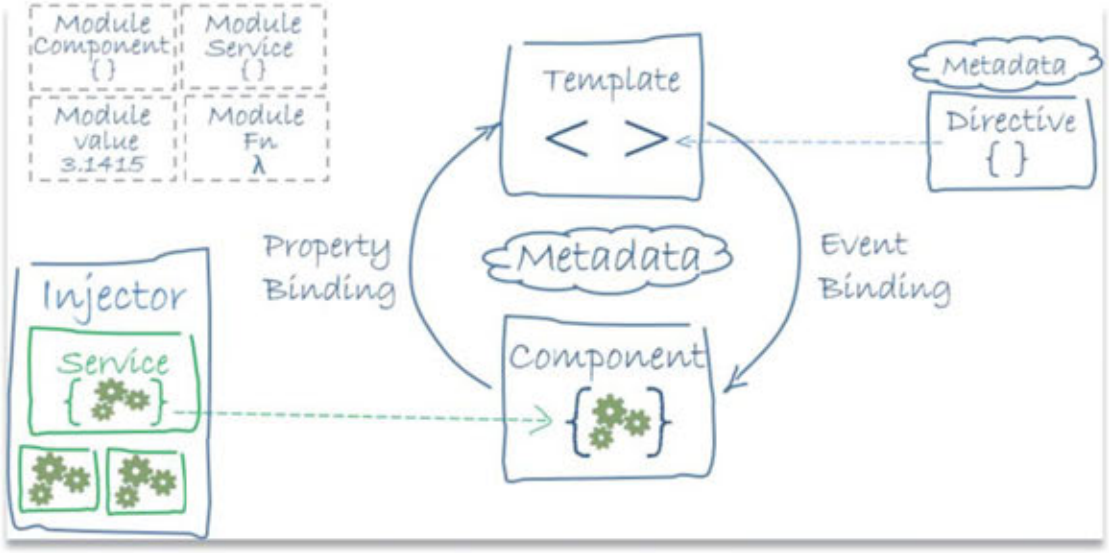
3 Módulos y Componentes

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

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



3.1 Módulos en Angular

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

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

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

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

1. En primer lugar una serie de instrucciones de importación de las librerías o paquetes Angular, asi como otros elementos externos que emplearemos en el módulo.

2. En segundo lugar, @NgModule, un decorador que recibe un objeto de metadatos que definen el módulo. Estos metadatos son los siguientes:

    - *Declarations*. Las declaraciones son las llamadas vistas de un módulo. Hay 3 tipos de vistas o declaraciones, los componentes, las directivas y los *pipes*.

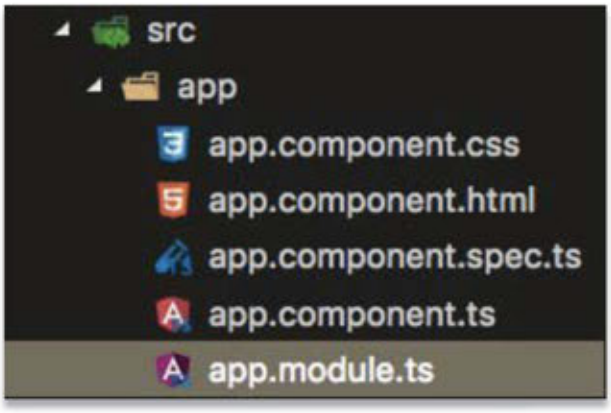
    - *Imports*. En este apartado se indican las dependencias o paquetes que empleará este módulo, cuyo origen se define en las importaciones al inicio del archivo.

    - *Providers*. Son los servicios utilizados por el módulo, disponibles para todos los componentes, y que centralizan la gestión de datos o funciones para inyectarlos en los componentes.

    - *Bootstrap*. Este metadato define la vista raíz de la aplicación y es utilizado solo por el módulo raíz. No confundir con el popular *framework* de estilos del mismo nombre.

3. Y el tercer y último bloque de código, es la instrucción de exportación del módulo como una clase Angular, que será posteriormente introducido en el archivo JavaScript principal de la aplicación.

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



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

src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
                                                          ①
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent
  ],                                                      ②
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule { }  ③
```

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

② El decorador @NgModule, que incluye:

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

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

③ Para finalizar, el archivo exporta la clase AppModule.

3.2 Componentes en Angular

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

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

Un componente estará también compuesto por tres bloques de código:

1. Imports. Las sentencias de importación de los diferentes elementos que empleará el componente.

2. Decorador @Component. Con al menos, los siguientes metadatos:

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

   Template. El archivo html con la vista del componente.

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

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

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

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

src/app/app.component.ts

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

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

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

En el cual:

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

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

- El selector. Que define en qué etiqueta html se renderizará este componente. En este componente raíz, la etiqueta es app-root que es la que tenemos en el archivo index.html.

- La template o plantilla. Que define el contenido html del componente. En este caso se usa templateUrl y la ruta del archivo template, siendo app.component.html.

- La referencia de estilo. Aquí se definen los estilos CSS particulares de ese componente. En este caso se usa styleUrls y la ruta del archivo de estilo del componente, siendo app.component.css.

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

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

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

src/app/app.component.html

```
<div style="text-align:center">
  <h1>
    Welcome to {{title}}!!
  </h1>
  <img width="300" src="…" />
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank"
href="https://angular.io/docs/ts/latest/tutorial/">Tour of Heroes</a></h2>
  </li>
  <li>
    …
  </li>
</ul>
```

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

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

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

src/app/app.module.ts

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

…
  declarations: [
    AppComponent
  ],
…
```

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

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

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

src/main.ts

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

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

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

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

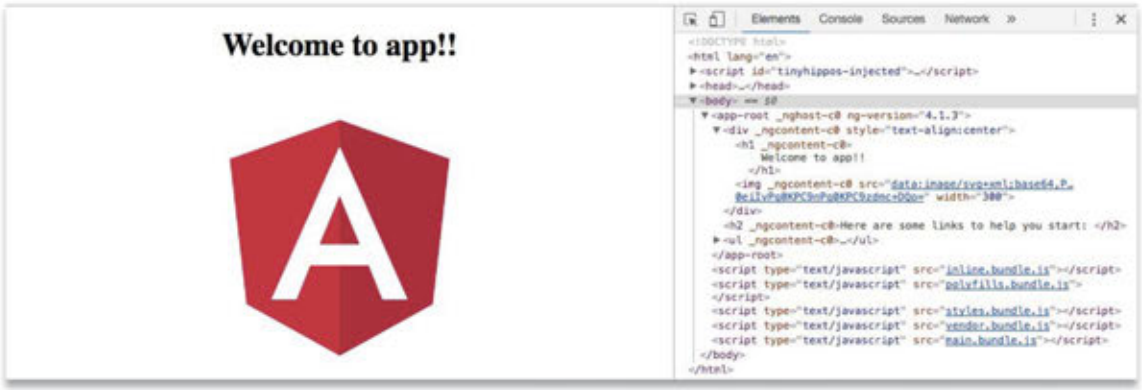
Que ① importa AppModule y en su última línea ② define AppModule como el punto de entrada de la aplicación.

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

src/index.html

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

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



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

src/app/app.component.html

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

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

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

src/app/app.component.ts

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

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

src/app/app.component.html

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

Volvemos al navegador y comprobamos:

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

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

src/app/app.component.css

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

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

src/app/app.component.html

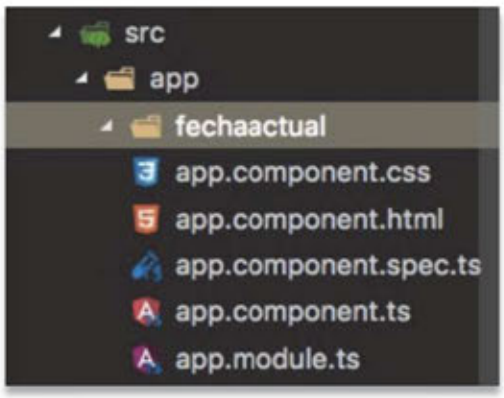
```
<div class="encabezado">
  <h1>¡Hola {{ destino }}!</h1>
</div>
```

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

3.3 Creación de un nuevo componente

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

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



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

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

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

src/app/fechaactual/fechaactual.component.ts

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

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

```
})

export class FechaactualComponent {
        hoy: any = new Date();                ③
}
```

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

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

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

src/app/fechaactual/fechaactual.component.html

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

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

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

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

src/app/app.module.ts

```
…
import { FechaactualComponent } from './fechaactual/fechaactual.component'; ①
…
```

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

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

② Y en el array de declaraciones la añadimos.

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

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

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

src/app/app.component.html

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

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

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

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

src/app/fechaactual/fechaactual.component.ts

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

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

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

3.4 Creación de componente con Angular CLI

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

La sintaxis del comando para generar nuevos componentes es:

ng generate component <nombredelcomponente>

o su versión abreviada:

ng g c <nombredelcomponente>

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

ng generate component copyright

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

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

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

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

src/app/copyright/copyright.component.ts

```
export class CopyrightComponent implements OnInit {

  copyright: String = '© ACME S.A.';
  hoy: any = new Date();

  constructor() { }

  ngOnInit() {
  }
```

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

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

src/app/copyright/copyright.component.html

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

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

src/app/app.component.html

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

Podemos comprobar en el navegador la correcta implementación del componente.

3.5 Anidado de Componentes

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

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

src/app/copyright/copyright.component.html

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

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

src/app/app.component.html

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

Siendo el resultado final el mismo.

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

src/app/copyright/copyright.component.html

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

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

src/app/fechaactual/ fechaactual.component.ts

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

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

src/app/app.component.html

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

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