1 - BASE

> Base para una aplicación Angular

Vamos a crear una **base sobre la que programar una aplicación Angular 7** profesional. Usaremos el *CLI* para generar una estructura sobre la que crecer. Será como una semilla para un desarrollo controlado. La idea de árbol se usa en muchas analogías informáticas. La emplearemos en dos conceptos básicos en Angular: **los módulos y los componentes**.

Partimos de la aplicación tal cómo la dejamos en el Hola Mundo en Angular. Al finalizar tendrás un esqueleto del que colgar módulos y componentes funcionales.

> Código asociado a este artículo en *GitHub*: AcademiaBinaria/angular-board/

1. Módulos

Los módulos son **contenedores para almacenar los componentes y servicios** de una aplicación. En Angular cada programa se puede ver como un árbol de módulos jerárquico. A partir de un módulo raíz se enlazan otros módulos en un proceso llamado importación.

1.1 Anatomía de un módulo

Antes de usar cualquier módulo hay que conocerlo. En Angular **los módulos de declaran como clases de TypeScript**. Estas clases, habitualmente vacías, son decoradas con una función especial. Es la función `@NgModule()` que recibe un objeto como único argumento. En las propiedades de ese objeto es donde se configura el módulo.

Mira el módulo `AppModule` original que genera el CLI en el fichero `app.module.ts`.

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

1.1.1 Importación de otros módulos

El módulo `App` también se conoce como **módulo raíz** porque de él surgen las demás ramas que conforman una aplicación. La asignación de los nodos hijos se realiza en la propiedad `imports:[]`, que es un array de punteros a otros módulos.

> En la situación original el módulo principal depende un módulo *custom* pre-generado (el `AppRoutingModule` que usarás más adelante) y de otro *del framework* para la presentación en el navegador (el `BrowserModule`).

Veremos esto con más profuncidad en el punto 4.

1.2 Generación de módulos

Hasta ahora los módulos involucrados son librerías de terceros o que se crearon mágicamente con la aplicación. Es hora de **crear tu primer módulo**. Para eso usaremos otro comando del *cli*, el `ng generate module`. En una ventana del terminal escribe:

```
ng g m core
```

Esta es la sintaxis abreviada del comando `ng generate` el cual dispone de varios planos de construcción o *blueprints*. El que he usado aquí es el de `module` para la construcción de módulos.

> Si no te gusta teclear en la terminal, también puedes lanzar estos comandos desde Angular Console

El resultado es la creación del fichero `core/core.module.ts` con la declaración y decoración del módulo `CoreModule`. Este módulo te servirá de **contenedor para guardar componentes** y otros servicios esenciales para nuestra aplicación. Pero eso lo veremos más adelante.

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

Por ahora hay que asegurar que **este módulo será importado por el raíz, el AppModule**. Para ello comprobaremos que la línea de importación del módulo principal esté parecida a esto:

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

> El módulo raíz, al igual que como verás más tarde con el componente raíz, es un tanto especial. Su nombre oficial es App, aunque todo la documentación se refiere a él como raíz o root.

2. Componentes

Los módulos son contenedores. Lo primero que vamos a guardar en ellos serán componentes. **Los componentes son los bloques básicos de construcción de las páginas web en Angular 7**. Contienen una parte visual en html (la Vista) y una funcional en Typescript (el Controlador).

> La aplicación original que crea el CLI nos regala un primer componente de ejemplo en el fichero `app.component.ts`. Según la configuración del CLI este componente puede haber sido creado en un sólo fichero o hasta cuatro: (el controlador, con la vista y los estilos en ficheros propios y fichero extra para pruebas unitarias).

2.1 Anatomía de un componente

Los componentes, como el resto de artefactos en Angular, serán **clases TypeScript decoradas** con funciones específicas. En este caso la función es `@Component()` que recibe un objeto de definición de componente. Igual que en el caso de los módulos contiene las propiedades en las que configurar el componente.

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

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

Los componentes definen nuevas etiquetas HTML para ser usados dentro de otros componentes. Excepcionalmente en este caso por ser el componente raíz se consume en el página `index.html`. El nombre de la nueva etiqueta se conoce como *selector*. En este caso la propiedad `selector: "app-root"` permite el uso de este componente dentro de otro con esta invocación `<app-root></app-root>`. En este caso el componente raíz.

> Particularidades del componente raíz. Su nombre oficial es `AppComponent`, y su selector debería llamarse `app-app`. Pero su *selector real* es `app-root`, formado a partir del prefijo de la aplicación y su supuesto nombre oficioso. Observa el prefijo `app` que se usará en todos los componentes propios, fue asignado por defecto durante la generación de la aplicación. Puede personalizarse usando el modificador `--prefix` de `ng new` y en distintos ficheros de configuración. Volviendo al componente raíz; está destinado a ser usado en la página principal, en el `index.html`. Eso obliga a registrarlo de una manera especial en el módulo raíz. Hay que incluirlo en el array `bootstrap: [AppComponent]`, es ahí donde se incluyen los componentes con la capacidad de lanzar *bootstrap* la aplicación.

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

Y en el `index.html`

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

La plantilla representa la parte visual del componente. De forma simplificada, o cuando tiene poco contenido, puede escribirse directamente en la propiedad `template` del objeto decorador. Pero es más frecuente encontrar la plantilla en su propio fichero *html* y referenciarlo como una ruta relativa en la propiedad `templateUrl`.

La propiedad `styles` **y su gemela** `stylesUrl` **permiten asignar estilos** *CSS, SASS o LESS* al componente. Estos estilos se incrustan durante la compilación en los nodos del *DOM* generado. Son exclusivos del componente y facilitan el diseño y maquetación granular de las aplicaciones.

Los estilos, ausentes en este ejemplo, podrían incluirse como un array de cadenas, o llevarse a un fichero propio como en el caso de la vista.

```
<div style="text-align:center">
  <h1>Welcome to {{ title }}!</h1>
  <img width="100" src="./assets/logo.png" />
</div>
...
```

En la clase del componente nos encontraremos la implementación de su funcionalidad. Normalmente expondrá propiedades y métodos para ser consumidos e invocados de forma declarativa desde la vista.

Una aplicación web en Angular se monta como un **árbol de componentes**. El componente raíz ya viene creado y convenientemente declarado; ahora toca darle contenido mediante una estructura de página y las vistas funcionales.

2.2 Generación de componentes

Para **crear nuevos componentes** vamos a usar de nuevo el comando `generate` del *CLI* . Pero ahora con los planos para construir un componente. La sintaxis completa del comando `ng generate component` o abreviadamente `ng g c` permite crear componentes en diversas formas.

Casi **todas las páginas tienen una estructura** similar que de forma simplista queda en tres componentes. Uno para la barra de navegación, otro para el pie de página y otro intermedio para el contenido principal.

Ejecuta en una terminal estos comandos para que generen los componentes y comprueba el resultado en el editor.

```
ng g c core/shell
ng g c core/shell/header
ng g c core/shell/main
ng g c core/shell/footer
```

Fíjate en el componente del fichero `shell.component.ts`. Su estructura es igual a la del componente raíz. Destaca que el nombre del componente coincide con el nombre del selector: `app-shell` y `ShellComponent`. Esto será lo normal a partir de ahora. Sólo el componente raíz tiene la excepción de que su nombre `App` no coincide con su selector `root`.

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-shell',
  templateUrl: './shell.component.html',
  styles: []
})
export class ShellComponent implements OnInit {
  constructor() {}
  ngOnInit() {}
}
```

Y esta es su vista asociada. La cual es de nuevo una composición de otros selectores. Estamos creando un frondoso árbol de componentes.

```
<app-header></app-header>
<app-main></app-main>
<app-footer></app-footer>
```

3 Visibilidad entre componentes

La clave del código limpio es **exponer funcionalidad de manera expresiva pero ocultar la implementación**. Esto es sencillo con los lenguajes de POO, pero en HTML no era nada fácil. Con la **programación basada en componentes** podemos crear pantallas complejas, reutilizables y que a su vez contengan y oculten la complejidad interna a sus consumidores.

3.1 Componentes públicos y privados

Los componentes no deciden por sí mismos su **visibilidad**. Cuando un componente es generado se declara en un módulo contenedor en su propiedad `declares:[]`. Eso lo hace visible y utilizable por cualquier otro componente del mismo módulo. Pero **si quieres usarlo desde fuera tendrás que exportarlo**. Eso se hace en la propiedad `exports:[]` del módulo en el que se crea.

La exportación debe hacerse a mano incluyendo el componente en el array, o indicarse con el *flag* `--export` para que lo haga el *cli*. Esto es lo que se ha hecho en el módulo *Core* para poder exportar el componente `shell`.

```
@NgModule({
  declarations: [ShellComponent, HeaderComponent, MainComponent, FooterComponent],
  imports: [CommonModule, RouterModule],
  exports: [ShellComponent]
```

```
})
export class CoreModule {}
```

> **Los componentes privados suelen ser sencillos**. A veces son creados para ser específicamente consumidos dentro de otros componentes. En esas situaciones interesa que sean privados y que generen poco ruido. Incluso, en casos extremadamente simples, si usamos el modificador `--flat` ni siquiera generan carpeta propia.

Por supuesto que `HeaderComponent` necesitará la propiedad `title` y también la moveremos desde `app.component.ts`. Dejando de esa manera el componente raíz en los huesos.

3.2 Importación y exportación entre módulos

Que un componente sea público es la primera condición para que se consuma fuera de su módulo. Ahora falta que quién lo quiera usar el selector `<app-shell>` importe su módulo `CoreModule`. Esto lo haremos en el `AppModule` para que lo use el `AppComponent`.

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

Como regla general, **cuando en una plantilla se incruste otro componente**, Angular lo buscará dentro del propio módulo en el que pretende usarse. Si no lo encuentra entonces lo buscará entre los componentes exportados por los módulos que hayan sido importados por el actual contenedor.

> Ahora mismo en `AppComponente` sólo puedo usar a `ShellComponent`, que es el único componente accesible. En `ShellComponent` se pueden usar sus vecinos *Header, Main y Footer*. Es un práctica recomendada el mantener el `AppModule` y el `AppComponent` tan simples como sea posible. Para ello movemos todo lo que podemos al módulo de ayuda `CoreModule` distribuyendo el contenido de `app.component.html` en las plantillas de *Header, Main y Footer* que corresponda.

3.2.1 Dos mundos paralelos: imports de Angular e import de TypeScript

Si es la primera vez que ves código TypeScript te llamarán la atención las primeras líneas de cada fichero. En el `app.module.ts` son algo así:

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

Estas **sentencias de importación son propias del lenguaje** y nada tienen que ver con Angular. En ellas se indica que este fichero importa el contenido de otros ficheros *TypeScript*. La importación se realiza en base a convenios personalizables. Si empieza con `./` entonces se busca a través de la ruta física relativa al fichero actual. En otro caso se busca en el directorio `node_modules` y se trata como código de terceros.

> En general no tendrás que preocuparte de estas importaciones físicas, pues el *VSCode* y las extensiones esenciales se encargan de hacerlo automáticamente según lo uses en tu código

4. Transitividad y Organización

4.1 Transitividad en una cadena de módulos

Un problema que reforzará tu conocimiento sobre el sistema modular surgirá al mover la etiqueta `<router-outlet></router-outlet>` del `app.component.html` al componente *Main*. En su vista `main.component.html` tendrás algo así.

```
<h2>Here are some links to help you start:</h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial">Tour
of Heroes</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">CLI
Documentation</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular
blog</a></h2>
  </li>
</ul>
<router-outlet></router-outlet>
```

Todo son etiquetas *html* estándar salvo la última `<router-outlet></router-outlet>`. El propósito de este componte lo veremos en la próxima lección dedicada a enrutado. Pero por ahora más que una ayuda es un dolor de cabeza porque es un desconocido para el módulo `CoreModule`. Resulta que el `RouterOutletComponent` está declarado en un módulo del *framework* llamado `RouterModule`. Dicho módulo fue importado de manera automática durante la generación del código inicial, pero ¿Dónde?.

Como digo el tema del enrutado es un capítulo aparte, pero las relaciones de los módulos debes conocerlas cuanto antes. Durante la generación inicial se crearon dos módulos: el `AppModule`, ya estudiado, y su asistente para enrutado `AppRoutingModule`. Este último aún no lo hemos visitado. Su contenido es:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
const routes: Routes = [];
@NgModule({
```

```
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Obviando la por ahora inútil instrucción `.forRoutes(routes)`, llama la atención que este módulo es dependiente del famoso `RouterModule`, es decir lo importa en su array `imports:[]`. Pero además va y lo exporta haciendo uso de la interesante **propiedad transitiva de los módulos**. Cada módulo puede exportar sus propios componentes o los de terceros. Incluso puede exportar todo un módulo al completo. Al hacerlo así, el `AppRoutingModule` estaba poniendo a disposición del `AppModule` todo el contenido de `RouterModule`, incluido el por ahora fastidioso `RouterOutletComponent`.

Pero el módulo *Core* no importa al *AppRouting*, así que nada sabe de un selector llamado *router-outlet*. Para solucionarlo sólo puedes hacer una cosa: importar al `RouterModule` en el `CoreModule`, que quedará así:

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';
import { FooterComponent } from './shell/footer/footer.component';
import { HeaderComponent } from './shell/header/header.component';
import { MainComponent } from './shell/main/main.component';
import { ShellComponent } from './shell/shell.component';

@NgModule({
  declarations: [ShellComponent, HeaderComponent, MainComponent, FooterComponent],
  imports: [CommonModule, RouterModule],
  exports: [ShellComponent]
})
export class CoreModule {}
```

4.2 Organización de la aplicación en módulos

Todos los programas tiene partes repetitivas. Los principios de **organización y código limpio** nos permiten identificarlas y reutilizarlas. Con los componentes ocurre lo mismo. El módulo y los componentes recién creados suelen ser comunes a casi todas las aplicaciones. Estos y otros muchos surgirán de manera natural durante el desarrollo de una aplicación para ser utilizados en múltiples páginas.

Son **componentes de infraestructura**. Conviene guardarlos en una carpeta especial. Aquí la he llamado *shared*, pero *tools*, *common*, o *lib* suelen ser otros nombres habituales. Para reforzar el uso del CLI escribe el siguiente comando que aprovecharemos en el futuro.

```
ng g m shared
```

> Por ahora déjalo huérfano, no lo importaremos hasta que tengamos módulos funcionales. Sólo anticiparte que es un módulo dónde se hace mucho uso de la propiedad transitiva del sistema modular de Angular.

El caso es **distinguir los componentes de infraestructura de los de negocio** o funcionalidad. Los módulos *core* y *shared* los trataremos como de infraestructura y todos los demás serán de negocio (aún no tenemos). El primero es para meter cosas de uso único esenciales para la aplicación. El segundo para meter bloques reutilizables durante la construcción de la aplicación. Recuerda que sólo son convenios de arquitectura de software; adáptalos a tus necesidades.

> En esta aplicación hasta ahora no es nada funcional,!y ya tiene seis módulos y seis componentes!. Puede parecer sobre-ingeniería, pero a la larga le verás sentido. Por ahora te permitirá practicar con la creación de módulos y componentes.

El bosque de módulos a vista de pájaro

```
AppModule
|
+--AppRoutingModule
|  |
|  +--RouterModule
|
+--BrowseModule
|
+--CoreModule
   |
   +--RouterModule

SharedModule
```

El bosque de componentes a vista de pájaro

```
AppComponent
|
+--ShellComponent
   |
   +--HeaderComponent
   |
   +--MainComponent
   |  |
   |  +--RouterOutletComponent
   |
   +--FooterComponent
```

Con esto tendrás una base para una aplicación *Angular 7*. Sigue esta serie para añadirle funcionalidad mediante Páginas y rutas Angular SPA mientras aprendes a programar con Angular7.

> Aprender, programar, disfrutar, repetir. -- *Saludos, Alberto Basalo*