2 - SPA

> Páginas y rutas Angular SPA

Las **aplicaciones Angular 7 son conjuntos de páginas enrutadas** en el propio navegador. Son las conocidas *SPA, Single Page Applications*. Estas apps liberan al servidor de una parte del trabajo, reducen la cantidad de llamadas y mejoran la percepción de velocidad del usuario.

En este tutorial aprenderás a crear una Angular SPA fácilmente usando `@angular/router`, **el enrutador de Angular**.

Partiendo de la aplicación tal cómo quedó en Base para una aplicación Angular. Seguimos usando el concepto de árbol, ahora como analogía de **las rutas y las vistas** asociadas. Al finalizar tendrás una angular SPA con vistas asociadas a sus rutas.

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

1. Rutas

Al crear la aplicación hice uso del flag `routing true` en el comando de generación del *CLI*. Esto causó la aparición de no uno, sino dos módulos gemelos en la raíz de la aplicación. Has estudiado el `AppModule` verdadero módulo raíz, y ahora verás en profundidad a su gemelo: el **módulo de enrutado** `AppRoutingModule` y el uso que hace del `RouterModule`.

1.1 RouterModule

El *Angular Router* necesita ser importado y configurado. El módulo `AppRoutingModule` cumple dos funciones. Por un lado **importa al** `RouterModule` de Angular, el cual contiene toda la lógica necesaria para enrutar en el navegador. Por otro lado, permite la **definición de rutas** en el array `Routes[]`.

```
import { Routes, RouterModule } from '@angular/router';
const routes: Routes = [
  {
    path: 'heroes',
    component: HeroesComponent
  },
  {
    path: 'not-found',
    component: NotFoundComponent
  },
  {
    path: '**',
    redirectTo: 'not-found'
  }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
```

```
})
export class AppRoutingModule {}
```

El array de rutas recibe **objetos ruta** con propiedades de configuración.

La primera es `path:` en la que se especifica **la dirección** que resuelve, en este caso la ruta vacía o raíz del árbol de rutas. Las otras son opcionales y las veremos poco a poco.

1.1.1 Component

Vamos a crear un componente donde guardar el contenido que el CLI nos regala de inicio. Los enlaces a las páginas oficiales de Angular y al ejemplo del *Tour Of Heroes*. Para ello crearé el componente HeroesComponent.

```
ng g c heroes
```

Lo hago en la carpeta raíz; algo poco aconsejado si queremos tener una estructura escalable. Pero es un buen anti-ejemplo 😉

Ahora debo decidir qué ruta asociarle... por ejemplo `/heroes`. Eso es lo que se ve en el inicio de la configuración de rutas. Para no perderme lo ideal es tener un nuevo enlace de navegación en el `HeaderComponent`

```
<header class="sticky">
  <a routerLink="/" class="logo"> <span class="icon-home"></span> <span>{{ title
}}</span> </a>
  <a routerLink="heroes" routerLinkActive="router-link-active" class="button">
    <span> Heroes</span>
  </a>
</header>
```

Vayamos casi al final y de paso hagamos algo útil para no volver a perdernos sin remedio. Un detector de rutas no contempladas, y una ruta a dónde redirigir a los usuarios perdidos. Para ello estudiaremos la propiedad `component` que es fundamental pues indica **el componente** que se debe mostrar cuando esta ruta se active.

Así es cómo funciona el enrutado. Un camino y un componente asociado. La tabla de enrutado se procesa de arriba a abajo y cuando un camino coincide con la ruta actual, se para y se carga el componente.

Vamos a crear un componente con la intención de mostrarlo sólo cuando las demás ruta fallen. Se llamará *not found* Lo creo asociado al `CoreModule` lo cual ayuda a organizar los elementos de la aplicación.

```
ng g c core/not-found
```

Ya podemos asociar dicho componente al camino `not-found`. Pero esto es poca cosa. Hay mucho más.

1.1.2 RedirectTo

La configuración de rutas no sólo permite asignar componentes a las direcciones. También se pueden hacer **redirecciones de unas direcciones a otras**. Y por supuesto puede haber **rutas no contempladas o errores** por parte del usuario, los infames `404 Not Found`.

> En este caso cuando se escriba la ruta `/not-foud` se mostrará un componente, el `NotFoundComponent`, cuyo contenido indicará al usuario que se ha perdido. Claro que nadie va voluntariamente a esa ruta. Mediante el `path: '**'` le indico que ante cualquier ruta no contemplada anteriormente se ejecute el comando `redirectTo: 'not-found'`, el cual nos lleva a una ruta conocida con un mensaje bien conocido. *Page Not Found*.

Pero ¿Cómo es eso de que se mostrará?, ¿Dónde se cargará?. Presentamos a `<router-outlet>`.

1.2 Router Outlet

La idea general de **una SPA es tener una única página que cargue dinámicamente otras vistas**. Normalmente la página contenedora mantiene el menú de navegación, el pie de página y otras áreas comunes. Y deja un espacio para la carga dinámica. Para ello necesitamos saber **qué componente cargar y dónde mostrarlo**. De esto último se ocupa el *router outlet* mediante la etiqueta `<router-outlet></router-outlet>`.

En el `main.component.ts` había un contenido *hard-coded*. Para hacer que el contenido sea dinámico se sustituye por el elemento de Angular `<router-outlet></router-outlet>`. Este elemento del framework inyectará dinámicamente el componente que le corresponda según la ruta activa. El `MainComponent` queda así:

```
<main class="container">
  <router-outlet></router-outlet>
  <!-- Dynamic content here! -->
</main>
```

1.3 Router Link

Los enlaces web tradicionalmente se han resuelto con elementos `<a href=""></a>` dónde en su atributo `href` se asociaba la dirección a la cuál navegar ante el click del usuario. **En Angular los enlaces se declaran con un atributo** especial llamado `routerLink`. Este atributo **se compila dando lugar al** `href` oportuno en tiempo de ejecución.

En el fichero `not-found.component.ts` pon algo así:

```
<h1>Not Found</h1>
<h2>404</h2>
<a routerLink="/">Go home</a>
```

> Por ahora la funcionalidad de `routerLink` no mejora en nada a `href`. Pero lo hará. Mientras tanto familiarízate con su sintaxis y... asegúrate de importar `RouterModule` en los módulos en los que lo vayas a usar.

Salgamos de este bucle creando más rutas y más componentes. Pero esta vez con un nueva técnica.

2 Lazy Loading

La web clásica funcionaba con un navegador pidiendo una ruta al servidor. El servidor buscaba o montaba un documento html y se lo devolvía al navegador para que lo renderizase. Una nueva ruta significaba repetir todo ese viaje. Hasta que aparecieron las Single Page Applications. En este caso el código cliente es el responsable del contenido asociado a cada ruta. Y eso es mucha responsabilidad.

Las *webs SPA* se crearon por una razón que casi acaba con ellas: **la velocidad**. Al realizar el enrutado en el cliente y querer evitar todos los viajes posibles hasta el servidor, se cargó a la única página web con todo el peso de la aplicación. Lo cual la hizo terriblemente lenta en la primera visita de cada usuario.

El **impacto de la primera visita** en una aplicación de intranet no suele ser un problema grave. Pero en internet esa visita puede ser la primera y si tarda mucho, también será la última. La solución viene de mano del concepto de *lazy loading* o carga perezosa. Consiste en diferir la carga de la lógica asociada a una dirección hasta el momento en que sea activada dicha ruta. De esa forma, **una página no visitada es una página que no pesa**. Y la carga inicial se hace mucho más liviana.

En Angular el *lazy loading* es tan sencillo que ya se recomienda implementarlo por defecto. Para hacerlo conoceremos más comandos del `Router` y algunas herramientas de compilación usadas por el *Angular CLI*.

2.1 Webpack y los bundles por ruta

Hay que saber que el *Angular CLI* usa internamente la herramienta de empaquetado *webpack*. La cual recorre el código *TypeScript* buscando `imports` y empaquetando su contenido en sacos o *bundles*. Luego introduce las referencias a esos *bundles* en la sección se *scripts* del index.html, haciendo que se descarguen todos nada más arrancar la aplicación. Esto puede ser muy pesado en aplicaciones grandes. Así que hay que buscar una manera de diferir esa descarga, repartiendo el *bundle* principal en otros más pequeños que se cargará bajo demanda.

Objetivo: adelgazar el peso del *bundle* principal, el `main.js`.Para conseguirlo hay que configurar las rutas de forma que no sea necesario importar los componentes a mostrar. Tal como se ha hecho con el `NotFoundComponent`, de hacerlo así con todos, *webpack* empaquetaría esos componentes como algo necesario... y por tanto serían enviados al navegador en el *bundle* principal sin que sea seguro su uso. Ese no es el camino, es una excepción para componentes poco pesados y muy utilizados.

La solución que ofrecen el *cli* y *webpack* consiste en **delegar la asignación del componente a otro módulo, pero sin importarlo** hasta que su ruta principal se active.

He creado unas vistas para ser usadas en las direcciones `/` y `/about`. Los componentes asociados se llaman `HomeComponent` u `AboutComponent`. Se han **declarado pero no exportado** en sus respectivos módulos `HomeModule` y `AboutModule`. No es necesario exportarlos porque no serán reclamados directamente por nuestro código.

```
ng g m home --routing true
ng g c home/home
ng g m about --routing true
ng g c about/about
```

Estos módulos no deben ser importados por el `AppModule`; no queremos saber de su existencia. Simplemente debe usarse su ruta relativa en el módulo de enrutado `AppRoutingModule` como un valor especial. Vamos a agregarlo al `app-routing.module.ts` que quedará así.

```
import { Routes, RouterModule } from '@angular/router';
const routes: Routes = [
  {
    path: '',
    loadChildren: './home/home.module#HomeModule'
  },
  {
    path: 'about',
    loadChildren: './about/about.module#AboutModule'
  },
  {
    path: 'not-found',
    component: NotFoundComponent
  },
  {
    path: '**',
    redirectTo: 'not-found'
  }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Fíjate que **la dirección del fichero es una cadena de texto** asignada a una nueva propiedad de objeto *route*, la propiedad `loadChildren:""`. No se está produciendo ninguna importación en *TypeScript* como ocurre con el componente `NotFoundComponent`.

Con esta información *webpack* va a generar un *bundle* específico para cada módulo. Si durante la ejecución se activa la ruta `/` (muy probable porque es la ruta raíz) o la ruta `/about` entonces se descarga ese paquete concreto y se ejecuta su contenido. Mientras tanto, se queda almacenado en el servidor.

> Esto hace que la aplicación de Angular pese menos y responda antes, mejorando el tiempo de pintado inicial. La combinación de estas y otras técnicas que veremos en este tutorial sacarán el mejor rendimiento posible a tu aplicación Angular.

2.2 El enrutador delegado

Ya sabemos que hasta que no se active la ruta `/` o la `/about` no hay que hacer nada. Pero si se activa, entonces se descarga un *bundle* que contiene un módulo y los componentes necesarios. Sólo falta escoger dentro de ese módulo el componente que se asignará a la ruta.

Para eso al crear los módulos Home y About use el *flag* `routing true`. Esto hace que se genere un segundo módulo de enrutado. El `HomeRoutingModule` y el `AboutRoutingModule` son prácticamente idénticos al enrutador raíz.

> Digamos que son **enrutadores subordinados** al primero. Sólo se llega aquí si en la ruta principal se ha navegado a una dirección concreta. Se hace notar esa distinción durante el proceso de importación del módulo de Angular `RouterModule`. En el caso principal se pone `imports: [RouterModule.forRoot(routes)]` y en todos los demás `imports: [RouterModule.forChild(routes)]`.

A nivel subordinado, la dirección `path: ""` se agrega al `path: ""` de su enrutador padre. Cuidado, es un error común repetir el *path* a nivel hijo. En este caso incluso parece redundante. Pero con *about* no quedan dudas. En el *root* lleva `path: "about"` y en el *child* solamente `path: ""`.

La ventaja real de este segundo enrutador es que irá empaquetado en el mismo *bundle* que el módulo de negocio y sus componentes. Descargando ese peso en el momento que se necesite. Aquí sí que asignaremos un componente concreto: el `HomeComponent` o el `AboutComponent`. Por ejemplo el fichero `home-routing.module.ts` quedará más o menos así:

```
import { HomeComponent } from './home/home.component';
const routes: Routes = [
  {
    path: '',
    component: HomeComponent
  }
];
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class HomeRoutingModule {}
```

2.3 Navegación

Ahora que ya tenemos un par de rutas reales, es buen momento para crear un mini menú de navegación. Vayamos al `core/shell/header.component.html` y pongamos algo así:

```
<header class="sticky">
  <a routerLink="/" class="logo"> <span class="icon-home"></span> <span>{{ title
}}</span> </a>
  <a routerLink="about" routerLinkActive="router-link-active" class="button">
    <img width="32" style="vertical-align: -0.5em" src="./assets/logo.png" />
<span> About us</span>
```

```
  </a>
</header>
```

3 Rutas anidadas

Cuando las interfaces se complican, es habitual que las aplicaciones dispongan de menús de navegación a distintos niveles. Dentro de una misma página podemos querer ver distinto contenido y además reflejarlo en la *URL*. Para resolver esta situación en Angular disponemos de la técnica de las *nested routes*.

> De una manera un tanto forzada la he incluido en la página `/about`. La cual disponen de su propio menú de navegación, y lo que es más importante, su propio `<router-outlet></router-outlet>`.

Para empezar veamos como queda el *html* del `about.component.ts`. Vamos a dotarlo de dos rutas nuevas `/about/links` y `/about/info`. Cada una mostrará contenido en un componente adecuadamente insertado en el `<router-outlet></router-outlet>` local.

3.1 Children

Para repasar conceptos de generación de componentes

```
ng g c about/about/links
ng g c about/about/info
```

Para que funcione empezamos por crear los dos componentes `LinksComponent` e `InfoComponent` de forma rutinaria. Y los asignamos en el `about-routing.module.ts` como subordinados a la ruta principal con el comando `children:[]`. Los caminos se van agregando sobre la ruta principal activa, la `/about`. Esto es así tanto el `routerLink` como el `path` de los `children`.

```
const routes: Routes = [
  {
    path: '',
    component: AboutComponent,
    children: [
      {
        path: 'links',
        component: LinksComponent
      },
      {
        path: 'info',
        component: InfoComponent
      }
    ]
  }
];
```

3.2 RouterOutlet anidado

Los componentes de las rutas `children` se inyectarán en el `<router-outlet>` del componente contenedor `AboutComponent`. Es como si todo volviese a empezar desde aquí.

```
<header class="sticky">
  <a routerLink="links" class="button"> <span> Tutorial Links</span> </a>
  <a routerLink="info" class="button"> <span> More Info</span> </a>
</header>
<router-outlet></router-outlet>
```

Con estos conceptos y la combinación de `children, loadChild, component, redirectTo` ... asociadas a `path` podrás configurar tu aplicación y responder a cualquier *URL* desde la misma y única página `index.html`.

4 Parámetros

Las rutas vistas hasta ahora se consideran estáticas pues se han definido usando constantes. Es muy habitual tener **páginas con la misma estructura pero distintos contenidos**. Un blog con sus posts, una tienda con sus productos, o un proyecto con sus tareas... hay miles de ejemplos así.

4.1 Variables en la ruta

Ese tipo de direcciones se consideran paramétricas, tienen unos segmentos estáticos y otros dinámicos. Estos últimos se definen con parámetros, algo así como **variables dentro de la cadena de la ruta**. Su sintaxis obliga a precederlas de dos puntos. Por ejemplo `countries/:country/cities/:city` resolvería rutas como *countries/usa/cities/new-york* o *countries/italy/cities/roma*. Rellenando los parámetros `:country` y `:city` con los valores necesarios.

> Esta aplicación no tiene un propósito de negocio concreto. Iremos creando rutas según sea necesario por motivos pedagógicos. Empezaremos con unas páginas destinadas a mostrar los autores del proyecto.

Vamos a crear rutas como */authors/albertobasalo* o */authors/johndoe*. Para ello necesitamos el segmento principal */authors* y una par de componentes.

Vamos a agregar los componentes necesarios como hasta ahora.

```
ng g c about/about/authors
ng g c about/about/authors/author
```

En las rutas del `about-routing.module.ts` agregamos un nuevos *children paths*

```
{
  path: '',  component: AboutComponent,
```

```
    children: [
      {
        path: 'links', component: LinksComponent
      },
      {
        path: 'info', component: InfoComponent
      },
      {
        path: 'authors', component: AuthorsComponent
      },
      {
        path: 'authors/:id', component: AuthorComponent
      }
    ]
  }
```

Esta configuración resuelve las rutas `about/links`, `about/info`, `about/authors` y `about/authors/cualquier-otra-cosa`. Y las carga con el componente adecuado. Lo novedoso en el camino `:id`. El prefijo dos puntos indica que es un parámetro. Algo así como una variable en el segundo segmento que se almacenará y será recogido con el arbitrario nombre `id`.

Para mostrar el uso de los nuevos enlaces he agregado el `authors/` al `AboutComponent` y he creado un listado en el `AuthorsComponent`. Familiarízate con las rutas relativas para componer la ruta completa.

```
<a routerLink="albertobasalo" class="button"> <span> Alberto Basalo</span> </a>
<a routerLink="johndoe" class="button"> <span> John Doe</span> </a>
```

Aún más interesante es el componente que muestra cada autor de la lista, el `AuthorComponent`. En este caso fíjate cómo accede a la ruta, cómo obtiene el valor del parámetro y cómo lo usa para mostrarlo en la web.

4.2 ActivatedRoute

El framework *Angular* trae muchas librerías para facilitar la vida al programador. Sólo hay que saber dónde están y cómo pedirlas. Para ello volvemos a la tecnología escogida, *TypeScript*, que permite las **importaciones y la inyección de dependencias**. Hay un tema dedicado a conocer en profundidad los servicios inyectables en Angular. Por ahora una breve introducción.

Contenido del fichero `author.component.ts` relacionado con la obtención del parámetro de la ruta activa:

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

export class AuthorComponent implements OnInit {
  public authorId = '';
  constructor(activateRoute: ActivatedRoute) {
    this.authorId = activateRoute.snapshot.params['id'];
  }
```

```
  ngOnInit() {}
}
```

La instrucción `import { ActivatedRoute } from "@angular/router";` pone a disposición del programador el código donde está definida la clase `ActivatedRoute`. Pero no se instancia directamente; en su lugar, se usa como un argumento del constructor de la clase del componente. Ese constructor es invocado por *Angular*, y dinámicamente el propio framework sabe cómo rellenar los argumentos que se pidan en los constructores. Es decir, sabe cómo inyectar instancias en las que dependencias declaradas.

Una vez que han **inyectan las dependencias en el constructor** ya están listas para ser usadas. En concreto `activateRoute` da acceso a métodos y propiedades para trabajar con la ruta activa y poder leer sus parámetros.

Obtenidos los datos desde la *URL*, ya se muestran en la vista de forma ya conocida. Fichero `/about/authors/author/author.component.html`

```
<h2>Author profile</h2>
<h3>{{ authorId }}</h3>
```