4. Rutas y SPA

RUTAS Y NAVEGACIÓN

Este tema dedicado a las **rutas** y a la **navegación**, veremos por primera vez lo que es una **Single Page Application (SPA)** y cómo llevamos al **browser** la responsabilidad de qué presentamos cuando el usuario navega determinadas **rutas**.

Obviamente, cuando vas de un sitio a otro, a lo mejor necesitas llevar equipaje. Son los **parámetros** que enviamos desde una página hasta otra que los recogemos como si fuesen señales.

Por último, veremos la diferencia que hay entre las páginas que se construyen en el **navegador** y las que ya vienen hechas desde el **servidor**. Cada una le sentará mejor a cierto tipo de usuario, persona o robot, intentaremos que todos queden satisfechos.

4.1 Conceptos de enrutado y Single Page Applications

Entramos en el nuevo mundo, el mundo de las **rutas**. Las **rutas**, o esas **URLs** que ponemos habitualmente en el **navegador**, y que hasta hace, pues no sé, 10-15 años pues era algo que resolvían los **servidores**.

Analizaban esta **ruta** y, seguramente, en base a una tabla que tenían en **base de datos**, más algunas **plantillas** que tenían en un disco local, eran capaces de componer un **HTML** que enviaban directamente al **navegador**.

Bueno, pero todo esto ha cambiado con el advenimiento de las **Single Page Applications**, que es la base del funcionamiento de **Angular**, en donde todo este esfuerzo se traslada al **navegador**. Es decir, el **navegador** va a necesitar algún tipo de **base de datos de rutas** y, por supuesto, **plantillas**. Y con eso montará, se autogenerará su propio **HTML**. De esta manera, el **servidor** estará más descansado.

Para que todo esto funcione, necesitamos que **Angular** nos ayude, y nos ayudará con todo lo que viene en lo que antes llamábamos **Router Module** y que ahora es simplemente el **Router** o la **Router Library**. En esta **Router Library** vendrán las capacidades para montar una tabla de **rutas**, el equivalente a esa **base de datos**, donde pondremos el **path** que tengamos que tratar. Ese **path** podrá ser complejo, simple, anidado, etc. Y le asociaremos un **componente**. En el futuro puede tener otro tipo de cosas, pero esencialmente es esto.

Así que, para cada **ruta** tendremos un **componente**. ¿Y dónde aparecerán esos **componentes**? Si recuerdas, desde el primer momento el **index.html** tenía dentro a un **componente** llamado **app-root**, el **componente raíz**, el cual esencialmente casi no tenía nada dentro, excepto algo que no le hemos prestado atención hasta ahora, llamado **Router Outlet**, o la salida, el lugar a donde irá el **componente** que toque según la **URL** que esté activa en ese momento.

En el fondo lo que tendremos que hacer entonces será crear una tabla de caminos, asignar los **componentes** a esos caminos, activar el **Router Outlet** y configurar alguna cosa en el **app.config**. Todo esto puede resultar un poco tedioso, pero también te diré que es muy fácil de mecanizar porque es siempre lo mismo.

Por último, decirte que para movernos desde una página a otra página ya no usaremos un ancla normal con un **href**, en lugar del **href** usaremos algo nuevo llamado **router-Link**. Este **router-Link** será un atributo no estándar, que también viene en la **librería Router** y que me permitirá hacer estos movimientos en local, sin necesidad de ir al **servidor**. Venga, vamos a ver cómo se hace eso en código.

4.1.1 Configuración y router outlet

Vale, pues nos vamos a dirigir al mundo de las **rutas**. Y para ello, por primera vez, necesito abrir aquí el **editor de código** con múltiples pestañas porque voy a necesitar o voy a relacionar cuatro ficheros. No hay que preocuparse demasiado porque en alguno no tocaremos nada y en otro muy poquito y en los demás pues va a ser evidente y así más fácil de ver los cambios que se producen. Pero lo primero es saber cuáles son y dónde están.

Empiezo por el **app.config** que está en la raíz de lo que sería la aplicación, el código de mi aplicación. Y este **app.config** fue generado por el **CLI** con el **n g new** y hasta ahora no lo hemos tocado. Algo haremos, pero muy poquito. Y por ahora simplemente nos fijamos en que tiene aquí una sección en la que habla del **router** por primera vez. Y además vemos también por primera vez o nos fijamos por primera vez en que hay importaciones sobre **@angular/router**.

@angular/router es el lugar, la **librería**, originalmente el **módulo**, en donde está todo lo que necesitaremos para hacer enrutado. Que no era ni más ni menos que determinar qué se va a ver en función de la **URL** que se esté presentando. Y esta es una labor que haremos desde el lado local de la aplicación, desde el **browser**.

Bien, este **provideRouter** lo que llama inmediatamente es a una exportación de aquí de un fichero **app.routes**. Está justo al lado, donde lo único que parece haber es una constante inicializada con un array vacío. Y que efectivamente aquí será donde declaremos las **rutas** y de alguna manera las instrucciones para que aparezca el **componente** adecuado.

Hablando de los **componentes**, os recuerdo que tenemos un **componente raíz** llamado **AppComponent**, cuyo selector es **lab-root**. **Lab** es mi prefijo, **root** es la indicación de que este es el **componente raíz**. Y dentro de él, entre otras cosas, llamamos a componentes normales, digamos como **header**, como **bookings** y como **footer**.

Y aparece aquí un componente, **router-outlet**, que de nuevo es algo, todo lo que lleve el prefijo **router**, pues evidentemente vendrá de **AngularRouter**. Y hoy sí le dedicaremos un momentito a pensar qué rayos hace este componente.

Por último, **lab.bookings** o **bookings.component**, visto desde el modo clase. Vamos a ver cómo utilizar **rutas** y cómo vamos a sustituir al **lab.bookings** exclusivamente por el **router.outlet**. Entonces, para ello, voy a empezar a rellenar este array que aquí me estaba vacío. Y estos objetos van a tener siempre la misma, o casi siempre la misma estructura. En un lado pondremos el **path** o camino que van a resolver, y en la otra la instrucción que se debe ejecutar. La instrucción será una propiedad con un valor. La propiedad que voy a poner no será **loadChildren**, sino **loadComponent**.

Y **loadComponent** podría ser un array, claro, podría serlo, podría ser un número, un objeto, pero no. Lo que va a ser, será una función. Las funciones, en principio, tienen esta firma. Entonces, **loadComponent** tendría una firma como esta, solo que aquí habrá que hacer algo. Lo que se hace aquí es retornar la importación de un componente. Retornar la importación de un componente puede hacerse en una sola instrucción. Por lo tanto, no vamos a necesitar llaves {} y simplemente pondremos la instrucción que nos retorne esa importación.

Esa importación se hace con la función **import**. Esta es una función del **JavaScript**, no va a venir de **Angular**. Y aquí lo que esta función hace es, por un lado, recibir una cadena de texto. Y por otro, que será la ruta al componente. Y por otro, un trabajo una vez que esa ruta esté cargada. Como esa ruta, potencialmente, puede tardar en cargarse porque no deja de ser la lectura de un fichero que, además, cuando estemos en ejecución estará remoto. Esto nos va a devolver una promesa y, por tanto, tendremos la opción de llamar a **then** y a **catch**. Normalmente lo haremos con **then**. Y esto es, más o menos, la anatomía de lo que será **loadComponent**. En la práctica esto es mucho más fácil e incluso hay maneras de autogenerarlo. Porque aquí lo único que hay que hacer es ir a poner la ruta del componente que quiero cargar. Que no es otro que este **BookingsComponent**. Si lo veo desde aquí, este **bookings.component** va a estar en la carpeta **bookings**. Y dentro de ella será el fichero **bookings.component**. Fíjate que no es necesario ponerle el **.ts**.

Vale, muy bien, pues esta parte ya estaría. Voy a recoger esto de aquí porque ahora ya no me hace falta seguir por ahí el caminito. ¿Y para qué me vale el **then**? Bueno, en el caso de que carguemos un fichero, este fichero va a tener una o más exportaciones. Lo habitual es que solo tenga una y la declare así simplemente. Export, class, lo que sea. Esto podría ocurrir. Entonces lo que tenemos que decirle es de todas las potenciales exportaciones que hay, con cuál se queda. Y para eso usamos este método **then**. El método **then**, de nuevo, es funcional, lo cual quiere decir que aquí cabría una función de estas de tipo Arrow. De nuevo, esta función de tipo Arrow no necesita llaves {} a este lado porque lo que vamos a hacer se resuelve en una sola instrucción. Y lo que recibe aquí, por convenio, se le llama simplemente **m** porque se refiere al módulo. Hablando en lenguaje **JavaScript TypeScript**, no en lenguaje clásico de **Angular**. Fichero, punto y vería todas sus exportaciones. Como la única que tiene y la única que me interesa es **bookings.component**, pues esa es la que pongo.

Entonces, una vez que tengo esto así, esto ya se lee más fácil y dice, cuando el camino sea vacío, cárgame el componente que esté en esta ruta. Esa ruta es un fichero y de todo lo que exporte ese fichero, seguramente solo esto, quédate con él. Lo siguiente será ver qué efecto habrá tenido este **loadComponent** y este **path**

en la ejecución. Y resulta que si yo vengo aquí a ver qué es lo que ha pasado, voy a recoger esto hacia aquí, me doy cuenta de que en la ruta principal lo que veo es **PaddleSurf**, **PaddleSurf**... Es decir, aquí aparece este componente repetido. ¿Y por qué ocurre que aparezca repetido? Pues ocurre porque aparece una vez, porque lo estoy invocando porque me da la gana, y una segunda porque lo hago de manera implícita a través del **router-outlet**, que es como debe ser. Es decir, que esta primera vez sobra, este componente lo voy a quitar, me quedaré exclusivamente con el **router-outlet**.

Primer efecto es que a partir de ahora ya solo se ve una vez, porque este **router-outlet** lo que hace es atender a estas rutas, ver que este camino coincide con lo que hay a partir de la raíz, esto es una dirección relativa, y en su lugar, o además del **router-outlet**, realmente carga este **lab-bookings**, lo carga por el nombre de la clase, fijaos, **bookings.component**, no por el selector. Tal es así que el selector, realmente en una situación como esta, por mucho que nos parezca y nos duela en los ojos de ver un componente sin selector, no es necesario, porque lo carga no por el HTML, sino por la clase.

Por lo demás, también podremos decir que **bookings.component** ahora ya no es necesario aquí, y por tanto, normalmente el componente raíz me va a quedar algo como esto. Bien, y ya está. Esto es lo básico del enrutamiento. En el **app.config** el programa se configura a partir de un array de rutas, que son pares de caminos e instrucciones, estas instrucciones pueden variar, van a variar, van a ser distintas, y lo que va a hacer el programa es, según la ruta que esté activa en ese momento, incrustar el componente que le toque, según esta instrucción, en el lugar del **router-outlet**.

4.1.2 Router link

Vale, hasta ahora no hemos mejorado funcionalmente nada la aplicación, porque lo único que hicimos fue sustituir a un componente por el **router-outlet** que lo carga dinámicamente cuando corresponde, que no está mal. Pero ahora vamos a hacer algo para un componente nuevo y para ello voy a dividir aquí mi terminal en dos para poder generar un nuevo componente.

Típico **login ng generate component (ng g c)** y ahora le voy a decir que me genere el componente **login** pero lo voy a crear dentro de una carpeta llamada **auth** para ahí agrupar todo lo que tenga que ver con autenticación, autorización, etc. Pero a mí me gusta que todas las **rutas** cuelguen de una determinada carpeta y por convenio suelo utilizar **routes**. Hay quien le llama **pages**, hay quien le llama **paths**, en fin. Bien, una vez que aparece, tengo la carpeta **rutas** con la carpeta **auth** y después el **login.component**. El **login.component** va a ser a todos los

efectos lo mismo que estaba haciendo aquí el **bookings.component**. Lo único que tendré que hacer será configurarlo bien aquí. **Rutas** sencillas, **rutas** complejas e incluso **rutas** paramétricas como veremos más adelante. Y la parte del **loadComponent** pues exactamente lo mismo. Esto es lo que tenemos. En este momento el **login.component** se cargaría si yo navego a la ruta **barra auth/login**. Login works. Pero ¿cómo podemos navegar hasta allí directamente desde la aplicación sin tener que escribir la ruta? Bueno, pues para eso teníamos aquí el componente **header**, lo voy a poner aquí ahora mismo, el cual ya tenía una etiqueta **a** con un atributo **href**, pero veremos que eso lo vamos a cambiar.

El atributo **href** a pesar de que efectivamente funciona, es la manera clásica que tiene cualquier navegador de enviar a un usuario hacia una determinada ruta. Pero en **Angular** lo que queremos es que esa navegación se gestione de manera local sin realmente involucrar al servidor si no es necesario o en cualquier caso pasando antes por estos filtros y estas instrucciones. Y para eso lo que haremos es no utilizar el **href**, esto da un poquito de miedo la primera vez, no utilizaremos el **href**, sino que utilizaremos algo llamado **routerLink**. Bueno, **routerLink**, como empieza por **router** ya me suena que tiene que ver con cualquier cosa del enrutado y **link** está bastante bien como nombre, pero dices, vale, **routerLink** está muy bien, pero **routerLink** no es algo estándar, así que habrá que importarlo. Entonces, no sólo importaremos componentes sino que importaremos atributos que en **Angular** llamamos directivas. A los atributos no estándar les llamaremos directivas. Bueno, pues este **routerLink** es una directiva, parece un atributo, en el cual podemos darle aquí un valor como es en este caso el valor vacío y podríamos darle otro valor, por ejemplo, para ir a **login**. Bueno, ya me aparece aquí esto y ahora puedo hacer click de uno para el otro y veo cómo se carga dinámicamente el contenido. El **login.component** no necesitaría realmente selector. El **login.component** es lo único que se está exportando desde este fichero **default**.

Esta es la exportación única y por defecto, entonces si te tomas la molestia de escribir este **default** en el componente antes de exportarlo, lo que podrás es tener pues una tabla de enrutados más limpia.

4.1.3 Page components

Vale, tenemos ya nuestra página principal, la de **login** y vamos a añadirle un poquito más de funcionalidad, por ejemplo, con una página de registro o alguna cosita más, pero sobre todo vamos a profesionalizar un poco nuestro código de manera que quede más limpio, más homogéneo y quizá por el camino aprendas alguna cosilla más de **Angular**. Entonces, para empezar, voy a generar un nuevo componente que va a ser el de registro, así que en principio me vale casi casi el comando anterior, porque sólo sería cambiar el nombre, pero es que además me

doy cuenta de que como estos componentes en principio sólo se van a utilizar desde el enrutador, no los voy a llamar por su nombre selector nunca, pues puedo evitarlo desde un principio. Así que puedo decirle que en este caso concreto no quiero tener un **selector**. Otra cosa que voy a hacer, y así aprendéis más cosas que pueden hacer los generadores, es que cada vez que yo genero un componente me lo nombra pues dándole el nombre que yo le pido y de apellido **component**, lo cual no está mal. Además, con el tema de color de iconos que yo tengo, pues le ponemos ya un icono aquí de **Angular**. ¿Pero se puede mejorar?

Sí, a mí me gusta que los componentes que se vayan a utilizar, en el caso de las rutas, tratarlos como algo un poco distinto y para eso le cambio el sufijo, si queréis, de aquí del **component**, mediante el atributo **type**. El atributo **type** le puedes poner en principio lo que tú quieras y yo, para los que sean de rutas, escojo llamarles **page**, porque esto es lo que va a aparecer en esa ruta, una página. Una ganancia es que yo tengo también configurado mi **Visual Studio**, esto puedes verlo después en el código fuente si te interesa, para que les cambie un pelín el color del icono y así este **register.page** lo vamos a tratar como algo especial. El problema es que en cuanto le cambio el sufijo a **page**, se me queja el **Eslint**, porque dice que esto no es un sufijo estándar y tiene razón.

Así que vamos a seguir aprendiendo cosas, abro el **eslintrc.json** y le voy a agregar aquí una regla extra. Lo que me va a permitir es definir mis propios sufijos no estándar, como por ejemplo **page**. Haciendo una exportación por defecto para que me quede sencilla la declaración de la ruta. De esta manera cada vez esto me queda más sencillo.

Y ahora sería cuestión de invocar esa ruta de alguna manera. Entonces en este formulario de **login**, que habrá un... Entonces en esta pantalla de **login**, además de un formulario, podría haber un enlace por si aún no tienes cuenta. Si ese fuese el caso, tendríamos que tener un **routerLink** previamente importado apuntando a esta nueva dirección. Como ves me lleva directamente a **auth/register**.

Podría hacer la misma jugada, por ejemplo, si estamos en la pantalla de registro, pero ya tenemos cuenta. Podríamos ir a **login** siempre importando aquello que necesitamos. Y ahora ya puedo navegar entre la página principal, la de **login** y la de registro. Bueno, para profesionalizar esto un poquito más, pues la página de **login** y la de registro, meteré lo que será la carcasa de un formulario. Por ejemplo, en **login** pondré algo como esto. De forma que cuando vayamos veamos algo así. Y en la de registro, pues, más de lo mismo, quizá un formulario aún un pelín más complejo, que nos permita poner **password**, repetirla, confirmar, etc. Dedicaremos una lección a los formularios que no tocan todavía.

Lo que sí toca es decirte que el **routerLink** se puede también expresar, y lo verás en muchas situaciones, como un atributo evaluado. Al meter cualquier atributo,

incluso las directivas entre corchetes [], lo que hacemos es que esto de aquí deje de ser un valor estándar y empiece a ser algo que debe ser procesado. La cuestión es que la alternativa a utilizar el **routerLink** sin nada más, sólo con una cadena, sería trocear esa cadena de caminos en un array y que el sistema los vaya componiendo. Esto veremos que va a tener más sentido en la próxima lección de los parámetros.

4.2 Parámetros en las rutas, señales en los componentes

Siguiendo con el recuerdo de cómo eran las páginas hace 15-20 años, estaban en un **servidor** con su **base de datos** y su **plantilla**, seguramente habría aquí una tabla, además de las **rutas**, pues quizá de productos, artículos, etc., donde todos estos registros se asociaban con la misma **plantilla** para producir **HTML** parecido, pero distinto, con los datos de cada uno de ellos. Seguramente en función de una **URL** que me mostraba aquí los productos, y ahora aquí el ID, 7A4... no sé qué.

Muy bien, bueno, pues a esto, este identificador, le llamaremos **parámetro**. **Parámetro**, en inglés, no confundir con los **query parameters** que van al final de la **URL**. Esto es un segmento de la **URL** donde una parte es significativa para localizar un dato, un dato, no una **plantilla**, y ese dato, mezclado con la **plantilla**, produce un resultado concreto.

Como sabemos, ahora con **Angular** estamos en una **SPA**, donde el **index HTML** es siempre el mismo, está siempre del lado del **browser**, y somos nosotros los responsables, así que necesitamos una manera de obtener esos **parámetros**. Para ello tenemos que, uno, en la tabla de **rutas**, cuando determinamos un **path**, pues habrá que, de alguna manera, significar dónde está ese **parámetro**.

Y eso se establece mediante un dos puntos : y después el nombre de un **parámetro**, que puede ser, normalmente, pues por ejemplo, **id**, el identificador de algo, **slug**, si es, por ejemplo, algo que se quiera ver bonito en Internet, legible, pero que sea también un identificador único, y esto lo que hace es determinar que un trozo de la **URL** actuará como clave, como identificador, como **parámetro**. ¿Y después qué? Pues después tendremos que recogerlo.

Lo recogeremos en un **componente**, y esto ha evolucionado mucho, y para bien, porque estos **parámetros** ahora los podemos recoger como señales. ¡Ajá! Así que sólo tendremos que identificar **id**, o **slug**, o como le hayamos llamado, esto es lo que tiene que coincidir, y decir que esto no es una señal, sino un **input** de un determinado tipo con un determinado valor.

A partir de este momento, esto se convierte en una señal, y podemos tener señales derivadas, es decir, **computed**, en base a esa señal, y **efectos**. Así que aquí aparecerán múltiples maneras de programar en base a estos datos de entrada para generar datos de salida que de nuevo se repinten, o efectos secundarios que cambien lo que sea.

Muy bien, pues vamos a ver cómo se hace eso en código.

4.2.1 Configuración y envío

Antes de empezar con el núcleo de esta lección dedicada a los **parámetros**, voy a incorporar un archivo llamado **activities.data**, que no es más que un **array** con una lista de actividades, esto en el futuro vendrá de un **servidor**, pero por ahora lo tenemos aquí **hardcoded** y así lo podré utilizar para varias cosas. Por ejemplo, en la pantalla principal ahora mismo tendríamos aquí un **componente** que nos ha servido para aprender lo básico de **Angular**, pero que funcionalmente no tiene mucho sentido, que sea esto lo primero que veamos. Quizá tendría más sentido que en la pantalla principal viésemos una lista de estas actividades. Y eso es lo que vamos a hacer.

Así que lo primero será que en las **rutas** voy a hacer unos cambios aquí para que veáis que nada está escrito en piedra, sino que todo es modificable. Y continuando con lo que hemos visto, voy a crear aquí un **componente** específico para la página **Home**. Entonces, esta **home.page** la dotaremos de un contenido que en principio también nos va a servir de repaso. Déjame estabilizar esto un poquito. Bien, una vez que tenemos esto estabilizado voy a explicarlo, pero son cosas, la mayoría de ellas, ya vistas.

Primero, todo lo que necesite utilizar, antes debo importar el **routerLink**, que lo usaremos ahora para algo novedoso, el **CurrencyPipe** y el **DatePipe** que ya habíamos visto antes. Segundo, en los datos, pues en este caso yo creo aquí una propiedad para esta clase pública que la igualo al contenido exportado de este **array**. Esto es una cosa que no es muy habitual en una aplicación profesional, pero que en nuestro caso para empezar nos puede valer.

Pensad que aquí está el **array**. Y ya que tengo aquí un **array**, repasamos también el iterador **for**, **@for**, **activity** of **activities**, con el **track** por su clave y a partir de ahí uso la variable **activity** para pintar por su **location**, **price**, **date**, **name** y esto es lo novedoso.

Dentro de la **app** voy a poner un **routerLink**, fíjate que lo tengo entre corchetes [] para que esto sea evaluable y ahora los segmentos de la **URL** que se generarán serán una combinación de una parte fija, que siempre pone **/bookings**, y una

parte variable que es lo que vamos a considerar el **parámetro**. Bueno, pero ¿qué es eso del **parámetro** y cómo puedo empezar a ver esta nueva página aquí? Pues volvemos entonces al **app.routes**. El **app.routes** recordamos que estaba configurado de manera que en el camino vacío me apareciese el **bookings.component**.

Esto va a dejar de ser así porque lo que quiero que me aparezca ahora será simplemente el **home.page**. Para que esto funcione debo ponerle aquí el **default** y a partir de este momento todas las rutas quedan homogéneas y simplificadas y por supuesto ya puedo ver aquí la lista de actividades. Hay que pensar que cada una de ellas tiene un enlace donde nos lleva a un **slug** que si yo hago click me intentaría llevar, pero veo que estoy teniendo aquí un error, voy a poner la consola, porque realmente no he programado esa ruta.

La ruta a la que intento ir es una ruta que pone **localhost, 4200, bookings, barra, kayaking, lake, leman, atlasal**, no sé qué, pero esa ruta yo no la tengo programada. Para hacerlo voy a agregar otra configuración donde el **path** ahora va a ser un pelín distinto. ¿Por qué? Porque tiene una sección normal, fija digamos, pero después algo raro que es este dos puntos :. El dos puntos : es la cosa rara. El dos puntos : lo que indica es que lo que va después no es un texto fijo sino por así decirlo una variable, lo que aquí llamaremos un **parámetro**.

Cuidado, en el futuro veremos la distinción con los **query parameters**, estos son **parámetros** tal cual, no **query parameters**. A partir de aquí sería más o menos lo mismo, sería cuestión de cargar cuál es el **componente** que quiero. Obviamente ahora mismo el **componente bookings** ni apellida **page** ni está en esta ruta, pero eso tiene más o menos fácil solución. Arrastro, cambio el nombre en el fichero y en la clase, lo exporto por defecto, pongo la ruta adecuada y después ya puedo probar de nuevo la navegación. Vemos aquí la **URL** completa donde esta parte de aquí será tomada como **parámetro**. Por ahora el código no está haciendo ningún caso de ello, simplemente puedo comprobar que si me voy a otra ruta funciona en el sentido de que me lleva, pero el contenido que veo es estáticamente siempre el mismo, lo que teníamos **hardcoded** del primer ejercicio. Eso cambiará ahora.

4.2.2 Recepción reactiva de parámetros como señales

Vale, pues vamos allá con la parte más chula y novedosa de cómo vamos a aprovechar los **parámetros**. El **parámetro**, concretamente, que hemos definido en la ruta **bookings/**

. Lo de **slug** viene simplemente de que es el nombre de una propiedad que se usa habitualmente en las **URLs** para que quede elegible para personas y que podría valer para indexar, etcétera. Bueno, pero eso es la definición del **parámetro** dentro

de la ruta. Hemos hecho un uso, vamos a decir, emisor de ese **parámetro** creando una ruta paramétrica donde se combina una parte fija con una parte variable, pero nos queda que, cuando llegamos al **componente** adecuado, obtener el valor del **parámetro** y utilizarlo.

Esto es un proceso de varios simples pasos que comienzan por volver a la configuración de la aplicación y aquí veíamos que teníamos una lista de **providers** donde, con la nueva sintaxis de **Angular** de manera funcional, estamos diciendo aprovisióname el **router**, aprovisióname algo de lo que ya hablaremos, pero el **router** con estas rutas. Y hablando de **con**, tendremos aquí también, usando la nueva sintaxis de **Angular**, una llamada, una función de configuración que le dice, oye, provéeme el **router** pero con **ComponentInputBinding**.

Esto no es otra cosa que decirle que los **parámetros** que tengas en la ruta, los enlaces con algo que haya aquí como **input** en este **componente**. Los **inputs** en la sintaxis moderna de **Angular** ya no necesitan llevar una decoración específica, sino que simplemente son un tipo en sí mismo. Al final es una propiedad, por ejemplo, en este caso le voy a llamar como el **parámetro**, **slug**, y lo voy a igualar a un **input** de tipo **string**, porque eso es lo que me llegará, sin un valor inicial. Este **input** viene del **@angular/core**, se importa, igual que habíamos visto en su momento con las señales, porque en el fondo esto va a ser una señal. Donde el **enrutador** emita, cambia el estado según la **URL** que tengamos aquí, lo que puedo hacer es empezar a tener propiedades computadas.

Por ejemplo, yo tenía aquí una propiedad **activity**, **hardcoded**, y ahora voy a cambiarlo por una propiedad computada que inicialmente va a hacer que mi código no compile, ya lo sé, pero vamos a dejarlo listo para después arreglar. Ahora haré que **activity** sea igual a algo **computed**. **Computed** será una función y dentro de esta función lo que haré será retornar una búsqueda en el **array** de actividades, buscando este **slug**, y si no lo encuentra, devolviendo una actividad nula que había previamente definido. Obviamente, si una función sólo tiene una instrucción con **return**, se puede hacer más sencilla, y esto, a partir de ahora, será para vosotros casi un patrón.

Esta es la parte de entrada de los argumentos y esta es la parte computada salida de los datos en base a esos argumentos. Obviamente, haber cambiado un tipo estándar por algo que es una señal hace que todo mi código deje de funcionar, pero el **refactoring** es más o menos sencillo, así que yo lo que voy a hacer es resolverlo y simplemente lo que he cambiado es que todas aquellas funciones computadas que partían de **activity** como un valor estándar, pues ahora tienen que invocarse, es decir, **activity** ahora es una señal computada.

Lo mismo me ocurre en el **HTML**. Lo cambio para utilizar la señal **activity**, pero con un truco. Aquí de nuevo un patrón que te vas a encontrar con frecuencia y es

que, a partir de ahora, como nuestros datos dependen primero de que obtengamos un **parámetro** en el futuro, de que vayamos a buscar esto, pues sabe dios dónde, a internet seguramente, es posible que esta, esta señal aún no tenga valor o tenga un valor nulo que a ti no te interesa. Así que habitualmente lo envolvemos dentro de una estructura condicional **if**, que sólo aparece cuando esto ya tiene un valor. Y de paso me sirve para crear un alias. Esto es otra cosa de la sintaxis de **Angular** que puedes aprender ahora mismo, que es que una señal se puede guardar su resultado en un alias, que sería como guardarlo en una variable y así el código ya no lo tengo que cambiar tanto, porque realmente lo que tengo por aquí es esta misma, el resultado de esta misma variable.

Por lo demás, el código en principio hace exactamente lo mismo, lo único que te darás cuenta de que he cambiado varios **status**, de forma que algunos ya no se permitan ni siquiera intentar hacer la compra. En fin, que aquí lo que quiero que veas es **input** de entrada, en este caso desde la **URL**, datos principales computados, datos derivados también computados, y efectos para calcular o preparar las cosas en función de los datos que haya inicialmente, por ejemplo, o de lo que vaya haciendo el usuario. Y todo apoyado en el concepto de **parámetro** de las rutas, tanto durante el envío como durante la recepción.

4.3 SEO y Server Side Rendering

Bueno, lo que te muestro aquí sería el esquema básico de cómo funciona una **Single Page Application** donde tenemos ya varias **rutas** predefinidas que, según se van activando, cargan y muestran su contenido dentro de un **index.html** que, en principio, está esencialmente vacío. Sí, tiene el **app** o el prefijo que tú le hayas puesto, **root**, pero nada dentro. Todo el contenido se monta dinámicamente.

¿Y cómo es eso? Pues porque se generan, **Angular** nos va a generar durante el proceso de compilación, unos ficheros de extensión **JS**, porque eso es lo que son, **JavaScript**, uno por cada ruta que se haya definido. Y el **enrutador**, cuando verifica que un determinado camino se ha activado, pues carga, o mejor dicho, descarga desde el servidor este **JavaScript**, lo procesa y genera contenido **HTML**. Este proceso es lo que llamamos **lazy loading**. ¿Por qué? Es **lazy** porque, hasta que no se activa este camino, no se produce esta ida a buscar este fichero, no se procesa y no se pinta.

Esta es la parte **lazy**. Y lo bueno es que, una vez hecho, yo puedo irme a otro camino y, cuando vuelva, como esto ya estará descargado, sólo tengo que volver a procesarlo y volver a generar nuevo contenido. Incluso permitiría, de alguna manera, un trabajo desconectado que, con la ayuda de otras herramientas como las **Progressive Web Applications** y demás, podría hacer que nuestra aplicación

funcionase de manera desconectada. Pero, aunque no sea así, aunque no funcione desconectado, lo que funciona es muy rápido porque sólo descargo aquello que visito y lo que nunca se visita nunca se descarga. Todo es configurable y este **lazy** se puede convertir en algo **Eager**, es decir, que yo predescargo esto de antemano, de manera que, cuando el usuario visita, ya no tiene que producirse la descarga desde el servidor y solamente el proceso. Bueno, todas esas son estrategias avanzadas que, por ahora, no tocan. Por ahora, lo que sí que interesa es ver la carcasa del **índex HTML** vacía, se rellena en función de las visitas y con el contenido que descarga según vamos visitando y procesando el **Javascript**.

Pues algunas soluciones traen sus propios problemas y es que estas **Single Page Applications** lo que hacen es que, cuando solicito cualquier ruta, la que sea, al servidor, lo que me devuelve, como hemos visto, es un **HTML** esencialmente vacío. Y claro, esto, para empezar, al usuario lo deja un ratito, sin ver absolutamente nada, hasta que, a partir de los **JSON** que aquí vienen, se solicitan esos **JS**, se descargan, se procesan y entonces sí, entonces sí que tenemos cosas que ponen contento al usuario. ¿Y qué ocurre con los robots? Me refiero a los robots de **Google**, **Bing** y demás que tienen que indexar nuestras cosas.

Bueno, pues que, por defecto, esos robots no esperan a que estos **Javascripts** bajen, es decir, no hacen esta llamada al **Javascript** y lo que indexan es el contenido recién descargado, es decir, nada, porque ahí no hay nada que ver. Así que, desde siempre, a los robots no les ha ido bien con las aplicaciones **Single Page Applications**.

¿Cómo se solucionan estos dos problemas, el de la indexación para los robots y el de la página vacía para el usuario? Pues volviendo a trasladar parte de la responsabilidad al servidor. Y aquí surge el concepto de **Server Side Rendering**. El **Server Side Rendering** me obliga a tener un servidor, propiamente dicho, no algo estático, seguramente algo con **Node** en los días actuales, y este servidor lo que hace es recibir esta petición, procesarla con un **Angular** local. Voy a simplificar así, ¿vale? como si hubiese aquí un **ng** que procesa esta petición.

El resultado de esta petición es **HTML Full**, totalmente formado, pesado también si quieres. Cuando esto llega al **browser**, el usuario ya inmediatamente, aunque tarde un pelín más, ya es feliz y, sobre todo, también hacemos felices a nuestros robots. ¿Por qué? porque ya tienen un contenido para indexar. Además, el usuario sigue siendo muy feliz porque lo que se ha descargado es una página, la que toque y, por tanto, su **JS**, es decir, ese **JSON**. Pero, para otra ruta, una vez que él haga una navegación local, el usuario esa navegación local la resolverá en local, con su tabla de rutas y con sus contenidos. Para el robot todo será siempre un mundo nuevo. El robot siempre hace las peticiones al servidor.

Digamos entonces que el **Server Side Rendering** nos ayuda a que la primera visita de una página siempre sea grata para el usuario y siempre sea indexable. Sobre todo, esto es interesante en aplicaciones que estén hacia internet para los robots, no tanto para aplicaciones de intranet o aplicaciones de gestión, aunque siempre tienes esta posibilidad al alcance de un clic.

4.3.1 SPA y navegación local y offline

Bueno, pues como ya llevamos algunas lecciones, ahora nos toca disfrutar de lo hecho. Y más que ver código, vamos a ver el resultado de este código. Así que para empezar voy a volver a lanzar a toda pantalla la compilación y la ejecución de nuestra aplicación. Y lo que estoy viendo es en la pestaña de la red, pues todas las peticiones que se han hecho. En concreto hay una de ellas, lo que pondrá aquí **localhost** de tipo document, que si voy veo que el preview se me aparece sin la aplicación de estilos, por supuesto. La respuesta, sin lugar a dudas, es el **HTML**.

Y en principio pues nada que objetar sobre esto. Esto es como quien dice, he pedido una página y me la han servido. Es curioso que la página viene ya servida, y esto veremos después, que es producto del **server-side rendering**. Pero por ahora aún no quiero entrar en eso, simplemente es que estoy viendo esta página.

Obviamente si navego, por ejemplo, para ver alguna de las actividades, lo que ocurrirá es que, sí, la veo, y además, oh, sigo viendo lo mismo, ¿qué es lo que ha pasado? Pero lo que está pasando es que realmente esta petición ya no se está respondiendo por **HTML**, sino que se ha respondido a través de alguno de estos **chunks**, grupos, trozos, pedazos, donde el **JavaScript** me viene aquí todo el código necesario para montar esta página.

Es decir, esta página, que estoy viendo aquí, no se ha solicitado directamente al servidor como página en sí, sino como **JavaScript**. Voy a hacerlo de nuevo para la página **login**. Fíjate que aquí tenemos 20 peticiones, hacer clic en **login**, va a ocurrir que tendremos, bueno, pues 22 peticiones. Si están bien ordenadas, y si no, pues no nos pasará nada, haremos clic en este **chunk** de aquí a ver si hay suerte, y vemos que sí, sí que la hay.

Resulta que aquí lo que nos viene es el código empaquetado responsable de pintar la página **login**, el componente **login**. Es decir, que durante el desarrollo lo que hemos hecho es empaquetado el código en esos imports, y ahora pues **Angular**, a través de sus empaquetadores, nos ayuda a localizarlo, a descargarlo y a mostrarlo. Pero como **JavaScript**, no como elemento de **HTML**.

¿Y qué ventaja tiene esto? Bueno, primero, has visto que hasta que no visité la página de **login** no se produjo la petición. Teníamos 20 peticiones que después

han crecido. Si yo vuelvo a visitar la página **login**, veo que tengo una petición extra, pero esa petición extra solamente es para comprobar que nada ha cambiado, que todo está bien. Entonces, ¿podríamos incluso visitar esa página sin que realmente fuese necesario que estuviese ahí el servidor?

Pues sí. Vamos a verlo, porque aquí en la pestaña de red puedo deshabilitar, poner totalmente en offline la aplicación, que es como quien dice que hemos apagado el servidor, o hemos desconectado el cable, mejor dicho. Y si yo ahora me voy a **login**, pues resulta que puedo ir a **login**.

¿Cómo es posible? Es posible porque realmente el **chunk**, el pedazo que estaba asociado... Ya no sé cuál era. Bueno, este quizá. No, es el del **home**. Pero al **home** le pasa lo mismo. Puedo volver al **home** y ver su contenido. Obviamente esto que me está pasando ahora de ver el contenido es porque el contenido realmente está en el código. No estamos necesitando peticiones a servidores API remotas.

Pero lo que te estoy mostrando aquí es que una vez que una aplicación se ha descargado, que una **single page application**, porque la única página que hay es esta. Esto es el **single page**. Todo lo demás yo me puedo mover que mientras, vamos a decirlo así, tengamos datos y la hayamos visitado, funcionará incluso desconectada.

Esto es lo que permite también hacer aplicaciones para móviles con esta tecnología. Obviamente hay que empaquetarlas de determinada manera, pero este es el fundamento que hay debajo. Y respecto a lo visitado y lo no visitado, quería mostrarte una última cosa. Como ves, yo puedo ir al **login**, al **home** y a todo esto sin ningún problema.

Pero estando en el **login**, te recuerdo que una funcionalidad que hicimos es el enlace al registro si no estoy registrado. Bueno, pues voy a hacer clic en esto y ¡tachán! no voy, no voy, no puedo ir. ¿Qué me está pasando aquí? Bueno, lo que me está pasando es que realmente como esa página yo nunca la había llegado a leer, voy a poner esto en el modo **console** porque acabo antes...

Bueno, las peticiones por cierto son las del **favicon**, no le deberíamos hacer mucho caso. Pero a esta sí, a esta sí. Quiero visitar la página de registro, el sistema solicita el **chunk-P5**.. no sé qué, no sé cuánto. Y obviamente como estamos desconectados no lo puedo obtener. Esto de no poderlo obtener me produce un fallo, fíjate que es aquí el **app route** es el que solicita eso y el **login component** no sale del plano porque no puedo irme de aquí a esta página que no existe. Fíjate que el error sigue persistiendo.

¿Puedo volver a todo lo que ya conocía? Por supuesto que sí. Tengo el problema del **favicon** pero ahora mismo no me preocupa. Y aquí tenemos el problema de que no puedo visitar eso. Si obviamente yo vuelvo a habilitar la página, recargo, me

voy al registro, es decir la visito, vuelvo al **login**, lo visito, vuelvo a la página principal, etc.

Una vez que las haya visitado todas, de alguna manera si tuviese los datos yo podría navegar por las páginas, por todas ellas. Fíjate que ahora ya estamos viendo registro sin problema. Podría navegar por todas ellas porque realmente lo único que estamos haciendo es cambiando de un **JavaScript** para otro. Esta visita que he hecho yo se puede forzar, se puede programar para que el sistema de alguna manera por debajo las visite.

No tiene que llegar a mostrarlas, simplemente hace las peticiones a los **chunks** para tenerlos descargados y a partir de ese momento tú navegas, pues vamos a decirlo así, de manera desconectada y en local. A esto se le llama **Lazy Loading**, que es sólo visito cuando lo necesito y por otro lado una vez descargado no lo necesito más veces y sigo con la navegación en modo local.

4.3.2 Indexación de contenido SSR

Bueno, desde el primer día cuando creamos la aplicación oímos hablar del acrónimo **SSR**, o **Server Side Rendering**, nos preguntó si queríamos esa funcionalidad, dijimos que sí, pero por ahora no le hemos sacado partido. Eso ya se acabó porque voy a lanzar el script **npm run serve**, que yo lo tengo automatizado para que primero haga una compilación física, es decir, genere en la carpeta de distribución, nombre de mi aplicación, y concretamente dentro de la carpeta **server**, un montón de código en **javascript** que necesito ahora mostrar, esta es la segunda parte, lo que llama a este script, este script vino gratis, nos lo generó el **CLI**, donde lo que hace es llamar a **Node** sobre el programa **server.mjs**, esto es un módulo de **javascript**, **Node** moderno, y a partir de ese momento me lo sirve, ojo, en **localhost 4000**, no 4200, esto se puede manipular, pero eso es lo que tenemos ahí. Ahora yo voy a este **localhost 4200**, lo que voy a hacer es abrir aquí un navegador, no en el puerto 4200, sino en el 4000, voy a hacerlo bien, y inspeccionar el **network** para ver qué es lo que se solicita y lo que no. Realmente es prácticamente lo mismo, tenemos aquí el **localhost** con el contenido, si los **chunks** que necesito, que son los de la página principal, si yo me voy, por ejemplo, como siempre, a **login**, aparecerá un nuevo **chunk** aquí, que es el del **login**, lo que pasa es que ahora ya ni siquiera me lo dice porque está una compilación más abreviada, más segura, pero esto es la misma historia, todo funciona exactamente igual, de forma que también si yo deshabilito la red, esto por volver a ver lo mismo en este modo, puedo volver a **login**, no puedo ir a **registro** porque nunca lo había visitado todavía o no lo había visitado todavía, así que si lo vuelvo a visitar, si lo visito, perdón, puedo ir, puedo desconectarme, puedo volver y ya funciona. Pero

esto es exactamente lo mismo que veíamos en el modo desarrollo. ¿Qué es lo que quiero mostrarte ahora aquí?

Cuando estamos en modo servidor, todas las páginas que se sirven inmediatamente tienen un código fuente, que ahora voy a ampliarte y ponerlos un poquito más visibles. Esta parte de aquí es el **CSS**, pero en esta parte de abajo, aquí vemos que aquí hay esto sobre **Paddle Surf**, esto es **Danubio**, esto es **Kayaking**, esto no sé qué, es decir, un robot podría tener material aquí para indexar esta página, saber de qué va, saber que, si alguien busca **Paddle Surf** en el lago **Leman**, pues seguramente podría llegar a esta página. Y esto ocurre también si me voy justamente a esa página de **Paddle Surf** en el lago **Leman**, es decir, yo puedo hacer esta petición, ir a ver el código fuente de la página y me fijaré que tengo otra vez todo el **CSS**, pero aquí ya tengo la información con cuánta gente está participando, cuántos son los participantes totales, pues la información específica del **Paddle Surf** en el lago **Leman**, pero no otra cosa, lo que estoy viendo es esta página concreta, es decir, lo que te quiero mostrar es que el **server side rendering** devuelve la petición completamente resuelta para todas y cada una de las páginas, la resuelve del lado del servidor, lo mismo ocurriría para este **login**, podríamos ver el código fuente de página y ver que esto en algún momento tendrá un formulario con un email, **login**, etc.

Y esto permite a los que estén interesados en que sus páginas sean descubiertas, claro que si tu aplicación es una aplicación local no tiene mayor interés, pero para los que quieran que sus páginas sean indexadas, el **server site rendering** es una mejora muy importante, porque satisface al robot dándole la información y satisface al usuario porque de todas formas la navegación local sigue siendo, sigue estando permitida como hemos visto e incluso la navegación offline una vez visitada. Y todo esto, claro, siempre que tengamos datos.

4.3.3 SEO y metadatos

Continuando con esto del **SEO** y viendo, por ejemplo, que estamos renderizando la página principal, hay cosas que no se ven a primera vista, como es lo que viene dentro del **HTML** en el **head**. Aquí estaría, por ejemplo, el **title**. Bueno, esto sí se ve, es el título de allá arriba, pero otros **metas** que no se ven. Y la pregunta es, ¿podemos hacer algo en **Angular** con esto? La respuesta es que sí, y para ello vamos a aprender un par de trucos nuevos. Me voy a la **home.page** y lo primero que voy a hacer es reclamar dos ayudantes de **Angular**, que se hacen mediante la solicitud de una inyección de dependencias. Esto ahora mismo me queda un poquito raro, pero la sintaxis es muy simple y ya dedicaremos un tiempo a ver cómo crear incluso nuestros propios servicios.

Ahora mismo, quédate con que cada vez que invoquemos a **inject**, lo que le estaremos es pidiendo a **Angular** que nos inyecte un servicio para poder utilizarlo. Normalmente, el uso que hagamos de estos servicios va a ser sólo durante el código y no en la **template** y, por tanto, los procuraré a hacer privados. Yo lo hago con el prefijo **#**. Y si he solicitado este servicio es porque quiero hacer uso de él y, para ello, en el constructor voy a invocarlo y veré que me ofrecen las posibilidades básicas, que realmente será poder modificar el **title** o obtener su valor. En este caso quiero modificar su valor y voy a poner algo bastante más explícito como este texto. En cuanto lo haga y si me vuelvo a la ejecución, veré que ahora el **title** pone, pues eso, **Activities to book**, que es lo mismo que se ve aquí arriba. No sé si podrás llegar a verlo, pero bueno, esto es el cambio que se ha producido.

Se podrían hacer otros cambios mediante otro servicio llamado **meta**. Así que, de nuevo, le pido a **Angular** que me lo inyecte. Todo esto viene de **@angular/platform/browser** y, a partir de aquí, hago uso de él. Normalmente puedo agregar una etiqueta con un nombre y una descripción o podría utilizar **updateTag**, que es un poquito más versátil porque si existe la modifica y si no la crea igualmente. Ahora veremos que por aquí se me ha agregado una etiqueta **meta** con ese texto que, por ejemplo, para un robot indexador podría ser muy útil. Y hablando de utilidad voy a hacer eso mismo, pero en la página de las reservas donde debería haber información específica de una actividad concreta. Igualmente agrego ya los dos servicios que necesito y en el constructor tendré que hacer uso de un nuevo efecto porque realmente para poder dar el título y la metadata necesito disponer del objeto **activity**, que te recuerdo es una señal computada a partir de la señal de entrada del **Slug**.

Así que este efecto se ejecuta una vez que tengo actividad o por si esto cambiase durante la ejecución obtengo ya su nombre, una descripción que me resulta interesante y vamos a verlo. Ahora al entrar en cualquiera de ellas ya puedo ver aquí una descripción que por supuesto el usuario no ve, pero a los robots les podría venir fantástico. Lo que el usuario sí que ve es como aquí me ha cambiado el título y pone pues **Hiking**, no sé si lo podéis ver ahí, este que es un poquito más largo pues **Paddle Surf**. En definitiva, **Meta** y **Title** me permiten cambiar elementos en la cabecera del **HTML** que pueden servir para visualizar en los títulos o simplemente para que los robots puedan indexar mejor el contenido de mis páginas.