6. Patrones de escalado

Entramos en el tema que más me gusta de **Angular**, aquel que me permite aplicar **patrones de diseño** a un **framework** de **frontend**. Y para empezar, el **patrón contenedor-presentador** que reparte las responsabilidades de un **componente** en uno que se dedica a los datos y otro que se dedica a presentarlos. Después continuaremos extrayendo la responsabilidad del acceso a los datos del **componente** a otra cosa, en este caso serán los **servicios** que inyectaremos para que lleven ahí la **lógica**.

Por último, intentaremos mantener nuestra **aplicación** lo más limpia posible y extraeremos todo aquello que sea reutilizable a algún sitio compartido.

6.1 Patrón Container/Presenter

Con el tiempo, cualquier **aplicación** que hagas en **Angular** va a acabar complicándose mucho, o quizá muchísimo. Pero nuestra intención será siempre mantenerla de tal manera que la podamos simplificar de alguna forma. La parte visual tendrá sus cosas, y asociado, yo lo suelo representar así, un **componente**,

acabaré teniendo aquí todas las propiedades que necesite esta vista, aquí tendrá los métodos... En fin, que un **componente** de verdad acabará siendo realmente complejo. Vamos a empezar a ver el **patrón container-presenter**, que lo que hace es determinar qué trozos de esta vista son, a su vez, pequeños **componentes**.

Por supuesto, cada uno de ellos tendrá su **lógica**, pero obviamente una **lógica** más sencilla que la del **componente** padre. La responsabilidad que le daremos al **componente** padre, y que se quedará con ese nombre, es que le llamaremos eso, Parent Smart, porque se ocupará también del acceso a los datos, o **container**, que es el nombre más formal, y a los de abajo les llamaremos **child**, o **children**, si lo quieres, **dumb**, pobrecitos, como si fuesen tontos, pero lo que quiere decir es que no se ocuparán de obtener los datos, sino que los recibirán, o **presenters**, que sería el nombre más formal.

Entonces, **parent-child**, **smart-dumb**, o **container-presenter**, en el fondo es el mismo **patrón**, que por si fuera poco, yo acabo distinguiendo físicamente, porque a estos **componentes** suelo ponerles un sufijo especial en el nombre, **.page.ts**, o **.widget.ts**, el primero para los casos de enrutamiento, el segundo para trozos suficientemente inteligentes. Y a los otros, bueno, pues les acabo llamando simplemente **.component**, o en algún caso, pues **.form**, puede ser.

Bueno, esta es la idea, cómo llevarla a la práctica, ha cambiado un poco en **Angular** en los últimos tiempos, y todo debido a las señales. Vamos a ver entonces cómo se produce este envío de información desde un **componente** padre, que seguramente obtiene los datos desde un **API**, para nosotros es una base de datos, hacia su **componente** hijo, cómo fluye esta información, y cómo se la devuelve mutada, si es que también queremos que el flujo vaya en ambos sentidos. Para ello, en el **componente** hijo,

se declararán propiedades públicas, que funcionarán después como atributos, y serán de tipo señal, aunque cada propiedad, propiedad 1 por ejemplo, podría ser un **input**, podría tener, o debería tener un tipo concreto, podría tener un valor inicial, y esto permitirá recibir cambios, y tratarlos en la interfaz del **componente** hijo, como si fuesen señales que han cambiado, por lo que sea desde su padre, pero que han cambiado y que tiene que repintarse.

Si a su vez el **componente** hijo también quiere notificárselo a su padre, por ejemplo, podría tener una propiedad 2, de tipo **model**. Estas son propiedades breakable, es decir, que son cambiables en el hijo, puede hacer cosas como prop2.set o .update, y notificarse, recibirse esos cambios automáticamente en el padre, o bien, incluso, mediante eventos. Esto sería como emitir un evento hacia arriba.

Y la sintaxis que vamos a involucrar en el lado del padre será corchetes **[]** para enviar información, como si fuese un atributo, y paréntesis **()** para recibirlo como un evento. O la mezcla de ambas, que será lo conocido como **banana in a box**, donde tenemos el nombre del modelo, dentro de unos paréntesis **()**, y envuelto en unos corchetes**[]**. Esto nos permite enviar información hacia abajo y hacia arriba, esto nos permite enviar información hacia abajo, y esto, enviarla hacia arriba. Todo esto se ve mejor en la práctica con un ejemplo.

6.1.1 Extraer presentación a un componente simple

Ahora que tenemos algo de funcionalidad ya en nuestra **aplicación**, es un momento para refactorizar y prepararnos para el crecimiento. Para ello voy a aplicar el **patrón container presenter** que va a repartir las responsabilidades de los **componentes**. Voy a tener donde había un **componente**, voy a tener dos. Uno se encargará o se quedará con la responsabilidad del acceso a datos, mientras que el otro se encargará de la presentación. Puede ser de todo o de una parte. Voy a empezar por una parte muy sencilla que es la presentación de una actividad.

Para ello voy a crear un **componente** nuevo en la misma carpeta donde tengo el **homepage**, pero será un **componente** sin un atributo especial en el nombre, será simplemente un **componente**. Distingo aquellos que se van a encargar de algo más, que estarán enrutados y que tendrán acceso a datos por el sufijo **page**. Y mantengo el sufijo **component** para todos los demás.

Una vez que tengo el **componente**, lo primero que haré será llevarme lo que me sobra de un sitio al otro. Y obviamente esto me traerá un montón de problemas. Entre ellos me dirá que aquí no se conoce el **routerLink**, por supuesto no saben que es una activity y aquí me habrá quedado un lugar vacío, pero todo a su tiempo. Para que las directivas **routerLink** o los **pipes currency** y **date** dejen de protestar, me traigo los **imports**.

Lo siguiente es que voy a necesitar aquí un objeto actividad. Este objeto actividad o esta propiedad pública de esta clase deberá rellenarse, pero no será responsabilidad suya, él solo se ocupa de presentar. Para que sea responsabilidad de otro lo voy a declarar de tal manera que se convierta en un parámetro de entrada y eso en **Angular** se le llama un **input**. Este **input** puede ser de un

determinado tipo, como por ejemplo el tipo **activity**, y puede o no ser requerido en este caso, y casi siempre será que sí.

Y a partir de este momento, este **input.required.Activity** se va a convertir en una señal que cada vez que cambie, cada vez que su padre me invoque y me envíe datos, ahora veremos cómo, va a reflejar esos cambios. Obviamente en cuanto deja de ser una propiedad normal y pasa a ser una actividad, tengo que refactorizar porque sabemos que las señales tienen que ser invocadas como funciones. Tras un breve refactoring, este **componente** queda listo para ser utilizado.

6.1.2 Refactorizar componente contenedor inteligente

Bueno, pues hecho el trabajo de extracción, ahora queda la recolección de los frutos, pero es ya muy sencilla. Lo que sí que ocurrirá es que este **componente**, para que sea conocido en el ambiente de este otro, necesitaremos agregarlo a su lista de **imports**. De hecho, casi diría que sustituye a todos los demás, porque se han ido a donde son necesarios, aquí ahora ya no son necesarios. Entonces, este **ActivityComponent** puede ser utilizado dentro de este **for**. Podemos simplificar este tipo de elementos cuando no tienen contenido.

Y lo que estamos haciendo, lo que estamos viendo, es que la propiedad **Activity** de este **componente** se convierte en un argumento, un atributo de entrada de este elemento **HTML**. Visto desde aquí, esto sería un elemento con su atributo y su valor, mientras que, visto desde aquí, es una propiedad pública que se expone, no por ser propiedad pública, sino por este **input**, se expone como algo utilizable. Una vez que yo tengo esto utilizable, lo relleno, por ejemplo, pues con esta actividad, etc. Y ya está. A partir de ahora, tengo una manera de descomponer mis **componentes**, valga la redundancia, pasándole los datos que necesiten y haciendo que estos presentadores se ocupen sólo eso, de la presentación, dejándome a mí el grueso de la **lógica** que también vamos a repartir.

6.1.3 Comunicación entre contenedor y presentador

Bueno, pues hemos visto la comunicación padre-hijo desde un **componente** contenedor que envía datos para que el presentador los presente y los trata aquí como señales de solo lectura, no los puede cambiar, y aquí pues le enviamos un valor cualquiera. Y ahora vamos a ver algo más. Vamos a ver que un **componente** hijo puede también mutar ciertas señales y el padre enterarse de esos cambios. A esas señales mutadas les llamaremos **modelos**. En esta división de **modelos** vamos a crear un array para almacenar las actividades favoritas. No será más que

un array de strings, pero envuelto en algo llamado **model**. Lo mismo que esto era un **input** y que en otras situaciones hemos visto el **signal**, aquí lo que tenemos es un **model**.

Desde el punto de vista de este **componente**, esta variable actuará como una señal writable, es decir, que la podrá modificar tanto con **set** como con **update**, que es lo que vamos a ver ahora. Voy a crear inmediatamente un método para manipular este array, que lo que hará es, recibiendo el slug de una actividad, actualizar esta señal mediante el método **update**, no el **set**. El **set** lo habíamos visto para sustituciones totales. Aquí nos interesa tratar con lo que había y modificar la señal. Ojo que en cualquiera de los dos casos, tanto que ya incluya el slug y por tanto lo quitemos, porque aquí estamos haciendo el típico toggle de te pongo favorito o te quito de favoritos, como agregándolo, siempre retornamos un nuevo array, no mutamos el array favoritos, lo usamos como punto de entrada, devolvemos otro distinto.

Esto es típico de la programación reactiva, de tratar a las variables como inmutables y devolver siempre clones. Pues una vez que hemos hecho esto, sería cuestión de agregarlo a algún **input** aquí en el **HTML**. Esto no es más que un **input** de tipo **checkbox**, donde asociamos el evento click con este método, pasándole el valor actual del slug correspondiente.

¿Y cómo se recoge esta cosecha desde el **componente** hijo en el **componente** padre? Pues facilísimo. Lo único que tengo que hacer es crear igualmente una señal de favoritos, la voy a hacer writable, y esta señal modificable es lo único que tengo que enlazar con este **modelo**. Fíjate que la actividad era de sólo entrada, era un **input signal**, pero esta es de doble entrada, de entrada y salida. Entonces, en estas situaciones de entrada y salida tenemos que utilizar el convenio del corchete **[]** y el paréntesis **()**. El corchete **[]** y el paréntesis **()** son conocidos como **banana in a box** por la forma, digamos, con los paréntesis del plátano y por la parte de encajar eso en una caja. Bueno, si te sirve de referencia para acordarte, pues está bien, pero lo interesante es que veas que los paréntesis **()** aportan la entrada, inicialmente vacía de datos hacia este modelo, y los paréntesis **()** actúan como un evento que comunican los cambios hacia arriba. Luego corchetes **[]** y paréntesis **()** son envío de datos hacia abajo y hacia arriba, **double binding**, enlace doble de modelos entre padres e hijos. Para ver en el padre qué efecto tiene esto, voy a agregar aquí un pie a esta página que me muestre cuántas actividades tengo, eso ya lo sabía, y ahora cuántos favoritos he seleccionado. Veremos cómo en ambos casos se tratan como señales y que, desde este punto de vista, son exactamente lo mismo. Pues ahí tenemos el contador, cómo va cambiando el número de actividades favoritas y el número de actividades totales. En el futuro

incluso llevaremos estos cambios más allá del **componente** padre al **componente** cabecera.

6.1.4 Comunicación de eventos desde el presentador al contenedor

Vamos a seguir con este **refactoring**, convirtiendo ahora el **footer component** en un **footer widget**, dotándolo de una mínima inteligencia para leer y escribir en un **repositorio** local el estado de las **cookies**. Para ello también agrego aquí un nuevo tipo que me permita tener una especie de enumerado con los estados posibles de estas **cookies**. Sustituiré también este sencillo **if else** por algo un poquito más complejo y así vemos un caso de uso para un **switch** que irá sobre una señal pues que va un poquito más allá de un boolean y puede empezar siendo una señal modificable con un estado inicial **hardcoded** o incluso leído.

Vamos a verlo desde el **localRepository** y convenientemente tipificado. No necesitaremos respuesta a eventos porque todo lo que hagamos lo haremos utilizando exclusivamente señales y efectos. He dejado sin programar este caso cuando aún el usuario no nos ha dicho qué piensa hacer con las señales para que aquí programemos el **componente presentador** que nos transmitirá, que nos emitirá el deseo del usuario. Abandonamos entonces el **componente inteligente** y le vamos a crear un **componente simple presentador** llamado **cookies**.

En este caso el grueso va a estar en la parte presentadora en la que meteremos un diálogo con tres botones que permitirán que el usuario rechace las **cookies**, acepte solamente las esenciales o las acepte todas. Tal como habíamos visto que podíamos utilizar un **input** para recibir información ahora utilizaremos un **output** para emitirla, pero esta función **output** no devuelve, no es de tipo señal sino que es un **output emitter**. De todas formas la sintaxis es exactamente la misma que hubiéramos hecho en el caso de un **input**. Pensaremos ahora que este **componente** que captura la intención del usuario al final se puede reducir a algo que o bien la rechaza o bien la acepta, parcial o completamente. Así que eso es lo que haré, tener aquí dos señales, una para el caso en que rechace las **cookies** y otra en caso de que las acepte que como ves puedes tipificarlo a algo más concreto. Es decir, durante el proceso **output** se puede enviar una señal vacía, sin nada, simplemente indicando que se ha cancelado sin ningún argumento extra o con un argumento que puede ser un número, un objeto, un enumerado, cualquier cosa.

Nos quedaría ahora invocar desde los eventos nativos del **HTML** como el click una suerte de transformación a eventos de negocio como puede ser el **cancel**. Y aquí la sintaxis será **emit**, similar a lo que podría haber sido un **set** en una señal o un **next** en un **observable**. Para el caso de que queramos transmitir, emitir,

argumentos pues será similar, pero enviándole ahora el argumento que queremos. Fíjate que al estar tipificado no puedo enviar un número ni cualquier texto que se me ocurra, tiene que ser algo previamente acordado. Y aquí se acaba la responsabilidad del **presentador**, que en este caso es un presentador de acciones, que emite algo que de este lado recibiremos como eventos. Así que incorporamos el **componente de las cookies** al **footer**.

Cookies es un presentador, el **footer** es un **widget inteligente** y este **componente** ahora me va a ofrecer como eventos, como si fuesen nativos, como si fuese el click, me va a ofrecer las señales enviadas como **output**. Yo las llamo señales porque dentro de este concepto de comunicar **componentes** a través de una sintaxis novedosa como son los **input**, **model** y **output**, aunque los **output** no sean técnicamente señales, se comportan de una manera parecida y al final acaban agrupándose dentro del mismo paraguas. Pero efectivamente son emisores hacia afuera. Lo que hagamos en estos métodos puede ser complejo o muy sencillo. Por ejemplo, en el caso de que el usuario quiera cancelar, pues rechazaría o pondría en estado rechazado las **cookies**. Y en el caso de que las haya aceptado, aquí ahora tendré la oportunidad de recibir un argumento que genéricamente se llama **$event**. Este **$event** está tipificado según el **output** que hayamos creado aquí, en este caso de tipo **acceptance**, que son estos dos strings que coinciden con estos que recibimos aquí y no tendremos problema.

Pero te recuerdo que aquí yo no puedo poner cualquier cosa, ni siquiera cualquier texto. Bien, de esta manera recibimos ambos eventos y son transformados a señales. Una cosa que puedo hacer ahora es guardar en el **localRepository** el estado de esta señal y para ello me serviré de los efectos, pero sin necesidad de crearlos en los **constructores**, sino creándolos también como propiedades. La sintaxis es esencialmente la misma. Simplemente declaro una propiedad en la cual le asigno un efecto que dentro, por supuesto, lleva una función donde en alguna de sus instrucciones se hace referencia a una señal.

En este caso lo que voy a hacer es guardar en el **localRepository** los cambios que ocurran en el estado de las **cookies**. Todo esto funciona gracias a que hay un entorno de ejecución donde las inyecciones son conocidas. Esto sólo ocurre de manera gratuita en los **constructores** y en las declaraciones de propiedades. En otras situaciones deberías tener que enviarle tú el contexto de inyección. Pero vamos a ver cómo funciona esto. Y ahora al arrancar la aplicación me aparece este diálogo. En la sección de **application** tendré también las **cookies** en estado **default pending**. Si las cancelo se pondrán **rejected**. A partir de ahora siempre quedarían así. Tengo que voluntariamente borrar esto para que me vuelva a preguntar y ya tenemos el sistema completo. Un **presentador** con **outputs** y un **contenedor inteligente**, en este caso no enrutado, por tanto un **widget** que es

capaz de hablar con los **repositorios** locales o remotos para recuperar o cambiar el estado.

6.2 Servicios e inyección de dependencias

Como vimos con un **componente** complejo, podemos dividir su vista en trozos más pequeños. Pero ¿qué hay de su **lógica**? Sobre todo la **lógica** de los **parent containers**, donde tenemos un montón de **propiedades** y también potencialmente un montón de **métodos**, y aquí sí, con bastantes líneas de código. ¿Cómo mantenemos a raya esta complejidad? Pues extrayendo parte de su contenido a otras clases que no tendrán representación visual, serán simplemente clases con **propiedades** y **métodos**, y que de alguna manera es como la programación que acostumbras a hacer en cualquier otro lenguaje. Y a esto le llamaremos **servicios**.

Estos **servicios** se decorarán, como muchas otras cosas en **Angular**, con una **@** que marca esto como algo **inyectable**. **Inyectable**. Que sea **inyectable** quiere decir que alguien después pueda reclamar una **propiedad**, que la pondremos privada, con el **#** para indicar que esto es algo privado, un nombre, y la llamada a la función **inject**, pasándole el nombre de la clase que hayamos puesto aquí. Pues en definitiva, extraemos código y nos lo llevamos desde un **componente** a un **servicio**, a uno o varios. Es decir, tú puedes tener **servicios** especializados e incluso reutilizables entre **componentes**, de forma que un mismo **servicio** pueda ser **inyectado** en varios **componentes** y reutilizar así la **lógica**. Al hablar de **inyección** se supone que tengo algo y le voy a inyectar otra cosa. Para que pueda hacerlo tengo que hacer, por un lado, la reclamación, eso de invocar a la función **inject** con el nombre de la dependencia. Este será la **dependency**. Pero esta **dependency** tiene que estar en lo que llamaremos una tabla de cosas **inyectables**. La tabla de cosas **inyectables** puede tener un ámbito, un nombre y un contenido. Al nombre habitualmente le llamaremos **proveedor**.

El ámbito, cuando generas un **componente** y le pones el **injectable**, pondrá **provided in root**. **Root** es el ámbito máximo que permite que todos los **componentes** de tu **aplicación** puedan disfrutar de una instancia de un **servicio**. Y no sólo de una, sino siempre de la misma. Es decir, esta instancia sería siempre la misma. Esto permite, por ejemplo, o nos permitirá en el futuro comunicar unos **componentes** con otros. Este ámbito **root**, que se automarca, cuando generas un **componente**, podría evitarse o podría no hacerlo, podrías cambiarlo y entonces elegirías tú el ambiente, que puede ser de un **componente** concreto o con un aprovisionamiento concreto, aunque sea en el **root**, pero voy a decir **custom**. Esto es lo que hemos hecho, por ejemplo, con el **servicio HTTP client**, que es un

servicio como los nuestros, sólo que generado por **Angular**, que no tiene el **providerInRoot** y que hubo que meterlo en un array, que esta tabla es lo que es en memoria, un array de **providers**, donde se le dice, oye, provéeme el **HTTP client** con una determinada configuración.

Bueno, pero eso se verá más adelante. Por ahora nos quedamos con que cualquier cosa sea un **componente**, o sea, otro **servicio**, puede delegar, puede depender de otros **servicios** llamados **dependencias**, habitualmente aprovisionados en el ámbito de la raíz para todo el árbol de **componentes** y, si no es así, pues en una tabla especial de aprovisionamiento, que puede ser local de un **componente** o de una **librería** o un **módulo**.

6.2.1 Extraer lógica y datos a un servicio fachada

Continuando con el proceso de **refactoring**, ahora te voy a presentar cómo crear un **servicio** y cómo **inyectarlo**. Para ello, usaré de nuevo el generador, sólo que esta vez en lugar del "ng g component" utilizaré "service" (ng s component), **s** en lugar de **c**, y la ruta que me parezca adecuada con el nombre que me parezca adecuado. Aquí aparece un **servicio** llamado **home.service**, un **servicio** auxiliar que suelo utilizar para ayudarle a cada uno de mis **componentes** contenedores para que manejen sus datos de una forma externa. Tal como hicimos al crear un **componente presentador** extrayendo presentación y llevándola a su lugar, ahora haré lo mismo pero con este tipo de información. Por ejemplo, voy a responsabilizar a este **home.service** del acceso al **API**.

Por supuesto, cuando eso ocurra, esto dejará de tener sentido y lo cambiaremos. Pero por ahora lo que voy es a replicar esta funcionalidad de una manera más limpia, de manera que encapsulo en este **servicio** todo lo que tenga que ver con la llamada al **API** y dejaré limpio a este **componente** de esa responsabilidad. Ahondando un poco en que es un **servicio** que como ves ha sido generado igual que un **componente**, sólo que utilizando la **s** de **service** en lugar de la **c** de **component**, te diré que se parecen también en su forma. También son clases con el nombre que tú le hayas dado y el apellido **service** y después llevan una decoración.

En este caso la decoración es una función que se llama **@Injectable**. Es el equivalente al **component** aquí. Esta **@** la ofrece **TypeScript** para los decoradores, que son funciones que agregan capacidades a clases, métodos y demás. Una clase decorada como **injectable** va a ser algo que se pueda reclamar y que haremos después, igual que aquí estamos reclamando la **inyección** de un **HttpClient**, haremos que se reclame la **inyección** de un **HomeService**.

Este **providedIn: 'root'** que me viene regalado querrá decir que este **servicio** está disponible durante toda la **aplicación**. Hay situaciones donde eso no es interesante y lo cambiaremos, pero por ahora y para empezar te diré que el 90% de tus **servicios** estará bien así, **providedIn: 'root'**. Puede que te haya llamado la atención también que junto con la generación del **servicio** ha aparecido otro fichero que tiene de apellido **service.spec** y que además el icono pues aquí me lo cambié un poco. Efectivamente, como te puedes imaginar, es un fichero para hacer **pruebas unitarias** contra el **servicio** que acabamos de generar.

No es el objetivo ahora mismo empezar con las **pruebas unitarias**, pero sí te diré que es una práctica recomendada probar unitariamente a los **servicios**, mientras que quizá ciertos **componentes** se puedan probar con otras tecnologías. En cualquier caso todo eso es controlable. Te recuerdo que el **angular.json** se puede configurar, se le pueden configurar sus **schematics** y tal como he dicho que los **componentes** en principio no quiero que tengan test, no he dicho nada para los **servicios**, pero podría hacerlo. Si pongo esto, a partir de este momento todos los **servicios** que cree se crearán sin su fichero de pruebas, como en principio. Quiero dejar constancia de que los **test** son una buena práctica. Voy a mantener lo que viene por defecto que sería algo así. Ya sabemos entonces cómo crear **servicios** con ng g **service** (ng Generate Service), **inyectables** y **testeables**.

6.2.2 Inyectar dependencias en el componente contenedor

De nuevo, tal como hicimos con el **componente** que lo importamos y lo utilizamos, toca recoger los frutos de este **servicio** y para ello va a ser aún más sencillo. Lo que tengo que hacer realmente es pedir que me lo inyecten igual que antes pedía que me inyectasen el **HttpClient**. Así que eso es lo que haré. Declaro una **propiedad privada** que sólo será utilizada dentro de la clase **home.page** y habitualmente le llamo simplemente **service** porque voy a procurar tener un único **servicio**, un poco siguiendo el **patrón fachada**, para cada **componente contenedor inteligente**. Dentro del **inject** pediré el **servicio** por su nombre de clase. A partir de este momento, si **Angular** lo encuentra como algo **inyectable**, si no pues obviamente fallará, si lo encuentra me dará una **instancia** de este **servicio** para que la pueda utilizar. Y recalco lo de una **instancia** porque así será una **instancia única** a través de un **patrón singleton** que se puede utilizar en toda la aplicación. Por supuesto esto va a cambiar la manera en la que yo estaba invocando **HttpClient.get** por simplemente algo más de negocio como **get activities**. En fin, el código no es que haya mejorado mucho pero ahora hemos encapsulado la **lógica de acceso a datos** y el hecho de que lo realizamos con **HttpClient** e incluso la **URL** a la que invocamos, esto ya no es necesario

conocerlo y nos queda algo más sencillo y más cercano a lo que hace este **componente** que será ocuparse de orquestar los datos con la presentación.

Es un **componente inteligente**, un **componente contenedor** o **componente padre**. Cualquiera de estas maneras también se le conoce. Cuando se enrutan, a mí me gusta distinguirlos con el sufijo o con el tipo **page**. Para finalizar y relacionarlo todo un poquito, te diré que si te preguntabas con **HttpClient** cómo es que funciona el **inject** es porque habíamos puesto en una lista de **providers**, sería algo así como **proveedores**, pues hemos apuntado aquí que tendremos **provide HttpClient**. Esto hubo que hacerlo porque este **servicio** es configurable y permite hacer más cosas. El nuestro es más sencillo y basta con hacer el **providedIn: 'root'** y es como si automáticamente este **HomeService** se hubiese añadido a esta lista de **providers** y estuviese disponible en toda la aplicación. Por ahora quédate con eso. Los **servicios** necesitan ser **inyectables** y aprovisionados en algún lugar o bien lo son automáticamente cuando son generados o bien hay que incluirlos a mano para ser utilizados. En cualquier caso, después se reclaman con **inject**.

6.3 Principio DRY con código compartido

Voy a hablarte ahora un poco de nuestra aplicación pero vista como un árbol de carpetas, así a vista de pájaro, y ver qué cosas contiene. Voy a empezar por decir que efectivamente he creado una carpeta **Routes** en la cual suelo guardar páginas que a su vez usan **componentes** y seguramente **servicios**. Con el tiempo tendré distintas páginas con distintos **servicios** y también distintos **componentes**, claro. Pero puede ocurrir que algo de esto sea común y reutilizable, por supuesto. Y todo eso lo llevaré a una carpeta llamada **Shared**.

En esta carpeta **Shared** se distinguirán las capas tecnológicas de la aplicación que serán el **User Interface**, la **lógica del dominio** y la **infraestructura para acceso a la API**. No todo lo que vaya en estas tres subcarpetas será igual. Obviamente aquí habrá **componentes**, **presentadores**... En **Domain** puede haber **servicios**, **tipos**, **clases**, **funciones**... Esta es la **lógica de negocio** compartida entre varios sitios o que tú quieras extraer. Y en el **API** lo que habrá serán **repositorios** para hacer llamadas **HTTP** o ese tipo de cosas también que puedan ser comunes o reutilizables. He dejado aquí un hueco para meter una carpeta que habitualmente se le llama **Core**, en la cual hay muy poquita cosa hoy en día. Quizá es una rémora del pasado pero que yo almaceno aquí, por ejemplo, **componentes** de un solo uso genéricos como el **Header**, el **Footer** y alguna otra cosa así. Lo importante, toda nuestra funcionalidad estará en **Routes** y todo aquello común en **Shared** dividido a su vez en **componentes** reutilizables, **lógica del dominio**,

reutilizable, **API** reutilizable, bien colocada, extraída del resto de la funcionalidad. Vamos a verlo en código.

6.3.1 Servicios y utilidades de datos comunes

Te presento aquí dos archivos aparentemente no relacionados. Por un lado, el recién creado **home.service**, dedicado a dar servicio exclusivamente al **home.page**. Y por otro lado, **bookings.page**, otra página que, sin estar relacionada directamente con la raíz, pues también tiene alguna lógica que se puede asemejar, ¿verdad? Vuelve a llamar aquí a un **HTTP**, tiene la **URL** de las actividades, y aunque formalmente no necesita el **getActivities**, sí que usa otras cosas por el estilo.

Lo que te quiero plantear es la posibilidad de crear **servicios** que sean comunes, que sean compartidos a varias páginas, a varios servicios o entre páginas y servicios. La novedad del servicio que voy a crear es que está en la carpeta **Shared**, una carpeta que aún no existe, existirá, incluso se tragará a **Domain** y permanecerá en la raíz junto a **Core** y **Routes** durante el resto del desarrollo. Profundizaremos más en el sentido que tiene cada una de esas carpetas.

Una vez creada la carpeta **Shared**, en este caso le he metido otra más, **API**, que es donde voy a guardar todo lo que tenga relación con llamadas al **API**, y después un servicio llamado **Activities**, que por ahora no tiene nada. De hecho, podría empezar con algo tan sencillo como la funcionalidad del **home.service**. Y a partir de ahí el **home.service** podría delegar absolutamente su trabajo en él. Para ello, lo que haríamos sería llamar al **ActivitiesService**, tal que así.

Sólo que si lo hacemos de esta manera, fíjate que la importación me queda bastante fea, porque empiezo a tener que moverme por varias carpetas hacia arriba, hacia abajo, etc. Para evitar esto, voy a enseñarte un truco que reduce y simplifica muchísimo estas importaciones. En el fichero **tsconfig** que tenemos en la raíz, puedo meter unas configuraciones extra, como estas, que me permiten definir un nuevo camino apuntando a una ruta específica y física en el directorio. Entonces, puedo sustituir cualquier llamada hasta **API** por simplemente **@API**. Como ves, esto va a dejar todo mucho más limpio.

Si yo tenía aquí un servicio que se llamaba **getActivities**, tengo que mantenerlo. Sólo que ahora mismo me queda así como una especie de **man in the middle**, porque delega específicamente todas las llamadas contra este otro método. Es un buen punto de inicio para que empecemos a pensar en los servicios asociados a **componentes smart** de páginas, como si fuesen **fachadas**, que a su vez utilizan a otros. En condiciones reales, aquí habría bastante más de un servicio. El **home.service** es fácil, pero el **bookings.page** ya no lo es tanto. Para empezar, las

llamadas a lo que podrían ser métodos que nos llevemos al **ActivitiesService** están por el medio del código, no serán tan sencillas de encontrar.

Entonces, un truco que uso yo es decir, bueno, ¿qué es lo que realmente tienen de común? Pues, por ejemplo, la **activitiesUrl**. Si comento esta propiedad, empezará a fallar indicándome un sitio donde debo refactorizar. La refactorización empieza por llevarme inmediatamente el código de lo que estamos haciendo aquí al nuevo servicio. Por ejemplo, aquí estoy intentando obtener una actividad a partir de un slug. Bueno, pues eso es lo que hago, **Get Activity by Slug**.

Un pequeño renombrado para homogeneizarlo todo. Y me fijo en que no solo estoy haciendo un **get**, sino que estoy creando una tubería que transforma la información y que captura errores. Quizá esto sea útil mantenerlo de este lado del servicio, de manera que a este lado podamos simplificar esta llamada. Una vez lo he inyectado, lo uso en lugar de todo esto, incluso del **pipe**. Eso sí, me falta la llamada al método. Pues se trataría de seguir buscando sitios donde estábamos usando la **URL** de las actividades, que son indicadores como este, de que esta lógica podría trasladarse para aquí. Por ejemplo, en esta situación lo que estamos haciendo es un envío, un **Put**, de una **Activity**.

Así que creamos el método adecuado, en el cual te cuento que de la misma manera que nos suscribimos a los errores y retornamos objetos nulos, simplemente podemos hacer un **console.error** para tener claro que ha ocurrido, pero se lo relanzamos para que el que nos llame se entere. Ahora la llamada aquí quedaría mucho más sencilla, porque en el fondo es llamar al método **putActivity**, pasándole una actividad y suscribiéndonos para cuando las cosas acaben bien, diciendo que todo ha ido bien, mientras que mantenemos el mensaje de error oculto del otro lado. Tú podrás decidir cómo balanceas la lógica de un sitio a otro.

Y hablando de lógica, aún queda una lógica bastante enrevesada por aquí, este **toSignal**, **toObservable**, con el **SwitchMap**... No es momento aún de que entiendas perfectamente todo esto, pero sí que es momento por lo menos de ocultarlo, porque la verdad es que esto, o algo como esto, lo vamos a utilizar mucho. Cada vez que un input cambie y necesitemos utilizarlo como entrada a una llamada asíncrona, se sucede este problema de pasar de **toSignal** a **toObservable** con un **SwitchMap**.

Para ello voy a crear aquí en la carpeta **API**, a mano, un fichero. Sí, también se pueden crear ficheros así, a mano. En principio, lo que he puesto aquí no es más que una función que intentará traerse toda la complejidad del **toObservable** y del **toSignal** a un sitio único y común, de manera que esto me quede más sencillo. Lo interesante es que quiero que veas aquí cómo exporto una función, que en este caso tiene la complejidad de usar otras funciones como **toObservable**, **toSignal**, etc., cosas que ahora mismo aún te costará un poquito entender, o el **SwitchMap**,

que no lo he explicado aún, pero lo interesante es ver que puedas tener ficheros de utilidades donde tengas funciones o tipos exportados que se puedan utilizar. Eso sí, no caigas en un cajón desastre y organiza las cosas. Por ejemplo, si esto tiene que ver con el **API** y trata de señales, pues se llamarán **signal.functions** y estará igualmente dentro de esa carpeta. Para utilizarla no tengo que inyectar nada, simplemente es una función pública exportada. La utilizo así, sin más, y obviamente va a requerir una importación. Que, como ves, con el **@api** quedan tan difíciles de localizar.

Veremos cómo hacer más de lo mismo, pero ahora con el dominio.

6.3.2 Lógica y tipos de dominio

Muy bien, pues en nuestro afán de encontrar cosas comunes que podamos reutilizar, voy a subir una capa, estamos en la capa **API**, y me voy a ir a la de la **lógica de negocio**, que ya había empezado a crear algo en una carpeta llamada **Domain**. Aquí, si tú le llamas **Business Logic Layer**, **Business**, etc., como tú quieras. Pero esta capa, esta carpeta **Domain**, voy a, antes de nada, moverla, desplazarla al **Shared**, porque será algo a reutilizar por las demás rutas. Por ahora aquí lo único que tenemos son definiciones de tipos, constantes que puedan utilizarse en lugar de valores nulos, para no tener que andar comprobando si algo es nulo o no. Pero está claro que esta carpeta **domain**, que si hubiera nacido ya aquí en **Shared** desde un principio, pues le hubiera venido fantástico tener este path, este alias, apuntado aquí.

Lo digo porque a partir de ahora, todas las llamadas al **domain** empezarán por **@domain**, pero las que ya estaban hechas, pues no es así. El truco más barato sería eliminarlas y esperar que las vuelva a crear el sistema. Como ves, ahora ya parten de **@domain**. Después de hacer eso, en todas partes, que por ahora son pocas donde se usa, pues vuelvo aquí a **bookings.page** y empiezo a pensar si dentro de todo este código que tengo hay algo que sea de **lógica pura de negocio**, que realmente casi ni tenga nada, nada, nada que ver con **Angular** ni sus aledaños como el **rxjs** y demás. Y me encuentro, por ejemplo, **lógica** como esta. Esto sería un buen candidato para extraer bien a **servicios**, como hemos visto hasta ahora, o bien simplemente a funciones. Os recuerdo que las funciones no tenemos algo fácil que nos genere un fichero así tal cual, pero obviamente podemos crearla a mano.

Entonces, en este **activity.functions** me voy a traer una implementación de ese mismo código. Esto simplemente recibe una actividad a los participantes, hace una pequeña lógica y lo cambia. Y ahora es simplemente utilizarlo. Te recuerdo que para utilizarlo no necesito hacer una importación desde el punto de vista de

Angular, sino simplemente desde el punto de vista de **TypeScript**, que debería ser automática. Esta es la llamada que tengo que hacer. Esta es la importación. Y lo que pretendo es limpiar mi código, extrayendo cosas que pueda tranquilamente llevarme a ficheros de funciones. También podrían ser entidades estas actividades. Yo las estoy declarando como tipos, pero si tenéis la buena práctica, ¿por qué no?, de utilizar clases con lógica propia, es decir, crear entidades, ya que estamos hablando aquí de dominio, podríamos hablar de entidades tranquilamente, pues estas funciones empezarían a ser métodos de esas clases. Cualquiera de los dos paradigmas me parece bien, siempre que tú extraigas aquello que esté molestando en donde no pertenece. No pertenece la **lógica de negocio** a algo de presentación.

6.3.3 Componentes reutilizables

Como ejemplo, vamos a hacer la última capa que nos quedaría en una arquitectura clásica donde tenemos el **API** o infraestructura de acceso a datos en una aplicación web, el dominio con la **lógica de negocio** y por encima podríamos poner aquí la **User Interface**.

Esta carpeta de **User Interface**, que tampoco ha nacido todavía, aparecerá aquí en la carpeta **Shared** en cuanto yo cree algo. ¿Y qué voy a crear ahí? Pues serán **componentes**. ¿Qué componentes? Pues aquellos que o bien vaya a reutilizar o simplemente que sean un poquito complejos. Pero hay algunas cosas que me molestan un poco. Por ejemplo, yo tengo aquí unos estilos que están muy asociados a este elemento de aquí, que a su vez necesito un **uppercase**. Podría interesarme encapsularlo y aún encima poder reutilizarlo en la **home.page**, que llama aquí a un elemento que muestra unas actividades y que le vendría de perlas agregarle aquí el estado de la aplicación. Estoy mostrando la localización, el precio y la fecha, pero no el resto. Obviamente puedo crear **componentes** que agrupen a otros componentes, todo aquello que encuentre yo que sea reutilizable lo podría hacer. Para empezar, lo que necesito es crear el **componente**, que le voy a llamar **activity-status**, porque va a estar específicamente dedicado a mostrar el estado de una actividad. Lo meto dentro de la carpeta **UI** y dentro de la carpeta **Shared**. Una vez que nace y está aquí, me lo traigo. Pues viene en pañales, pero vamos a empezar a rellenar un poquito. Según rellenemos las cosas aparecerán problemas, obviamente, porque reclama dependencias, es decir, le falta aquí una propiedad para esto, le falta aquí otra propiedad para esto, le falta aquí el **uppercase**. Voy a empezar por él porque es el más sencillo. Me lo llevo de aquí, de hecho, seguramente ya no lo necesitaré más, así me queda una marca de qué es lo que me he movido, y me lo traigo para aquí.

Otra parte bastante sencililla es que los estilos que los tengo asociados a esta clase, pues también me los puedo llevar. Esto también es un corta y pega. Y una vez hecho lo fácil, pues vamos a intentar hacer lo que es un pelín más difícil, que tampoco es para tanto. Es simplemente pensar que aquí necesito acceder al estado, al **activity.status** en concreto, pero si me doy cuenta, pues solo con el **status** también me vale, no necesito la actividad completa.

Por lo tanto voy a crear, para empezar, una propiedad pública llamada **status** que actúe como un **input**, porque me la va a pasar el **componente contenedor**. Será requerido, este componente no tiene sentido si no tengo un estado, y el tipo concreto pues ya será el de **ActivityStatus**, que viene de un enumerado que teníamos aquí, donde la definición del tipo **Activity**.

Una vez hecho eso, este **status** será lo único que necesite, y como sabemos que los **inputs** son señales, pues esto habrá que invocarlo. Desde este punto de vista ya está todo correcto, lo que necesita lo pide como un **input** y lo utiliza, en este caso como una señal, que además, pues si tiene que pasar por un **uppercase**, pues lo usa, y si tiene que utilizar una determinada clase **CSS** la usa. ¿Y ahora qué? ¿Cómo utilizamos esto del otro lado, donde estaba anteriormente? Muy fácil, lo primero es que antes de utilizar algo por su selector, debo importarlo, así que lo agrego aquí al array de importaciones, y lo siguiente es eliminar esto y cambiarlo, efectivamente, por una invocación por el selector, pasándole como **input** de entrada el valor del estado. Una cosa muy a tener en cuenta, los **inputs** son señales, técnicamente hablando, pero cuando los invocamos le pasamos valores, lo que sea, en cada momento el que toque. No le estoy pasando aquí, no tiene que ser esto una señal. De esta manera, una vez que tengo esto creado en un **componente**, puedo utilizarlo agregándoselo a cualquier otro. Lo que tienes que ver es cómo reutilizo lógica, representación, y en caso de que aún no la reutilizase, al menos la encapsulo y me la llevo a donde tiene que ir. De esta forma, mis **componentes** son cada vez más sencilos y además, si puede ser, se homogeneizan.