5. Comunicaciones HTTP

Angular es un **framework** para hacer aplicaciones web que obtienen sus datos normalmente desde un **API REST** y esas comunicaciones siempre han sido asíncronas. Vamos a ver cómo **RxJS** sigue ayudándonos con ello. Pero claro, esa comunicación asíncrona ahora convive con las señales que notifican los cambios en la presentación. Esa interoperabilidad es parte de **Angular** Moderno.

Y si te preguntas cómo vas a controlar todo ese intercambio de información entre el cliente y el servidor, la respuesta es con tuberías de **RxJS** que te darán una potencia extra con sus operadores inteligentes.

5.1 Consumo de un API

Hasta ahora habíamos supuesto que teníamos siempre un servidor que se comunicaba con un **browser** enviándole **templates** de **HTML** y **Javascript**. Se supone que este **Javascript**, junto con las plantillas, generaba contenido para satisfacción del usuario. Pero nos faltaba algo más, que eran los datos. Hasta ahora iban incrustados dentro de este **Javascript** y eso no tenía ningún sentido.

Lo que sí tiene sentido es aprovisionar un servidor, normalmente un **API REST**, que puede estar en la misma máquina física o no que la que sirve las plantillas de **html** y los **javascripts**. Esto nos da un grado más de libertad, por supuesto, con el que nos comunicaremos enviando información en el formato **JSON**.

Entonces, este intercambio de información lo que va a hacer es llevar los datos a una base de datos y también mostrárselos al usuario. Claro, la comunicación entre estas máquinas del **browser** y el **API** tiene que ser lo más fluida posible, pero para que el usuario no se resienta de ninguna manera entraremos en el mundo del asincronismo. Las tecnologías asíncronas para comunicar **browsers** con **APIs** llevan desde los tiempos inmemoriables de **AJAX** y después han pasado por las **Promises** hasta llegar al **Async Await**, el **Fetch** moderno.

Pero en **Angular** utilizaremos una tecnología propia, o por lo menos una tecnología apropiada, mejor dicho, porque el propietario, la librería propietaria se llama **RxJS**, **Reactive Extensions for JavaScript**, y estas extensiones reactivas funcionan con una tecnología llamada **Observables**. Estos **Observables** me permiten suscribirme y obtener los datos cuando regrese la petición. Haré una petición y me suscribiré a ese resultado, a esa **response**. Tradicionalmente los **Observables** los podemos tratar como si fuesen fuentes de un **stream**, que es como un río, que va a llegar a un sumidero a alguien que lo consume, como si fuese el mar, y que emiten distintos eventos. Pero en el caso del **HTTP** lo que vamos a tener son observables de un solo disparo. Fluirá el estado de la petición, de manera que cuando lleguemos al final, donde habrá una suscripción, obtendremos o bien los datos, si las cosas han ido bien, o un error, si es que algo ha pasado.

Típico **404**, no encontrado, **401**, no tienes permisos, un error **500**, el servidor no responde... En definitiva, usaremos la tecnología observable con fuentes de **streams** para suscribirnos a los resultados de las operaciones. Esto tendrá mucha más potencia cuando estos **streams** realmente se canalicen en tuberías que permitan operar con esos datos. Y para ello, **Angular** me ofrece un servicio llamado **HTTP Client**, el cual a su vez tiene varios métodos, **GET**, **POST**, **PUT**, **DELETE**, para hacer el típico **CRUD** contra un **API** que actuaría como una base de datos. Todos ellos van a disponer de tipos de datos para especificar qué es el

resultado esperado y todos ellos tendrán la necesidad de una suscripción en la cual meteremos código de varias maneras distintas para hacernos cargo de esa respuesta. Y esta comunicación con **RxJS** es esencial que la conozcas porque te hará falta para cualquier comunicación no trivial. Sí es cierto que está disponible también las **APIs Fetch** con **Async**, **Await**... para casos muy muy muy sencillos, pero la potencia real verás que la tendremos con estos **pipes** y sus operadores. Así que vamos a familiarizarnos con **RxJS**.

5.1.1 Lectura asíncrona de datos

Bien, pues aquí estamos con nuestra aplicación de reserva para actividades, donde aparecen las actividades, podemos entrar a ver cada una de ellas e intentar reservar, pero todos estos datos vienen de un fichero que tiene una constante aquí hardcoded con la información de estas actividades y esto tiene que dejar de ser así para ser un poco más realista y venir de un servidor con un **API REST**. Para ello lo que necesitamos es tener al menos un servidor, aunque sea de pruebas, en local. Yo he instalado, y te dejo en la documentación las instrucciones para que tú también lo hagas, el **json-server-auth**, incluso con su versión de autorización, aunque por ahora no lo usaremos, de forma que nos lance un servidor con unos datos ya más o menos preparados. Entonces para eso te he puesto aquí los scripts que vamos a utilizar. Concretamente vamos a lanzar el **api**

, que lo que hace es lanzarnos el **API** pero ya con unos datos, digamos, de semilla para que tengamos cosas.

Y una vez que eso esté en marcha, aquí en **localhost3000** es donde nos escuchará este servidor. No hay por qué venir a ver esta página web, esto solo vengo yo ahora para que tengamos esta idea, sino que llamaremos, por ejemplo, a **localhost3000/activities** y nos dará pues algo similar a lo que teníamos hardcoded, pero ahora vendrá de un servidor. Los datos están en una carpeta llamada **db**, fuera de la aplicación, porque esto no quiero que me contamine la aplicación el día que usemos un servidor de verdad, y simplemente pues nada, unos ficheros que indican qué colecciones tenemos, con qué permisos, que por ahora es todo abierto, y dejaremos de utilizar poco a poco el **activities.data**. Pues lo primero, para empezar a utilizar un servidor, es tener un cliente, y ese cliente **HTTP** necesitamos aprovisionarlo, será un servicio de **Angular** que utilizaremos, tal como hemos aprovisionado el **router**, ahora lo haremos con el **HttpClient**. Client que se me incorpora, ¿vale?, desde **@angular/common/http** y estará listo para empezar a usarse desde nuestras páginas.

Y la primera página donde lo vamos a usar será la página home, en la cual yo ahora mismo estoy mostrando el famoso array de actividades, que es esto, un array de

estos tipos, voy a ponerle aquí el tipo específico, porque ahora mismo lo que voy a hacer es eliminar esta referencia, este array, poner un array vacío, y como verás lo que pasará es que nuestra página ya no tiene actividades que mostrar, pero eso tiene fácil solución. Para solucionarlo voy a volver un momentito aquí al **app.config** y hacer con el **HttpClient** algo parecido a lo que hicimos con el **router**, ver que se pueden configurar cosas, entonces una de ellas que le vamos a poner es **withFetch**, esto tendrá importancia para poder utilizar la aplicación tanto del lado del servidor como del lado del cliente, dedicaremos tiempo a eso, pero ahora mismo para que sea congruente le digo **HTTP withFetch**. Y ya que hemos venido aquí, este aprovisionamiento, ¿cómo puedo utilizarlo? Bueno, en el caso del router, por ejemplo, habíamos utilizado el **routerLink**, etc., dentro de la template, pero ahora vamos a hacerlo dentro del código, porque vamos a necesitar esto como código. Al necesitar un aprovisionamiento como código, lo primero que haré es guardarlo en algún sitio, así que voy a crearme una propiedad aquí, decirle que es de tipo **HttpClient** y pedirle por favor a **Angular** que me lo inyecte. **Inject** es una función propia de **@angular/core** que es capaz de darte una instancia de algo previamente aprovisionado, o sea que aquí en la sección de providers decimos lo que tenemos y aquí ahora reclamamos esa inyección. Normalmente estos servicios sólo se deberían utilizar dentro de la propia clase, así que yo como buena práctica te recomiendo que los hagas privados. Yo lo hago con el **#**, podrías hacerlo con la palabra private, pero así es como reclamamos una instancia de cualquier servicio previamente aprovisionado.

Y teniendo ya el servicio, es hora de usarlo. ¿Hora? ¿Cuándo? ¿Cuándo vamos a usar ese servicio? Buena pregunta. En este caso sería lo antes posible para disponer de la lista de actividades, así que no hay nada que ocurra antes que la ejecución del constructor y yo dentro de este constructor voy a llamar a **this.httpClient** y voy a ver qué me ofrece. Lo que me ofrece aquí son los típicos métodos que podríamos asimilar como verbos del protocolo **HTTP**. Está el **put**, **post** y **get**. Vale, **get**, inicialmente el método que voy a utilizar para obtener cosas, lo único que me va a requerir es una url. La url será efectivamente el lugar donde están los datos, así que se la voy a poner aquí. No es obligatorio, pero una cosa muy recomendable es que estos métodos, en concreto para el **get** es el más utilizado, pero los demás también, pueden tipificarse de manera genérica. Tiene un tipificador genérico y aquí decirle que yo lo que espero que venga es un array de actividades. Así que muy recomendable que hagamos esto.

Una vez hecho esto, habrá que pensar, oye, este **get**, esta instrucción, parece que hace lo que tiene que hacer, que hace la petición. Parece, pero no es así, porque resulta que este **get** está implementado con una tecnología llamada **observable** y de todas las encarnaciones que tiene la **observable** pues lo han hecho con uno que ellos llaman **observable frío**, que es que sólo funciona cuando alguien está

mirando. ¿Qué significa mirar a un **observable**? Significa suscribirse a sus eventos y para eso tengo aquí la función **subscribe**. La función **subscribe**, vamos a ver que ahora en cuanto yo guarde, recompile y se reejecute esto, cómo la llamada **get** se ha producido y ha devuelto estos datos.

Muy bien, pues ahora que ya tenemos la llamada con los datos, lo que tenemos que hacer es hacer uso de ellos. La función **subscribe**, en su manera más sencilla también de programarla, admite una función en la cual su primer argumento serán los datos obtenidos como resultado, voy a llamarle por ahora **result**, y aquí lo que haríamos es hacer uso de ellos, pues por ejemplo **this.activities** es igual a **result**. Bueno, pues esto nos queda así, donde nos hemos suscrito al resultado, este resultado se guarda en la variable y la variable ahora se recompila y ya nos muestra, fijaos, más datos que antes y más realistas. Un par de cosas más, para profesionalizar esto un poquito, como mínimo vamos a hacer que el **API URL** esté en una propiedad, incluso más adelante vendrá de servicios de configuración, y otra es que este **HTTP Client** que tiene el prefijo del **#**, porque es privado, le voy a poner un sufijo, voy a cambiarle aquí el nombre, voy a poner aquí un sufijo que será un **$**, porque esto es el convenio que más se utiliza para significar que algo va a ser observable. Lo veremos en todos los ejemplos a partir de ahora.

5.1.2 Envío asíncrono de cambios

En esta sección, después de ver cómo obtenemos datos, vamos a ver cómo los modificamos. Por ahora teníamos en la página principal la lista de actividades que si me voy a ver una de ellas, por ejemplo, navegar por el Mediterráneo, a quien no le apetece, ¿verdad?, pues vamos a intentar hacer una reserva y para ello utilizaremos un nuevo endpoint llamado **bookings**, donde haremos las reservas, y también modificaremos la propia actividad porque pasará a cambiar de estado cuando le corresponde. Vale, para ello me voy a la página **bookings.page** y en ésta me he tomado ya la libertad de rellenar con estas tres propiedades que aún no he utilizado. Recordamos, obtener el **HTTP Client** y tener las urls mínimamente preparadas. También os quiero presentar a un nuevo tipo, que es el **booking.type**, donde almacenaremos a qué actividad nos hemos apuntado con cuánta gente.

Cuando vimos el **GET**, nos suscribimos de la manera más simple posible, que era una función que trabaja con el resultado, sin más. Con el **POST** podríamos hacer lo mismo, pero os voy a enseñar una manera un pelín más evolucionada de trabajar con las respuestas. Esto valdría también para el **GET**, que es en lugar de poner una función pondremos un objeto, y este objeto tendrá dos o hasta tres propiedades, normalmente dos. La primera se llamará **next** y lo que va a hacer es tener código para ejecutarse cuando llegue el siguiente dato de esta llamada. El siguiente y único dato de una llamada **HTTP**, porque las llamadas **HTTP** o

devuelven o no devuelven cosas. Y hablando de no devolver, porque podríamos tener cualquier error desde autenticación o que los datos no estén bien, esta es la segunda propiedad que vamos a poner aquí, que será la propiedad **error**, que también vamos a decir que tiene la misma pinta. Es decir, estos serán objetos con dos propiedades donde cada una de ellas son funciones. Claro que estas funciones son totalmente diferentes porque en una de ellas me llegará un resultado y en la otra me llegará un error. Lo que haga con cada uno pues dependerá de mis circunstancias. Por ejemplo, con un error por ahora podríamos simplemente pintarlo por consola. Y con el resultado, pues en este caso por ahora voy a pintarlo también por consola, aunque haremos más cosas después.

Vale, vamos a probar si esto funciona o no. Voy a poner aquí al menos un participante, le voy a decir que quiero reservar la plaza y me fijaré en las llamadas que aquí ocurran. Efectivamente tenemos una llamada a **bookings**, **localhost:3000/Bookings**. La respuesta ha sido que sí, que me lo han creado correctamente. Puedo ver que la payload enviada se corresponde bastante bien con lo que estábamos rellenando por aquí y veo también que la respuesta, que es el objeto de reserva perfectamente creado ya en el servidor. Estoy indicando aquí que hemos llegado bien por este **next**. Puedo ver que después de ese guardado aquí en el servidor efectivamente está esa reserva. Y lo siguiente que quiero hacer es algo funcional. Resulta que cuando tengo una actividad que está publicada y quiero hacer una reserva con suficiente cantidad de gente, llega un momento en el que se confirma. Así que no sólo quiero guardar la reserva, sino que la actividad cambie de estado y pase de estar meramente publicada a estar confirmada.

¿Cómo hago eso? Primero, desde el punto de vista de código y funcional, lo que he creado aquí es un método que llama, en lugar de a **post**, llama a **put**, porque lo que va a hacer no es crear algo nuevo en el servidor, sino cambiarlo. ¿Dónde vamos a cambiar? Vamos a cambiar una actividad. ¿Qué actividad? Hasta ahora estábamos haciendo llamadas **get** contra toda una colección o **post**, que también es contra toda una colección, pero las actualizaciones, lo que vendría a ser un **update** en **SQL**, se hace contra un registro y entonces la url tiene que ser más precisa. Lo que estoy combinando aquí es la **activity url** con la idea actual. Con lo cual, el **put** es muy similar al **post**, sólo que en su url lo que llevamos es el acceso directo a una actividad. Por supuesto, en el **body**, en la payload, si quieres, va exactamente lo equivalente. En este caso, la actividad ya actualizada y, a partir de ahí, la suscripción funciona exactamente igual. El siguiente tema interesante es ¿cuándo vamos a llamar a este método? A este método lo llamaremos una vez que hayamos guardado la reserva. Es decir, el cambio desde el anterior ejercicio es simplemente que, al guardar una reserva, actualizo el estado de la actividad y, para ello, **put** contra una url precisa enviando la actividad. Vamos a comprobarlo. Como veis, se ha producido el **post** a **bookings** con la reserva actual y, después,

se ha producido el **put** a la actividad número 4 y con la payload que me lleva el estado actualizado, que es confirmado. Si yo vengo aquí y me fijo antes de actualizar, veré que la actividad número 4 estaba como published pero, en cuanto recargue, aparece como confirmed y, por supuesto, la lista de reservas habrá ido creciendo.

Pues entonces ya sabemos obtener información, crear información y modificar información, todo de manera asincrónica, suscribiéndonos a **Observables**, que nos ofrece **HttpClient**.

5.2 Asincronismo y señales

Vamos a empezar ahora con uno de los temas más novedosos y también más complejos de **Angular** Moderno y es aquel que mezcla el tema de las señales con el mundo asíncrono, especialmente de **RxJS**.

Partiremos de esta primera división. Usaremos señales para todo aquello que tenga que cambiar en **HTML**, todo lo que tenga que ver con la detección del cambio. Y usaremos los **observables** para todo el mundo asíncrono. Obviamente las señales nos ofrecen mucha potencia. Nos ofrecen la posibilidad de tener la propia señal con el set para cambiar, pero también el computed para cualquier dato derivado y los efectos para generar nuevos cambios.

Atención porque al principio lo mostraremos como una posibilidad, pero se refactorizará con técnicas más complejas y vamos a procurar evitar usar efectos con **observables** internos. Es decir, que dentro de un efecto hagamos alguna llamada **HTTP**. Esto será algo que intentaremos evitar.

Por lo demás, si ya sabemos cómo hacer llamadas, cómo suscribirnos y cómo funcionan los cambios con las señales, pues esto es la mezcla perfecta. Señales para toda la gestión del cambio síncrono y **observables** para toda la obtención de respuestas asíncronas. Vamos a verlo.

5.2.1 Señales con los datos recibidos

En esta lección vamos a combinar dos temas que hemos aprendido: los **observables** asíncronos, a los cuales nos suscribimos para obtener sus datos, y por otro lado las señales, que podrían venir como input, se podrían crear manualmente, y que después podríamos hacer actualizaciones o cómputos sobre ellas o incluso efectos secundarios. Vamos a mezclar esos dos conceptos y ver cómo maridan.

Para ello voy a mostrarte algo muy curioso que nos ocurre aquí en la **home.page**. Obviamente esto al arrancar llama **Http get**, obtiene el listado de actividades y las pinta. Si yo visito cualquiera de ellas y vuelvo, oh, resulta que no se me repintan aquí las actividades. Pero puedo comprobar que la llamada sí se ha realizado, que la respuesta sí ha llegado, e incluso podríamos hacer aquí un **console.log** y comprobar que efectivamente si me voy y vuelvo, el resultado, el array, sí que ha vuelto, pero no se me ha repintado. ¿Por qué?

La respuesta a esa pregunta requiere comprender el concepto de detección de cambios en **Angular**. Por si no te lo habías preguntado, alguien en algún momento tiene que decidir si esta plantilla que produce esta respuesta debe ser repintada o sigue siendo válida. Y eso sólo te lo da preguntarte si los datos han cambiado o no. Claro, aquí surge el problema de cada cuánto tiempo pregunto si los datos han cambiado, y por otro lado, qué significa que hayan cambiado. Por ejemplo, un array puede haber cambiado una única propiedad, imagínate, el precio de esta actividad, o pueden que sean totalmente distintos. Bueno, detectar esos cambios es costoso.

Así que **Angular** nos ofrece dos posibilidades. La moderna recomendada, **onPush**, requiere por parte de los programadores que hagamos algo más, pero hace que las aplicaciones sean más ágiles, funcionen mejor, porque detectará en menos situaciones los cambios y lo hará de una manera mucho más liviana. El problema es que si tú no haces lo que debes, pues nos pasan estas cosas. Que es que vamos y volvemos y no se nos repinta. Las soluciones serían una vuelta al pasado, poner esto en modo default, y en este caso sí que podemos ir y volver y se repintan, porque digamos que el detector de cambios funciona más veces.

Pero esto es malo para el rendimiento. Volvamos, por tanto, a **onPush**. Dentro de esto hay dos soluciones. Una también clásica, que es utilizar un **pipe** llamado **async**, que notificaba los cambios de las llamadas observables, o bien aprovecharnos del conocimiento que tenemos de las señales. Para ello lo que voy a hacer es convertir este array en una señal y cambiar el código para que esto compile y sea utilizable. Así que lo primero que haré será que una señal no se actualiza así como así, sino que tenemos que invocar a su método **set**. Una vez hecho eso, también sé que las señales se invocan como si fuesen funciones, y esta es la clave. La plantilla estará atenta a que esta señal le indique justamente eso, una señal de cambio.

Así que ahora al volver, esto ya funciona perfectamente, porque los datos cuando llegan se guardan notificando que ha habido un cambio en esta señal. La plantilla de alguna manera escucha, se suscribe a los cambios de estado en esta señal, y cuando los detecta, se repinta.

5.2.2 Señales para enviar cambios

Vamos a ver ahora un caso un poquito más complejo, porque hasta ahora en la página **home** lo que ya teníamos claro desde hace tiempo era cuándo llamar al **observable Http** y era en el constructor, al visitar la página. Tanto da que venga directamente como que me recargue. Lo primero que hago es llamar a **Http** y cuando lleguen los datos activar la señal. Pero, ¿qué ocurre si queremos utilizar esa misma estrategia en el **bookings.page**? Es decir, aquí yo no puedo hacer una llamada hasta que no tenga el **slug** por el cual hacer la invocación.

Hasta ahora estábamos haciendo que la actividad fuese una señal computada en la cual íbamos a la raíz de **activities** si lo buscábamos, etcétera. O ahora yo aquí lo tengo **hardcoded** para que obtenga siempre la misma. Bueno, esto **hardcoded** entonces deberíamos cambiarlo por algo que efectivamente se ejecute con **Http get** cuando el **slug** obtenido de la URL cambie. Podríamos seguir con la estrategia de mantener que **activity** sea una señal derivada de **slug**. Y aunque eso es posible, te voy a enseñar una solución que es menos farragosa para empezar y que no requiere de ninguna librería externa para que esto funcione. Y es aprovecharnos de los efectos.

Recuerdo que un efecto no es más que un código que se ejecuta cuando alguna de las señales que participan en él cambia. Así que yo podría plantearme aquí un efecto que haga una llamada **HTTP** cuando **slug** cambie y que como resultado rellene **activity**. Para ello voy a hacer esto por partes. Para empezar, convertiré a **activity** en una señal en sí misma, no en una señal derivada. Después lo que haré será crear aquí un efecto. La estructura de los efectos es siempre la misma, paréntesis implica llaves. Y ahora el contenido te lo voy a desmenuzar un poco. Lo primero, vamos a centrarnos en dos cosas fundamentales. Una, hacemos una llamada contra una url que depende del **slug**.

Esta url es un texto que combina la parte básica de las url de actividades y después una query contra un **API**. Lo segundo es ver que este **slug** es una señal y por tanto este efecto se ejecutará cada vez que **slug** cambie. Es decir, cuando recibamos la url entrará por aquí, se quedará con la parte del parámetro que corresponda y eso activará este efecto. Lo siguiente será hacer una llamada **Http** que hará un **get** con una raíz de actividades contra esta url. Y a partir de aquí el trabajo ya es más o menos el de siempre con **observables**. Sería suscribirnos y trabajar con el resultado.

Un tema interesante es que **get**, cuando se le invoca de esta manera, devuelve un array y yo me quedaré con su primer valor. Obviamente podríamos proteger esta situación. Por ejemplo, con la **NULL_ACTIVITY** que nos evita trabajar con nulos de verdad y así tenemos el código un poquito más limpio. Limpio estará, pero aquí abajo yo tengo algo en la consola que me gustaría mostrarte. Dice aquí la consola,

recuerda que no puedes escribir en las señales desde las funciones computadas o los efectos, que es justamente lo que estamos haciendo aquí con este efecto. Bueno, pues lo mismo que habíamos hecho antes lo volvemos a hacer ahora, que es poner un segundo argumento aquí a los efectos para que me permita ahora sí cambiar, escribir en esta señal cuando esta otra señal cambia.

Recuerda, esta señal produce la ejecución de este efecto que como efecto secundario cambia esta otra señal. Y por tanto, ahora ya tenemos la posibilidad de hacer preguntas y mostrar la actividad adecuada. Vemos que en el **network** se nos combina perfectamente el **slug** que estamos buscando.

5.3 Operadores RxJS

Vamos a ver ahora un par de conceptos avanzados de **RxJS** y para ello voy a tratar de hacértelo llevadero, volviendo con el símil de una fuente que emite un chorro, supongamos que de agua, que llega hasta el océano donde hay un suscriptor. Este suscriptor consume el contenido del agua de esa fuente, ya no se puede reutilizar más, pero por el camino podemos meter tuberías.

En estas tuberías las rellenaremos con operadores. Estos operadores los habrá de dos tipos. Algunos van a mutar el contenido y otros no, otros generarán efectos. Fíjate que estos conceptos de mutación y efecto aparecen también durante las señales con los **computed** y los **effects**, porque esta es la base de la programación reactiva. Entre los operadores los que te vas a encontrar que mutan, por ejemplo, estará el **map** que transforma algo una cosa en otra.

Entre los efectos, el más conocido, estará el **tap** que hace algo con el **X** sin transformar nada. Esto que parece un dibujo para niños pequeños del ciclo del agua se complica bastante cuando lo que entra por esta tubería es a su vez otra fuente y entonces la canalización es una mezcla de ambas. Esto lo veremos con los operadores de primer nivel, entre ellos aquí se verá el **switchMap**, aunque por ahora sólo a modo introductorio. Habrá una lección dedicada a este tipo de operadores. El otro problema que puede aparecer es un error, que algo, y así debería ser verlo, nos reviente la tubería de manera que el flujo se interrumpe. Ese operador, llamado **catchError**, se parece a los demás en cuanto a que recibe una entrada, en este caso será un error, y produce una salida. Pero ojo, porque esta salida de alguna manera tiene que volver a ser una fuente, porque recuerda que cuando se produce un error, una excepción, el flujo se interrumpe.

Esto es lo raro y novedoso. El **catchError** requiere ser rehidratado para que continúe o para que siga generando algo. Si lo que quieres hacer es simplemente un log de ese **catchError** o un efecto secundario, no tiene por qué devolver nada

aquí. Otro tema a tratar será la interoperabilidad de las señales con los observables, y para eso **Angular** nos ofrece un par de funciones de interoperación. Esencialmente permitirá llevar una señal al mundo de los observables, es decir, poder suscribirnos a cambios provocados por señales, o bien hacer que flujos observables se conviertan en señales. Un patrón muy habitual que utilizaremos aquí será, por ejemplo, que, dado una url que tenga un parámetro, esto implícitamente es una señal, que cuando cambia no requiere, por ejemplo, hacer una llamada **Http**, pero vamos a intentar evitar a toda costa hacer **Http** como un efecto de una señal. Vamos a intentar evitar esto y para ello tenemos que primero transformar la señal en un observable, pasarlo por una tubería y de esa tubería obtener un nuevo resultado, que si lo queremos presentar tiene que volver a ser una señal. Bueno, esto no es la cosa más sencilla del mundo, pero es siempre igual. Haremos **signal to observable**, tuberías para transformar lo que necesitemos, y el resultado que es un observable, **to signal**. Vamos a ver entonces la interoperación entre signals and observables.

5.3.1 Tuberías funcionales

Bien, pues en esta última lección del tema dedicado al asincronismo, que hemos hablado bastante de señales y de **observables**, vamos a aprender un par de trucos más. El primero se trata de aplicarle al **observable** algo que consideraremos una **tubería**. ¿Por qué? ¿De dónde viene este nombre y para qué va a servir?

Pues verás. Tienes que pensar que los **observables** de un solo uso, como estos con **HTTP**, son la excepción y no la regla, sino que en principio los **observables** emiten grandes cantidades de datos unos después de otros. De ahí nos viene el concepto de **next**, el siguiente, que pase el siguiente, que pase el siguiente.

Bueno, podríamos pensar que las funciones que generan **observables**, como en este caso el **get**, son una especie de fuente, un surtidor, de un **stream**, ¿vale?, que podríamos traducir como arroyo, riachuelo o conjunto de datos que va uno detrás de otro, y que el **subscribe** sería el sumidero o el último lugar donde es accesible ese arroyo de datos, ese **stream**.

Y por el medio entonces sí que podríamos poner algo más, **pipes**. ¿Con qué sentido? Bueno, en principio el sentido sería que dentro de la **tubería** metamos todo aquel código de transformación, filtro y demás que estamos poniendo en las suscripciones, de forma que a la suscripción le llegue ya el dato limpio y listo para ser usado. Por ejemplo, en este caso podríamos querer quedarnos sólo con el primer elemento del array y en caso de que no viniese ninguno, porque no encontrase nada, devolver pues esto, una actividad nula.

¿Cómo se puede hacer eso? Pues se puede hacer mediante un operador. Dentro del **pipe** puedo meter múltiples operadores y estos operadores vienen predefinidos en **RxJS** y eso, el más conocido es el **map**. El **map**, con la intención de transformar algo, es de nuevo una función. En principio, el **map** recibe, está en la **tubería**, está en el arroyo medio del río, y recibe pues el chorro de información. En este caso, el array de actividades. ¿Y qué hace con él?

Bueno, pues puede retornarlo, si es en una sola instrucción, que suele ser así, pues no necesito llaves{} ni punto y coma;, pues podríamos retornar directamente **activities[0]**, si es que existe, y si no, **NULL_ACTIVITY**. Este operador **map** que recibe el array ya no devuelve un array, sino uno solo, por eso aquí se me está quejando. Fíjate, voy a ponerle los tipos ahora mismo a esto, y aquí además cambiaré el nombre, porque lo que me llega aquí es efectivamente ya sólo una actividad, no un array, aquella que corresponde con este **slug**, y ese dato ya lo puedo usar aquí directamente.

Entonces, en principio lo que estamos haciendo aquí sería sustituir el lugar donde llevarme la lógica, mejor dicho, que estaba en el **subscribe**, me la llevo al **map**. Y esto que he hecho con el **map** podría hacerlo con más operadores. Los operadores se separan con comas y se ponen uno detrás de otro. Voy a poner, por ejemplo, uno que capture errores y resuelva para esa situación. Para ello voy a agregar el operador **catchError**. Este es un operador que recibe también, como todos, una función y retorna algo, lo que quieras. Entonces ya tenemos aquí el error capturado. Pero una cosa muy interesante es que, ante un error, podríamos querer actuar de muchas maneras.

Por ejemplo, hacer de manera que el error no parezca tal. Por ejemplo, yo puedo devolver aquí una **NULL_ACTIVITY**. Como decía, podría estar tentado a hacer algo como esto, pero se produce un error de compilación que lo que me está diciendo es que el tipo actividad que es esto no es equivalente a un tipo **observable** de no sé qué. Pero bueno, ¿qué me estás contando? Si en el **map** yo recibía un array, me quedaba con un objeto y a eso no le ponías problema y ahora resulta que aquí, en el **catchError**, retorno también, pues en este caso la actividad nula, pero una actividad, a fin de cuentas, y a esto sí le pones problema. ¿Por qué? Pues la cuestión es que cuando se produce un error es como si el flujo se hubiese interrumpido completamente, de forma que este **catchError** entra por una vía distinta y este return no re-hidrata el **observable**. Para re-hidratar el **observable** tendríamos que utilizar aquí una función fuente. Vaya, cuántas cosas nuevas, ¿no?

Una función fuente. Bueno, pues las funciones fuente las hay de distintos tipos, pero la más sencilla de todas es la función **of**. La función **of**, sí, con este nombre así tan sencillo, **of**... Por cierto, todas estas cosas vienen... **RxJS**, aquí está, **catchError**, **map**, **of**, ¿vale? Como decía, la función **of** lo que hace es convertirse

de nuevo, o re-hidratar o regenerar una fuente **observable**, de forma que esto ahora funcione de esta manera. **Http**, obtén por favor, con esta **URL**, un array de actividades y el resultado me lo canalizas por aquí. Si el resultado ha ido bien, bien significa que no ha fallado, es decir, algo ha venido. Pues escoge el primer elemento del array.

Si no hay primer elemento del array, devuelves directamente una actividad nula, pero un objeto actividad. Muy bien. ¿Qué es que ha habido un error? Vale, pues si ha habido un error, entonces tengo aquí un capturador y vas a re-hidratarlo mediante la función **of**, de manera que los que vienen por detrás ni se enteran de este error. Esto es como haber hecho un catch que continúa la ejecución.

Bueno, pues esto es un aprendizaje que nos servirá para hacer muchas más cosas, pero sobre todo para entender cuál es la potencia real real de los **observables**, que es la capacidad de canalizar en ellos múltiples operadores. Estos son de los más básicos posibles. El **map** que transforma lo que entra en otra cosa y el **catchError** que es capaz de re-hidratar algo que se había roto para que, en principio, la ejecución continúe.

5.3.2 Interoperabilidad básica de señales y observables

Bien, así que hemos visto que tenemos **observables** a los cuales nos podemos suscribir, podemos setear esos datos en señales **Writable**, del tipo adecuado, y a partir de aquí vamos a ver qué puntos de mejora puede haber sobre potenciales problemas que a lo mejor no son tan evidentes.

El primero es que esta señal, al ser **Writable**, cualquiera, fuera del uso del que estaba más o menos previsto, que es cuando lleguen los datos, setearlos, pues podría hacer un setting en cualquier momento. Por otro lado, cada vez que hacemos una suscripción estamos consumiendo unos recursos de memoria que habría que liberar cuando perdiese ya su interés. Es decir, habría que guardar esta suscripción, que se puede hacer, y después desuscribirse. El problema no es el cómo, sino el cuándo.

Y hacer este tipo de cosas siempre es un poquito engorroso, no siempre te acuerdas, etc. Con lo cual, tenemos señales que pueden modificar fuera de su uso pretendido. Segundo, el problema de las suscripciones que tendríamos que eliminar a posteriori.

La pregunta es, ¿habrá alguna manera de arreglar estos dos problemas y juntar **observables** y señales? Y la respuesta viene de la mano de **Interop**.

Interop es una librería propia de **Angular**, voy a agregarla, antes de nada, que tiene dos funciones, **toSignal** y **toObservable**, ahora solo voy a usar **toSignal**. Y lo

que me va a permitir es crear una señal a partir de un **observable**. De manera que en lugar de crear la señal con su función constructora normal, lo crearé con **toSignal**.

Y como primer argumento de entrada, le daré la parte **observable**, no el subscribe, la parte **observable** que provee los datos. Y cómo es posible que esta señal, mientras no se produzca la llamada, tenga un valor indefinido, tengo que agregarle un valor inicial que no es más que poner en un objeto, en un segundo argumento, el **initial value** a cero.

A partir de ahí, la función **toSignal** estaría bien construida, esto ya no sería necesario, y mi siguiente problema, que es una solución en el fondo, es que **activities** ya no es una **writable signal**, sino una signal normal y corriente, y esto tiene la gran ventaja de que no puede ser reasignado su valor más allá de esta declaración inicial, con lo cual lo tengo perfecto.

Tengo el **toSignal** que se rellena a partir de un **observable** y solo de ese **observable**, y la segunda parte es que este **toSignal** es tan inteligente que es capaz de gestionar la suscripción y de suscribirse él solo, con lo cual es otra ventaja añadida que ahora mismo no somos muy conscientes, pero que está ahí.

Por último decirte que, claro, al eliminar la posibilidad de hacer el subscribe aquí, quizá los datos según vengan no te valgan, pero como ya sabemos que tenemos la opción de agregar **pipes**, por ejemplo con un **catch error**, que me permitiese que cuando llegue un error re-hidratar esto con un array vacío, podríamos hacer esta llamada **get**, capturando el error, ir retornando, re-hidratando el **observable** a partir de algo vacío, o podríamos haber hecho un **map** si hubiese que transformar cualquier cosa previamente.

Es decir, siempre tendríamos la oportunidad de transformar en la canalización el flujo de un **observable**, y al mismo tiempo canalizar su resultado, haciendo el set automático contra las actividades, teniendo señales no reescribibles.

De esta manera conseguimos lo mejor de ambos mundos, que los **observables** hagan el trabajo asíncrono de obtener los datos, que los **pipes** transformen lo que necesiten o resuelvan los problemas cuando se produzcan, y que **toSignal** se dedique a desuscribirse para que no consuma más allá de lo necesario, y aportar señales de solo lectura para que solo él pueda modificarlas.

toSignal, recuerdo, viene de **rxjs-interop**, pero **@angular/Core**.

5.3.3 Usos avanzados de rxjs-interop

Vamos ahora a un caso más complejo. Tenemos una actividad **writable** que la estamos seteando en el **subscribe** de un **observable**. Hasta ahí vemos que esto funciona, es más o menos sencillo, pero presenta los problemas de que la suscripción se queda abierta. Esto a la larga podría ser un problema. Segundo, el **activity** es **writable**, con lo cual me lo pueden cambiar en otros sitios que quizá no era lo que se pretendía. Y tercero, por si fuera poco, el problema es que para hacer la llamada **Http** necesito otra señal de entrada para montar la URL.

Este es mi gran problema porque el identificador de la actividad que quiero obtener me lo darán en algún momento. Sí, pronto, inicial, etcétera, pero no sé cuándo y además puede cambiar. Por lo tanto, necesito una señal como punto de entrada de setear otra. Esto lo he resuelto con un efecto secundario, un efecto que además he tenido que ponerle que permita la escritura de señales. Esto no es la manera más limpia y óptima de hacer las cosas. Para resolver el problema voy a pasar de esta señal de entrada a esta señal de salida y por el medio utilizaré un par de **observables**. Como tengo que transformar interoperar de unos para otros, voy a necesitar aquí de nuevo la librería **Interop** de **Angular** con **RxJS** y con dos funciones, no sólo **toSignal** sino **toObservable**. Lo primero, esta señal que yo tengo aquí, voy a transformarla en un **observable** usando la función **toObservable**. Esto me debería devolver un **observable** de **string**. Lo quiero para acabar transformándolo en un **Observable** de **Actividad**.

El **Observable** de **Actividad**, que ahora voy a declararlo así con un **of(NULL_ACTIVITY)**. Esto siempre genera un **observable** del tipo del dato que le pases ahí. Es como un string de un solo disparo inmediato y acabaré utilizando este **observable** de actividad no como en el **signal** de este caso sino como un **toSignal** donde la fuente efectivamente es este **observable** y el valor inicial mientras es **observable** no produzca nada será igualmente la actividad nula. De esta forma ya no tengo una **writable signal** sino simplemente una **signal** que en el fondo es lo que quiero y todo mi problema está en este intermedio como paso de este **observable** a este otro.

Obviamente no será tan sencillo como crear un **of** sino una transformación sobre el **observable** inicial. Las transformaciones vendrán siempre dentro de algún tipo de **pipe** sólo que ahora no utilizaré **map** sino **switchMap**. **SwitchMap** es uno de los operadores conocidos como de "orden superior". Eso ya suena feo, orden superior, pero lo que hace realmente **switchMap** es recibir el dato de un **observable** y generar otro. No recibir un dato y generar otro dato sino recibir un dato y generar un **observable**.

El dato que recibe obviamente es el **slug** que estemos tratando ahora mismo. Esto, recuerda que vendrá en el futuro, pero no sé cuándo y puede cambiar, pero

no sé cuándo. Bueno pues eso se convierte en un **stream** de cosas que pueden ir pasando y ahora con este **slug** tengo que hacer lo que estaba haciendo antes aquí en el efecto. Obviamente cambiando la llamada a la señal por simplemente este parámetro y después voy a hacer por ahora solamente el **this.get** para que esto vaya tomando un poquito de forma. Aún no está perfecto porque me dirá aquí que estamos retornando un **observable** de actividades y esto esperamos un **observable** de una sola actividad. Muy fácil, simplemente tengo que efectivamente utilizar los mismos operadores, la misma tubería sobre la llamada **Http**.

Esta tubería no es la misma que esta. Esta tubería va sobre el **observable** que tiene los datos de entrada, esta tubería va sobre las respuestas que vienen del **Http** y de cada respuesta se queda con el primer elemento si es que lo encuentra o nada, si no encontró nada, incluso también supera los errores de esta forma. A partir de este momento yo ya no necesito este efecto que efectivamente ya estaba provocando aquí un problema porque no se puede setear pero esto a la larga es buena cosa porque procuraré usar los efectos secundarios lo menos posible y desde luego que no tengan que cambiar señales. Ahora que ya sé que tengo **toObservable** y **toSignal**, puedo hacer todo este tipo de jugadas que si bien puede resultar intimidante la primera vez que se ve, también te digo que esto es una receta que utilizarás con mucha frecuencia y que si aquí desapareciese el **console.error** esto podría quedar bastante más sencillo y legible y menos intimidante.

Interoperabilidad entre **observables** para el tratamiento asíncrono y señales para el renderizado de plantillas cuando los datos cambian de la manera que sea más eficaz posible y con menos consumo de recursos eliminando las suscripciones cuando no sean necesarias.