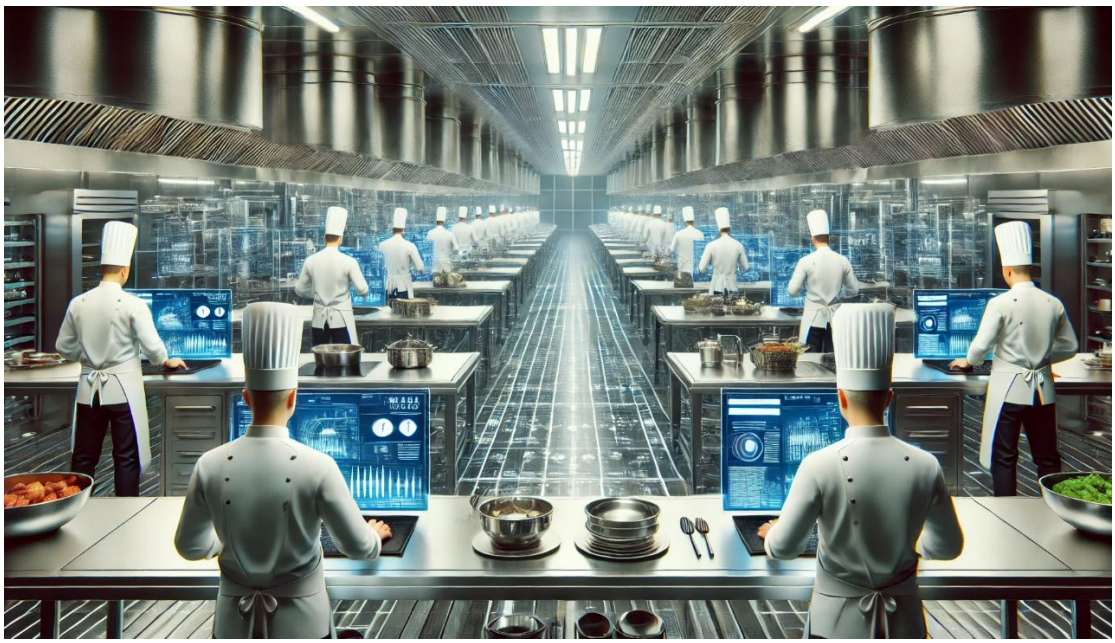
7. Programación reactiva

Empezamos el tema reactivo con un **almacén global** que permite comunicar **componentes** que no se conocían, que no eran familiares, que no estaban dentro de un **container**. Ahora, cuando uno cambia, el otro se enterará a través de una señal.

Otra manera de transmitir el estado desde un **componente** hasta otro será a través de la **URL**. Utilizamos el **router** con un **store** que será reactivo porque podremos escucharlo clásicamente con **observables** o más modernamente con señales. La **programación reactiva** tarde o temprano te proveerá de eventos que generan otros eventos, que a su vez generan otros eventos, y necesitarás algo para aplanar ese maremagnum. Los **operadores de primer orden de RxJS** están ahí para ayudarte.



7.1 Un almacén global basado en Signals

La programación reactiva viene a resolver varios problemas, entre ellos uno que en principio el patrón **container-presenter** no resuelve. A pesar de tener toda la buena intención de enviar o recibir a través de modelos **input-output** o puramente **models**, esto solo funciona si los **componentes** son padre-hijo, es decir, si hemos seguido el patrón **container-presenter** que no siempre es posible.

Es muy habitual que en una aplicación realista tengamos, por ejemplo, un **componente** que está en el **header**, típico carrito de la compra, y otro **componente** en el cual tenemos el botón de comprar, los elementos que sean. Obviamente, estos **componentes** no tienen por qué ser padres e hijos, entonces, ¿cómo hacemos que los cambios que genera este **componente** se reflejen en este otro? Bien, pues lo vamos a hacer a través de un **servicio común** basado en señales.

Realmente, vamos a suponer que tenemos un **componente uno** y un **componente dos**, pueden estar enrutados o no, lo que sí que serán, será inteligentes en el sentido de que, al depender de lo que yo voy a crear aquí, un **servicio** que siempre apellidaré de tipo **Store**. Esto es para seguir un poco la nomenclatura de patrones más evolucionados como **Redux**, en el cual tendré una propiedad **state**, normalmente la haré privada, para que los cambios que cualquiera de ellos hagan sobre ella, sea a través de métodos controlados, algún **setter**, y para obtener los datos, algún **getter**. Claro que este **getter** puede ser perfectamente una señal.

De esta forma, un **componente** puede provocar un cambio en este **Store**, y el otro puede recibir esas notificaciones. Este patrón, con este almacén más o menos globalizado, si lo haces absolutamente global, está disponible para todo el árbol de carpetas, si simplemente se lo agregas a una de las ramas, estaría disponible para los **componentes** de esa rama, pero sí que es bastante común que esto sea basado en el **root**.

Una cosa que conviene es familiarizarse con esta nomenclatura de llamarle **Store** a una clase que va a almacenar internamente, en una propiedad privada, el **State** como fuente única de la verdad. A este **Store** le llegarán, por una parte, modificaciones vía acciones que se despachan, así se habla de despachar acciones, donde aquí con algún tipo de método habrá algo de **SetUpdate** contra la señal del estado. Obviamente también publicaremos propiedades computadas que, desde fuera, serán vistas o serán seleccionadas. Por eso se habla de visiones seleccionadas, pero se calcularán mediante propiedades computadas. Además, puede haber efectos que generen cambios en otros sistemas o en otros estados. Bueno, esencialmente esto es una simplificación del patrón **Redux** aplicado

simplemente con señales y con las funciones básicas que vienen con ellas. **Efectos**, **Computed**, **Set** y **Update**.

7.1.1 Crear un store basado en signals

Bien, vamos a empezar este tema de la **programación reactiva** creando una carpeta **State** dentro de la carpeta **Shared**, porque esto va a ser compartido, y en ella guardaré los **stores** que voy a crear a partir de ahora. Esta carpeta también le agrego un **path** para que me sea cómodo de reutilizar. Y el comando que voy a utilizar será simplemente **ng g service** ¿En dónde? En **Shared State**. Y le llamaré **favorites-store** porque será el almacén para guardar los favoritos que hemos seleccionado previamente de nuestras actividades.

El nombre ahora mismo me queda un poquito redundante y yo suelo hacer una cosa que es eliminarle este último sufijo **service**, dejarlo exclusivamente como **favorites.store**, y ya puestos también renombro aquí para que tenga una naturaleza específica, que es el convertirse en un **store** y no algo tan genérico como un **servicio**. De paso esto también, tal como tengo configurado mis iconos, le cambia el icono y lo puedo ver más fácilmente.

Bueno, ¿y qué es un almacén? Un almacén en cualquier tecnología que se haga, cualquier tecnología reactiva, va a componerse de algo privado que será el estado, y en este caso para mí el estado va a ser una **writable signal**, donde voy a guardar los favoritos, en este caso fuertemente tipado como un array de strings, y el valor inicial que le daré será el ser una señal con un array vacío. A partir de ahí debería disponer de dos maneras públicas de leer y escribir.

Voy a utilizar nombres muy genéricos, pero lo que estoy haciendo aquí es hacer pública esta variable retornando valor, pero como algo de sólo lectura. De esta manera cualquiera se podrá enterar de sus cambios, pero no podrá modificarla. Si quiere hacerlo la protejo a través de un método **set** que de vuelta utiliza la variable privada. Bien se podría haber hecho público y nos evitábamos estos dos trabajos, pero esta manera de proceder es muy común en **programación reactiva** y me permite hacer validaciones, chequeos, clonados y demás durante el seteo, además de fomentar que el uso del almacén sea de una forma controlada. Incluso se podrían crear métodos más específicos tipo **addFavorite** o **removeFavorite**, que no va a ser el caso, no los voy a hacer ahora, pero serían cosas como estas.

7.1.2 Uso desde páginas o componentes inteligentes

Vale, pues ahora que tengo el almacén voy a empezar a utilizarlo, a consumirlo si quieres, desde los sitios donde me interesa. Por ejemplo, en la **home.page** tenía aquí algo que cuando un usuario marcaba un favorito, pues simplemente hacíamos un **console.log**. Bueno, pues es hora de hacer algo más inteligente. Si solicito aquí tener mi **FavoritesStore** podría, desde aquí directamente, invocar al **this.#favoritesStore** para que modifique este estado. ¿Con la intención de qué? Pues consumirlo desde el **header.component**. Aquí igualmente podría tener el **favoritesStore**.

Aquí donde tengo un cero empezaré a poner algo un poquito más inteligente. Entonces ahora podría crear aquí una propiedad computada a partir de esta señal que tiene el array, que me dé la cuenta. ¿Cuántos tengo? Un número así, pero va a ser algo **computed**. Muy bien. Que tendría, que tendría acceso al estado y cuando este cambie calcular su tamaño. Esto lo ponemos aquí, obviamente invocándolo como una función, porque esto es una señal. Y veremos cómo a partir de ahora, al marcar algo, la cuenta de favoritos aumenta tanto a nivel local, como ya lo teníamos, como a nivel global.

Un **refactoring** sencillo para esto sería llevarnos este **favCount** directamente a donde se calcula, sustituyendo obviamente, y a partir de este momento ya sólo tengo que seleccionar así. He, digamos que, ocultado el problema ahí. Bueno, una vez hecho esto, deciros que este **header.component** se comporta ahora como algo inteligente, porque es capaz de obtener sus propios datos. No lo recibe vía **input** de ningún padre y por tanto lo considero **smart**. Haré un **refactoring** que consiste en renombrarlo como algo distinto a un **component**, a falta de un nombre mejor, yo le llamo **widget**, y para que esto se refleje tanto en el icono, como aquí en los ficheros y sea congruente, pues también le cambio este nombre a **header widget**. Normalmente, si lo hubiera generado ya con esa predisposición a ser un **widget**, ya hubiese puesto **ng g c core/header --type=widget**. Pero bueno, ahora lo que tenemos son **componentes** no relacionados, una página y un **widget** nada relacionados entre sí, excepto porque comparten puntero a un almacén y de esta manera se comunican los cambios de estado.

7.1.3 Persistencia y reutilización

He agregado una carpeta más llamada **services** dentro de **shared** y la he añadido al **tsconfig** para poder reutilizarla, porque mi intención a partir de ahora es crear más **servicios** de utilidad según los necesite. Voy a crear uno que me va a indicar si estoy trabajando en el servidor o en el navegador. Este es un tema que volveré sobre él más adelante, pero que esencialmente dice que desde que hemos

establecido que **Angular** se puede ejecutar localmente, como ha sido siempre, pero también desde el lado del servidor para fomentar mediante el **server-side rendering** temas como el **SEO** y la velocidad de la carga inicial, esa carga inicial se ejecuta en el servidor. Y habrá cosas que no tengan sentido, como algo que voy a hacer después, que será darle la posibilidad de guardar en el **local storage**, cosa que en el servidor no existe, eso solo existe en el navegador.

Para resolver ese problema, el código, por arcano y raro que te parezca, es súper sencillo. Esencialmente lo que hacemos es pedirle a **Angular** que nos inyecte algo que no será una clase, sino un valor, viene marcado en mayúsculas y viene de **angularCore**, y esto es un valor que, una vez que se le pase a una función también propia de **Angular**, nos retornará una respuesta para indicarnos si estamos en el servidor o estamos en el browser. Como esto es bastante rebuscado, yo lo suelo guardar directamente en un **servicio** llamado **platform**, donde por ahora, como mínimo, facilito al resto del código encontrar respuesta a esta pregunta. ¿Estoy en el servidor o estoy ya, como normalmente así será, en el browser?

Bien, y una vez que tengo ese **servicio**, voy a crear otro, en la misma carpeta, que le llamaré **local-repository**. Lo de **service**, ya sabéis, es lo que siempre le agrega **Angular**, pero no siempre me tiene tanto sentido. A los **repositorios** suelo cambiarles el sufijo del fichero y dejarlos así, **local.repository**, porque así también le cambio el icono y entiendo que aquí se va a tratar de leer y escribir. El código no es nada específico de **Angular**, así que lo pego y te lo explico. La idea es valernos del **PlatformService** para que, si estamos en el lado del servidor, no hacer nada, pero si no es así, tengo un tercer método de ayuda para trabajar con el **localStorage**, serializando elementos y guardándolos en una clave, recuperándolos y parseándolos ya definitivamente. La idea es usar este **LocalRepository** ahora en el **FavoritesStore** para guardarlos ahí y que me sirvan de una ejecución para otra. Para ello, obviamente, tengo que solicitar que me inyecten una instancia de este **LocalRepository** y después, únicamente en su constructor, haré uso de él. Ahora te explico.

De dos maneras. Están las dos en el constructor, pero no suceden al mismo tiempo, ni siquiera de manera puramente secuencial. La primera línea lo que dice es que, usando el **LocalRepository**, obtengamos los favoritos, inicialmente como un array vacío, y lo que de ahí venga se use como estado inicial de este estado local. El segundo lo que dice es que cuando el estado cambie, por lo que sea, se guarde en favoritos. A partir de ahora, al ejecutar la aplicación, deberíamos irnos a la pestaña **Application**, la voy a ampliar un poquito. **Localhost** por ahora está vacío, pero en cuanto yo empiece a marcar cosas, veis cómo van creciendo los valores, se están guardando directamente en el **localhost** y aquí lo que me gustaría es que ahora pudiera ir a una página cualquiera y volver, o incluso tener

una página de favoritos y que esto se remarcase, cosa que no está haciendo. Para ello me falta ir a donde se presentan los favoritos, que es en la página **home**, **activity.component**, y agregarle esta marca de **Checked**. Esto lo que va a hacer es que, si ya estaban checados, los obtenga. Para ello, en la **home.page**, cuando empezábamos diciendo que mis favoritos era un array vacío, ahora diremos que mis favoritos es lo que diga el estado inicialmente.

Volvemos a la aplicación. Vemos que si navego y vuelvo, las marcas de favoritos se mantienen. Recargo la aplicación. Siempre aparecen porque su valor se obtiene del **local storage**. Yo me voy a una página, vuelvo y ahí están. Y ya puestos, podría reutilizarlo en una página especial solo para ver la lista de favoritos. Para ello, simplemente genero una nueva ruta. En cuanto tenga la página, la agrego a la tabla de rutas. Página está aquí. Vamos a hacer que esto sea la exportación por defecto para que se me facilite el agregarla aquí como un camino con un **loadComponent** sencillo. Y de nuevo, aquí no tengo nada más que volver a solicitar el puntero al almacén de favoritos, disponer del propio estado, es decir, de la lista de favoritos, obtenida a partir de ese **Store**. **Store** es el almacén, **State** es el contenido, lo que vale. Y aprovechando la nueva sintaxis con estructuras repetitivas integradas de **Angular**, pues aquí tengo un recorrido sobre el array de favoritos. Ahora sólo me queda hacer que naveguemos a esa página desde el **header**.

Si todo va bien, me llevará a ella y me lista los favoritos. Esto, aunque cerrase el navegador y lo volviese a abrir en otra pestaña, o incluso directamente esta **URL**. Porque el almacén es local, es reactivo. Si estas cosas cambian, todo cambia y tenemos un almacén global basado en señales. Y yo sinceramente creo que es muy muy sencillo.

7.2 Usando el router como almacén

Otra manera de comunicar reactivamente dos **componentes** que no sean padre e hijo sería utilizar la **URL**, sí, el **router**. Porque un **componente** puede escribir y el otro puede leer, siempre que esta lectura sea bien mediante **observables** o **signals**. Y para escribir, bueno, hay una posibilidad de navegar sin moverse del sitio pasando simplemente los **query params**.

Esta manera de comunicar es muy útil porque, de alguna forma, podemos señalizar en nuestros parámetros el estado de la aplicación. Por ejemplo, para filtros que se puedan compartir con otros usuarios o recuperar esa sesión más adelante. En cualquier caso, merece la pena que dediquemos un momentito a ver cómo comunicar **componentes** a través de su **URL**.

7.2.1 Un widget con señales

Vamos a continuar haciendo **programación reactiva** añadiendo una funcionalidad que le falta ahora mismo aquí a esta aplicación que sería tener un buscador, algo que me permitiese filtrar las actividades buscando por cualquier texto y que además me las presentase ordenadas por algún criterio como podría ser el nombre, la fecha o el precio. Para ello empiezo por definir lo que será el tipo **filtro** en donde guardaremos lo que se busca, el campo de ordenación y si la ordenación será ascendente o descendente.

Lo siguiente será crear un **componente** que voy a utilizar como filtro, pero como estamos en una lección sobre **programación reactiva** voy a considerarlo un **componente inteligente** por tanto le cambio el tipo a algo distinto del simple **component**. Le llamo **widget** porque tampoco quiero confundirlo con una **page** porque no será enrutado pero es desde luego algo distinto. Aprovechando la potencia de las **señales** voy a crear los tres campos que necesito marcándolos como **señales writable** dándoles como valor por defecto lo que hemos determinado aquí es decir vacío por id y ascendente y todo eso lo voy a enlazar con un formulario del que te pego aquí el **html**. Es un formulario en principio normal y corriente sólo que con un **ngModel** en un **banana in a box** asociado cada uno de ellos a una señal. Para que esto acabe funcionando te recuerdo que tenemos importar el **FormsModule**. Esta manera sencilla de hacer formularios enlaza desde la vista al controlador y de vuelta los datos con los elementos.

Aprovechando que estamos en el mundo de las señales podemos crear una **señal derivada** de sólo lectura de tipo **filter** que me permita en el futuro tener un efecto para hacer algo útil con ella. Por ahora simplemente un **console.log** podría valer. Para comprobar que esto funciona agrego el **FilterWidget** a la **home.page**. Ahora ya puedo seleccionarlo. No uso ninguna entrada salida con él simplemente para que aparezca y para ver cómo los cambios tienen repercusión inmediata en el **console.log**. Vamos a ver cómo hacer esto un poquito más útil.

7.2.2 Señales desde Query Params

Vale, tenemos aquí el **FilterWidget** que, cuando cambiaba algo el usuario, escribía en el **console log**. Esto me parece poco. Yo creo que lo suyo sería que escribiésemos, por ejemplo, en el **router** para poder recoger esa información desde otra página.

Vamos a utilizar el **router**. Ojo, que no es el de **express**. Estas cosas pasan a veces. Vale, hay que asegurarse que es el **router** que pillamos desde

@angular/router. Y una vez que lo tengamos, podemos mejorar este efecto haciendo una navegación en la cual pasamos el valor de este filtro, que te recuerdo que es una señal de tipo **filter**, y que cuando esto cambia, porque el usuario teclea en cualquiera de estos sitios, lo que hará es una navegación sin moverse de la ruta en la que esté, pero cambiando los **queryParams**. Vamos a verlo.

Fíjate ahora en la **URL** que ya está poniendo ahí **Search**, **Order By** y **Sort**. Y en cuanto yo escriba, voy a escribir así a lo largo para que se vea que la **URL** está cambiando. Si además pongo algo con interés, como por ejemplo por fecha descendente, lo copio, me lo traigo aquí para que puedas verlo. Ahí está. Estamos de alguna manera serializando el valor de esta propiedad computada a la **URL**. ¿Con qué intención? Con la intención de recogerlo, por ejemplo, en la **home page**. Para ello, y aprovechando que ya tenía en la configuración ya habíamos puesto que queríamos utilizar el **router** con el **component InputBinding**, esto quiere decir que cualquier parámetro, no sólo los parámetros que van fijos en las rutas, como por ejemplo dos puntos **slug**, sino que **query parameter** también se puede recoger.

Esto es lo que estoy haciendo aquí, recogiendo a partir de la **URL** los valores que pueda haber que se llamen así, **search**, **orderBy** y **sort**. Para ver que eso trae algo, lo voy a pintar en el **footer**, de forma que ahora, si te fijas aquí abajo, esto irá cambiando según teclee.

¿Por qué? Porque el widget filtrador escribe en la **URL** del **router** y el **router** de vuelta se lo pasa a la **home page**. Es interesante que pienses que estos dos **componentes**, a pesar de que están muy cerca uno del otro, no los estoy tratando como padre-hijo ni **pare-container**, de hecho, tranquilamente este **componente**, que ves que está cambiando aquí las cosas, podría estar físicamente, por ejemplo, en la cabecera, si es que tuviese un poquito más de sentido estético y de tamaño. De esta forma puedes comunicar **componentes** que no tienen una relación directa a través, por ejemplo, del **router**.

7.2.3 Query params observables

Una de las razones para escribir en la **URL** es que me permitiese, por ejemplo, guardar esta, vamos a decir, sesión de filtro, de forma que, si navego a una página y vuelvo, volviese y se re-hidratase con los datos que tendría que haber ahí. Digo tendría porque, como ves, se han perdido. Esto vuelve a estar vacío. No sé si el zoom te permitirá verlo, pero esa es la realidad. Lo que quiero es que el valor inicial que le doy aquí al **FilterWidget** se pueda tomar también de la cabecera, de la **URL**. El problema es que ahora no tengo una manera cómoda de hacerlo con

señales. Las **señales** me funcionan bastante bien, pero son de solo lectura las que vienen del **query parameter**. Aquí realmente lo que puedo hacer, y de paso repasamos un poco cómo se accede a los datos a través de **observables**, es obtener acceso al **queryParams** a partir de **ActivatedRoute**. **ActivatedRoute**, igual que **Router**, es un **servicio propio de Angular** que puedo solicitar su inyección y me da un puntero a la ruta actualmente activa.

A partir de ahí puedo acceder a su **observable** de **queryParams** y recuperar en una **señal** que rellenaré a partir de la función de **interrupt toSignal**, que parte del **observable** y que requiere un valor por defecto. Lo que estoy haciendo aquí es transformando un **observable** a una **señal**. ¿Para qué? Para utilizarla después como punto de entrada de otra **señal**. De esta manera lo que venga de los **queryParams** entrará como valor inicial de mi **señal** de búsqueda. Una vez que lo hago para todos compruebo el resultado. Ahora, si todo va bien, al escribir una actividad, solicitar un tipo de ordenación, aunque me vaya de la página al volver, esto se almacenará. Y no sólo se almacena sino que se recupera, me rehidrata de alguna manera el **componente de filtro** y además, por supuesto, se lo pasa a la página para que filtre. Pero eso será en la próxima lección.

7.3 Operadores avanzados de RxJs

Vamos a explicar ahora cómo se comportan los **observables** de primer orden con **RxJS** en una situación que espero que te resulte familiar.

Voy a sustituir a los trabajos típicamente de llamar a un **API** por los trabajos de desarrollo de un sistema en **backend**, de una base de datos, de un sistema **front** y de una tarea de **testing**. Algo que te puede resultar común en cualquier desarrollo y cualquier proyecto. Obviamente, estos trabajos tienen duraciones distintas. Unos serán más largos que otros y, por supuesto, alguno puede fracasar. Claro, ¿por qué no?

También, para hacer una simulación un poquito realista, distinguiremos una situación donde tengamos a un equipo de desarrolladores de una situación donde tengamos a un único desarrollador. Y los resultados se producirán en el momento del tiempo que sea propicio. Es decir, en algún momento se entregará el **back** y en otro momento se entregará el **data**. Y en esta situación no habríamos entregado ni **front** ni **test**.

Pero bueno, esto es el punto de partida para que entiendas cómo van a ser y cómo se van a comportar cada uno de los distintos operadores de primer nivel.

Bien, vamos a ver el primer operador que es útil cuando tenemos un equipo. Es decir, que tengamos cuatro desarrolladores, por ejemplo, y sepamos las tareas

desde un primer momento toda la especificación de lo que se puede hacer. De esta manera, todas pueden comenzar en paralelo. Y obviamente, cada una se resolverá según el tiempo que le lleve. En una situación como esta, cada trabajador entrega su resultado cuando termina, pero **fork join**, una vez que ha hecho la parte **fork**, después hace el **join**. Y el **join** no lo puede hacer hasta que el último haya entregado su trabajo. A partir de aquí, sí que entrega el trabajo de todos ellos. Entregará el **back** bien hecho, la base de datos bien hecha y el **front** y el **testing** completos. De esta manera, se garantiza la entrega de todo en una sola respuesta.

Esto es útil cuando tienes una batería de datos de entrada, haces 20 peticiones en paralelo al **API** y pintas el resultado una vez que tienes todos ellos resueltos. El siguiente operador va a ser el **Merge Map**.

Igualmente, también trabaja en equipo, así que podrá tener a los cuatro trabajadores disponibles. Lo que pasa es que en esta situación, las peticiones de trabajo no son conocidas todas en el mismo momento, sino que van surgiendo. De esta manera, yo puedo tener un **backend** que sé que empieza en un determinado momento y me lleva un tiempo. Puede ocurrir que la base de datos no empiece a desarrollarla hasta otro momento y que la acabe también muy rápido. Puede ser que el **front** no me decida hacerlo hasta pasado un buen tiempo y puede ser que el **testing** decida hacerlo en este momento y me lleve todo este tiempo. Como tengo al equipo de cuatro desarrolladores, no tendré problema y obtendré los resultados según cada uno de ellos termine.

En esta situación, yo no espero por el último, sino que obtengo la base de datos, en un determinado momento obtengo el **backend**, en otro tengo el **testing** y en otro tengo el **frontend**. Es decir, las cosas ocurren en paralelo y el primero que acaba, según va terminando, va entregando su resultado. Esto es útil cuando respondes, por ejemplo, a eventos del usuario, que vas lanzando peticiones quizá a un **API** en el momento en que el usuario hace cosas y cada una de ellas termina cuando le toque.

Esto es lo que hace el **Merge Map**, mezclar todos los trabajos de manera que el que los recibe piensa que esto es un **observable** único que ha producido estos resultados en el tiempo. Hasta este momento suponíamos que teníamos un equipo de desarrolladores y entonces podían trabajar en paralelo. A partir de ahora te presento situaciones como la del **Concat Map** que sólo va a poder trabajar porque es lo que tiene con un único trabajador. Al tener un único trabajador está muy bien que las peticiones lleguen cuando tengan que llegar.

Supongamos que primero empieza a hacer el **backend** y en un determinado momento alguien le pide que haga también la base de datos. Vamos a suponer que esto ocurre aquí, pero como ves aún no ha terminado de hacer el **backend**.

Así que sus opciones son varias, pero en este caso este trabajador decide terminar lo que está haciendo y retrasar, posponer el inicio del **data** al momento en el que haya terminado con el **back**. Es decir, entrega el **back** y después se pone con el **data**. Cuando termine el **data** lo entregará.

Si tuvo la suerte de que nadie le interrumpió pidiéndole el **front** hasta que hubiese terminado el **data**, pues lo empezará cuando le toque y en ese momento entregará el **front**. Si alguien decide interrumpirle pidiéndole el **testing** mientras estaba haciendo el **front**, pues esperará y no empezará a hacer el **testing** hasta ese momento y lo entregará cuando acabe. Entonces, lo que ocurre aquí, obviamente es que los tiempos se dilatan porque, claro, sólo tengo un trabajador. Esto es así. Y él va entregando todo, no deja nada sin hacer, lo que pasa es que hace uno detrás de otro según el orden de las peticiones que le hayan llegado.

Esto sería útil si tú quieres guardar datos según el usuario vaya marcando o haciendo clic en un botón de Save, Save, Save y se van retrasando, concatenando uno detrás de otro hasta que el proceso esté libre. Simplemente no haces ninguna petición en paralelo.

Una versión similar es el **Switch Map**. Seguimos con un único trabajador y seguimos con nuestras peticiones de hacer un **backend**, de hacer una base de datos, de hacer un **front** y de hacer un **testing**. Y todas estas llegan cuando al jefe de equipo se le ocurre que tengan que llegar. Entonces, en esta situación, si hemos recibido la petición de hacer un **backend** y por el camino nos aparece otra petición para hacer la base de datos, atención porque este trabajador deja lo que está haciendo y se cambia a hacer el **data**. Si tiene la suerte de que nadie le molesta mientras hace la **data**, pues lo entregará.

Pero, después, por ejemplo, si empieza a hacer el **front** más tarde y alguien le interrumpe pidiéndole que haga el **testing**, dejará el **front** a medio, sin terminar, y se pondrá a hacer el **testing**. Obviamente, cuando termine el **testing**, claro, lo entrega. Resultado. Este trabajador sólo nos entrega la base de datos y el **testing**. ¿Y qué hay del **backend** y del **front**? Pues los dejó a medias. Fueron trabajos que canceló y se cambió de tarea. De ahí viene el **switch**. Este **switch** es porque deja de hacer una cosa para hacer otra.

Esto es útil, por ejemplo, en un caso de un buscador donde el usuario está tecleando algo y va cambiando su petición. Bien porque se arrepiente, porque corrige o porque está completando la pregunta. Entonces, la pregunta anterior no tiene sentido para nosotros, ese proceso se inició, pero se cancela su proceso y se cambia a otro. Esto es lo que hace **switch map**.

Y ahora vamos a ver lo que hace **Exhaust Map**. De nuevo, tenemos un trabajador. Obviamente lo he dejado para el final porque estará exhausto el hombre. Y como

fuentes de trabajo, pues volvemos a tener que le piden hacer un **backend**, que le piden hacer una base de datos, que le piden hacer un **front** y un **testing**. De nuevo, estas cosas ocurren en cualquier momento del tiempo. Este trabajador lo que va a hacer si le piden hacer un **backend** es ponerse a trabajar en él y si por el camino alguien le pide hacer un **data**, recuerda que en el **switch** se cambiaba, dejaba el **backend** y se pasaba el **data**, pues aquí hará al revés. Lo que hará es ignorar completamente la existencia de esa petición y seguirá trabajando en el **backend** hasta que lo entregue.

Si afortunadamente nadie le pide el **front** antes de eso, pues el **front** será su siguiente tarea y la hará hasta que la termine. Si por el camino alguien le pidió hacer un **testing** será ignorado, es que ya ni lo inicia, simplemente no lo entrega. En esta situación el trabajador nos entrega el **back** y en algún momento el **front**. Del **data** y del **testing** nada se supo, pero también es verdad que no le dedicó ni un segundo de su tiempo, no se cambió de tarea, como hizo en el caso del **switch map**, donde abandonaba lo que estaba haciendo y se pasaba a la siguiente, y en esta situación es hasta que no termine, no se pone con ninguna otra cosa, olvidando las que pudieron pasar por el medio.

Bueno, esto también podría ser útil en el caso de un buscador, de manera que ni siquiera hagas caso de lo que te teclee si piensas que la primera petición es realmente la buena, porque eso es lo que está pasando. Esta petición es la que considera buena y esta la considera mala. En el caso del **switch**, lo que está haciendo lo deja y se cambia. En el caso del **concat**, pues todo lo considera bueno y lo va haciendo, cuando buenamente puede, uno detrás de otro. De esta manera, tarda mucho pero lo entrega todo. Y claro, ninguno de ellos compite con el **merge** ni con el **fork**, porque estos son equipos de desarrolladores que pueden trabajar en paralelo.

La diferencia es que en el caso del **fork** sabemos todo el trabajo que hay que hacer desde el segundo cero y los lanzamos en paralelo, mientras que en el caso del **merge** las cosas ocurren cuando buenamente pueden y se resuelven también cuando buenamente pueden.

Espero que estos ejemplos te puedan servir para entender cómo se comporta cada uno de estos operadores de primer nivel y escojas el que más adecuado es al caso de uso que tú tienes.

7.3.1 Observando y operando con eventos de usuario

Regresamos aquí al **FilterWidget** que se encargaba de rellenar los tres campos de filtro: término de búsqueda, cómo se va a ordenar y el sentido del orden. Y nos vamos a centrar en esta parte de la búsqueda para darle una funcionalidad extra

que me va a permitir responder a eventos del usuario de una manera mucho más controlada, evitando ráfagas de tecleo, evitando valores repetidos y evitando quizá términos de búsqueda demasiado pequeños. Como es una lógica suficientemente compleja la voy a llevar a un componente específico. Hasta cierto punto lo consideraré tonto porque él no obtiene ni modifica los datos por su cuenta, pero no lo será para nada tonto porque aquí le vamos a meter mucha inteligencia.

Ya que es un componente presentador, empezaremos por tener un modelo de entrada-salida que me permita tener datos para presentar aquí. Y lo haré mediante un **input**, en principio muy sencillo porque no necesita siquiera tener un **FormsModule** ni nada por el estilo. Simplemente le daremos un valor, este es el término de búsqueda que podrían pasarme, ni siquiera tiene evento porque el evento lo vamos a obtener de una manera programática. Te presento aquí otra de las novedades del mundo de las señales que es que podemos hacer consultas sobre la plantilla y retornar eso como una señal. La consulta lo más cómodo es hacerlo sobre un elemento pre-identificado con el sostenido de **Angular**. Te recuerdo que el sostenido de **Angular** es una manera de generar identificadores que son muy fácilmente utilizables desde el controlador. Mi intención es en el constructor obtener la referencia a ese elemento, ya que esto es una señal, declaro un efecto y en ese efecto obtengo el valor que tenga la señal en ese momento. Si no la tiene, me voy. Si es que, por ejemplo, no se ha construido la template, esto antes había que esperar a que la template estuviese construida y utilizar algún **hook** dentro del ciclo de vida del componente, pero ahora no es necesario porque aquí recibo señales cada vez que esto cambie.

La primera vez puede valer indefinido, pero la siguiente en cuanto tenga el valor del elemento ya puedo trabajar con él. Vamos a ver ahora una fuente **observable** que es **fromEvent()**. Es una función que viene de **RxJS** y que recibe como primer argumento un puntero a un elemento. En este caso va a ser el **inputEl** y dentro de eso nos vamos al **nativeElement**. Tengo un acceso directo al elemento **HTML**. Su segundo argumento será el evento al que me interesa suscribirme. Aquí se produce una suscripción a un evento y el resultado va a ser algo **observable**. Este **fromEvent** se puede tipificar.

¿Para qué hago todo este trabajo? Al final del día lo que intentaré hacer en este **subscribe** será, con un valor que me llegue, voy a poner por ahora x, me gustaría asignárselo al término de búsqueda. Esto por ahora no es válido porque lo que recibo sería esta variable x de tipo event y yo lo que necesito pasarle aquí es un **string**. Para ello empiezo a trabajar con las **tuberías**, que es de lo que va a tratar fundamentalmente esta lección. En las **tuberías** lo primero que puedo hacer para ver lo que sale por ahí es usar **tap** y **map** para transformar. Tanto **tap** como **map**

son funciones que vienen de **RxJS**. Puedo meter dentro de cada una de ellas otras funciones. Recuerda que estamos en programación funcional y cada uno de estos operadores a su vez ejecuta una función con el dato que recibe y le envía al siguiente de la **tubería** un dato procesado. En el caso de **tap** realmente no mutamos nada, no retornamos nada y es muy habitual que después del implica simplemente haya una instrucción, un **console.log** para ver qué demonios está viniendo por ahí. Y veo que me llegará un evento y ahí lo publicaré. El **map** se queja porque efectivamente sí que tiene que retornar algo. En el retorno lo que voy a hacer es una transformación de un objeto evento al valor que tiene dentro.

Lo que hace es recoger el evento, hacer un **casting** a un elemento **input** que tiene la propiedad **value**. Así que en el siguiente lo que recibiré por aquí le llamaré **value**, diré que es de tipo **string** y lo que haré con él es imprimirlo. Fíjate que ahora el **subscribe** ya no se queja porque x es de tipo **string**. De hecho, ya no le llamaré x sino **searchTerm**, el término de búsqueda y declaro su tipo un **string**. Es hora de ver esto en ejecución. Pongo en marcha la aplicación. Mientras se compila voy a arreglarlo para que este **SearchComponent** que acabamos de crear sea utilizado dentro del **FilterWidget**. En el futuro sustituirá a este **input**, por ahora simplemente lo va a duplicar.

Si todo va bien deberíamos ver dos buscadores, pero el que me preocupa es el de abajo en el cual voy a escribir. Ha ocurrido un evento de entrada, nos hemos quedado con su valor. Eso es lo que hacemos aquí. Del evento quedarnos con el valor. Podemos hacer más cambios por supuesto. Uno que nos preocupa por ejemplo es no filtrar con términos demasiado cortos. Así que puedo utilizar el operador **filter** que dado un valor tecleado sólo lo tiene en cuenta si su tamaño es superior a 2. Vamos a comprobar esto metiendo otro **tap**. De nuevo si escribo una S vemos que el evento ocurre, el valor se obtiene, pero no pasa al siguiente nivel. Eso sólo ocurrirá cuando aparezca la tercera letra. ¿Qué ocurre si el usuario escribe en modo ráfaga? Bueno pues que se generan un montón de eventos y a lo mejor no todos son interesantes. Habría que descartar los valores intermedios. Eso se hace mediante un operador que se llama **debounce** que es como que me desentiendo de los eventos que transcurran más rápido que 300 milisegundos. Para comprobarlo vamos a poner otro **tap** más. Después los quitaremos y voy a escribir muy rápido cualquier otra cosa. Pero fíjate que el **debounce** me ha evitado ver… Voy a quitar muchas casas de aquí y si te fijas de nuevo ha habido un montón de eventos **filter** porque por aquí ha pasado por supuesto los **inputs** que obtienen el valor, los propios eventos, ha habido un montón de ellos pero el **debounce** ha evitado que muchos de estos de estas ráfagas lleguen.

Otro operador muy interesante para estos casos se llama **distinctUntilChanged**. Creo que tiene un nombre bastante específico, pero aun así vamos a comprobarlo

con un nuevo **tap** porque lo que pretende es que dado un texto si yo borro y vuelvo a escribir la f muy rápidamente el **distinctUntilChanged** me protege y no se emite porque realmente es el mismo valor que el anterior. Si yo en lugar de una f pongo una t entonces sí, si lo corrijo y vuelvo a poner una f vuelve a surgir. Bien, y ¿qué hacemos con el widget que estábamos recogiendo este valor? Bueno pues claramente el **lab-search** va a sustituir a este **input** así que lo podríamos comentar. Tengo entonces que utilizar lo que antes era un **ngModel** ahora es el término de búsqueda. Es decir, cualquier componente de más bajo nivel que exponga un model puede usarse con un **banana in a box** porque permite leer y escribir a la vez. Si yo escribo aquí un texto pues aquí aparezca filtrado y allí aparezca el texto para filtrar. Esta primera ejecución tiene el problema de que aparece aquí un **undefined** porque ahora como no tenemos aquí ningún elemento en el **URL** no hay un filtro previo pues este **undefined** se propaga hacia abajo. Requiere una corrección mínima que es simplemente protegernos que si aún no han llegado valores pongamos los valores por defecto en nuestras señales.

Ahora ya empezamos bien. Si yo hago una ráfaga sólo cuando paro se emite el evento y si quito y pongo una p muy rápido como el evento en sí no ha cambiado nada cambia. Si voy borrando a trocitos sí, sólo cuando me paro. Es decir, tengo ahora mismo un buscador donde estoy bastante protegido en cuanto a ráfagas, búsquedas demasiado cortas y valores repetidos. En la próxima lección empezaremos a hacer uso de estos términos de búsqueda para realmente filtrar.

7.3.2 Operadores observables de primer orden

Bien, habíamos quedado en que necesitábamos usar los **filtros** para realmente hacer una consulta en el **API**. Pues en base a estos **filtros** te dejo aquí la composición de la **URL**. No tiene demasiado misterio, pero ya para que no nos dé trabajo pues está creado en el **activities.repository**, el cual es invocado desde el **home.service**. También tenemos aquí un **getActivitiesByFilter**, que asegura que el **filtro** vaya completamente relleno. Ya sabes que puedes no buscar nada en concreto o no haberlo ordenado por ningún criterio en concreto o en un orden ascendente o descendente concreto. Así que esto es lo que le da soporte a la **home.page**, que visualmente pues tiene su parte del buscador, su parte del listado, su parte de informe, pero que por ahora no está haciendo ningún tipo de ordenación. Para que la haga voy a aprovechar estas tres señales, que son las que me llegan bien del **router**, bien de lo que hace el usuario, y voy a cambiar, y mucho, esta línea que hasta ahora obtenía las actividades así directamente, pero que ahora lo va a hacer a través de **observables** y **switchmaps**. Esto es algo que ya habíamos visto hace unas lecciones, pero que ahora me voy a centrar para explicártela lo mejor que pueda.

Bueno, lo primero es que voy a necesitar un **filtro** computado a partir de estas señales para mantener sencilla la llamada a este **API**, es decir, pasarle directamente el **filtro**. Lo segundo, eso que es una señal, la voy a pasar a algo **observable**. Para eso utilizo la función **toObservable** que venía con **interop, rxjs interop**. Ahora ya podría hacer una **tubería** a partir de este **observable**, pero ¿con qué intención? Pues la intención va a ser sustituir este **observable** de **filtros**, es decir, todos los cambios en **filtros**, por actividades. Para eso lo que hago es guardar aquí en una propiedad auxiliar una función. Fíjate que esto es una **arrow function** definida y almacenada en una propiedad. Recibe un **filtro** y llama al **getActivitiesByFilter**. Esto es para que me quede más sencilla la siguiente línea que te voy a poner. La siguiente línea lo que hace es, a partir de la fuente **observable**, que es este **filter** de aquí, hacer una **tubería** en la cual utilizamos un operador de primer orden, el **switchMap**. El **switchMap** lo que me dice es, cada vez que mi entrada cambie, desecha lo que estás haciendo y sustitúyelo por un nuevo **observable**. Y esto es lo que haremos. Cada vez que aparezca un nuevo criterio de **filtro**, olvidaremos lo que estamos haciendo y lo sustituiremos por el resultado de esta función. Esta función bien podría estar programada aquí dentro de este **switchMap**, pero me queda así más limpio. Ahora que tengo el **observable** que emitirá las actividades acordes al último **filtro** obtenido, me queda devolver eso a lo que teníamos, que eran señales. Recuerda que esto es lo que esperaba la interfaz. De hecho, por eso se queja, porque le faltan esas señales.

Bueno, pues aquella línea tan sencilla ahora se ve sustituida por esta otra, que no es más que volver a utilizar el **interop**, esta vez con el **toSignal**, desde un **observable** y pasándole un valor inicial. En definitiva, esto parece un gran rodeo y algo muy complejo, pero lo que dice es, si tengo una entrada que es una señal, puedo convertirla a un **observable** con la intención de canalizar a través del operador **switchMap** un cambio de ese **observable** fuente a un **observable** destino, a un **observable** target, si quieres. Y ese **observable** target lo transformo, sus resultados de nuevo en señales, para que lo consuma la interfaz. Bien, el resultado de todo esto es que cada vez que **filtro** se produce la llamada, si yo me cambio ahora aquí de **filtro**, pues se produce una llamada distinta, incluso el campo de ordenación también afecta. Y una cosa interesante que hace el **switchMap** es que esto sólo se nota si pongo, por ejemplo, que tengo unas llamadas lentas, porque si no, no va a ocurrir, pero fíjate que aquí la última llamada producida es esta que incluye surf por nombre y descendente. Simplemente lo voy a cambiar a ascendente y a descendente muy rápido. Y lo que ha ocurrido es que la primera llamada, a pesar de que sí se lanzó y al servidor le llegó, fue cancelada. Eso quiere decir que no se procesó la respuesta, no se ha perdido tiempo en ese proceso de respuesta porque no nos interesaba. Hemos

decidido cambiar de opinión. Queríamos ahora, por ejemplo, voy a irme a date y rápidamente a name. Tengo que hacerlo suficientemente rápido para que yo cambie de opinión más rápido de lo que tardan en atenderme. Esto es lo que hace el **switchMap**. Olvidar la primera consulta que se hizo y quedarse con la siguiente, devolver siempre la última de la que se hizo una petición. Esto es útil en condiciones de red lentas y en general, como buena práctica, el **switchMap** es perfecto para estas situaciones.

7.3.3 Peticiones paralelas

Para continuar con las soluciones de **observables** de alto nivel, vamos a plantear primero un problema. Tenemos aquí nuestro listado de actividades, de las cuales podemos marcar como favoritas las que nos parezcan bien, e ir a su página, en la cual nos aparece el listado. Bueno, nos aparece el listado simplemente de sus **slugs**, que es lo único que hemos recuperado de aquí.

Imagínate que quiero obtener el dato completo de todas y cada una de estas actividades, pero obviamente no hay una consulta que me lo devuelva todas ellas, sino que tengo que ir una por una solicitándoselas al **API**. Vale, en una situación como esa, lo que tendremos es la posibilidad de lanzar todas estas peticiones en paralelo, de manera que lleguen lo antes posible y que la interfaz sea responsiva en todo momento. Vamos a verlo en código.

Bueno, pues para agregar esa funcionalidad nos venimos a la página de favoritos y mantenemos aquí al lado el **activities.repository**, que me permitirá obtener los datos de una actividad basado en su **slug**. Esto ya teníamos este método, **getActivityBySlug**, que nos devuelve un **observable** con esa actividad cuando llegue. Por ahora almaceno ahí, inyecto aquí este repositorio para uso futuro.

Lo siguiente que voy a hacer es decir que este array de favoritos realmente lo voy a renombrar como una propiedad privada que ya no va a utilizar la vista, por eso se empezará a quejar, en la cual le daré un nombre más correcto, que serán los **slug**, los identificadores visuales de **URL** de las actividades que tenga como favoritas. Y esto es lo que tengo.

Lo siguiente será que si yo disponía de unos favoritos aquí, puedo volver a ponerlos, pero de tipo **signal**. Esto es lo que quiero acabar teniendo. No es lo que voy a tener, sino lo que me gustaría. Y por ahora, como no tengo nada mejor, simplemente les asigno una señal vacía. Realmente, como esto van a ser actividades, y ya por no gastar más la palabra favoritos, puedo poner aquí que estos serán las actividades.

Entonces, partimos de una señal de un array de strings a una señal de un array de actividades. Para facilitarme la vida, y ya que el estado de favoritos no va a cambiar una vez que visite esta página, puedo también invocarlo y decir que esto no será una señal, sino que serán ya el array de favoritos.

Lo siguiente que necesito es invocar para cada uno de ellos a esta función. Para que me sea cómoda esa invocación, estoy guardando aquí, en una propiedad privada aún no utilizada, la llamada a la función **getActivityBySlug** como una función invocable. ¿Con qué intención? Con la de mapear este array. Vamos a decir que cualquier array, por el hecho de serlo, tiene el método **map** al cual se le puede aplicar una función.

Bien, esto que estoy haciendo puede tener cierto sentido lógico, pero poco sentido sintáctico. Lo que quiero decir es que esta instrucción, por sí sola, no vale para nada, debería guardarla en otra propiedad. Esta propiedad me almacena ese concepto. Mapear un array a otra cosa. Esta otra cosa de aquí van a ser **observables**, así que me gustará marcárselo fuertemente, de forma que digo que esto es un array de **observables** de una actividad.

Es decir, para cada entrada de este array de cadenas, voy a tener un **observable**. No una actividad, sino un **observable** de una actividad. Y ahora me queda, digamos, la parte donde **RxJS** me va a ayudar con un operador de primer orden que realmente se usa distinto a todos los demás. No se usa dentro de un **pipe**, sino que se usa como fuente original. Y es el **fork join**.

Fork join lo que me permite es, dado un array de **observables**, es decir, esta propiedad, recoger su resultado en una sola variable. Como aquí lo que van a venir ya serán actividades, declaro una variable privada, pero de tipo **observable**. Y ahora, atención al tipo concreto. Aunque esto te pueda parecer un trabalenguas, aquí tengo un array de **observables** de una actividad, y aquí tengo un único **observable** de un array de actividades.

A todos los efectos, si alguien ve solamente esto, es como si hubiésemos hecho una llamada al **API** preguntando por todas las actividades o todas las que contienen surf. No nos daríamos cuenta de que es el resultado combinado de múltiples **observables**. Esto es lo que hace el **fork** dividir, y el **join** volver a juntar.

Una vez que tengo este **observable**, que viene de un **fork join**, como podría haber venido de cualquier otra cosa, ya puedo volver a utilizar, no un **signal**, sino un **toSignal**, para que esto refresque la interfaz y se desuscriba automáticamente. Son las cosas que nos hace la función **toSignal**.

Pero los pasos, lo repito, son los siguientes: Partimos de obtener el array de cadenas. Definimos una función que, dada una cadena, devuelve un **observable**. Mapeamos el array usando esa función, con lo cual nos devolverá un array de

otras cosas. ¿Qué cosas? Un **observable** de una actividad, de una, de la que se haya pedido aquí. A través de **fork join** lanzamos todos esos **observables** a la vez y recuperamos el resultado como un único **observable** del array de actividades. Esto tiene de bueno que se lanzan en paralelo. Y, por tanto, en principio, genera un mejor rendimiento total.

Por último, una vez que sabemos que existe este **observable** de actividades, simplemente lo transformamos a una señal con un valor inicial y lo pintamos. Para que esto ya se vea un poquito mejor, y dispongamos de una actividad, pues vamos a decir, de las buenas, no una simple string, ya podemos poner esto un poquito más bonito y ver en ejecución qué es lo que tenemos.

En ejecución, entonces, una vez que teníamos marcadas una serie de actividades como favoritas, si las queremos ir a ver, veremos cómo se lanzan prácticamente en paralelo todas las llamadas. Obviamente obtenemos los resultados, obviamente está aquí cada una de ellas por separado, porque eso es lo que ha ocurrido. Hemos lanzado en un **fork** que nos abre una serie de caminos y hemos juntado todo al final.

Y ese juntado es lo que nos devuelve aquí. Así que los operadores de primer orden unidos a los cambios interoperables entre **observables** y señales nos dan lo mejor de los dos mundos. **Potencia de RxJS para temas complejos asíncronos** y la capacidad de las señales de simplificar el repintado y la detección de cambios en **Angular**.