3. Señales y control

Tema estrella en Angular Moderno, las **señales**, que nos permitirán resolver un problema que
viene de años atrás, que es cómo detectamos los cambios para repintar las vistas de la
manera más eficaz posible. Con las **señales** entramos de lleno en el mundo de la **programación funcional** y aprenderemos a crear **señales derivadas** y **efectos secundarios**. Esos poquitos.

Para finalizar, agregaremos algo de lógica a nuestras **plantillas** para controlar, mediante **estructuras condicionales** y **repetitivas**, lo que se ve con los datos.

3.1 Actualización de datos y refresco de pantalla usando Signals

Bueno, pues vamos con el cambio más drástico que nos ha presentado **Angular** en estas últimas versiones modernas, que no es otro que el de las **señales**, porque ataca de lleno al corazón de cómo veníamos guardando y recuperando información.

Estoy pintando aquí una caja para hacer el famoso símil de una **variable** con una etiqueta, por ejemplo, saldo, donde yo guardo el valor 100. Si yo quiero recuperar el valor 100, pues por ejemplo, porque estoy haciendo un **console.log** o algo por el estilo, console.log(saldo), pues obviamente recibiré este 100. Si algo cambia el valor de la variable y pasa a valer 110, en principio o se vuelve a ejecutar este código voluntariamente o si no, el 100 no se cambiará mágicamente a un 110.

Y esto es importante cuando en una **template** tengo una referencia a la variable saldo, porque en este caso pintará el 100 y no sabrá exactamente decidir cuándo esto ya vale 110. Este es el problema que en **Angular** llamamos el del **change detection** o detección de cambios, que es un problema serio que ahora las **señales** vienen a resolver.

¿Cómo lo hacen? Bueno, pues encerrando a esta caja en un sistema que dice, bueno, yo me voy a comprometer a que, si tú llamas a unos determinados métodos que ahora veremos de determinada manera, voy a notificar a quien esté interesado los cambios que hayan ocurrido en el momento que hayan ocurrido, de forma que siempre estén actualizados.

Parece razonable. Así que saldo, a partir de ahora, va a admitir un método **set** en el cual le pondremos un nuevo valor o un método **update** que recibirá una función que mute el valor. Simplemente pensar que, si aquí ahora ponemos que esto vale 120, esto va a valer aquí 120, pero lo más interesante es que esto debería repintarse a 120.

Digo debería porque para que cumplamos la siguiente parte del contrato es que ya no llamaremos a la variable de cualquier manera, sino que llamaremos a la **señal** con unos paréntesis () porque convertiremos las variables en funciones de manera que se evalúan cada vez que hay un cambio. Es decir, saldo deja de ser una variable, una propiedad normal, tanto en plantilla como en código, y empieza a convertirse en una función, pero esa función se reevalúa ella sola cada vez que se sabe que ha pasado por el **set** o el **update** cambiando su valor.

Claro que para que todo esto funcione, saldo ya no vale definirlo como un número, ni igualarlo a un número, sino que habría que definirlo como una **signal**. Eso sí,

con un valor inicial, podría ser un valor indefinido también, se le puede asignar un tipo, tipo **string**, tipo **number**, tipo **boolean**, tipo **array** de **strings**, **array** de **objetos**, lo que se quiera.

Pero lo interesante, a partir de ahora las **señales** emitirán a quienes se suscriban los cambios en sus valores, y por suscripción entendemos simplemente la evaluación de las funciones señal.

3.1.1 Señales por todas partes

Bueno, pues es hora de darle algo de vida a esta aplicación que ahora mismo es muy estática y que sólo presenta información en modo solo lectura, como pueden ser los datos de una actividad, o ahora que he incluido aquí un contador de nuevos participantes que vamos a querer hacerles una reserva.

Para hacer eso voy a necesitar como mínimo tener un **formulario** y ahora voy a pediros un acto de fe porque tengo aquí listo el código para hacer un **input** que está decorado con un par de algo que llamaremos **directivas** que vienen en módulos externos y que tendré que importar. Pido un salto de fe porque dedicaremos una lección especialmente a todo el tema de los formularios y en ellas explicará qué es este **FormsModule** y otros que existen y qué es lo que hacen.

Por ahora os diré solamente que esto es un **input** de tipo numérico en el cual lo que estaremos haciendo es recoger el valor de una propiedad y dándola como valor del modelo asociado a este **input**. En cada cambio invocaremos a un método que también tenía aquí comentado y listo para ser usado, que lo que hace es recibir el número, el valor que tenga en ese momento y asignarlo a los **participantes**. Esto se puede hacer de más maneras, pero no es el momento ahora de aprender formularios.

Simplemente lo que quiero es que este 0 que aparece aquí lo hace porque este es el valor inicial y según yo lo cambio va cambiando aquí en **newParticipants**. Bueno, esto es una magia de **Angular** que viene desde tiempo inmemorial que es lo que se llama el **double binding** o el enlace doble entre vista y controlador o vista y propiedades.

Este es el modelo clásico de trabajo con **Angular**, pero yo os voy a presentar el modelo actual moderno que está basado en **señales**. Las señales van a ser tipos específicos de **Angular**, los vamos a importar, van a venir desde **Angular Core**, es decir, que es algo oficial pero que genera nuevos tipos de datos.

Para empezar, voy a convertir este entero en una señal y eso es tan sencillo como declararlo así: **Signal** con un valor inicial de 0. Muy bien, a partir de este momento

yo ya no puedo invocar a **newParticipants** como si fuese un valor numérico cualquiera, sino que tengo que hacerlo con una sintaxis específica que es como si fuese una función. Realmente esa es una función que evalúa el valor que tenga en ese momento la señal porque la señal justamente recibe este nombre porque emitirá un destello, emitirá un evento cada vez que su valor cambie.

Esto es algo que **Angular** aprovechará muy bien para saber cuándo debe repintar y qué trozo de pantalla debe repintar. Esto tiene también consecuencias en la manera de cambiar el valor, de asignarlo, ya no se hace una asignación tal cual porque lo que tengo a la izquierda ya no es un número normal, sino que es una señal y para ello tendré que utilizar el método **set**. Tampoco es que esto sea muy difícil, pero esto es un cambio fundamental porque **newParticipants** deja de ser, vamos a decirlo así, un número para ser una señal que vigila a ese número y emite cambios cada vez que ese número cambia.

Desde el punto de vista funcional esto no ha mejorado ni empeorado nada, pero una particularidad, solo para que seamos conscientes, es que a partir de ahora este valor no tiene por qué cambiar de referencia, lo que cambia es su contenido y esto sigue siendo completamente funcional.

¿Se puede hacer para más cosas? Sí, por ejemplo, yo podré tener otra señal para indicar o con otro tipo, por ejemplo, si ya hemos procedido a hacer una reserva o no y, por ejemplo, podríamos tener aquí en el botón de **book now** un evento **click** que cuando se haga click llame a **onBookClick**. Ese método aún no existe, pero yo puedo crearlo fácilmente así. A partir de este momento puedo ponerle el **booked**, ¿vale? para ver qué se ha hecho.

Si sigo haciendo click no pasa nada, pero me permitiría seguir haciéndolo así que una cosa que podríamos hacer es deshabilitar este botón en función de que se haya hecho click o no. Y entonces esto, una de las maneras sería convertir el atributo **disabled** en algo evaluable y asociarlo con la señal **booked**, de forma que a partir de ahora el botón está activo, pero en cuanto yo haga la reserva dejará de estarlo y eso es gracias a que esta señal ha cambiado.

Pues esta es la introducción a las señales que no son más que tipos de datos oficiales de **Angular Core** que admiten otros subtipos para sus valores. Por ejemplo, esta será una señal para un **number** y esta será una señal para un **boolean**. Los cambios se producen mediante **set** y ya veremos más adelante también con **update** y las lecturas se producen como si fuesen funciones. Es como llamar a la función que me devuelve el valor de la señal.

3.2 Programación reactiva con Signals

Retomamos el tema de las **señales**, viendo que cuando invocamos a su método **set**, se reevalúan y muestran su último estado cambiado. Pero lo que las hace realmente potentes es la capacidad de componerse unas sobre otras, como si fuesen derivadas funcionales, a través de la función **computed**.

El ejemplo que voy a mostrarte sería una variable que se llamase "tieneDeuda" y que vendría de un cálculo sencillo como el siguiente: es decir, que el saldo valga menor que cero. Esto puede estar en código imperativo o, sobre todo, nos interesa si estuviese dentro de unas llaves {}. En este caso, cada vez que la variable saldo cambie, se repintaría este trozo de **HTML**.

Pero, claro, realmente no siempre que cambia el saldo cambia el valor de tieneDeuda. Por ejemplo, nuestro amigo cada vez tiene una deuda mayor, pero no por ello deja de tenerla, sigue teniendo deuda. ¿Cómo se resuelve esto? Redefiniendo "tieneDeuda" como una señal computada a partir de saldo. **Computed** es una función que viene con **@angular/core**. Tienes que importarla y empezar a utilizarla. Dentro llevará otra función. Habitualmente esta función no recibe argumentos y en su cuerpo suele tener una única instrucción, así que normalmente las llaves sobrarán, pero formalmente sería algo así: lo que voy a hacer aquí dentro es programar un return de saldo menor que cero. De esta manera "tieneDeuda" ahora se puede utilizar como si fuese una señal.

Ojo, exactamente igual que las señales, necesita ser invocada como una función. La gran ventaja es que solo repintaré este **HTML** cuando **tieneDeuda** varíe, no cuando saldo varíe. Por tanto, no volvería a tener que repintarlo hasta que, afortunadamente, alguien haga un saldo **set** positivo.

Otro elemento crítico de la programación reactiva con señales va a ser la de los **efectos**.

Volvamos al ejemplo que nos ocupa, donde suponemos que tenemos una variable saldo de tipo señal con un valor inicial, algo computado que se evalúa cada vez que saldo cambie y después cambios sucesivos de la variable saldo, de la señal saldo, que desencadenarán el "tieneDeuda". Y hablando de desencadenar, vamos a suponer que necesitamos un efecto secundario que sea enviar un email. Ojo, ya no es pintar algo en la pantalla, sino enviar un email para avisar de que tienes deuda o para notificarte de que ya no la tienes. A eso es a lo que llamaremos un efecto secundario.

Es algo serio y que se programa dentro del **constructor**, porque realmente es como un registro. Para ello tengo que invocar a la función **effect** que se comporta de manera parecida a la **computed**. Tendrá unos paréntesis y una implementación

interna. En esta implementación interna habitualmente hay código que muta o genera eso, un efecto secundario. No devuelve un cálculo, no tiene que retornar nada, sino que hace cosas. Por ejemplo, si tiene deuda, enviar tipo A y si no, enviar tipo B.

Realmente, lo que está ocurriendo en estos dos métodos no es nuestra preocupación, sino que es un efecto secundario que se ejecutará cada vez que **tieneDeuda** cambie de cierto a falso. **Ojo**, no cada vez que saldo pase de 100 a 120 o de menos 100 a menos 200, sino cada vez que pase, por ejemplo, de menos 100 a más 50. Ahí **tieneDeuda** pasará de cierto a falso y esto se reevaluará.

La programación reactiva funcional se basa en estos dos principios: **computed** y **effect**. Uno para generar valores derivados y el otro para mutar o generar efectos secundarios más allá de lo que tenemos delante.

3.2.1 Computación derivada

Vale, pues vamos a continuar dándole funcionalidad a esta aplicación y repasando o aprendiendo algo más de **Angular** antes de meterle más caña a las **señales**.

Por ejemplo, teníamos aquí a la vista información sobre la actividad y participantes que ya estaban registrados de antes. Se supone que estos datos, vale, nos vendrán de algún servicio, pero por ahora los tenemos aquí como algo fijo. ¿Se pueden hacer cálculos y utilizar esos datos para más cosas? Pues por supuesto. En la actividad sabemos que hay un máximo y un mínimo de participantes y también ya hay gente que está apuntada. Luego, ¿cuántos me quedan que pueda apuntar yo? Pues actualmente serían los máximos participantes posibles menos los ya apuntados, es decir, me quedarían seis plazas disponibles.

Y eso lo puedo utilizar como un cálculo aquí dentro del atributo **max**, que es estándar de cualquier **input**, lo único que estoy haciendo es haciéndolo calculable y también limitándolo al mínimo, que en este caso es un valor fijo, de manera que yo no me pueda pasar para abajo ni demasiado para arriba. Como veis, esto es una pequeña mejora. Igual que el disponer aquí de unos **displays** que me vayan diciendo cuántos son los participantes actuales, lo único que cambia, cuántos son los totales, los que ya había más los nuevos, y cuántas plazas me quedan que, de nuevo, pues es un cálculo similar a este. Y esto prácticamente sin usar **señales** excepto que esta señal, **newParticipants**, se evalúa de una manera distinta a cómo se evalúa un número. Hay que llamarla como si fuese una función.

Y ahora viene lo novedoso que os voy a enseñar sobre funciones y computación, porque así es como se le llama, a un cálculo que se hace en base a **señales**, pero que se guarda de una manera especial, que se programa de una manera especial, que quizá al principio sea un poco más engorrosa, pero que también, en un acto de fe, os pido, lo veremos más adelante en este curso, que va a sentarle bien durante la ejecución, a acelerar la ejecución de los componentes con **Angular**.

Y para eso voy a presentaros la función **computed** y vamos a empezar, por ejemplo, por utilizarla para calcular el total de participantes. Esto que ahora mismo no es más que esta suma, lo voy a redefinir de esta manera. Y ojo, esta función **computed**, voy a guardar, antes de nada, viene, se importa, igual que **signal**, de **@angular/core**, es decir, es algo propio del framework. La función **computed** lleva como argumento otra función.

Aquí ya el asistente me ha ayudado demasiado, quería simplemente que viéseis que esto va a ser una función, con paréntesis, implica, etcétera, donde después haríamos un cálculo. Vale, el cálculo, efectivamente, sí es este cálculo sencillo, que dice el total de participantes va a ser los que ya estaban más los nuevos. Lo interesante es que tenéis que pensar que esta función se va a reevaluar sólo cuando esto de aquí cambie.

Ahora mismo, quizá no tengas muy claro cada cuánto se va a reevaluar una plantilla para volver a repintarse. Es normal, eso es un tema avanzado, pero sí te diré que las **señales** vienen a dar respuesta a esa gran pregunta, que es ¿cuándo se va a reevaluar? Pues cuando esto emita una señal de que ha cambiado. Cuando esto ha cambiado, este **computed** se va a reevaluar y cuando el resultado de esta reevaluación también cambie, pues se repintará donde corresponda.

Por ejemplo, yo ahora, **total participants** y, ojo, desde el punto de vista de la plantilla esto es como si fuese una señal. Este **computed** es, a su vez, una señal read-only para él y por tanto se evalúa con llaves {} como si fuese una función. Entonces **computed** lleva dentro una función y se invoca como función. Estamos en una programación reactiva funcional. Bueno, pues de la misma manera que he hecho el **total participants** haría también para el **remaining places**, que sería otra computación, donde me irá calculando cuántas plazas me quedan.

Muy bien, bueno, pues con esto ya tenemos un previo de lo que sería la computación derivada.

3.2.2 Efectos colaterales

Bien, vamos entonces a terminar esta introducción recordando que las **señales** se declaran con una función constructora **signal**, con un valor inicial. Sus valores pueden cambiarse, de eso volveremos a hablar ahora mismo, a través del método **set**, y esto notifica a quien esté interesado que ha habido un cambio. Puede ser directamente llamando a esa señal o indirectamente a través de la función **computed**, que una vez que te lias a hacer funciones **computed** pues las haces para todo.

Yo, por ejemplo, he hecho aquí una para deshabilitar el botón de bloqueo cuando aún no has seleccionado a nadie o cuando ya has hecho la reserva. En ese caso, ya no podríamos reservar. Incluso también muestro aquí el cálculo del importe que va a tener que pagar el que se suscriba. Y hablando de suscripciones, resulta que aquí teníamos una actividad que tiene un status **published**, que además está asociado a unos estilos, por ejemplo, en este caso aparece así como en color navy, en color azul.

Y la cuestión es que nos dicen ahora que cuando el número de participantes supere el mínimo, pues deberíamos automáticamente cambiarlo a **confirmed**, y que cuando llegue al máximo pues se pase a **sold**. ¿Cómo hacemos eso? Porque ahora mismo con el **computed** somos capaces de derivar cálculos, pero en este caso sería cambiar algo, actuar sobre algo, no necesariamente una señal, en este caso de hecho no lo es, y eso es lo que llamaremos un **efecto**, como un efecto secundario, lo mismo.

La particularidad de los efectos es que se van a parecer sintácticamente a las funciones **computed**, van a ser también funciones, pero van a tener que declararse en el constructor de la clase, que hasta ahora no se había utilizado prácticamente para nada, el constructor de la clase, donde se quieran aplicar. ¿Por qué? Porque realmente un efecto lo que es es una función que se registra al principio, y una vez registrada se ejecuta cuando algo en su interior cambie.

Entonces, igual que hacíamos con las funciones **computed**, tendremos que importarlo, ¿vale? Y de la misma manera que el **computed** lleva dentro una función, el efecto también lo llevará. Esta función normalmente tiene bastante código, así que para no aburrirte con ese código lo voy a copiar y a pegar ya directamente, y lo único que es de interés es esta primera línea. El sistema se da cuenta de todas estas líneas, las evalúa y encuentra que una de ellas es una señal, o algo derivado de una señal, una señal computada, no tiene por qué ser una señal original.

A partir de ese momento dice, vale, muy bien, cada vez que esto cambie yo voy a ejecutar este efecto, o sea, que este efecto se ejecutará cada vez que esto

cambie, y esto cambiará pues cada vez que el resultado de este **computed** cambie, y esto lo hará cada vez que el resultado de esta suma, en particular, de este **newParticipants** cambie.

Una vez que tenemos esto programado, ya podemos hacer efectos secundarios. Por ejemplo, podemos hacer que este **status** cambie en función de estos cálculos. Vamos a comprobarlo. Vemos que como mínimo creo que tenemos aquí 5 y empezamos con 3, así que cuando yo agregue otro más pues pasará a **confirmed**, y si llego al tope pasará a **sold**. Muy bien, pues ya tenemos entonces un efecto secundario que cambia algo en función de un cambio de una señal.

Normalmente trabajaremos con **computed** para cálculos derivados y con **effect** para cambios derivados.

3.3 Estructuras de control declarativas.

Uno de los grandes avances que nos ha traído **Angular Moderno** es la capacidad de hacer que dentro de un elemento **HTML** que tenga opciones que aparecen cuando una condición se cumple y otras opciones que aparecen cuando esa condición no se cumpla, sea realmente muy sencillo. Y lo hacemos estableciendo la condición **if** y la condición **else** directamente sobre el código. ¿Y cómo? Pues utilizando una nueva sintaxis de control de flujo directamente sin necesidad de importar nada. @if() con una condición y entre llaves {} el **HTML** que quiero que aparezca cuando las cosas van bien, cuando el **if** se cumple. Y @else con el contenido en **HTML** que quiero que aparezca en caso contrario. Sobre esto hay mejoras y evoluciones, pero esta es la clave. Estas @ nos dan el control de flujo declarativo para mostrar u ocultar **HTML** en función de los datos. Y eso es lo que tenemos para las estructuras condicionales.

¿Habrá algo para las repetitivas? Por supuesto. Imaginemos el típico **HTML**, por ejemplo, con una lista donde tenemos que poner un montón de elementos li que son esencialmente iguales simplemente cambiando algún contenido o algún atributo. En estas circunstancias nuestro resalvador es @for(). @for() me permite repetir un conjunto de elementos según un array, es decir, aquí dentro tendré un iterador sobre algo iterable, habitualmente un array.

Una cosa interesante es que necesitaré también marcar cuál es el id que identifica a este elemento y lo distingue de este otro. De esa manera el sistema sabrá que la colección es distinta y que necesita repintarla. Tal como con el **if** había el **else**, con el **for** tendremos el @empty{} que nos permitirá meter un **HTML** que aparezca cuando no hay nada. Por ejemplo, para mostrar un placeholder vacío o un mensaje vacío. Así que las @ para **for** y **empty** nos dan estructuras repetitivas

igual que el **if** y el **else** nos daban estructuras condicionales. Programación declarativa directamente en el **HTML**.

3.3.1 Condicionales

Bueno, pues después del tema de las **señales** y de las funciones **computadas** con valores derivados a partir de las señales, vamos a sacarle partido mejorando la interfaz del usuario, por ejemplo, con estructuras condicionales, que es algo que ya existía en **Angular** desde hace tiempo, pero que se ha simplificado y mejorado mucho.

Y para ello aparece la sintaxis @if(){} con una condición y un bloque de llaves {}, como si fuese **TypeScript**, donde yo puedo seleccionar qué trozos del **HTML** quiero que aparezcan si esta condición se cumple. Ahora voy a poner aquí un **true**, así como una catedral, y ahora le ponemos aquí un @else{}, que sería la contrapartida, para que aparezca un elemento de **HTML** cuando esto no sea cierto.

Pero ¿qué condición voy a poner aquí? Pues, por ejemplo, voy a hacer que si me quedan plazas libres, es decir, si **remaining places**, recordamos, es una señal, luego tengo que invocarla, aunque sea derivada, computada, si esto es mayor que cero, quiero que me muestres el **label** y el **input**, pero que cuando pongamos por caso que se nos han agotado las plazas libres, esta parte desaparezca. ¿Y en su lugar qué puedes poner?

Bueno, pues podríamos poner desde un sencillo para indicar que ya no nos quedan plazas disponibles, a incluso le voy a meter un botón que lo que haga sea resetear los participantes nuevos a cero para que de nuevo queden las plazas libres, aunque solo sea para que podamos jugar un poco con esta situación, con el **if** y el **else** que me permiten tener estructuras condicionales con bloques de **HTML** que aparecen en función del valor de los datos utilizando una sintaxis declarativa muy sencilla, parecida al **TypeScript**, y que además no requiere de la importación de nada porque está incorporada directamente en el compilador de **Angular**.

3.3.2 Repetitivas

Muy bien, pues ya hemos visto las estructuras condicionales con el **@if-else** y qué estructura te suena o echarías en falta. Pues sí, la repetitiva, ¿con qué? Si esto era un **if-else**, pues era con el **for**. Y de la misma manera, prefijado con el **@**.

Vale, ¿para qué nos puede valer una repetitiva como esta? Pues sobre todo para iterar en algún tipo de array, cosa que yo ahora mismo no tengo aquí, pero me voy a inventar uno de participantes, no solo con el número de los participantes, sino con los participantes en sí mismos. Y de paso nos sirve para revisar el concepto de **señal**.

Las **señales** tienen o pueden tener un valor inicial cuando son números, cuando son booleans, lo puede tomar por sí solo, pero cuando es una cosa así, lo que le digo es que, oye, yo voy a tener una **señal** que va a ser un array de objetos con una propiedad **id** que va a ser de tipo numérico.

Bueno, y vamos a lo que nos interesa. Vamos a suponer que quisiéramos mostrar aquí en un **div** el listado de los participantes, que seguramente deberían tener más que el **id**, tendrían que tener un nombre, un email, un teléfono, etcétera, lo que necesitasen.

Pero para ello empezamos con el **@for**. El **@for** lo voy a programar exactamente igual que había hecho con el **if**. No tengo que importarlo, lleva aquí una condición de iteración y después un bloque de contenido. La condición de iteración va a ser un iterador **of** algo iterable. Pues este iterador **of** iterable, técnicamente lo que haremos será poner aquí un array, que en mi caso va a ser la lista de participantes. Recordamos que como es una **señal** tengo que invocarla para que me dé el resultado. Y después esto no es más que el nombre de una variable, pues **participant** podría estar bien. Así que, para cada participante de ese array, para cada objeto de ese array, me vas a hacer algo. Pero me pide una cosa más que en versiones anteriores de **Angular** era opcional y ahora es obligatorio que exponerle el **track** de cuál es la clave principal del array, del objeto iterador. Como aquí solo tengo una propiedad y además resulta que sí, es el identificador, pues esto no tengo más que pensar, pero en caso de que tuviésemos un objeto más complejo, lo que le indica el sistema es que si cambia el identificador querrá decir que ese objeto es otro y por tanto debe repintarlo.

Esto es una optimización para el propio compilador. ¿Y dentro qué ponemos? Pues lo que se nos ocurra. Cualquier cosa que pongas aquí se va a repetir tantas veces como haya en este elemento y va a utilizar sus datos, pues por ejemplo ahora mismo me aparece en este 1, 2 y 3. Si yo cambiase cualquier identificador aquí, obviamente pues eso cambiaría. Y si, por ejemplo, se me ocurre que cuando el usuario teclea aquí, esto no está afectando a ese array. Te voy a copiar aquí un código que lo único que hace es devolver el array de participantes a quedarse solo con los que había inicialmente y después sobre los nuevos agregarlos. ¿Esto qué va a hacer? Pues que cuando empecemos con los primeros participantes, por cierto, para no tener problemas voy a poner los identificadores correlativos. Según

los agreguemos van a ir creciendo hasta que lleguemos a un reset que de nuevo nos los va a agregar o a quitar.

Muy bien, así que hemos visto que tenemos estructuras condicionales y estructuras repetitivas que visitan e iteran sobre arrays de objetos, de números, de cualquier otra cosa, siempre con un track que nos permita saber cuál es lo que lo discrimina de manera única.

Tal como el **if** tiene de contrapartida el **else**, quizá te preguntes si hay alguna para el **for**. Por ejemplo, para el caso concreto de que el array estuviese vacío. Sería una cosa interesante. Voy a vaciar este array y ahora mismo lo que ocurre es que esto no produce ninguna salida porque obviamente este **for** sobre este array no tiene nada sobre lo que iterar. Pero ¿qué ocurre si yo quiero indicarle específicamente eso al usuario? Pues para eso tenemos aquí la sintaxis **@empty{}**. **@empty{}** funciona como en el caso del **else**. "No participants yet", ¿vale? Bueno, pues ya está.

Es simplemente ver que el **for** tiene asociado un **empty** igual que el **if** tiene asociado un **else**. A partir de ahora esto pues ya funcionaría así de esta manera, que es iterar cuando hay algo y mostrar el contenido vacío cuando no lo hay. A modo de repaso me voy al componente **footer** para agregarle una funcionalidad de que cuando aceptemos las cookies desaparezca este botón y aparezca en su lugar un mensaje indicando eso. Para ello voy a empezar creando una propiedad de tipo **señal** booleana que dice que las cookies por ahora no están aceptadas y haré que cuando el usuario haga clic ahí las marque como sí que están aceptadas. Recordamos que esto se hacía con el **set(true)**.

A partir de este momento puedo utilizarlo dentro de una template con una sintaxis condicional **@if** que me diga ¿están las cookies acepted? Si es así puedo mostrar este texto y si no mostrar el botón. En cuanto el botón se haga clic desaparecerá y aparecerá este texto.

Luego tenemos aquí el repaso de la sintaxis condicional declarativa utilizando **señales** para tener la plantilla siempre actualizada.