8. Seguridad y validación de datos

A los usuarios lo que les gusta es ver información, no tener que rellenar formularios tediosos, y sobre todo no tener que repetirlos, o no tener un **feedback** inmediato de si lo está haciendo bien, mal o regular. Bueno, afortunadamente todo eso en **Angular** es fácil con los **Reactive Forms**.

La seguridad es un tema serio, pero sobre todo en el lado del servidor. En el lado del **browser** no es para tanto, solo tienes que enviar las credenciales y guardar el **JSON Web Token**, que claro, habrá que enviar en cada una de las peticiones. Fácil, con los **interceptores**. Darle un buen **feedback** al usuario es la manera de que se sienta cómodo y seguro utilizando tu aplicación.

Y para ello, podemos utilizar los **Error Handlers** para capturar cualquier información, cualquier error, y mostrárselo de la manera más cómoda y sencilla posible.

8.1. Formularios para recogida segura de datos

El tema de los formularios suele ser algo tedioso para los usuarios que tienen que rellenar un montón de campos y a veces incluso hacen mal las cosas. Así que vamos a hacerles la vida lo más fácil posible y en **Angular** nos ofrece enlazar fácilmente lo que será la **template** que el usuario vea con el **modelo de datos** que nosotros queremos recuperar, porque al final de eso se trata. De enlazar de alguna manera estos **elementos HTML**, estos controles, con estos campos.

Y para ello, en lugar de hacerlo directamente, nos ofrecen un intermediario. Algo que llamaremos un **abstract control**, o abreviadamente un control. Este control va a disponer internamente de un nombre. Este nombre coincidirá con el nombre de una propiedad y se asociará en la **template** a través de un atributo llamado **Form Control Name**.

Por supuesto que lo más interesante es que este control lo que tendrá es un **value** que coincidirá con lo que escriba el usuario y que será lo que se asigne aquí a este valor. Y todo esto, que está muy bien, tiene además dos estados o dos máquinas de estado que nos indican si el valor que se ha introducido está en un estado válido o inválido.

Y además, todo esto, si el usuario ha hecho algo con el valor o no. Es decir, si está **pristine** o no, ya sé que este nombre es un poquito raro, **pristine**, **touched**, si el usuario ha hecho algo con él o no, no quiere decir que lo haya cambiado, cambiado sería **dirty**, que es que sí que lo ha cambiado.

Bueno, pues la unión de todas estas cosas, más el hecho de que al final un formulario es un conjunto de estos controles, **control 1**, **control n**, y que el formulario al final también tiene valor y estado, valor como la unión de todas estas propiedades, es decir, sería este objeto completo y estado, pues bueno, si todos los controles son válidos, el formulario se considera válido, si alguno es inválido, se considera inválido, y demás.

Bueno, pues todo esto viene en algo que su nombre nos recuerda a otros frameworks, porque empieza por reactivity, de normals, **reactive forms**, que es de lo que trata, y **module**, que nos recuerda al pasado de **Angular**. Bueno, pues este **reactive forms module** es el causante, el culpable de que tengamos todas estas herramientas a nuestra disposición.

Y para que las podamos usar de una manera más o menos sencilla, nos ofrece tipos de datos, como será el **FormGroups**, y herramientas, tools, como será el **FormBuilder**. Y con esto podremos hacer formularios reactivos que informen al usuario fácilmente de cómo está rellenando las cosas, si van bien, mal o regular.

Al final, cuando haces formularios, te vas a encontrar casi siempre poniéndoles aquí un **label**, teniendo aquí algún tipo de **input**, y seguramente algún **small hint** o error. Y esto lo vas a repetir varias veces. Claro, con el tiempo acabarás diciendo, oye, los **hints** quiero que aparezcan de una manera, los **errors** quiero que aparezcan de otra, y sólo cuando los hay, por supuesto.

Los **labels** también me interesa que aparezcan así, en recuadrado, yo qué sé. Y te preguntarás, ¿cómo puedo reutilizar esto? Bueno, hay varias maneras. La más completa y compleja se basa en crear controles que implementan una interfaz llamada **Control Value Accessor**, pero para empezar en **Angular** yo te ofrezco una solución de andar por casa un poquito más sencilla.

Esencialmente se trata de crear un contenedor donde la parte **label** está predefinida, la parte de los errores también, **small errors** o **hints** también, y admite, esta es la parte interesante, que el **input**, o también, por qué no, un **select**, un **checkbox**, cualquier otra cosa, se incruste aquí. Para ello, en **Angular** disponemos de una directiva llamada **ng-content** que nos permite incrustar un elemento dentro de otro.

De forma que el nombre que se va a poner aquí puede venirnos como un input, los errores pueden venir como un input, pero la parte más compleja, la que gestiona el **reactive-forms**, esta, la delegamos o la mantenemos en el sitio original. Y esto es lo que haremos utilizando **ng-content**.

Termino esta introducción hablándote de las funciones de validación, que, en principio, reciben argumentos que podríamos asociar con los controles al que vamos a validar, obviamente, y después tienen una lógica, habitualmente con algo que comprueba que el **value** coincide o no con lo esperado, y que cuando las cosas van bien, retorna **null**, curioso, **null** significa bien, porque quiere decir que no se ha incumplido nada, y cuando detecta algún error, retorna cualquier otra cosa, habitualmente un objeto que lleve como propiedad el nombre de la regla rota, y como valor algo que pueda servir, o como mínimo un **true**.

Bueno, estas funciones las puedes crear tú a voluntad para validar cosas que, en principio, el framework no te ayude, que te ayuda con mucho, por ejemplo, con cosas requeridas, con cosas de longitud mínima y máxima, por ejemplo, para cadenas de texto, con valores mínimos y máximos, por ejemplo, para rangos de números y fechas, pero otras cosas tendrás que hacerlas tú, y serán, en principio, así de fáciles.

Otra cosa es que, a veces, te encontrarás con funciones que requieran dentro otras funciones, retornen otras funciones, y esto será porque aquí sí que estará el control, pero fuera podemos tener argumentos. Entonces, este tipo de funciones

que encadenan a otras funciones, o que envuelven y retornen otras funciones, también tendrás que familiarizarte con ellas.

Con todas estas dos cosas, más la posibilidad de hacer validaciones asíncronas, pues tienes todo en tu mano para hacer validaciones de formularios que satisfagan a los usuarios.

8.1.1 Formularios reactivos

Vamos a continuar con algo que normalmente a los usuarios les da un poco de pereza, que es rellenar formularios. Sería lo que tendríamos aquí, tanto para el **login** como para **registro**, que es por lo que vamos a empezar ahora. Para ello, voy a aplicar la buena práctica del **Container Presenter** y llevarme toda esta lógica de presentación a un **componente**, pues eso, de presentación. Nos sirve un poco de repaso el generar un **componente** con un tipo específico.

No es obligatorio, pero a mí me gusta cambiarles el tipo con la implicación de ajustarlo también en el **ESlint** para que admita **form** como un sufijo válido. Y a partir de ahí viene el típico trasplante y empezaremos a utilizar este **componente** como un presentador. Fíjate que al ser este un **componente enrutable** no necesita un **selector**. Si lo tuviese no podrían coincidir, pero como no lo tiene, pues ahora mismo me estoy ahorrando inventarme un **selector** extra que normalmente hubiera sido algo como esto. Para darle vida a este formulario voy a empezar por importar algo que viene en **Angular Forms**, pero que se llama **ReactiveForms**.

ReactiveFormsModule contiene todos los elementos para convertir este formulario, ahora inanimado, en algo que sea útil y que ayude a validar la entrada del usuario. ¿Y qué cosas vienen dentro de ese **módulo** y qué demonios es un **módulo**? Bien, un **módulo** es la manera clásica que tenía **Angular** de agrupar **componentes**, **servicios** y demás. Hoy en día estamos haciendo que todos nuestros **componentes** sean **stand-alone** y por tanto no necesitan **módulos**, pero algunas librerías como esta, **ReactiveForms**, siguen viniendo dentro empaquetadas como **módulos**.

Como decía, dentro tenemos, por ejemplo, tipos de datos y el primero que te voy a presentar es **FormGroup**, que es un grupo de **controles**. Realmente un formulario se establecerá como un conjunto de **controles**, normalmente serán **inputs**, **selectores** y demás, que tendrán una contrapartida en el **modelo**. Por eso estos formularios también les llamaremos **conducidos por el modelo**. Mancharemos, utilizaremos poco la **template** y habrá bastante negocio aquí en el **controlador**. Para cada elemento que queramos enlazar de la **template** con el **modelo de datos**, crearemos una propiedad en un objeto. Esto no es más que un objeto

donde cada una de sus propiedades coincide con el **modelo de datos** que voy a querer recoger. Para cada uno crearé un **control de formulario** con un valor inicial, normalmente una cadena vacía, y con algún **validador**, que ahora veremos para qué se usa, pero que por defecto aquí mi amigo **Copilot** me ha regalado que el hecho de que sean requeridos. Si necesitase más validaciones, como por ejemplo para el caso del **email**, debería meterlas en un array. Vamos a decir que un **form control** tiene como primer argumento el valor inicial y como segundo argumento uno o múltiples **validadores**.

Los **validadores** son funciones que importamos también de **AngularForms** y que después veremos que podemos crear nosotros mismos. ¿Qué validaciones tenemos? Bien, voy a agregar una manualmente, por ejemplo, aquí para la **password**, y veréis que en cuanto escriba **validators** tengo aquí una serie de opciones como por ejemplo controlar el tamaño mínimo, como ya hemos visto validar si es un **email**, el tamaño máximo, el valor máximo si es que fuese un número, el valor mínimo si es que fuese un número y demás. Para una **password**, por ejemplo, me vendría bien que el tamaño mínimo pues fuese, por ejemplo, de cuatro. La confirmación de **password**, por supuesto, requeriría estas mismas validaciones, así que copio y pego. Por último, el de los **términos** lo doy por bueno así. Bien, de esta manera he creado el modelo de un formulario que es un grupo de **controles**.

¿Cómo lo engancho con la **template**? Pues usando directivas. La primera será **formGroup**, que igualmente, que aquí esto es un tipo, aquí esta directiva está esperando que le demos una variable de ese tipo, que no será otra que esta variable **form**. Por supuesto siéntete libre de nombrar esta propiedad como te venga bien. Hay quien lo hace de una manera minimalista y le llama **f** porque después hay que usarlo en múltiples sitios. Hay quien le llama como lo que va a recoger, por ejemplo, **RegisterForm** en este caso, y yo me quedo un poco a medio camino creando **form** que no es muy largo y es un poquito más explícito que una simple **f**. Esto lo que hará es enlazar el formulario con esto de aquí. ¿Para qué me vale? Bueno, voy a poner ahora una etiqueta **pre** con el típico chivato para mostrar el valor del formulario según lo relleno para que veas cómo voy a ir paso a paso enlazando estos **inputs** con estos **controles**. Lo único que hay que hacer es declarar un atributo no estándar, formalmente una directiva llamada **formControlName** y que también viene dentro de este **ReactiveFormsModule** y asociarle el **control** que quieres enlazar con este elemento **html**. Desde este punto de vista a esto le llamaríamos **propiedades del modelo** y aquí pues son elementos nativos **html**. Hago lo mismo con los demás y vamos a comprobar cómo funciona. Como puedes apreciar tenemos aquí el objeto de registro que está vacío pero que yo puedo rellenar con cualquier **password** y aceptando o no

los términos y condiciones. Como ves todo esto está perfectamente vivo y puedes cambiar lo que necesites.

8.1.2 Validaciones personalizadas

Vale, muy bien, esto enlaza entonces elementos nativos con propiedades de un modelo a través de algo llamado **FormControl**. ¿Y los validadores qué? Bien, pues los **validadores** nos van a servir para controlar si un elemento cualquiera es o no es válido, en función de que cumpla con las validaciones establecidas. Por ejemplo, acabo de agregar el atributo **aria-invalid**; esto es estándar, aunque **Angular** me obliga a prefijar algunos atributos con **ATTR** antes de poder incluirlos.

En el fondo, lo que estoy haciendo es asignar el atributo **Invalid** cuando realmente lo sea. ¿Cómo accedemos a este valor de **Invalid**? A través de **Form**. Aquí es donde nos conviene que la variable no sea muy larga. **Controls**, buscándolo por su nombre, nos da acceso a un montón de propiedades como **Errors** para saber qué errores tiene, **Valid** para saber si está válido, **Invalid** que sería su contrario, **Pristine** para indicar que nunca fue tocado, **Dirty** para indicar que su valor ha sido cambiado, y **Touched** para indicar que el elemento HTML fue tocado por el usuario aunque el valor no haya cambiado.

Yo en este caso lo dejo con un simple **Invalid**, que hace que el framework CSS que estoy usando coloque un color rojo o verde en función de su valor. Observamos cómo aparece en rojo y con esta marca cuando no tiene nada y se pone en verde cuando las cosas van bien. Si hago lo mismo para todos los demás elementos, ya tengo algo usable.

Todo empieza mal, las cosas pueden ir más o menos bien y cualquier cambio hace que las cosas dejen de funcionar correctamente. Las validaciones operan a nivel de control, sí, pero también a nivel de formulario. Así que puedo, por ejemplo, hacer que este formulario se ponga **Disable** en base a que él mismo sea inválido, si lo prefieres. Así, el botón estará deshabilitado hasta que yo rellene algo que considere mínimamente decente y luego ya podría hacer clic.

A los usuarios no sólo les interesa saber que han cometido un error sino también saber cuál han cometido. Por ahora no voy a poner gran cosa, pero sí te diré que podemos serializar al menos el objeto **errors** de manera que ahora mismo tengamos una pista de qué es lo que está pasando, aunque hay mucho que mejorar en la manera de presentar esta información. Como ves, sólo se trata de agregar pequeños trozos de texto que contendrán los errores.

Ahora, aquí no sólo veo que tengo un problema, sino que tengo una pista de cuál es y que desaparece, aunque quede un poco feo, según lo tecleo. Ahora me dice

que el problema está con que esto no tiene pinta de **email**. Lo mismo ocurre con la **password**, incluso dándome bastante información sobre el problema real que estoy teniendo. Ya podría enviar. Si queremos evitar que aparezcan errores cuando realmente no los hay, podríamos envolver cada uno de estos elementos en un **if**; de esta forma, cuando las cosas van bien, no aparece nada.

Bien, vemos entonces cómo podemos validar y dar **feedback** básico y sencillo a los usuarios en función de lo que van tecleando. Para que tengas un mayor conocimiento de cómo funcionan los **validadores**, voy a crear uno particular, **custom**, validaciones más allá de las que trae el framework, que además aplicaré a todo el formulario y no a un control en particular. Bueno, realmente lo aplicaré a dos controles, pero eso excede lo normal, que es aplicar cada validación a un solo control. Y es el caso de la **contraseña**, que tiene que ser la misma cuando la confirmas.

Ese tipo de validaciones, no solo para igualdad de **contraseña** sino, por ejemplo, para fechas o rangos que estén uno por encima del otro, se colocan en una sección extra donde se agregan las validaciones grupales para todo el formulario. La validación que voy a poner aquí no existe aún como tal; la voy a programar en una función y luego la llevaré a otro fichero aparte. Las funciones de validación puedes nombrarlas como quieras; yo aquí le voy a llamar **matchValidator**. Va a recibir como argumentos el nombre de los controles involucrados, en este caso será **password** y **confirm**.

Contrariamente a lo que podrías esperar, esta función no va a retornar nada inmediatamente, ni siquiera va a retornar un **boolean**. Lo que hará esta función cuando reciba argumentos extra es actuar como una factoría y retornar a su vez otra función. Si no estás acostumbrado a trabajar con programación funcional, quizá esto te resulte extraño, pero esta es la realidad: una función podría admitir una función como argumento o puede retornar otra función como resultado.

Esta función interior puede retornar sus propios resultados, como por ejemplo, aquí voy a hacer que retorne **null**. Pero obviamente no siempre será así. Cuando use esta función **matchValidator**, aceptará estos dos argumentos, **password** y **confirm**. Realmente lo que estamos diciendo es que esta función de aquí, que recibe estos dos argumentos, devolverá otra función que será la que se use como validación. La función que se usará como validación es esta interior, que por ahora siempre retorna **null**, pero entenderás que en algún momento tendrá una condición donde retorne veces unas cosas y otras, otras.

De hecho, te voy a empezar por explicar que en estas funciones se retornará **null** cuando las cosas vayan bien, porque eso significa que no hemos obtenido ningún error y retornaremos cualquier otra cosa cuando las validaciones no se cumplen. Es habitual devolver un objeto que indique el nombre de la regla rota, por ejemplo,

en este caso, que los datos no coinciden. Pero ¿cómo comprueba esto? Para ello tiene que usar estos strings, que son los nombres de los controles, contra el formulario. Y sí, cada vez que utilizamos un validador, implícitamente esta función recibe como argumento a su control, o en este caso, al formulario.

Así que aquí, implícitamente, nos estarán enviando un puntero al formulario. El formulario no es de tipo **form**, ni siquiera de **formGroup**, sino que es de tipo **AbstractControl**. Un control abstracto es lo que nos separa de una implementación HTML concreta y de un modelo de datos. Bueno, esta es la terminología. Lo siguiente sería obtener punteros a los controles concretos. Se puede llegar a través de **form.controls** o con **form.get**. Cada una de estas variables tendrá en este momento un control al que acceder y comprobar sus valores. Si no coinciden, tendremos un problema. Pero si coinciden, retornaremos nulo, que quiere decir que todo ha ido bien y que esta regla se ha satisfecho.

Debo decirte que, además de retornar este valor, si quieres asignárselo concretamente a uno de ellos para que forme parte de su grupo de errores, si quieres que su grupo de errores aumente, puedes hacerlo directamente agregándole este error. Bien, de esta forma, ahora cuando tengamos aquí que poner una contraseña, la de abajo no será suficiente con que tenga el tamaño adecuado, sino que su valor tiene que coincidir. Ves que este nulo viene exactamente de aquella respuesta. Como decía, esta función se puede utilizar desde varios formularios, así que yo suelo crear un fichero en el que agrego estas y otras validaciones, de forma que se puedan reutilizar en varios formularios. Como ves, viene de UI, que es la carpeta **Shared** que tengo aquí para cosas de **User Interface**.

8.1.3 Control de presentación

Bueno, pues esto hecho con el formulario de registro sobre la página de registro, a la cual envío una señal cada vez que el usuario hace **submit**. De esto volveremos a hablar más adelante. Una vez que tengo esto, como digo, me gustaría expandir esta manera de proceder a la página de **login**. Y entonces me daré cuenta de que nuevamente vuelvo a tener algo tipo **Label**, **Span**, **Small Input**, que sustituya a este **Label**, **Span**, **Input** de aquí tan sencillos por un poquito más, pero que siempre es muy parecido.

Cuando encontramos cosas muy parecidas y que nos gusta que sean parecidas, lo ideal sería hacerles un **componente** que fuese reutilizable. Que por lo menos nos dé la estructura para que todos nuestros controles, por eso le he llamado **control**, se parezcan. Es decir, que esta estructura **Label**, **Span**, **Small** y lo que sea, siempre sea de esta manera. El "lo que sea" es lo que **Angular** nos permite

hacer con **ng-content**. Un elemento de **Angular** que permite que cualquier otro componente reciba una entrada contenida. Esto es más fácil de ver en código que de explicar. Así que déjame que le agregue las propiedades de entradas requeridas, como son el nombre del **control**, lo que va a visualizar el usuario y los errores si los hubiera. Y por supuesto, claro, el **JsonPipe**, que si hago las cosas bien, esta importación debería desaparecer de ahí, venirse para aquí, porque ahora será el **control.component** el que se ocupe de todas estas cosas.

Así que voy a ponerte aquí uno debajo del otro cómo va a quedar esto. Sustituiremos este **Label**, **Span**, **Small**, **Input** por un **lab-control** que recibe las cosas como argumentos. No es que ahorremos muchas líneas, no es ese el objetivo, sino el hecho de que tengamos en un solo sitio la capacidad de determinar cómo se van a presentar, por ejemplo, los errores. En algún momento esto será mejorable o quizá, por ejemplo, había tenido yo la pereza de no meter **@if** a todos los **Small Controls** y, por tanto, seguían apareciendo nulos, pero ahora, al sustituirlos de esta manera, todos me quedarán igual. Como ves, de esta forma no me va a costar tanto crear el formulario de **login** y otros. También, como siempre, te dejo aquí en **Domain**, pues las definiciones de los tipos que voy utilizando.

El formulario de **login** es incluso más sencillo que el de registro y, como siempre, se trata de importarlo para tenerlo dentro del contexto, seleccionarlo, recibir sus eventos y, próximamente, pues enviar esto a través de servicios a un **API** que, como estamos tratando con cosas bien delicadas, como son el registro y el **login** de usuario, enlazarán con el tema de la seguridad del cual aún no hemos hablado. Aunque aquí se toca de refilón un tema que también es seguro, que es la validación de entrada de los datos del usuario, tanto en el formulario de **login** como en el de registro. Ahora las cosas funcionan.

8.2. Interceptores de comunicaciones y guardias de Navegación

Al hablar de seguridad, lo primero que tienes que tener claro es que lo que protegemos normalmente serán unos datos que estarán accesibles detrás de un servidor que nos expone un **API** y que esta es realmente la parte segura, tanto en el envío como en la recepción de esta información hacia el **browser**. Este cliente lo que tiene que hacer es identificar al usuario a través de unas credenciales, normalmente será el **email**, la **password**, etc.

Envíalas, espera que el servidor valide a este usuario y normalmente devuelva una identificación que será un **JSON Web Token**. Este **JSON Web Token** es algo firmado por el servidor que él reconoce y que tú como desarrollador del **browser** lo único que tienes que hacer es almacenarlo de alguna manera local, bien en la

memoria o incluso en el **storage**. En este caso, habría que tener algún tipo de protección extra y a partir de ahí todas las demás peticiones que hagas, sean **GET**, **POST**, etc., podrías incluir este **token** dentro de una cabecera especial de autorización con el prefijo adecuado, de forma que sea el servidor el que te lo valide.

Pero es fundamental entender que toda la lógica de seguridad real está del lado del servidor y que tú lo único que tienes que hacer es recuperar este **JSON Web Token** y enviarlo de la manera más cómoda posible de cada vez. A la hora de enviar las peticiones desde el **browser** al **server** solemos verlas como algo de uso único que aquí va una **request** y después nos viene una **response**. Pero la realidad es un poquito más compleja y resulta que hay una serie de eventos intermedios en todo este proceso.

El caso es que **Angular** aprovecha estos eventos para permitirte meter **interceptores** en todas las peticiones. **Interceptores** no serán más que funciones que tengan un puntero a esta petición y al siguiente procesador. Será lo que llamaremos el **request** y el **next**. La idea es que tú crees **interceptores** con el comando **ng g Interceptor**, les pongas un nombre y programes esta función dentro. Programar la función no suele ser complicado una vez que uno entiende que lo que puedes hacer normalmente será o modificar la **request** o modificar la **response**, pero en ambos casos no lo puedes hacer así de cualquier forma.

Modificar la **request** requiere clonarla porque la considera inmutable y se la pasa de un procesador de un **next** al siguiente, copiada, sin que sea realmente un puntero. Y por otro lado la **response** también tendremos la oportunidad de procesarla. Realmente lo que haremos será canalizar a través de un **pipe** de **RxJS** los procesos que puedan trabajar con ella. ¿Por qué? Porque consideramos que cada uno de estos procesadores es como si fuese un evento, transmitiese un evento en el proceso de una petición. Por lo tanto lo que hacemos es canalizar ese evento y mutar lo que entra y lo que sale.

Bueno, todo esto que parece bastante complicado al final **Angular** te lo hace fácil con el generador y con el hecho de que, para apuntártelo, para registrarlo, solo tienes que ir a donde esté el **app.config** y en los **providers** tendrás el **httpClient**. Bueno, pues habrá que decirle que lo use with **interceptors** y dentro de esos **interceptors**, que será un array, podrás meter los tuyos ordenadamente. Esto antes era bastante más complicado, pero ahora no es más que focalizarte en esta lógica que hay dentro de la función de intercepción, que por ejemplo es muy, muy, muy utilizada para transformar una **request** añadiéndole las cabeceras de autorización, es decir, el **JSON Web Token**.

Bueno, ahora que estamos focalizados en la situación de que entre dos pasos que puedan parecer contiguos hay otros intermedios potenciales, te voy a plantear

que, si estuviésemos hablando del **router**, podríamos estar en un **path 1** y movernos, el usuario puede hacer click y moverse o navegar a un **path 2**, pero que el **router** nos ofrecerá una serie de puntos de control intermedios para hacer ciertas cosas. Por ejemplo, podremos comprobar si puede abandonar o no de activar, se llama, esta ruta, si puede activar la siguiente o incluso si podemos resolverle algún problema antes de entrar.

A grandes rasgos, esto es lo que quiero que pienses, que hay posibilidad de introducir algo que llamaremos **guardias** y **resolvers**, ambos con su propio generador, así que tendremos el **ng g guard** y **ng g resolve** y a partir de ahí todo será rellenar estas funciones, donde recibiremos como entrada el **path** en el que estamos y como salida, pues realmente aquí será true si dejamos proseguir, esto es un **guardia** que deja continuar o no, abandonar una ruta, llegar a otra, o false, pero con la particularidad también de una navegación alternativa, es decir, esto es muy utilizado cuando, por ejemplo, el usuario no está autenticado y lo enviamos a **login**. En el caso del **resolver**, lo que devolveremos será un **observable**, pero ese **observable** se asignará a un valor y una vez asignado a ese valor se puede utilizar directamente en la página. Todo también se configura en las **routes**, el fichero de **routes** admite para cada **path** que le pongamos **guardias** y **resolvers**.

8.2.1 Envío de credenciales y almacenaje de Token

Bueno, nos adentramos en el oscuro mundo de la seguridad y partimos de nuestra página de registro con su formulario de registro perfectamente validado. Cuando el usuario lo envía, hace uso de una señal de salida que emite los datos registrados por el usuario con la intención de recogerlos en la página y después enviarlos. Ahora empezaremos a ocuparnos de enviar los datos tanto de registro como de **login**. Para ello, dejo de lado estos formularios y voy a estudiar los tipos de datos que he creado aquí en el dominio, que más o menos eran ya los esperados.

He creado un tipo para el registro, donde tenemos el nombre del usuario, el **email**, la **password** y los términos aceptados. Para el **login**, es incluso un poco más sencillo, y algo novedoso que depende del servicio que estoy utilizando ahora mismo. Además de lanzar la aplicación, lo otro que hacíamos es arrancar un servicio llamado **json server auth**, ese apellido **auth** es interesante porque es lo que nos va a llevar al tema de la seguridad.

Este servicio **auth**, que aquí nos habrá dejado un rastro, expone las rutas de actividades y **bookings**, que hasta ahora era lo que habíamos utilizado funcionalmente, pero también la de **users**. Y lo hace además con unos códigos que parecen un poco extraños, pero que en el fondo lo que están indicando son

una serie de permisos sobre estas ramas. Sin profundizar en lo que esto está haciendo ahora mismo, lo que haremos será que para crear actividades tengas que estar registrado y para modificarlas tengas que ser el dueño de esa actividad. Lo mismo o muy parecido también para las reservas.

Habrá también una ruta de usuarios a las cuales podemos llamar para hacer registros y **logins**. El servidor nos contesta con un objeto que tiene esta forma, un usuario y un **access token** en modo string. Este **access token** lo tendremos que guardar para utilizar en todas nuestras llamadas y el usuario, pues no es más que la representación del lado del servidor excluida la **password** por motivos de seguridad.

Para comunicarnos con ese servicio he creado un repositorio en **Shared API**, llamado **OutRepository**, que tiene lo que hemos visto hasta ahora siempre. La dependencia del **HttpClient**, una **URL** que por ahora sigue estando ahí **hardcoded** y después los métodos de negocio que necesite para comunicarme. En este caso recibo un objeto de registro y lo envío mediante **post** a la ruta **users** que acabamos de estudiar, que existe, enviándole este objeto y recibiendo el **UserAccessToken**. Con el **login** es algo similar, solo que la ruta que nos proveen es específica, no es sobre el objeto **users**, sino que tiene este nombre específico para enviar las credenciales y que nos devuelvan igualmente un **UserAccessToken** si es que las cosas han ido bien.

Armados con este servicio inyectable podríamos reclamarlo directamente en la página de registro y empezar a utilizarlo directamente dentro del método suscrito al evento. De esta manera haremos el envío, nos suscribiremos para que efectivamente la llamada se produzca, aunque por ahora no estamos haciendo nada con el resultado. Y de eso quería hablarte, de este resultado que viene por aquí, que por ejemplo vamos a sacar por consola. Vamos a ver qué devuelve.

Bien, tenemos esto programado no en la página de **login**, sino en la de registro, en la cual meteremos un usuario, haremos el envío y comprobaremos que este **post**, una vez enviado, nos responde con este usuario creado en el servidor, su identificador, etc., y su **access token** que tenemos que almacenar. Y para almacenarlo vamos a utilizar dos lugares. Uno, la memoria con un **store** nos servirá un poco de repaso de ese concepto de guardado para reutilización, y por otro lado también vendremos al **application** para guardarlo como en el **local storage** y poder reutilizarlo entre ventanas o incluso con una recarga parcial de la página.

Para ello, voy a generar un servicio en la carpeta **shared/state** donde guardamos todo lo que se refiere a nuestro estado de la aplicación, teníamos hasta ahora el tema de los favoritos, y ahora aparecerá también el **Auth-Store.Service** que yo

después, ya sabes, le cambio el nombre para identificarlo mejor, al apellido simplemente **Store**, igualmente aquí simplemente **Store**.

¿Y qué habrá en este **Store**? Bueno, pues un state privado que será una señal **Writable** sobre **UserAccessToken**. Esta señal parte con un valor inicial nulo, pero es lo que me va a permitir almacenar el usuario y el **token**. Para ello, como siempre, método público que reciba el dato y lo asigne y una serie de propiedades computadas a partir de este estado, de manera que el resto del código utilice lo que se necesite. Por ejemplo, saber si está autenticado o es anónimo en función de tener o no **accessToken** y también obtener fácilmente el **userID** que lo necesitaremos para firmar nuestras creaciones. Otra señal computada muy interesante es la que ya nos selecciona exclusivamente el **accessToken** que utilizaremos más adelante en todas las peticiones al **API** para identificar al usuario mediante ese **JSON Web Token** que el servidor reconoce.

La siguiente cuestión es, ¿dónde y quién usa a este **AuthStore**? Bueno, las posibilidades son dos, o bien hacerlo con los datos recogidos, recuerda que este result al final es de ese tipo **UserAccessToken** en cada una de las páginas, o delegar esa función directamente al repositorio, hacer como que el repositorio tiene una inteligencia extra que le permite automáticamente guardar esa información en el almacén. Si optamos por esta última, necesitaríamos no suscribirnos sino canalizar mediante un **pipe** y generar un efecto secundario, que en este caso sería con el operador **TAP**, de forma que el **UserAccessToken** se guardase en ese **AuthStore**. Lógicamente, esto requiere un puntero al **AuthStore** y una lógica simple, en la cual cuando recibamos el **UserAccessToken** lo guardemos ahí directamente.

Para ver esto en marcha podemos agregarle una funcionalidad extra al **AuthStore**, que es la de guardarse en el local repository. Como esto ya lo teníamos programado del caso de los favoritos, sólo hay que reclamar una inyección del local repository y, sin más, pues guardarlo durante el **setState**. Obviamente esto se podría hacer también como un efecto secundario de la asignación, pero por ahora esto es suficiente. Y para que las próximas veces se recoja el valor inicial a partir del local repository, también cambio la declaración inicial del estado, la inicialización del estado con esto. Bien, vamos a ver si esto ya funciona.

Para ello, registro otro usuario y voy a comprobar que se me guarda aquí como **AccessToken**, que inicialmente es vacío, pero al hacer el envío veo que en la recepción se ha guardado ya en la aplicación. Esto será útil cuando empiece a hacer que todas mis peticiones **Network** vayan firmadas en algo que por ahora no estoy utilizando, que serán las cabeceras de seguridad.

8.2.2 Interceptores de comunicaciones

Para hacer uso de este **AccessToken** guardado en este **AuthStore**, vamos a crear un **interceptor**.

Vaya, con ese nombre parece algo muy serio y complejo, pero la realidad es que como nos lo genera **Angular**, pues tampoco es para tanto.

Lo que sí es que lo que vemos aquí es un pelín raro, porque declaramos una constante que en el fondo tiene que ser de un determinado tipo, y se inicializa con algo parecido a una función. Bueno, es una función, una **fat arrow function**, donde recibe unos argumentos y retorna una salida.

Para conocer un poquito más sobre los argumentos y la salida, podemos analizar un poquito cómo es la declaración de este tipo función. Y veremos que el primer argumento, la **request**, es un puntero a la petición en curso, porque eso es lo que hará un **interceptor**, interceptar todas las peticiones en curso. Cada petición es manejada por varias funciones, y aquí lo que tenemos es un puntero a la siguiente.

Nuestra idea será meternos en el medio y utilizar esta petición antes de enviarla a la siguiente, pues quizá con alguna mutación o con algún cambio. Un tema interesante es que la función en sí devuelve un **observable** de eventos **HTTP** desconocidos. Bueno, difícilmente te encontrarás una firma más extraña que esta, pero vamos a tratar de analizar esto. Digamos que cada petición **HTTP** pasa por una serie de eventos, sale de tu navegador, llega al servidor, se recoge, se procesa, etc.

Y lo que tenemos es un **observable** que nos va emitiendo esos sucesos. Es decir, habrá un momento en el que la petición salga y otro en el que la petición llegue. Bueno, pues eso es lo que llamaremos eventos de **HTTP**.

En la práctica, en la mayor parte de las situaciones, lo único que tendremos que hacer es modificar de alguna manera la petición antes de ser enviada o bien canalizar a través de un **pipe** esta sucesión de eventos, esos **observables**, y meter algún tipo de transformación normalmente con la respuesta obtenida. Esto es a grandes rasgos lo que hace un **interceptor**. Quizá modificar la petición y quizá procesar la respuesta.

Modificar la petición podría significar o podría parecer algo sencillo como, por ejemplo, llegar y decir que la **request** tiene una serie de **headers** y dentro de esas **headers** tendrá una como **authorization** y que a esa le podríamos asignar algún valor. Pero las cosas no son tan sencillas porque realmente la **request** es algo inmutable que si queremos cambiar debemos clonarlo. Bueno, esto es un pequeño engorro pero que fomenta la programación funcional.

Y afortunadamente, sabiendo que esto es así, nos proveen de un método **clone** el cual devuelve una copia exacta de la petición con las mutaciones que queramos ponerle. Por ejemplo, podemos decirle que agregue las **cabeceras** que necesitemos, por ejemplo, la de **autorización**, la cual será una cadena de texto que haya que rellenar convenientemente.

Para rellenarla voy a hacer uso del **AuthStore** que teníamos por aquí. Así que pediré que me lo inyecten y lo usaré para obtener el **accessToken** a partir de esta señal computada. De forma que si tengo algo se lo asigno en esta autorización prefijado con el token **bidder** que le indica al servidor que tipo de **JSONWebToken** estamos enviando. Bueno, este **interceptor** que hemos generado habrá que utilizarlo en algún sitio.

La cuestión es desde dónde y cómo se registra. Bueno, te diré que, aunque es una función y no tiene el arroba **injectable** que hemos visto, por ejemplo, en estas clases, también es algo que tenemos que proveer. Para ello vamos a venir al **app.config** donde están las listas de **providers** y vemos que tenemos el de **HttpClient** que ya venía como una indicación de utilizar **Fetch** del lado del servidor. Eso era útil para utilizar el **server-side rendering**.

Bueno, pues también tenemos en esta nueva sintaxis **standalone** la posibilidad de utilizar **withInterceptors**. Y en este **withInterceptors** se admite un array de potenciales **interceptors** al cual yo le voy a agregar este. Tener varios **interceptors** será útil cuando queramos hacer funciones distintas en cada uno de ellos. Por ejemplo, este estará dedicado a la autorización, pero otro podría estar dedicado al proceso de respuestas, a la gestión de errores, a la repetición, a la caché y demás.

Por ahora solo tienes que entender que existe el concepto de **interceptor** que es una función que durante el proceso de esa función tenemos que retornar un puntero al siguiente procesador y que previamente podemos hacer cualquier tipo de mutación sobre la **request** que es lo que estoy haciendo aquí. Vamos a ver cómo funciona esto en la práctica. He lanzado la aplicación y podemos dirigirnos a la pestaña de **network** y ver aquí cómo las peticiones se han realizado y en principio en las **cabeceras** de la petición la **autorización** va vacía.

También puedo ver que por ahora no tengo ningún **accessToken** guardado. Esto tiene fácil solución si registro a un nuevo usuario y me devuelven ya este **accessToken**. Esto también lo puedo ver aquí como la **payload** subió y regresó con los datos adecuados.

¿Qué ocurre ahora cuando voy a visitar **Activity Bookings** por ejemplo? Las peticiones que se hacen responden igualmente, pero la **autorización** ahora va rellena con la palabra clave **BIDDER** más el token que habíamos recibido aquí. Te recuerdo que esto es justo lo que me ha devuelto el servidor. Esto le sirve a él para

identificar a este usuario, al usuario 4 que en el fondo es lo que necesita. Y de esa manera pues por ejemplo me permitirá que cuando creen mis nuevas actividades o mis registros de mis reservas de actividad queden pues efectivamente al nombre de este usuario.

Bueno pues así es como enviaremos los **tokens** que hayamos guardado previamente. ¿Y qué ocurre si por lo que sea no enviamos las credenciales correctamente? Bueno pues normalmente el servidor nos contestará con un código de error tipo 401. Podemos detectar esos errores en este **interceptor**. Para ello aprovecho la canalización que tenía y meto ahí otro operador de **RxJS** con el cual más o menos deberíamos estar familiarizados que es el **CATCHERROR**.

El operador **CATCHERROR** igual que la función **THROWERROR** vienen pues de **RxJS** y me permiten detectar cualquier error y relanzarlo él mismo o una mutación de él. Hacer esto es como un **TRY-CATCH** en el cual dentro del **CATCH** vuelvo a lanzar el mismo error. No tiene demasiado sentido si no hacemos algo más. Y lo que voy a hacer aquí de algo más es comprobar si este error tiene un código de estado 401 que significaría que el usuario no está autorizado.

¿Qué podemos hacer en esa situación? Bueno, por ejemplo, forzarle a que vaya a la página de **login**. Para ello voy a reclamar aquí también una dependencia al **router** de **Angular** y utilizarlo para navegar, en este caso a **auth**, **login**.

Por ahora no estoy utilizando ningún API que esté protegida y que me devuelva este código de error, pero todo llegará. Solamente es para que veas cómo un **interceptor** puede modificar la petición o procesar la respuesta, por ejemplo, en este caso, para la situación de un error. Es interesante que veas que siempre acabo retornando el error porque otros en la cadena podrían querer hacer uso de él. Aquí la única particularidad es que redirijo al **login** al usuario cuando no está autenticado, pero el **interceptor** vale para más cosas y debe permitir que el flujo continúe.

8.2.3 Guardias de navegación

Puede que te preguntes si hay algo más que **interceptores** para el tema de la seguridad en **Angular**.

Y la respuesta es que sí, claro que sí.

Tenemos también lo que consideraremos **guardias**.

Las **guardias** se generan con la palabra clave **guard** o simplemente la abreviación **g**, con lo cual nos queda así de simpático **ng g**.

Pero una **guardia** se parece bastante a un **interceptor** en el hecho de que también es una función que tiene una firma un tanto peculiar. En la práctica, lo que tiene que hacer una **guardia** es retornar cierto si el usuario puede acceder a una página y falso o una ruta alternativa en caso de que no pueda acceder a esa página. Es decir, actuarán antes de que la navegación se complete de una ruta a otra pasando por aquí.

Nos dan dos argumentos que podríamos utilizar: la ruta actual a la que queremos ir y el estado actual del **router**. Si tuviese interés lo usaríamos, pero yo por ahora voy a hacer una cosa más sencilla. Simplemente preguntarle a mi **store** si el usuario está autenticado o no. En ese caso retornaré **true** y podrá navegar, pero en caso de que no lo esté, pues haremos algo similar a lo que hacíamos con el **interceptor** y es redirigirle amablemente a la página de **login**. Es decir, esta es una protección de navegación.

En cierto modo es muy adecuada para los usuarios en cuanto a **feedback**, porque no le deja visitar páginas que no deberían poder, pero en la práctica tampoco es lo más seguro del mundo. La seguridad siempre debe recaer del lado del servidor en las **APIs**. Aquí esto no es más que algo que impide temporalmente visitar una página.

¿Y cómo lo hace? Tal como para registrar el **interceptor** hemos tenido que venir al **app.config**, para registrar una **guardia** hay que ir al **app.routes** y simplemente a una página que nos interese. Por ejemplo, voy a hacer que para poder hacer una reserva sobre una actividad previamente tengamos que estar autorizados o digamos autenticados. Y esto se hace simplemente agregándole aquí la propiedad **canActivate**, que lo que dice es si puedo activar o no esta ruta, y le paso la función que lo determina. Si esta función devuelve **true** podrá activarse, si no lo devuelve pues obviamente no lo hará.

El hecho de que pueda activarse o no ya lo decidí durante la generación. No te quería liar en ese momento, pero aquí cuando generamos una **guardia** le decimos que implemente uno de los supuestos de control de acceso a una ruta, que puede ser activar, incluso también la salida, no tiene tanto que ver con la seguridad sino con si ha guardado o no ciertas cosas, no vamos a entrar ahí.

Simplemente quiero que sepas que estas **guardias** se ejecutan antes o después de una navegación para prevenir que el usuario pueda llegar a donde no debe o salir sin haber hecho las cosas como debe.

A partir de ahora, al ejecutar la aplicación, si intentamos visitar cualquiera de las actividades nos redirige automáticamente a **Login** porque realmente no estamos registrados. Si nos registramos y ya disponemos de un token, ahora sí podemos

visitar las páginas. ¿Por qué? Porque ha preguntado al **Store** si disponíamos de un **accessToken** y le ha dicho que sí, que tenemos este.

Es justo lo que preguntamos aquí. En este está autenticado, comprobamos que el **accessToken** no es nulo.

Bueno, pues ya hemos visto entonces que las **guardias** protegen la navegación y los **interceptores** protegen las comunicaciones con el **API**. Además de estos dos artificios, tenemos también a nuestra disposición la oportunidad de usar **resolvers**.

No soy muy fan, no me encantan los **resolvers**, pero como mínimo para que veas lo que hacen te voy a enseñar uno.

Un **resolver** se ejecuta también antes de cada navegación. La generación es muy sencilla, simplemente ponerle la palabra **resolver** o la inicial **R** y una vez creado es de nuevo una función que habrá que asignar a una ruta. En este sentido es igual que lo que habíamos hecho con las **guardias**.

Este **resolver** de aquí, lo que me va a permitir es, dada una ruta, agregarle aquí una entrada **resolve** que dice ejecuta esta función y con su resultado relléname un campo, el que sea. Por ejemplo, yo donde lo quiero utilizar es en la página **bookings.page**, la cual requiere para empezar que a partir del **Slug** obtengamos la actividad actual, la que le toque a este usuario y para eso tiene que hacer todo este cambio de **Slug** a **getActivityBySlug**, común **observable**, etc.

Bueno, pues todo eso se puede programar dentro de un **resolver**, de forma que a partir de los datos de la ruta obtenga el **Slug**, use un servicio y retorne el **observable** a ese servicio. Obviamente, como aquí lo que vendrá será una actividad, tengo que ajustar el tipo de respuesta esperado pero lo que quiero que veas es que este **resolver**, una vez importado, se ejecutará cada vez que viajemos a esta página y él sólo rellenará esta propiedad. Y en la página ya no necesitaremos rellenar la actividad a partir de este **toSignalMap** sino que ahora podremos incorporar también un puntero a la ruta activada que nos permitirá acceder a sus datos.

Esto lo haré, por ejemplo, guardando la actividad resuelta, **resolvedActivity**, obtenida a partir de la instantánea que se haya guardado en la propiedad **activity**. Esta propiedad de aquí es esta propiedad de aquí. Y no tengo que suscribirme a pesar de que el **resolver** devuelva un **observable**, estará resuelto.

Y por tanto este dato ya estará directamente utilizable, no hay que esperar por él, puedo acceder a él a su **snapshot**. Esta actividad, si la quiero transformar en una señal, puedo hacerla de una manera bastante más sencilla de lo que tenía y sustituye a todo esto.

Bueno, efectivamente he transferido cierta complejidad de la página a una función especializada en obtener datos para esta página, pero al mismo tiempo, al apartar las cosas de un sitio para otro y obligarme a utilizar **ActivatedRoute** como una dependencia en lugar de los parámetros que venían ya automáticamente como input, pues no siempre me sale muy a cuenta y por tanto ya te digo que yo no soy muy fan de ellos.

Eso sí, mereces conocerlo. Eso sí, merece ser conocido y que tú lo evalúes.

8.3. Presentación de feedback al usuario

Un tema final que tiene que ver no tanto con la cuestión de seguridad que estamos tratando respecto a los datos, sino con la seguridad que le damos a los usuarios de que las cosas que hacen van bien o si van mal, que lo entiendan y lo sepan lo antes posible, es el hecho de darle **feedback**, sobre todo de todas las operaciones que sean asíncronas, como son las llamadas al **API**, que pueden demorar y habría que explicarle que estamos, bueno, pues trabajando en ello, o que han terminado bien o se han terminado con un error, pues mostrárselo también. Hay muchas maneras de hacer esto en **Angular**, pero ya que estamos en un curso de introducción, como mínimo vamos a tener la buena práctica de generar un componente que nos provea de este **feedback** de una manera homogénea, de forma que en todas nuestras páginas tengamos siempre el mismo componente y el usuario se acostumbre a ver el error o el estado del proceso siempre de la misma manera. También conseguiremos que, desde los códigos de esta página, pues sean más o menos siempre los mismos, porque lo que haremos será invocar a un **observable**, pero antes notificar que estamos trabajando, suscribirnos para el caso de que lleguen los datos bien o los errores y, bueno, pues que el usuario vea eso de la mejor manera posible, lo antes posible.

Obviamente hay soluciones complejas a todo este tema, incluso algunas que involucran patrones como **Redux**, pero esto es el inicio. Un tema un poquito más complejo ocurre cuando en una página estamos haciendo aquí algo con el usuario, aquí pueden estar ocurriendo otras cosas, pero queremos de alguna manera notificar, quizá en el **footer** o quizá aquí, pues que ha habido algo excepcional. Algo excepcional podría ser no necesariamente malo, quizá una información y a veces sí, puede ser un error que queramos notificar aquí, pero no un error con este proceso en concreto, sino con algo que puede haber ocurrido en cualquier sitio o en segundo plano. Para hacer algo así vamos a necesitar involucrar a varios actores.

Por un lado necesitaré tener algún tipo de **store**, ya hemos visto que esto almacena privadamente un **state** programado con **signals**, de forma que

cualquier elemento puede escribir aquí y cualquier otro elemento pueda leer a partir de él. El problema de escribir es que si queremos escuchar a cualquier tipo de error que se produzca en cualquier sitio deberíamos tener un capturador general. Afortunadamente **Angular** nos da el **error handler**, que enmascararlo, sustituirlo usando inversión de control y darle nuestro propio **custom error handler**. Todo esto se hace dentro de la configuración de los **providers**, donde se dice, oye aprovisióname el **error handler** tuyo usando el mío. Bien, a partir de aquí nuestro **error handler** puede escribir también en este **store** notificando los errores y cualquier otro elemento podría estar escuchando y mostrándolos. Vamos a ver todo esto, sobre todo la parte del **error handler** y repaso del **store**.

8.3.1 Feedback de operaciones

Bueno, pues vamos a terminar con algo que, no siendo estrictamente de seguridad, sí que le aporta seguridad al usuario y es tener **feedback**. **Feedback** inmediato de la validación. Ya lo hemos visto en los formularios y ahora lo veremos con esta estructura tan sencilla para operaciones contra el **API**, de manera que podamos indicarle al usuario si estamos trabajando y si hemos acabado bien o mal e incluso con un mensaje explicativo. Para ello lo que voy a hacer es crear un componente compartido en el cual haré una parte de presentación y otra de lógica donde empezaremos por declarar una propiedad requerida que nos tiene que llegar como entrada, un par de propiedades **computed** como pueda ser el estado para simplemente obtenerlo de forma fácil y el mensaje que puede ser un pelín más complejo. Bueno, en este caso lo único que hago es que si no tuviésemos mensaje pues utilizo el **status** en mayúsculas como algo a mostrar aquí.

Y aquí pues bueno lo que podemos hacer es, aprovechando nuestros conocimientos de la sintaxis declarativa, podemos utilizar un **switch** sobre el estado y para cada una de estas casuísticas mostrar unos textos. Yo aquí lo hago reutilizando inputs en modo **disabled** pero para poder utilizar sus elementos **aria** que me permiten poner colores adecuados para el éxito y para el error en cada una de las situaciones. Toca ahora utilizar este componente en las páginas por ejemplo voy a mostrártelo aquí en la **register page** importándolo como siempre para que esté en el contexto y seleccionándolo para que sea utilizable. Se queja obviamente de que le falta la señal requerida de **feedback** pero para ello tendremos que trabajar un poquito. Como mínimo vamos a crear una propiedad en modo señal con un valor nulo válido inicial y le enviamos su resultado, sea el que sea.

Ahora de lo que se trata es de que aprovechemos esta señal para enviarle los cambios de estado y mensajes adecuados en función de las situaciones en las que estemos. Habitualmente esto será indicar que hemos empezado a trabajar,

que hemos terminado bien o que ha habido algún error. Veámoslo en ejecución. Bueno tengo aquí el formulario de registro listo para ser enviado y eso es lo que voy a hacer, enviarlo y ver inmediatamente que se ha puesto el mensaje de que se ha registrado bien y que todo ha salido bien. Lo he hecho especialmente con el formulario de registro porque si repito, si intento repetir, tendré un problema porque seguramente esto esté repetido. No tengo por qué darle ese **feedback** exactamente al usuario, sino que ha habido un fallo durante el registro. También quiero mostrarte que, si estuviese en una situación de una lentitud exasperante en el **API**, pues veríamos un mensaje de que estamos trabajando en ello hasta que al final termine.

Bueno, **feedback** inmediato para cada operación al usuario.

8.3.2 Notificaciones de errores

Vamos a continuar ahora con el **feedback**. Esta vez, no estamos enfocados tanto en operaciones específicas, sino en aspectos más genéricos que pueden surgir en la aplicación, como las **notificaciones**. Estas las almacenaremos en un **store**. Ya estamos familiarizados con esto, así que no es necesario desarrollarlo desde cero. Básicamente, es una clase **inyectable** con un estado privado en forma de señal **printable**, y una parte **read-only** pública.

Este **store** incluirá selecciones de señales computadas que necesitemos. Por ejemplo, podríamos tener un contador de cuántas notificaciones tenemos pendientes y un método para agregar una notificación que actualice ese estado, o incluso para vaciarlo y limpiar las notificaciones.

En el siguiente punto, quiero detenerme para hablar sobre la generación de una clase que, aunque no será un servicio, llevará ese apellido: **ErrorService**. Esta clase tendrá la particularidad de exigir que implemente una interfaz de **@angular/core**. Esta interfaz obliga a tener un método con una firma específica. Este **servicio** será invocado por el sistema cada vez que ocurra un error.

No solo implementará esta funcionalidad por la interfaz, sino que también necesitaremos configurar algo más. Específicamente, iremos al **app.config** y añadiremos una entrada para realizar lo que se conoce como inversión de control. Aquí indicaremos que cuando **Angular** necesite un **error handler**, debe usar la clase **ErrorService** que acabamos de configurar. Esto nos permitirá utilizar de manera personalizada las notificaciones que hemos creado.

A continuación, agregaremos, como siempre, una inyección al almacén de notificaciones y estableceremos una lógica básica. Por ejemplo, esta podría

permitirnos generar una notificación con un mensaje y un tipo que se ajusten adecuadamente a la situación que se presente.

El siguiente paso será crear un componente que consuma las notificaciones desde este **store**. A este componente le crearemos una propiedad pública **input**, donde recibiremos el array de notificaciones. Para su visualización, incorporaremos un diálogo que presentará el listado de notificaciones, mostrándolas con una apariencia que variará según su tipo.

Para cerrar el diálogo, continuaremos con el enfoque basado en señales y simplemente usaremos un **output** que se activará durante el clic en el botón de cerrar. Este componente de notificaciones lo integraremos y utilizaremos dentro del **footer.widget**. Todo lo que necesitamos es un lugar desde el cual lanzarlo como un modal, que es lo que ocurrirá con este diálogo cada vez que sea necesario.

Como mínimo, necesitaremos acceso al **notificationsStore** para saber si tenemos notificaciones. Esto lo verificaremos accediendo directamente al estado de ese **store** o usando alguna computación que tengamos configurada. Tendremos nuestra propia señal local para gestionar cuándo mostrar u ocultar las notificaciones, dependiendo de si están siendo visualizadas y necesitamos ocultarlas cuando ya no sean relevantes.

Podríamos empezar con algún tipo de condicional **IF** para cuando tengamos notificaciones pendientes. Quizás podríamos añadir un botón que alerte al usuario de que hay algo que necesita revisar, y que en el **Tooltip** ya muestre cuántas notificaciones están pendientes de ser vistas.

El componente de notificaciones, una vez añadido al contexto del footer, se utilizará si es necesario mostrar las notificaciones. De no ser necesario, no se mostrará. Este botón, que activará las notificaciones, estará configurado para que cuando el usuario haga clic, las notificaciones se muestren y se oculten al cerrar.

Necesitaremos un par de métodos para esto: uno cambiará el estado de visualización de mostrar a no mostrar con una simple actualización inversa. Este **toggle** de notificaciones se vinculará al clic del botón. Cuando las notificaciones se cierren, necesitaremos otro método que las oculte y también las vacíe para que no se muestren como notificaciones no leídas sin resolver.

Este último paso lo vincularemos al evento de cierre. Así, el componente de notificaciones llamará a **onNotificationsClose**. Vamos a ver si esto funciona. La mejor forma de probar las notificaciones es provocando un error para que estas se muestren según se generen, apareciendo y desapareciendo en función de las acciones del usuario.

Todo esto es posible gracias a la capacidad de usar un **store** para almacenar información que se propaga desde cualquier lugar, especialmente desde un servicio que es provisto usando la inyección de dependencias con inversión de control. Aunque suene a un patrón de diseño de alto nivel, es simplemente una manera de decirle a **Angular** que use nuestro **error handler** personalizado, que a su vez notificará y mostrará adecuadamente las notificaciones. Y con esto, tenemos una aplicación funcional que notifica al usuario todo lo que va ocurriendo.