DWC

(Desarrollo Web en entorno cliente)

Índice

1.- Introducción

Los **observables** son una de las herramientas clave en Angular para crear aplicaciones reactivas y robustas. Permiten trabajar con flujos de datos asíncronos de manera eficiente y flexible.

1.1- ¿Qué es la Programación Reactiva?

La Programación Reactiva es un paradigma de programación asíncrona que se centra en el flujo de datos y la propagación del cambio. En lugar de pensar en términos de procedimientos secuenciales, la programación reactiva se basa en la idea de flujos de datos continuos que se transforman y se combinan a lo largo del tiempo.

Las características principales son:

- **Asíncrona:** La programación reactiva se basa en llamadas asíncronas no bloqueantes, lo que permite un mejor uso de los recursos del sistema y una mayor escalabilidad.
- **Orientada a eventos:** Los programas reactivos se basan en la emisión de eventos que representan cambios en el estado de la aplicación. Estos eventos se propagan a través de flujos de datos, lo que permite una respuesta rápida a los cambios.
- **Declarativa:** En la programación reactiva, se define qué es lo que se quiere obtener, no cómo se debe obtener. Esto facilita la escritura de código modular y reutilizable.
- **Componibilidad:** Que puede componerse o formarse uniendo varios elementos. Los flujos de datos en la programación reactiva se pueden combinar y componer fácilmente, lo que permite crear aplicaciones complejas a partir de componentes más simples.

Los beneficios del uso de observables son varios:

- **Mejora la capacidad de respuesta:** Las aplicaciones reactivas son más sensibles a los cambios en el estado del sistema y en la entrada del usuario.

- **Facilita la escalabilidad:** La naturaleza asíncrona de la programación reactiva facilita la escalabilidad de las aplicaciones a grandes conjuntos de datos y cargas de trabajo.
- Simplifica el desarrollo: La programación reactiva puede ayudar a simplificar el desarrollo de aplicaciones complejas, especialmente aquellas que manejan una gran cantidad de datos dinámicos.

1.2.- ¿Qué es RxJS?

RxJS es una biblioteca de JavaScript para la programación reactiva. Se basa en el **patrón Observador** y proporciona un conjunto de herramientas para trabajar con flujos de datos asíncronos.

Las características principales son:

- **Implementación de Observables:** RxJS proporciona una implementación del patrón Observador en JavaScript. Los Observables son objetos que representan flujos de datos asíncronos y pueden ser utilizados para crear aplicaciones reactivas.
- **Operadores:** RxJS ofrece una amplia gama de operadores para transformar, combinar y filtrar flujos de datos. Estos operadores permiten escribir código conciso y expresivo para manejar tareas complejas de forma asíncrona.
- **Soporte para múltiples plataformas:** RxJS está disponible para JavaScript en el navegador y en Node.js, y también hay implementaciones para otros lenguajes de programación.

Los beneficios del uso de RxJS son varios:

- **Simplifica el desarrollo de aplicaciones asíncronas:** RxJS proporciona un marco de trabajo consistente para trabajar con flujos de datos asíncronos, lo que facilita el desarrollo de aplicaciones reactivas.
- **Mejora la capacidad de respuesta y la escalabilidad:** Las aplicaciones reactivas son más sensibles a los cambios en el estado del sistema y en la entrada del usuario, y también son más fáciles de escalar a grandes conjuntos de datos y cargas de trabajo.

- **Código más modular y reutilizable:** RxJS permite escribir código modular y reutilizable para manejar tareas asíncronas.

RxJS es una biblioteca poderosa para la programación reactiva en JavaScript. Permite crear aplicaciones asíncronas con mayor facilidad, capacidad de respuesta y escalabilidad. Si desarrollamos aplicaciones complejas que manejan una gran cantidad de datos dinámicos, RxJS es una herramienta casi imprescindible para el desarrollo de aplicaciones Web modernas.

1.3.- ¿Qué son los Observables?

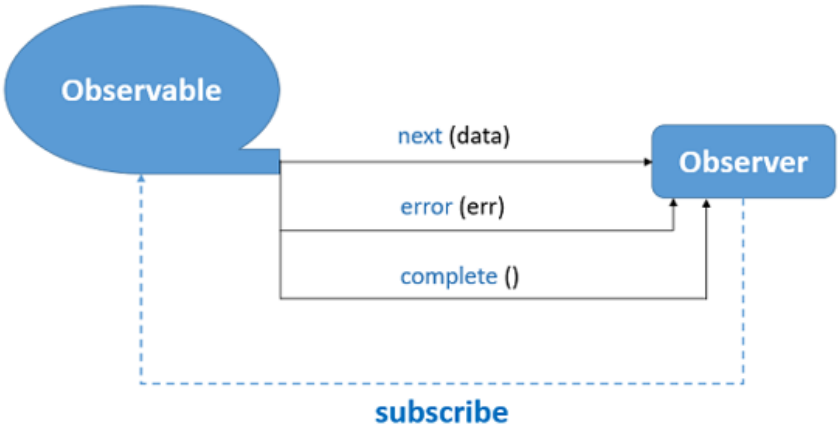
En la programación reactiva, **un Observable es un objeto que representa un flujo de datos asíncrono**. Los Observables emiten valores a lo largo del tiempo, y los programas pueden "suscribirse" a estos Observables para recibir y procesar los valores emitidos.

Imaginemos que estamos observando una cascada. La cascada es un flujo continuo de agua que cae sin parar. Podemos suscribirnos a la cascada para recibir notificaciones cada vez que una gota de agua cae. De esta manera, podemos procesar las gotas de agua de una en una, sin necesidad de esperar a que toda la cascada caiga.

De la misma manera, un Observable es como una cascada de datos:

- Los datos fluyen continuamente a través del Observable.
- Podemos suscribirnos al Observable para recibir notificaciones cada vez que un nuevo valor de datos está disponible.
- Podemos procesar los valores de datos uno por uno, sin necesidad de esperar a que todo el flujo de datos termine.

Los observables disponen de tres tipos de notificaciones que se utilizan para comunicar diferentes estados del flujo de datos. Estas notificaciones son **next, error y complete**



Next:

- Es la notificación más común y representa la entrega de un nuevo valor al observador.
- Puede ocurrir varias veces durante la vida útil del observable.
- Cada vez que se emite un valor, se llama a la función next del observador.

Error:

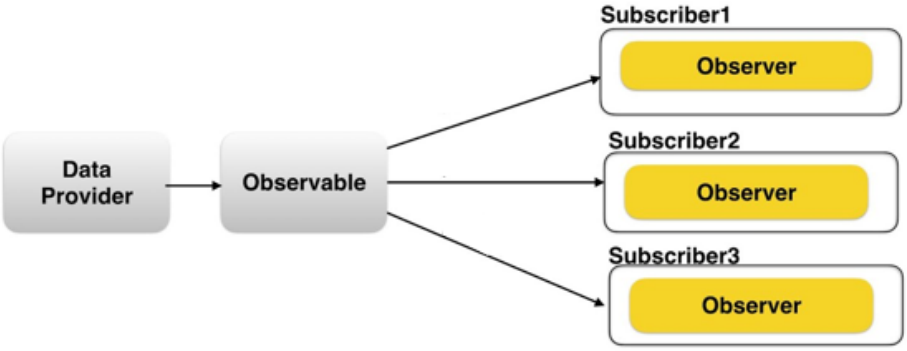
- Indica que ha ocurrido un error durante la ejecución del observable.
- Solo puede ocurrir una vez por observable.
- Cuando un observable produce un error, se llama a la función error del observador.

Complete:

- Indica que el observable ha terminado de emitir valores.
- Solo puede ocurrir una vez por observable.
- Cuando se completa un observable, se llama a la función complete del observador.

1.4.- Patrón de diseño Observer

El patrón de diseño Observer, es un patrón de diseño de software que define una **dependencia uno a muchos entre objetos**. En este patrón, un objeto, conocido como **Sujeto (Subject)**, mantiene una lista de objetos dependientes, conocidos como **Observadores (Observers)**. Cuando el estado del sujeto cambia, este notifica a todos los observadores de este cambio.



Imaginemos un periódico que publica noticias. Los lectores del periódico son como los observadores. Cuando el periódico publica una nueva noticia, notifica a todos los lectores que se han suscrito a él. De esta manera, los lectores se enteran de las nuevas noticias sin tener que consultar el periódico constantemente.

De la misma manera, el patrón Observer funciona de la siguiente manera:

- **Sujeto:** El sujeto es el objeto que cambia de estado. Puede ser cualquier objeto, como un modelo de datos, un componente de interfaz de usuario o un servicio.
- **Observadores:** Los observadores son los objetos que están interesados en los cambios del sujeto. Pueden ser cualquier objeto, como otro componente de interfaz de usuario, un controlador o una clase de utilidad.
- Notificación: Cuando el estado del sujeto cambia, este notifica a todos los observadores de este cambio. La notificación puede incluir información sobre el cambio, como el nuevo valor del estado o el tipo de cambio que se ha producido.

Las ventajas del uso de observables son varias:

- **Acoplamiento débil:** El patrón Observer desacopla los sujetos de los observadores. Esto significa que los cambios en el sujeto no afectan directamente a los observadores.
- **Flexibilidad:** El patrón Observer es flexible y se puede utilizar en una variedad de situaciones.
- **Escalabilidad:** El patrón Observer es escalable y se puede utilizar con un gran número de observadores.

También tiene una serie de desventajas:

- **Complejidad:** El patrón Observer puede ser complejo de implementar, especialmente si hay muchos observadores.
- **Overhead:** Overhead es una sobrecarga de tiempo de cálculo, memoria, ancho de banda u otros recursos excesivos o indirectos que se requieren para realizar una tarea específica. El patrón Observer puede añadir un overhead de rendimiento, especialmente si hay muchos cambios en el estado del sujeto.

2.- Observables en Angular

Ya hemos visto que mediante los observables podremos mantener comunicación entre varios componentes utilizando un subject que será observado por todos aquellos que se quieran suscribir a él.

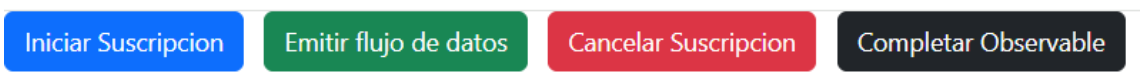
En Angular podemos utilizar los observable de diferentes maneras, lo más habitual suele ser:

1.- Usar objetos y clases de Angular que ya llevan implementados observables, en este caso deberemos hacer uso de ellos como están implementados, es decir utilizaremos objetos ya predefinidos con observables. Este es el caso de las clases HttpClient, Router y librerías UI como Angular Material o Ng-bootstrap

2.- La otra forma es que seamos nosotros los que implementemos objetos de tipo observable. Esto es útil cuando queremos mantener una comunicación entre varios componentes que no mantienen una relación de padre-hijo.

Para ver el funcionamiento de los observables vamos a crear un componente que utilice un observable al subject mensaje y que permita realizar todas las operaciones que dispone un observable.

Para ello en la vista del componente definiremos la siguiente interface:



2.1.- Creación de Observables

Para utilizar los observables en Angular **deberemos hacer uso de la librería RxJS**. Tendremos que importar la librería y utilizar los objetos predefinidos para trabajar con observables.

Para crear los observables utilizaremos las clases **Subject** o **BehaviorSubject**.

- **Subject:** Es un tipo de observable que permite emitir valores y recibir notificaciones de nuevos valores o eventos. Actúa como un canal de comunicación bidireccional entre emisores y observadores.

```
//Importamos clase de la librería RxJS

import { Subject } from 'rxjs';

//Dentro de la clase definimos el subject, en nuestro caso una variable llamada mensaje de tipo string

mensaje:Subject<string>=new Subject()
```

- **BehaviorSubject:** Es un tipo especial de Subject que, además de emitir valores, guarda un valor actual. Cuando un nuevo observador se suscribe, recibe el último valor emitido.

```
//Importamos clase de la librería RxJS

import { BehaviourSubject } from 'rxjs';

//Dentro de la clase definimos el subject, en nuestro caso una variable llamada mensaje de tipo string

mensaje:BehaviorSubject<string>=new BehaviorSubject('Valor inicial')
```

Las diferencias son las siguientes

- **Subject:** No guarda un valor actual. Los nuevos observadores solo reciben valores emitidos después de suscribirse. No puede inicializar con un valor
- **BehaviorSubject:** Guarda un valor actual. Los nuevos observadores reciben el último valor emitido, incluso si se suscriben después de que se haya emitido. Podemos inicializar con un valor

En nuestro ejemplo utilizaremos BehaviorSubject ya que nos permitirá inicializar el observable con un valor

2.2.- Emisión de información

Una vez creado el observable podremos empezar a emitir la información para ello haremos uso del método **next**.

Para ello definiremos en nuestro componente el método emitir, que cada vez que se ejecute emitirá información por el observable

```
emitir(){
    console.log('Emitimos valores por el observable')
    this.mensaje.next("Valor 1")
    this.mensaje.next("Valor 2")
    this.mensaje.next("Valor n")
}
```

Este método lo asignaremos al botón “emitir flujo de datos”. Una vez que tenemos creado el observable y emitimos información esta información estará disponible para cualquier observador que se haya suscrito.

2.3.- Suscripción a Observables

Para suscribirnos a un observable deberemos hacer uso del **método subscribe.**

Para ello en nuestro componente definiremos el método suscribir que nos permitirá crear una suscripción al observable y poder recibir la información que se emita.

Este método lo asignaremos al botón “Iniciar Suscripción”

```
suscribir(){
 console.log('Nos suscribimos. Recibiremos todos los datos que se
emitan')
 this.mensaje.subscribe({
      next: dato   => console.log(dato),
      error: error => console.error(error),
      complete: () => console.log('Observable completado')
    })
}
```

Con el método subscribe nos subscribimos al observable y recibimos un objeto con los tres estados del observable, cada uno de ellos implementa una fat arrow para gestionar cada estado.

En versiones anteriores, se puede utilizar la suscripción y gestionar los estados del observable con una fat arrow para cada estado y no hace falta utilizar un objeto. Esta forma aún se puede utilizar, pero nos indica que esta **deprecated**.

```
suscribir(){
  console.log('Nos suscribimos. Recibiremos todos los datos que se
emitan')
  this.mensaje.subscribe(dato  => console.log(dato),
                         error => console.error(error),
                            () => console.log('Observable completado')
                        )
}
```

Desde el momento que nos hemos suscrito al observable ya estaremos recibiendo la información que se emita. **Nada más suscribirnos ya tendremos disponible el ultimo valor emitido por el observable**.

2.4.- Cancelación de una suscripción

Un observador puede cancelar la suscripción en cualquier momento. En ese caso dejaría de observar al subject

Para poder cancelar la subscripción deberemos de utilizar el **método unsubscribe**.

Si vamos a utilizar la cancelación deberemos modificar la forma de suscripción, **tendremos que utilizar una variable en la cual se almacenara la referencia del observable** que vamos a utilizar, luego la cancelación se realizara sobre esa variable.

Con lo cual añadiremos en el componente una variable llamada miSuscripción

```
miSuscripcion:any
```

Ahora modificaremos el método suscribir para que haga uso de la variable. Lo que haremos es lanzar la suscripción como antes, pero guardando el valor de referencia en la variable miSuscripción que acabamos de crear, de esta manera podremos hacer referencia a la suscripción en el método cancelar.

```
suscribir(){
 console.log('Nos suscribimos. Recibiremos todos los datos que se
emitan')
 this.miSuscripcion=this.mensaje.subscribe({
      next:  dato  => console.log(dato),
      error: error => console.error(error),
      complete: () => console.log('Observable completado')
    })
}
```

Ahora implementamos el método cancelar que será el que cancele la suscripción. Este método lo asignaremos al botón “Cancelar Suscripción”

```
cancelar(){
 this.miSuscripcion.unsubscribe()
 console.log('Cancelacion realizada. No se reciben mas datos')
}
```

Desde el momento que **un observador cancele la suscripción ya no recibirá más información** del observable. Los demás observadores que sigan suscritos sí que seguirían recibiéndola

2.5.- Finalización del observable

El observable también puede completar su transmisión de información, es decir dejará de emitir información porque habrá finalizado. Para finalizar un observable utilizaremos el **método complete**.

Vamos a implementar en nuestro componente el método finalizar que finalizara con la emisión de información del observable

```
finalizar(){
 this.mensaje.complete()
 console.log('Observable completado. Ya no se emiten mas datos')
}
```

En este momento **el observable se ha completado y ya no se emitirán más datos, con lo cual todos los observadores ya no recibirán ninguna información**. Además, al finalizar la emisión de datos **a los observadores suscritos se les ejecutara el estado complete de la suscripción**.

Hemos visto el funcionamiento de los observables creando nosotros los objetos y métodos necesarios para su uso. Los observables los utilizaremos más adelante en objetos de Angular como el **Router** o el **HttpClient.**

En esos casos lo que haremos es suscribirnos a los observables que nos facilitarán los objetos de Angular para obtener la información necesaria que emitirán

3.- Comunicación entre componentes con Observables

No tiene gran utilidad utilizar observables para observar a un subject que está definido en el mismo componente.

Los observables son un mecanismo de comunicación entre componentes, para ello se utiliza un servicio para que todos los componentes tengan acceso al mismo subject y no repetir código innecesario.

- **En el servicio** definiremos el subject y el observable con los métodos para notificar los cambios y para completar el observable.
- **Los componentes** inyectarán el servicio y se suscribirán al servicio. También accederán para hacer notificaciones sobre el observable y que se reflejen en los demás componentes.

Los componentes al estar suscritos reflejaran los cambios del subject de forma inmediata