**Curso de programación reactiva RxJs.**

1. **Sección 2: Introducción a las extensiones reactivas y a la programación reactiva.**
2. **Sección 3: Observables.**
3. **Sección 4: Funciones para crear Observables.**
4. **Sección 5: Operadores básicos.**
5. **Sección 6: Operadores no tan comunes.**
6. **Sección 7: Operadores que trabajan con tiempo.**
7. **Sección 8: Ajax - Peticiones ajax usando RxJs/ajax.**
8. **Sección 9:** Operadores de transformación.
9. **Sección 10: Operadores y métodos de combinación de  
   observables.**
10. **Sección 11: Ejercicios de reforzamiento.**
11. **Sección 12: Fin del curso.**

**Sección 2: Introducción a las extensiones reactivas y a la programación reactiva.**

En esta sección aprenderemos sobre:

* Conceptos generales de RxJs y las extensiones reactivas
* ¿Qué es ReactiveX?
* ¿Cómo funciona?
* ¿Qué es el patrón observable?
* ¿Cómo leer los diagramas de canicas?
* Y otros temas

Es una sección corta llena de teoría, pero es indispensable que lo aprendamos para poder tener un mejor panorama de la programación reactiva.

**Video #07: conceptos generales.**

En esta clase empezamos con la pregunta “¿por qué usar extensiones reactivas?”, la respuesta que nos dan es porque desaseamos información en tiempo real, hoy en día no es viable para los usuarios hacer una recarga en la página para obtener información de manera de tiempo real sin que el usuario deba interactuar con un botón para obtener esa información.

¿En qué momento es correcto utilizar extensiones reactivas?

* Eventos de interfaz de usuario.
* Cuando es necesario notificar sobre cambios en un objeto.
* Comunicaciones por sockets.
* Cuando necesitamos trabajar con flujos de información.

Los pilares de la programación reactiva son los Observables Subscribers Operators.

**Observables:**

* Son la fuente de información.
* Pueden emitir múltiples valores, sólo uno o ninguno.
* Pueden emitir errores.
* Pueden ser infinitos o finitos. (completarse)
* Pueden ser síncronos o asíncronos.

**Subscribers**

* Se subscriben a un observable, es decir, estar pendiente de lo que realizar el observable.
* Consumen / observan la data del observable.
* Pueden recibir los errores y eventos del observable.
* Desconocen todo lo que se encuentra detrás del observable.

**Operators**

* Usados para transformar Observables (map, group, scan).
* Usados para filtrar observables (filter, distinct, skip, debounce).
* Usados para combinar observables.
* Usados para crear nuevos observables.

**Video #08: ¿Qué es ReactiveX ?,¿Como funciona?**

ReactiveX funciona con las tres siguientes bases, el patrón observador, el patrón iterador y la programación funcional.

**Observer Pattern**: es un patrón de diseño de software que define una dependencia del tipo de uno a muchos entre objetos, de manera que cuando uno de los objetos cambia de estado, notifica este el cambio a todos los dependientes.

**Iterador Pattern:** En POO, el patrón iterador define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección.

**Programación funcional:** Es básicamente, crear un conjunto de funciones que tengan un objetivo específico. Es decir, si tengo una función que reciba ‘A’ y retorna ‘A+1’, siempre que yo llame esa función retornara ‘A+1’.

Sin efectos secundarios, sin mutar la data.

**Video #09: ¿Cómo leer los diagramas de canicas?**

Para identificar un observable se usa el signo de dólar. Esto es un estándar para identificar observables en nuestro código.

Este observador trabajará con una línea de tiempo donde se representará en canicas los valores emitidos por este. diagrama canicas




**Nota:** Al momento de trabajar con extensiones reactivas es recomendable que la librería este en las dependencias de desarrollo.

**Sección 3: Observables.**

Estamos a las puertas de la sección que nos enseñará sobre los observables, el primer eslabón de la programación reactiva, veremos temas como:

1. Observers
2. Subscriber
3. Unsubscribe
4. Subjects
5. Hot y Cold Observables
6. Observables en cadena
7. Y más

Antes de utilizar funciones que crean observables, es importante comprender las bases para entrar a temas de la generación automática de observables.

**Video #13: Nuestro primer observable.**

Podemos crear un observable de la siguiente forma:



este es el ejemplo de un observable, podemos ver que para identificar que es un observable utilizamos el signo de dólar, dentro de los observables existe un objeto llamado Subscriber, este permite crear suscripciones que nos permitiría estar al pendiente de las emisiones de nuestro observable.

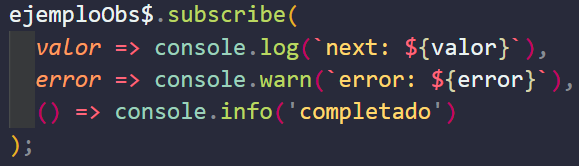
Para ejecutar un observable debemos suscribirnos a él para que nos notifique los cambios de nuestro observable.



Con el Subscriber.next() podemos emitir información a nuestras suscripciones, y con Subscriber.Complete() detenemos las emisiones de nuestro observable.

**Nota:**  es recomendado que nuetros obserbable debe tener la expecificacion del tipo de datos que fluyen en el.

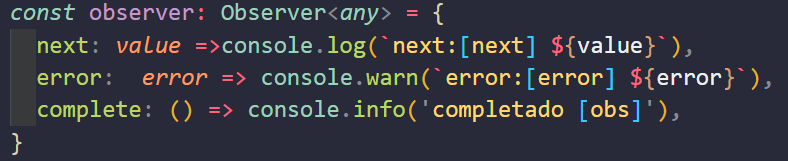
**Video 14#: Observer y Subscriber.**



Existe tres posibles funciones que puede recibir nuestro Subscriber, la primera, es la respuesta correcta de nuestro observable es esta el que procesa el next del observable, la segunda, es el posible error de nuestro observable a la hora de ejecutarse, la tercera una función que no recibe ningún parámetro y notifica que el observable ya termino su ejecución.

Estas funciones son los argumentos de nuestro Subscriber.

Existe otra forma de recibir los datos de nuestro observable y es crear un objeto que obtenga nuestros argumentos. Ejemplo:



Me parece que esta forma es un poco más ordenada y ocupa mucho menos código en nuestro Subscriber ya que solo debemos enviar el objeto como parámetro.

**Video #16: Subscription y Unsubscribe.**

Para realizar una suscripción a un observable debemos llamar a nuestro observable y agregarle el método subscribe. Ejemplo

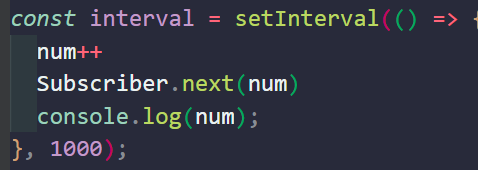


Para cancelar nuestra suscripción de nuestro observable es un tanto diferente, pues debemos crear una variable que alojara nuestra suscripción, esta variable debe ser de tipo Subscription, una vez realizada este proceso a nuestra variable la invocamos junto al método Unsubscribe. Ejemplo:





Primero guardamos el dato y después lo llamamos y le pasmos el método

**Nota:** conozco una nueva función de tiempo llamada ‘**setInterval**’ la cual se ejecuta su contenido cada espacio de tiempo que nosotros especifiquemos. Ejemplo: 

Y podemos terminar su ejecución de la siguiente forma



**Video #18 y #19: Subject part-1 y Subject part-2.**

Esta clase la iniciamos con una pregunta a resolver la cual es, que pasa cuando tenemos más de una suscripción a un observable y deseamos que los datos que arroje a cada suscripción sean los mismos.

Es aquí cuando llega el Observer **Subject** para solucionar nuestros problemas, ya que normalmente los Observer simples generar una ejecución independiente para cada suscripción, el Subject nos permite generar multidifusión lo que facilita para tener los mismos datos en cada una de nuestras suscriciones, en resumen, el Subject permite que cada una de nuestras suscripciones emita los mismos datos.

Para más información puede consultar la documentación <https://rxjs-dev.firebaseapp.com/guide/subject#subject>.

características del Subject

1- casteo múltiple: se refiere que muchas

suscripciones van estar sujetas al observable

y servirán para distribuir la misma data a

cada una de ellas

2- También es un Observer

3- maneja Next, Error, Complete

**Cold Observable:** cunado la data es producida por el mismo observable.

**Hot Observable:** Cunado la data es producida fuera del observable.

**Sección #4: Funciones para crear Observables.**

¡Es momento de utilizar funciones para crear observables!, esto nos ayudará a reducir mucho el tiempo a la hora de crear nuevos observables.

Los temas principales de la sección son:

* of
* fromEvent
* interval
* timer
* asyncScheduler

Mas adelante veremos otras funciones que crean observables, pero por ahora enfoquemos nuestra atención en las básicas y más comunes.

**Video # 23:** **OF**.

La función of devuelve un observable en base a un listado de elemento.

Este operador va emitir los valores uno por uno de manera asíncrona, cuando termina el último valor se finaliza la ejecución del observable.

**Nota**: cada argumento debe ir separado por una coma, en el caso que nuestro argumento sea un arreglo podremos usar el operador spread lo que nos va a desintegrar dicho argumento.



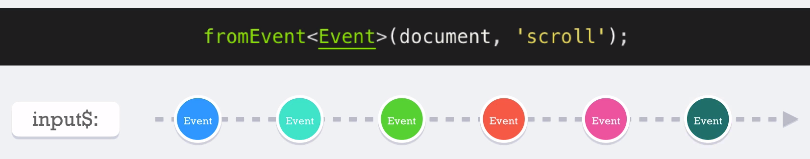
En resumen, el OF emite un observable por cada argumento que se le ingrese.

pero es recomendable tener un tipado estricto de datos.

Para mayor información es recomendable que ingrese a la documentación.

[**https://rxjs-dev.firebaseapp.com/api/index/function/of**](https://rxjs-dev.firebaseapp.com/api/index/function/of)

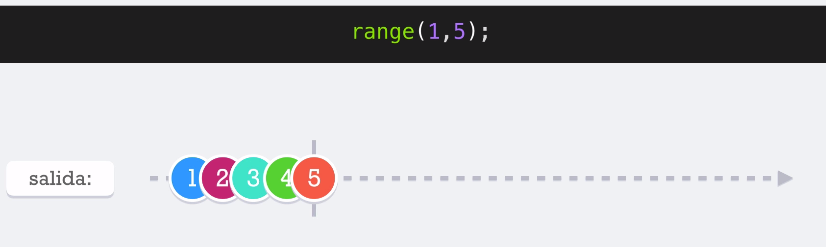
**video #24: fromEvent.**



En FromEvent emite eventos en forma de observables lo que permite suscribirnos a sus cambios y controlar la data que recibe las suscripciones. Para mayor información entra a la documentación oficial.

<https://rxjs-dev.firebaseapp.com/api/index/function/fromEvent>.

**Video #25: Range.**

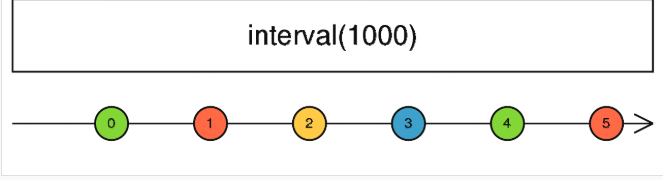


Range nos crea un observable que emite una secuencia de números en base a un rango. Que por defecto son síncronos, pero estos también pueden ser asíncronos, mediante otro proceso llamado scheduler.

Este observable recibe dos argumentos obligatorios, el primero, la primera posición del rango que por defecto esta es 0, el segundo, luego cuantos elementos se desea mostrar. El ultimo es un argumento opcional el cual es el scheduler.

**Video # 26: Interval y timer.**

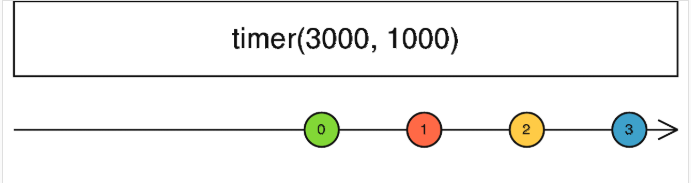
Crea un observable cada intervalo de tiempo especificado, en resumen, por ejemplo, enviamos por parámetro un segundo, apenas se complete ese segundo comenzara a emitir una secuencia de números accedente cada intervalo de tiempo asignado.



Para más información consulte la documentación:  
 <https://rxjs-dev.firebaseapp.com/api/index/function/interval>.

**Timer:**

Esta función, aunque muy parecida a al interval su funcionamiento es diferente, porque, aunque emita valor en un tiempo determinado a esta función le diremos en cuanto tiempo comience emitir y cada cuando queremos que emita nuestro valor.



Para más información ingrese a la documentación oficial

[**https://rxjs-dev.firebaseapp.com/api/index/function/timer**](https://rxjs-dev.firebaseapp.com/api/index/function/timer)

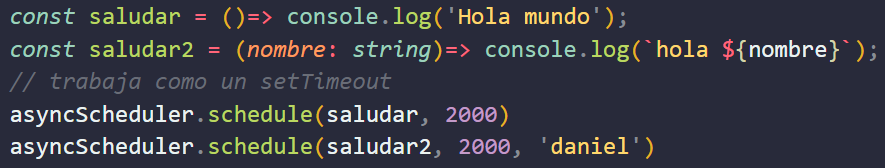
**Nota:** Recordemos que estas funciones se activan o ejecutan tan pronto el stock de col backs de JavaScript lo permita.

**Video #27: timer – configuraciones especiales.**

En esta clase nos mostró algunas posibles utilizaciones del timer, para más información diríjase a observables -> 07- interval&Timer, en la cual está el ejercicio realizado.

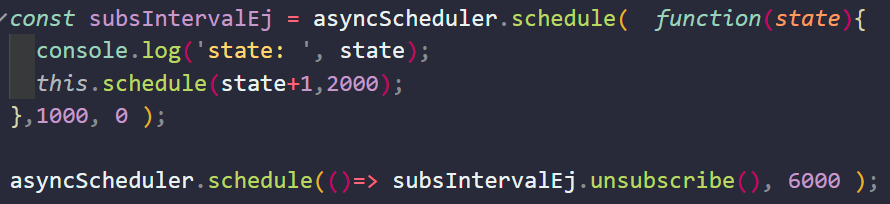
**Video #28: asyncScheduler.**

el asyncScheduler realiza el comportamiento del setTImeout y setInterval, pero la ventaja que tiene este método sobre los otros dos es que puede generar un Subscriber.



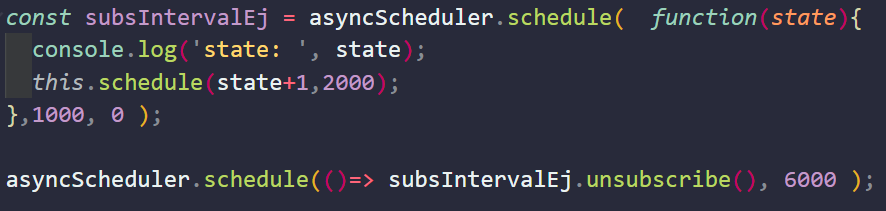
El código de arriba es un ejemplo de cómo trabaja el asyncScheduler como el setTimeout.

El siguiente ejemplo te mostrare como trabajar con el asyncScheduler como un setInterval



para trabajar con Scheduler como setInterval el parámetro que recibe no puede ser una función de flecha debe ser una función normal, esta función como parámetro recibe el state.

Para que cumpla con la función del setInterval debemos llamar nuestro Schedule dentro nuestra función y determinar cuál va ser su nuevo state y el intervalo de tiempo en el cual será ejecutado de nuevo.



**Nota**: la principal ventaja de estas formas de realizar la función del setTimeout y setInterval, es que genera suscripciones de sus procesos los que permite conocer su estado.

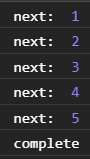
**Video #29: Mas ejemplos con From y of.**

a primera vista el from desintegra el arreglo enviado como parámetro y emite sus valores. El of en cambio emite el arreglo con sus elementos dentro, ósea genera solo una emisión por el argumento enviado y Form por cada uno de los elementos genera una emisión.

Ejemplo Form:



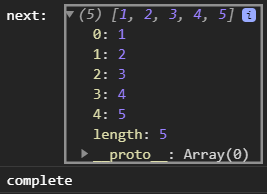
Resultado:



Ejemplo of:



Resultado:



Podemos ver en ejemplos de la explicación anterior.

Nota: conozco una nueva función que me permite generar una petición http en JavaScript, esta es **Fetch** a continuación dejo un abstracto de cuál es la función de Fetch. Esta función es propia de JavaScript.

El método fetch () toma un argumento obligatorio, la ruta de acceso al recurso que desea recuperar. Devuelve una [Promese](https://developer.mozilla.org/es/docs/Web/API/Promise) que resuelve en [Response](https://developer.mozilla.org/es/docs/Web/API/Response) a esa petición, sea o no correcta. También puede pasar opcionalmente un objeto de opciones inti como segundo argumento (ver [Reques](https://developer.mozilla.org/es/docs/Web/API/Request)).

Link de la documentación: <https://developer.mozilla.org/es/docs/Web/API/Fetch_API>

**Sección # 5: Operadores básicos.**

Es momento de entrar al tema de los operadores. Este tema me encanta y es donde ReactiveX brilla, porque la fuerza de los operadores nos permite jugar con la data y el flujo de información de una manera sorprendente.

Los temas principales de esta sección son:

1. Explicación de los operadores
2. Operadores como:
   1. map
   2. pluck
   3. mapTo
   4. filter
   5. tap
   6. reduce
   7. scan
3. Encadenamiento de operadores

Entre otros temas que verás poco a poco... estos son los operadores comunes y más usados...

**Video #33: ¿Qué son los operadores?**

Los operadores de RxJs son funciones que pueden ser encadenadas en lo que llamamos la cadena o pipeline de operadores y que se sitúan entre medias del Observable (productor de la información) y el Observer (consumidor de la misma) con el objetivo de filtrar, transformar o combinar los valores del Observable/Observables.en pocas palabras y en un gran resumen los operadores nos permiten transformar la data y majarla e interactuar con ella.

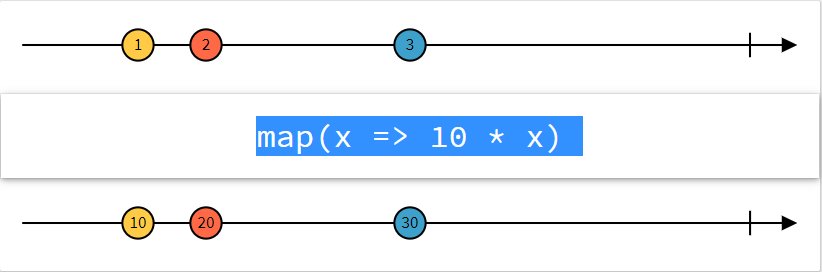
Puedes ingresar al siguiente campo en el que nos explican más a detalle el funcionamiento de los operadores más comunes.

[**https://pablomagaz.com/blog/como-funcionan-operadores-rxjs#:~:text=Los%20operadores%20de%20RxJs%20son,los%20valores%20del%20Observable%2FObservables**](https://pablomagaz.com/blog/como-funcionan-operadores-rxjs#:~:text=Los%20operadores%20de%20RxJs%20son,los%20valores%20del%20Observable%2FObservables)

También te invito a ingresar al siguiente sitio web en el que podemos ver los diagramas de canicas de la mayoría de elementos de RxJs[**https://rxmarbles.com/**](https://rxmarbles.com/)**.**

**Video #34: operador map.**

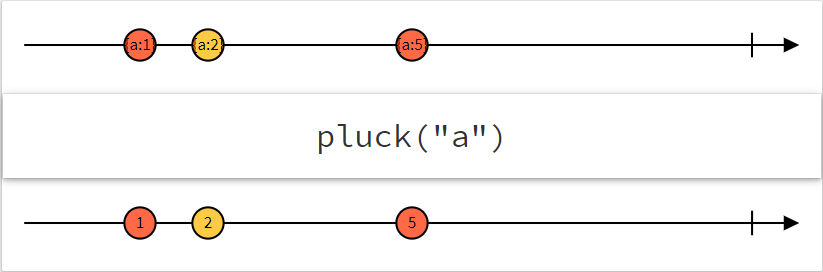
Este operador transforma la data ingresada y devuelve una data de cuerdo a nuestros requerimientos, se verá más claro en el siguiente diagrama:



Podemos observar que por cada valor ingresado en el operador este los multiplica por 10 y como resultado de este operador arrojará cada valor ingresado multiplicado por diez.

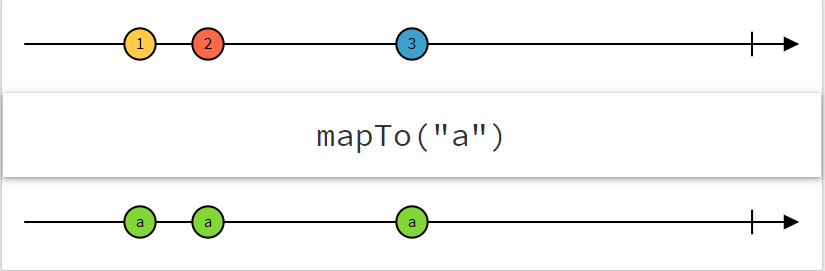
En pocas palabras por cada valor ingresado en el map realizara la operación que tengamos dentro de nuestro operador.

**Video # 35: Operador pluck.**



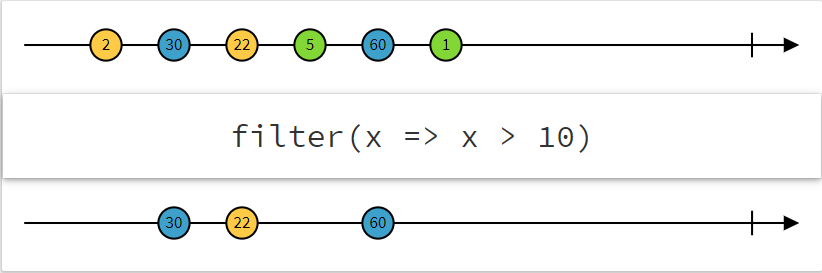
Este operador extrae el valor de un objeto que se recibe, ese será el valor emitido por el operador. Como sabemos el valor ingresado será objeto, pero como parámetro dentro de la función recibiremos la key que deseemos que se encuentre en nuestro objeto.

**Video #36: Operador MapTo.**



Este operador, como vemos en su diagrama en la parte superior, nos permite transformar la data de entrada en una salida en especifica.

**Video #37: Operador filter.**



EL operador filter, como su mismo nombre lo dice filtra los elementos que cumplan un criterio plenamente establecido por nosotros.

El filter por cada uno de los ítems recibidos ejecuta nuestra condición y devuelve un valor booleano, este valor es el que decide si pasa o no el ítem ingresado, si es negativo no pasa continua con el siguiente ítem, si es positivo pasa y sigue con el siguiente ítem.

Para más información ingrese a la documentación oficial:

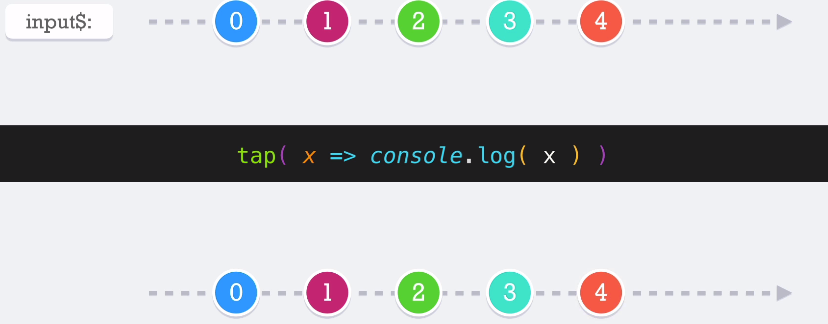
<https://rxjs-dev.firebaseapp.com/api/operators/filter>

Video de ayuda: <https://www.udemy.com/course/rxjs-de-cero-hasta-los-detalles/learn/lecture/16531710#content>

**Video # 38: Cadenas de operadores.**

Lo más importante que puedo sacar de esto, es que los operadores se ejecutan de arriba hacia abajo en el momento que los encadenamos.

**Video #39: Operador tap.**



Intercepta cada emisión en el origen y ejecuta una función, pero devuelve una salida que es idéntica al origen (observable original) siempre que no se produzcan errores.

Este operador es útil para depurar los Observables para los valores correctos o realizar otros efectos secundarios.

**Nota:** el Return no tiene valides en este operador.

<https://rxjs-dev.firebaseapp.com/api/operators/tap>

**Video #40 & #41: Laboratorio-ProgressBar.**

Este video explica cómo realizar una barra de progreso mediante observables, para emitir estos observables utilizamos el FromEvent que emite cada vez que se interactúa con el cuerpo de documento, recordemos que este fromEvent recibe dos valores, el primero el document, el segundo, el evento con el cual se desea interactuar.

Conceptos a tener en cuenta.

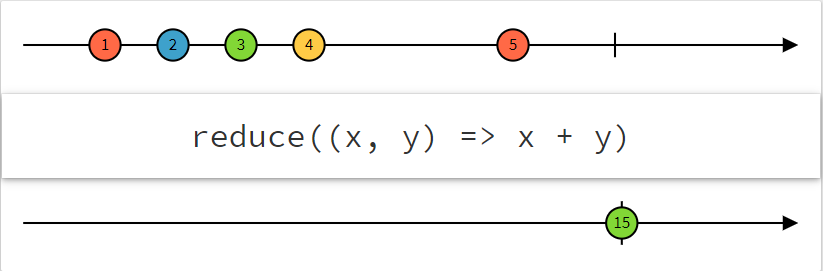
**ClientHeight: Este** es el tamaño total de la vista que posee el cliente.

**ScrollHeigth:** el tamaño total del documento HTML

**ScrollTop**: determina la posición en la que estamos en relación al top.

Estas variables esta ubicadas dentro de target-documentElement, este se encuentra en el evento scroll.

**Video #42: Operador reduce.**



Combina todos los valores del observable de origen, utilizando una función de acumulador, que suma el valor emitido anterior con el nuevo valor emitido.

En pocas palabras aplica una función acumuladora a las emisiones de nuestro observable.

El primer parámetro que recibe esta función es el valor acumulado, el segundo es el valor actual que posee la función.

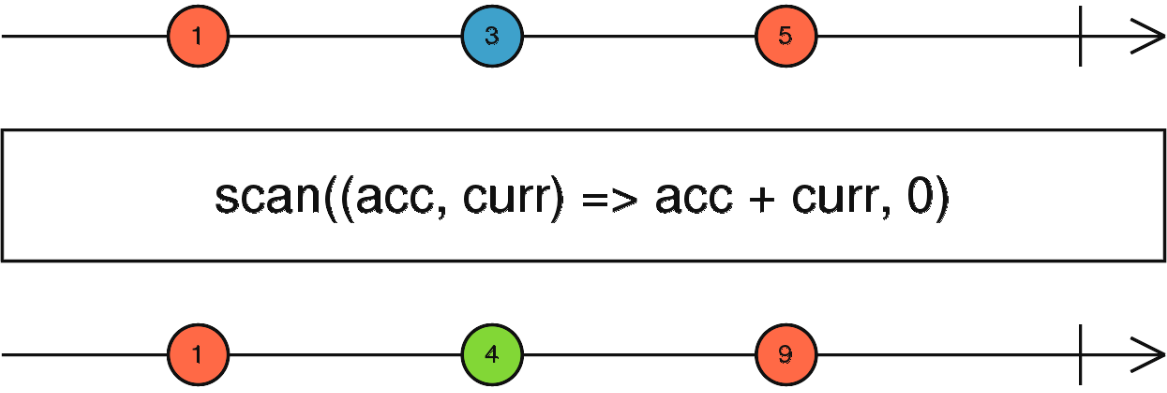
Después de haber iniciado la función, y que esta haya retornado algún valor, estará un ultimó parámetro numérico que indicara el valor inicial del acumulador.

Por ultimó el valor del acumulado no se emitirá hasta que la función de origen se complete.

**Nota:** al parecer ya existe una función con el mismo nombre en JavaScript, estos métodos tienen el mismo funcionamiento.

<https://rxjs-dev.firebaseapp.com/api/operators/reduce>

**Video #43: Operador Scan.**



Como podemos ver en el diagrama de canicas realiza una tarea similar a la que cumple la función **reduce**, pero con la diferencia que esta emite el valor acumulado cada vez que observable de origen emite un nuevo dato u/o valor.>

Scan como base del patrón **redux**, realizando un resumen muy corto, controla la información y el estado de esta dentro de nuestra aplicación, permitiendo un control constante, además que el flujo de la data será de una sola dimensión dentro de la misma aplicación limitando la comunicación entre componentes, pues estos para conocer el estado deberán ir al store, que contienen el estado actual de la información, para conocer el state de la app.

Fuente <https://desarrolloweb.com/articulos/que-es-redux.html>

https://rxjs-dev.firebaseapp.com/api/operators/scan

**Sección 06: Operadores no tan comunes.**

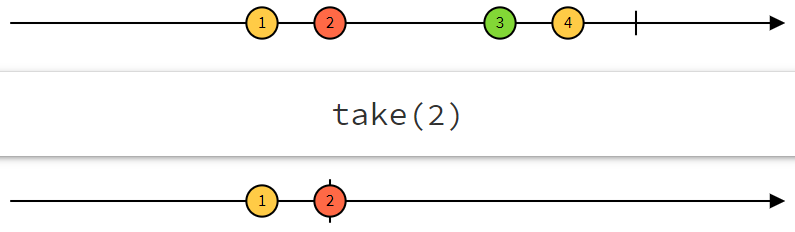
Operadores no tan comunes, aunque este término es relativo porque sé que muchos de ustedes usan estos operadores todos los días, quiero agruparlos como la segunda categoría de operadores para aprender.

Los temas principales son los siguientes operadores:

1. take
2. first
3. takeWhile
4. takeUntil
5. skip
6. distinct
7. distinctUntilChanged
8. distinctUntilKeyChanged

¡Espero que en todas las secciones no sólo aprendamos Rx, sino que también un poco de JavaScript y TypeScript!

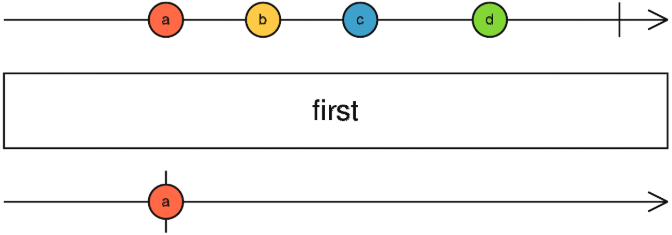
**Video #47: Operador Take.**



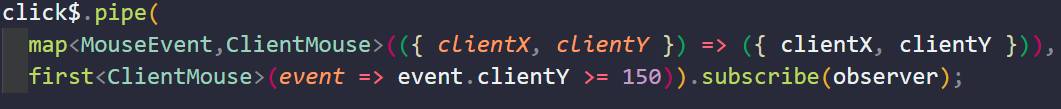
Como se puede ver en el diagrama de canicas este operador limita a una cantidad de dos las emisiones que llega a la suscripción por parte del observable de origen, completándolo después de esas dos emisiones, el parámetro que se pasa en el operador será la cantidad de veces que deseemos escuchar esas emisiones.

<https://rxjs-dev.firebaseapp.com/api/operators/take>

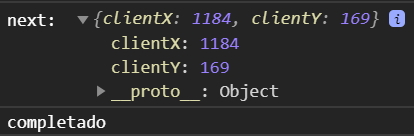
**Video #48: Operador first.**



Como podemos ver en el diagrama de canicas, emite el primer valor de nuestro observable de origen, esto sucede si el operador se envía sin argumento alguno. SI el argumento es alguna función que contenga una condición este se completara si la condición se cumple.



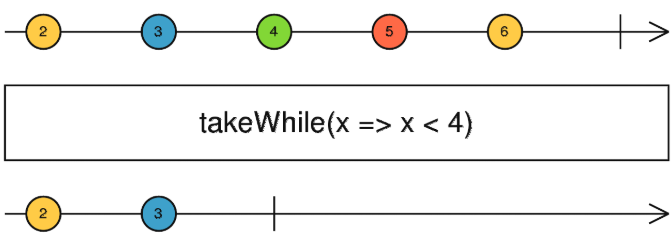
en el segmento de código de la parte superior podemos ver como el operador first trabaja con una condición en la que el clientY debe ser mayor a 150, si esta se cumple, la suscripción recibe la emisión y después se completa.



Este será resultado positivo.

<https://rxjs-dev.firebaseapp.com/api/operators/first>

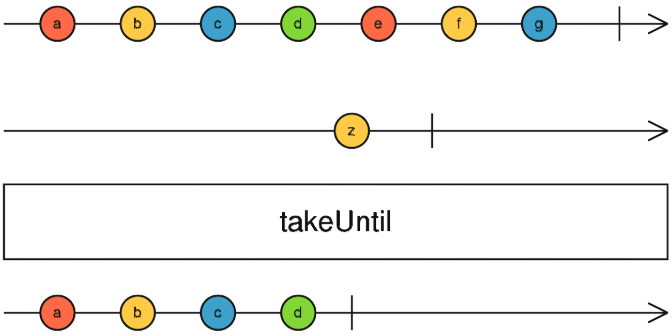
**Video #49: Operador TakeWhile.**



Recibe las emisiones del observador de origen hasta que la condición lo permita. Si esta condición llegase hacer **false** el operador completa el observable de origen, esta es la explicación del primer argumento que recibe este operador.

El segundo argumento de este operador es de tipo booleano, su nombre es **inclusive** y es opcional, el **inclusive** por defecto es false, si llegase a ser true emitiría el ultimo valor que emite nuestro observable de origen, ósea la emisión que rompe la condición de nuestro operador.

<https://rxjs-dev.firebaseapp.com/api/operators/takeWhile>

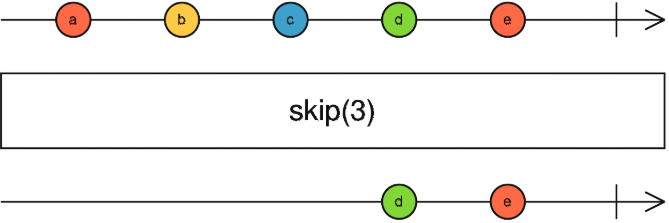
**video #50: Operador TakeUntil.** 

Este operador trabaja con las emisiones de otro observable, este deja pasar las emisiones del primer observable, que es al cual nos suscribimos, esa suscripción le agregamos el operador TakeUntil que estará pendiente del segundo observable, apenas este realice su primera emisión, el operador no dejará pasar más emisiones del primer observable y completará la suscripción que realizamos a nuestro primer observable.

Documentación Oficial:

<https://rxjs-dev.firebaseapp.com/api/operators/takeUntil>

**Video # 51: Operador Skip.**

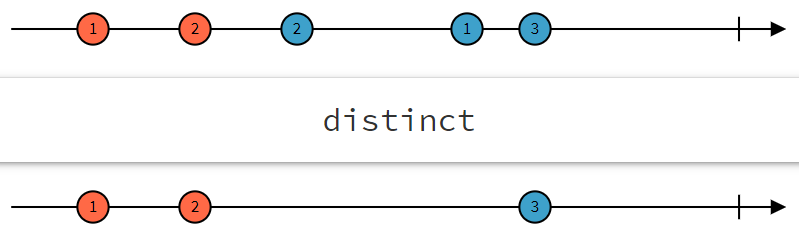


Este operador omite las primeras emisiones del observable de origen, nosotros declaramos mediante variable tipo number cuantas emisiones se van a omitir, por ejemplo, en el diagrama de canicas, vemos como el operador skip tiene un tres como parámetro y omite las primeras tres emisiones de nuestro observador de origen. Esa es la función que cumple este operador.

**Nota:** si el operador no recibe ninguna emisión del observable este nunca se ejecutará

<https://rxjs-dev.firebaseapp.com/api/operators/skip>

**Video # 52: Operador distinct.**



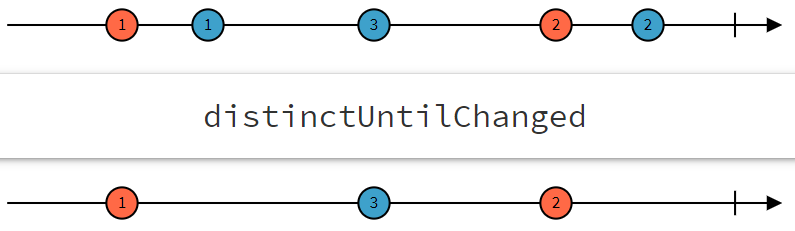
Devuelve un Observable que emite todos los elementos emitidos por el origen Observable que son distintos en comparación con los elementos anteriores.

Este operador evalúa las emisiones del observable de origen, si los valores de estas emisiones ya han sido emitidos con anterioridad este operador las bloquea y sigue con la siguiente emisión que viene del observable de origen.

Vale aclarar que este operador utiliza el triple igual para comparar cada emisión del observable.

<https://rxjs-dev.firebaseapp.com/api/operators/distinct>

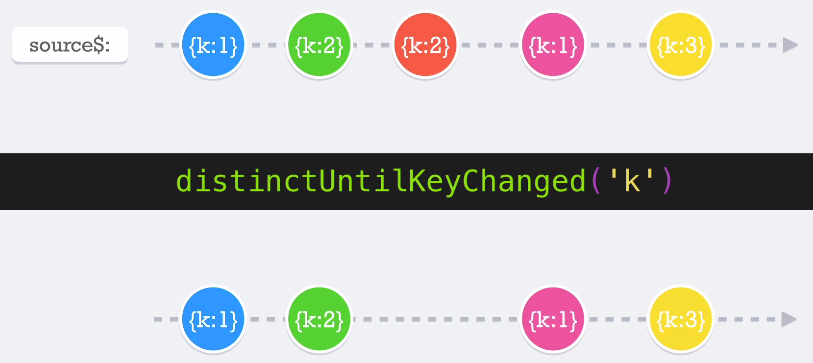
**Video #53: Operador DistinctUntilChanged.**



Este operador todas las emisiones de observable de origen siempre y cuando el valor de la emisión sea igual al valor de la emisión anterior.

Este operador al igual que el **distinct** usa el operador lógico triple igual (===) para su comparación.

**Video #54: Operador distinctUntilKeyChanged.**



Al igual que operador **DistinctUntilChanged** evalúa que el valor emisión anterior, si coinciden, retorna true y la emisión será omitida por el operador.

Pero hay algo particular en este operador ya que, en vez de evaluar directamente el valor, evalúa el key del valor, si el valor que traer key es el mismo al anterior lo omite.

**Sección #7: Operadores que trabajan con el tiempo.**

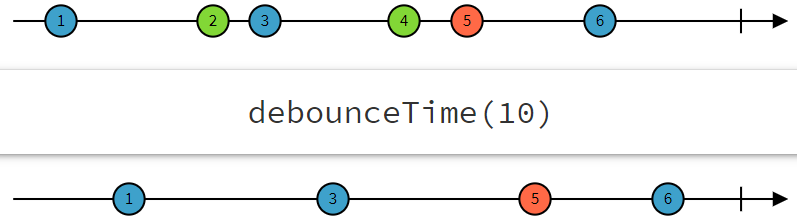
Entramos a los operadores del tiempo, los cuales se enfocan en trabajar de alguna manera con intervalos de tiempo.

Los temas principales son los siguientes operadores:

1. debounceTime
2. throttleTime
3. sampleTime
4. sample
5. auditTime

Aunque hay más operadores que podemos colocar en esta categoría, tienen las bases para poder investigarlos si así lo necesitan.

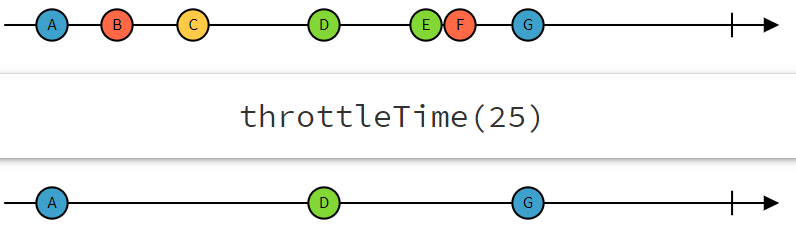
**Video #58: Operador DebounceTime.**



Este operador emite el valor del observable de origen en un intervalo de tiempo, enviado como parámetro dentro del operador, claro está, que si se realiza una nueva emisión por parte del observable de origen el valor anterior emitido será omitido, reiniciando la cuenta para emitir el ultimo valor que provino del observable de origen.

En el diagrama de canicas de la parte de arriba se puede evidenciar más claramente el comportamiento del operador.

**Video #59: Operador throttleTime.**

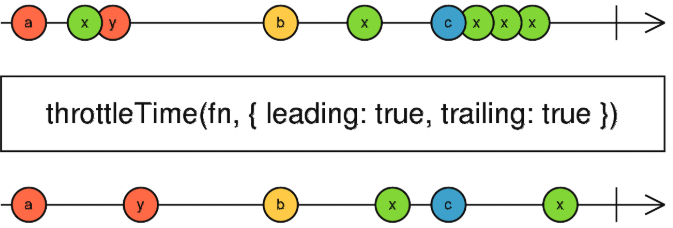


Esté como el operador anterior en esta guía funciona con un intervalo de tiempo en sus emisiones, pero a diferencia del **debounceTime**, el intervalo está inhabilitado hasta que el primer valor del observable de origen es emitido, en ese preciso momento inicia el intervalo. Hasta que el tiempo del intervalo termine las emisiones que vengan del observador de origen serán omitidas.

Igual que en el **debounceTime** el tiempo que asignamos al operador es pasado como parámetro dentro del operador.

Este operador puede servir para controlar las emisiones de observables que emiten valores muy frecuentemente.

Ya que este operador emite los valores del observable de origen con un tiempo determinado de separación entre cada emisión, y lo más común es que solo emita el primer valor, pero existe una manera de que emita el ultimo y primer valor.

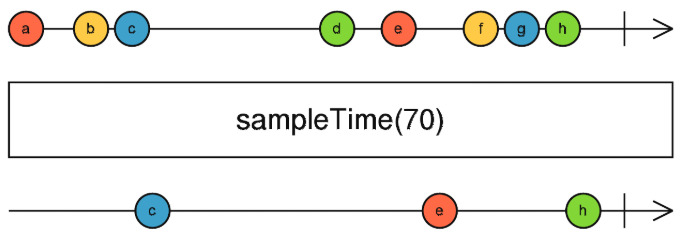


Este sería el ejemplo en diagrama de canias.  


Así quedaría el código.

De Esta manera obtendremos el primer y último valor de la emisión del observable de origen.

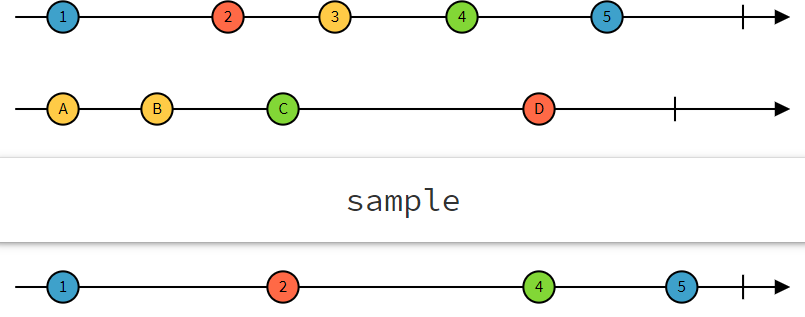
**Video #60: Operador SampleTime.**



Este operador emite al suscriptor el ultimo valor ingresado en el observable de origen después de haber terminado el intervalo de tiempo, este intervalo de tiempo es enviado como parámetro dentro del operador, en caso de que la fuente de nuestras emisiones no genere ningún valor el operador no emitirá ningún valor más.

Este operador realiza este proceso estando pendiente periódicamente del observable de origen.

**Video #61: Operador Sample.**



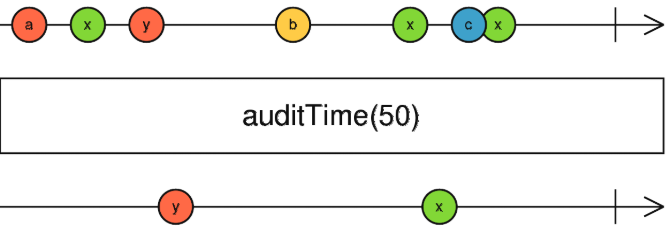
Tiene un comportamiento similar al **sampleTime**, ya que emite el ultimo valor emitido por el observable de origen, pero a diferencia del operador anterior en esta lista, este solo emite el ultimo valor del observable de origen cuando un segundo observable emita algún valor, este segundo observable es pasado como paramento dentro del operador.

Si el observable de origen no á emitido ningún nuevo valor y se emite un nuevo valor en el segundo observable no se generará una nueva emisión por parte de operador hasta que haya un nuevo valor que provenga del observable de origen.

La suscripción a este operador se completa en el momento que nuestro observable de origen realiza el **complete**.

<https://rxjs-dev.firebaseapp.com/api/operators/sample>

**Video #62: Operador auditTime.**



Este operador funciona con un intervalo de tiempo, que es enviado como parámetro del mismo, este intervalo estará desactivado hasta que el observable de origen realice su primera emisión, la finalizar el intervalo de tiempo del operador emitirá el ultimo valor que venga del observable de origen.

Si durante el intervalo de tiempo el observable de origen da **complete,** la suscripción también se completará y se dejara de emitir valores.

**Sección 8: Ajax – Peticiones ajax usando RxJs/ajax.**

**Estamos entrando a la parte de la librería de RXJS enfocada en peticiones Ajax. Aquí no sólo trabajaremos con funciones propias para realizar llamadas HTTP, sino que también configuraremos headers y trabajaremos con las respuestas.**

**Los temas principales de esta sección son:**

1. **Conceptos básicos de una petición ajax**
2. **Manejo de errores**
3. **Fetch API**
4. **getJson**
5. **Ajax**
6. **Diferencia entre getJson y Ajax**
7. **PUT, POST, DELETE, GET**

**Espero que muchos de ustedes vean el potencial que esta librería tiene, sin contar que se le suman todos los operadores que hemos visto hasta el momento.**

**Video #66: Conceptos generales de una petición ajax usando Fetch.**

El método fetch () toma un argumento obligatorio, la ruta de acceso al recurso que desea recuperar. Devuelve una Promise que resuelve en Response a esa petición, sea o no correcta. También puede pasar opcionalmente un objeto de opciones init como segundo argumento

Una vez que Response es recuperada, hay varios métodos disponibles para definir cuál es el contenido del cuerpo y como se debe manejar.

La interfaz **Response** de la [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) representa la respuesta a una petición.

La interfaz **Response** de la [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) representa la respuesta a una petición.

Para más información sobre este método de recuperación de información entra a los links que dejo a continuación

El **then** en las promesas se utiliza para manejar la respuesta exitosa, el **catch** para manejar el error en la respuesta

<https://developer.mozilla.org/es/docs/Web/API/Fetch_API>

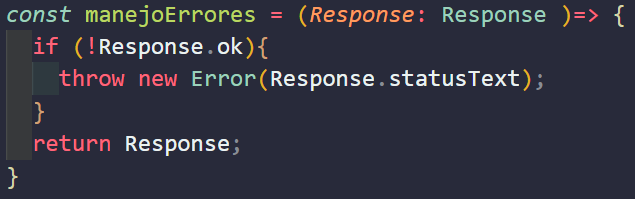
<https://developer.mozilla.org/es/docs/Web/API/Request>

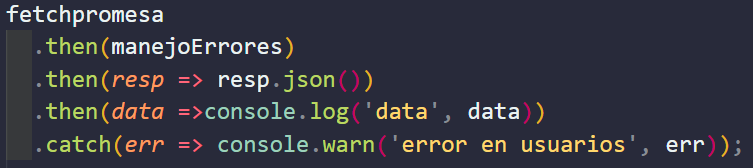
<https://developer.mozilla.org/es/docs/Web/API/Response>

<https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Utilizando_Fetch>

**Video #67: Manejo de errores con el fetch Api.**

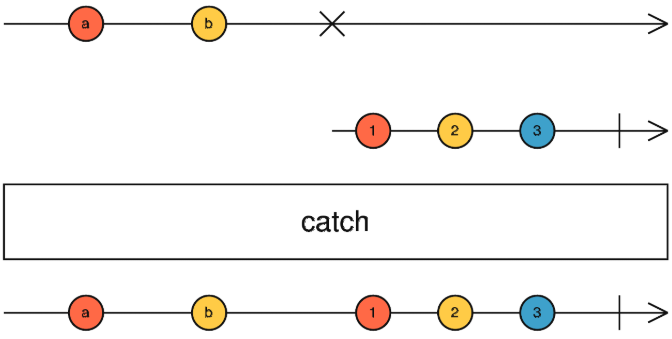
Adjunto un fragmento de condigo, el cual es una función con la cual podemos manejar los erros en el fetch. Esto solo es una de las tantas formas que existen para poder controlar el **catch** con esta **Api**.



Esta función es llamada por referencia dentro del primer **then** de nuestra promesa. 

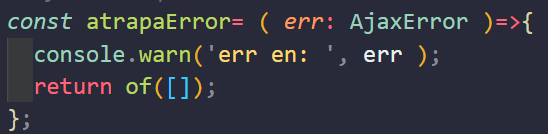
**Video #68: Petición usuando ajax de RxJs.**

**En** video conocemos un nuevo operador con el cual nos permite controlar el error de una petición ajax o un error ocurrido en las emisiones del observable de origen el cual es **catchError.**



La función principal de este operador es obtener o atrapar los errores que se generan en el observable de origen, lo cual permite el manejo de errores y tener más posibilidades de actuar. Al momento de obtener error el operador puede, por orden nuestra, enviar un mensaje o crear otro observable que será emitido a la suscrición.

Como parámetro el operador debe recibir las instrucciones de cual el proceso que realizara con el error recibido.



Esta es la función que se envía por referencia dentro del operador.

**Sección #9: Operadores de transformación.**

Vimos en la sección anterior (ajax), que existe la necesidad de subscribirse al producto de un observable, para poder obtener la información que necesitamos, lo cual lleva a que perdamos control de la legibilidad de nuestro código y la facilidad de trabajar con observables y operadores...

Por suerte, el equipo de ReactiveX pensó en esto y nos ayuda con las siguientes funciones y operadores:

1. mergeAll.
2. mergeMap.
3. switchMap.
4. concatMap.
5. exhaustMap.

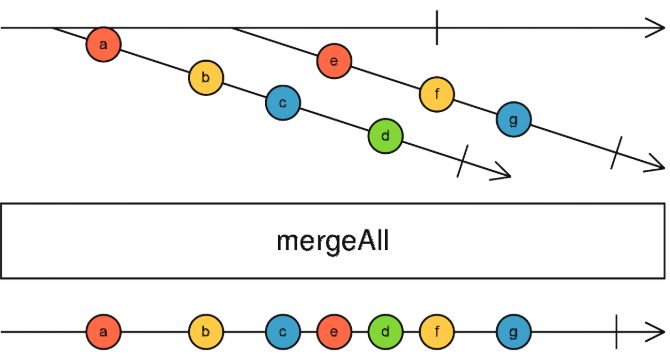
Aquí tendremos formas muy interesantes de poder trabajar con el concepto del aplanamiento, que veremos y explicaré más adelante en esta sección.

**Video #75: Introducción al problema y necesidad de operadores de transformación.**

**Buena Practica**: Lo ideal es que cada operador trabaje en una sola tarea al momento de recibir data.

El problema que se plantea en esta sección del curso es, exactamente el retorno de la información de un observable sin tener la necesidad de hacer un encadenamiento de operadores muy largo. O modificar el retorno de la data a la suscripción.

**Video #76: Operador mergeAll.**



La función principal de este operador es unificar todas las emisiones que vengan de los observables, sea el padre o lo hijos. Para que se complete la suscripción inicial que realiza el **mergeAll** todos, padre y lo hijos, deben haber terminado sus emisiones de valores y tener el complete realizado.

Esta operación se conoce como operadores de aplanamiento.

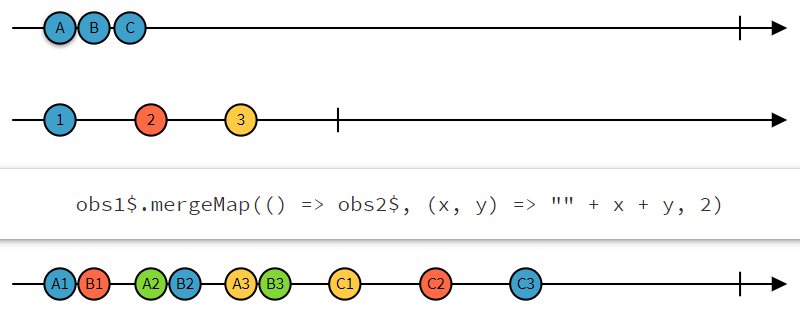
**Video #77: Tipando los operadores.**

En clase se tocó lo fundamental que es el tipado en las operaciones que realizamos con operadores, ya que se determina el tipo de data que entra en tu operador y la que sale de este con esto se genera un control más grande en los procesos que se realiza.

Te dejo un link en el cual puedes crear tus interfaces con los objetos.

<https://app.quicktype.io/>

**Video #78: mergeMap.**

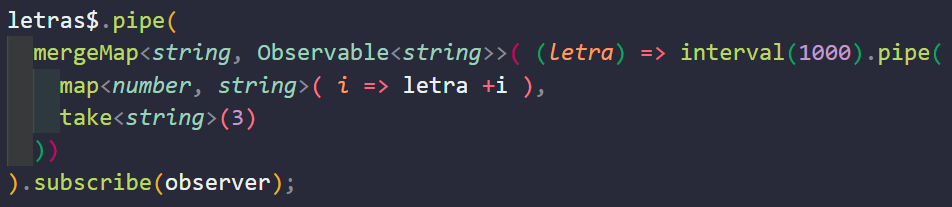


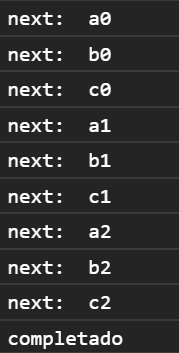
Este es otro operador de aplanamiento, recordemos que estos operadores crean un nuevo observable apenas le es ingresado el valor emitido por nuestro observable de origen, el valor devuelto por este operador no es un observable, pues estos operadores se suscriben automáticamente al nuevo(hijo) observable, el valor devuelto a la suscrición inicial será, los valores emitidos por estos observables.

Por cada nuevo valor emitido de nuestro observable de origen se crearía un nuevo observable hijo, pero todos los valores emitidos por estos se visualizarán por la suscripción inicial.

Para que nuestra suscripción inicial de **complete** deben estar todos los observables hijos completados y también su observable padre.

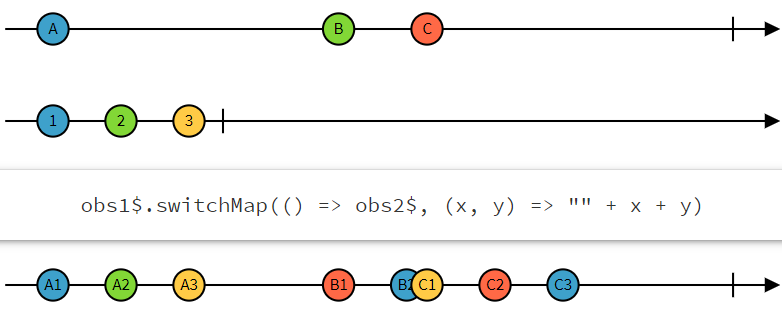
Ejemplo:





Este el un ejemplo muy parecido al ejemplo enseñado en el diagrama de canicas, podemos ver que por cada valor ingresado atreves de mergeMap se crea un nuevo observable.

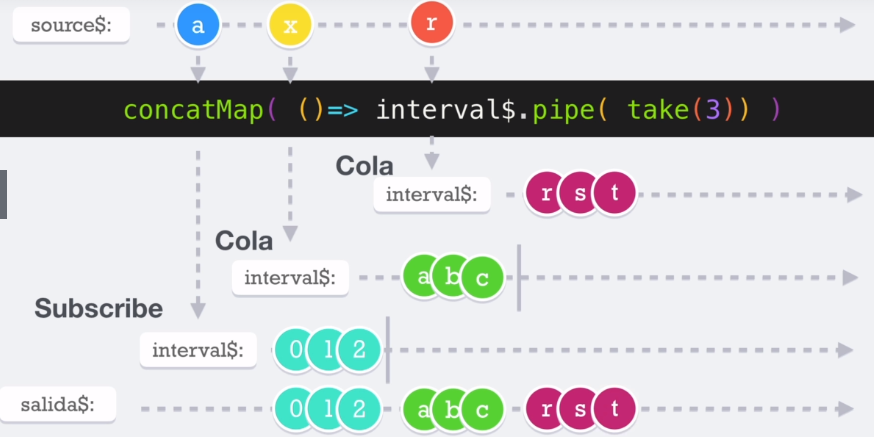
**Video #80: Operador de aplanamiento switchMap.**



Este operador al igual que el **mergeMap** crea un nuevo observable al cual realiza una suscripción interna cada vez que observable de origen emite un nuevo valor, pero a diferencia del **mergeMap**, este operador cada vez que recibe una nueva emisión de valor de observable de origen (padre) completa su observable interno anterior y crea un nuevo observable al cual se suscribe automáticamente y los valores emitidos por este se muestran en la suscripción original.

Este proceso sucede con cada emisión nueva del observable padre.

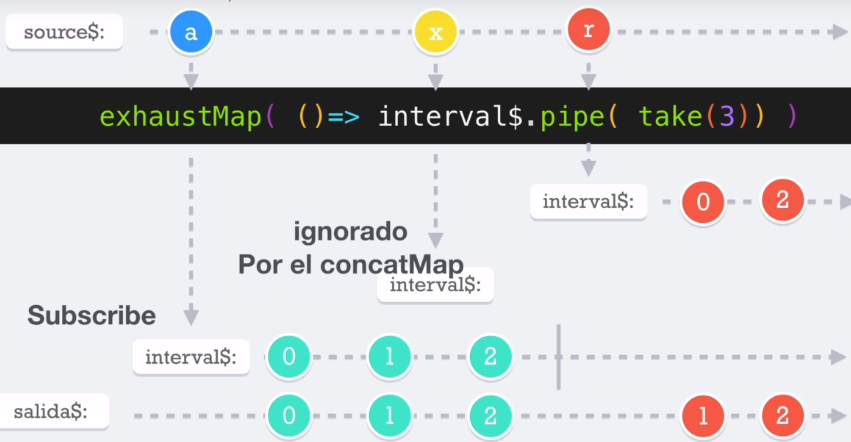
**Video #82: Operador de aplanamiento concatMap.**



Este operador al igual que los anteriores, crea un observable cuando recibe un valor emitido por el observable de origen o padre, estos observables creados, se ejecutar uno por uno hasta que genere le complete.

Si del observable de origen emite mas de un valor, cada observable hijo será añadido a una cola hasta que los observables hijos anteriores se hayan completado.

**Video #83: Operador de aplanamiento exhaustMap.**



Este operador crea un observable hijo por cada valor recibido del observable de origen, pero a diferencia de los operadores anteriores, si este operador tiene un observable hijo emitiendo algún valor y recibe un nuevo valor del origen, este valor recibió será omitido y todos los valores posteriores hasta que la suscripción actual al observable hijo se complete.

Este operador es útil, por ejemplo, para controlar las emisiones de algún botón con el primer click omitiendo los siguientes clicks hasta que se complete su función.