RXJS

Patron reactivo

RxJS

npm install rxjs

Observable

Un observable es un objeto que podrá emitir valores tipo genérico que se le especifique, puede emitir en un cierto o indefinido periodo de tiempo X cantidad de emisiones hasta que el observable se complete, para poder acceder a sus valores emitidos será necesario subscribirse al observable.

Subscripción / Observer

Un observable cuenta con un metodo subscribe, se recibe un callback con la accion a realizar al recibir la emisión.

```
observable$.subscribe(value => console.log(value));
```

O bien recibir un objeto (Observer) con las especificaciones de acción en emisión, error o que se complete dicho observable

```
const observer: Observer <any> =  {     //* Objeto Observer para los 3 casos
    next: emitido => console.log("next: "+ emitido),  //* Cuando se emmita
    error : error => console.error("Error: "+ error), //* Cuando se haya error
    complete: () => console.log("Se completo!!!"),    //* Cuando se complete
}

observable$.subscribe( observer );
```

Cada subscripcion crea un nuevo objeto Observable, por lo que pueden no ser iguales los valores de una subscripcion a otra

Creación de observable por constructor

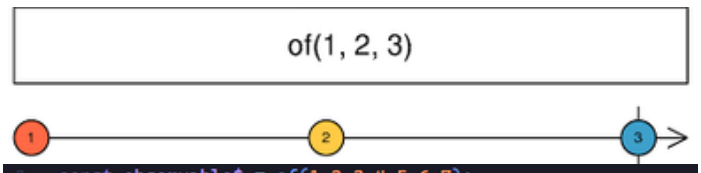
```
const observable$ = new Observable <number> (
    subscriber =>{          //? Subscriber permite emitir y completar
        subscriber.next(12);   //* Emito un 12

        const a = undefined;   //* Generamos un error
        a.nombre = "Hola";

        subscriber.complete(); //* Termina el observale
        subscriber.next(50);   //* Ya no se emite dicho valor
    }
);
```

El callback en el constructor recibe de parametro un subscriber capaz de emitir y completar el observable.
El observable puede emitir, completar y generar error. Al emitir un error o completar, ya no se emitiran valores.

Métodos creadores de observables

- **Of**

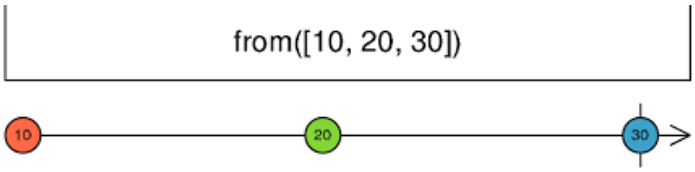
Recibe n cantidad de parámetros y el observable retornado los emitirá por separado de forma secuencias,




```
const observable$ = of(1,2,3,4,5,6,7);

observable$.subscribe(
    value => console.log(value),
    null,
    () => console.log("Terminamos el observable")
);
```

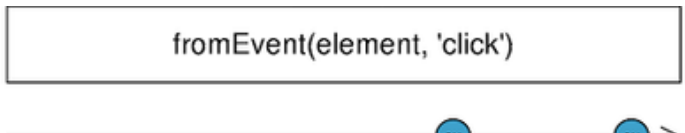
- **From**

Recibe un parámetro que será una estructura iterable y sus miembros serán emitidos de forma individual y consecutiva



- **FromEvent**

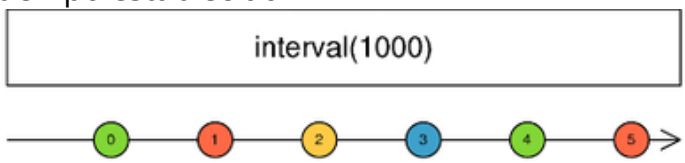
Recibe como parámetro un objeto HTML y el evento, cada que suceda dicho evento, el observable emitirá un objeto del tipo de evento.




```
const src1$ = fromEvent<PointerEvent>( document, 'click' );
const src2$ = fromEvent<KeyboardEvent>( document, 'keyup' );
```

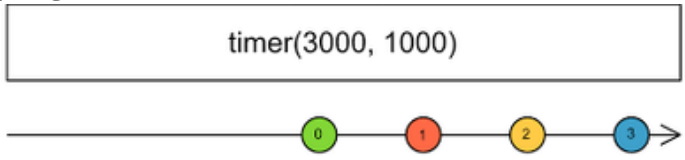
- **Interval**

Emite números consecutivos cada periodo de tiempo establecido



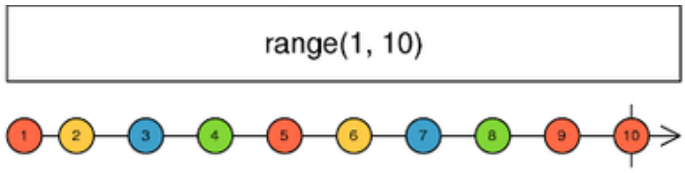
- **Timer**

Emite una solo valor en con el delay establecido, si recibe un segundo parámetro funcionara como un Interval con delay. También puede recibir un Date para programar una emisión exacta



- **Range**

Recibe un valor de inicio y como segundo parámetro los N números siguientes a emitir



RXJS

Patron reactivo

Subject y BehaviorSubject

A diferencia de un observable, estos no crean un nuevo objeto en cada suscripción, por lo que si hay una o mas subscripciones, los valores recibidos en cada una de ellos serán los mismos.
También puede subscribirse y desuscribirse a otro observable y emitir a la misma vez.
Puedes emitir directamente con su método "next" en cualquier parte de su scope

```
const subject$ = new Subject<number>();

//* El subject al ser un observable tambien me puedo subscribir a el
//* Ambas subscripciones reciben el mismo dato emitido
const sub1 = subject$.subscribe( console.log );
const sub2 = subject$.subscribe( console.warn );

setTimeout(
    () => {
        subject$.next(10);
        subject$.complete();

    },
    3500
)
```

A diferencia de Subject, BehaviorSubjtect forza a una emisión principal en su constructor, la cual todos recibiran siempre.

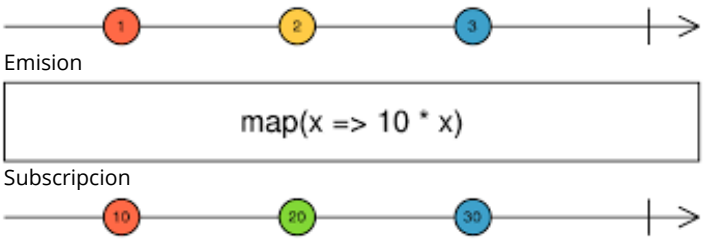
Operadores

Un observable o Subject pueden filtrar o transformar los datos emitidos para cada suscripción mediante el método pipe, que recibe N cantidad de operadores para transformar los valores

```
observable$.pipe(
    map( value => value + 2 )
).subscribe( data => console.log( data ) )
```

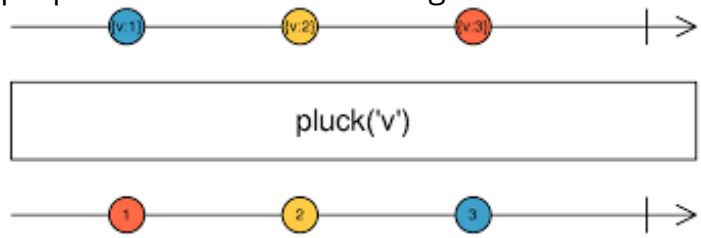
- **Map**

Recibe un parámetro que será una estructura iterable y sus miembros serán emitidos de forma individual y consecutiva



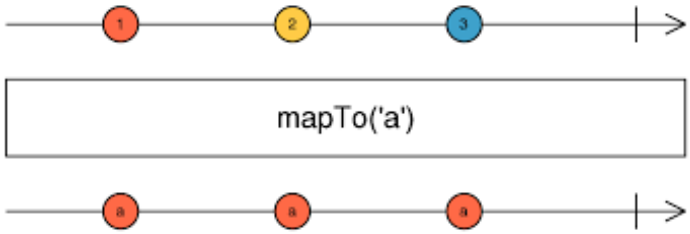
- **Pluck**

Si se es emitido un objeto, accede a la propiedad mediante un string



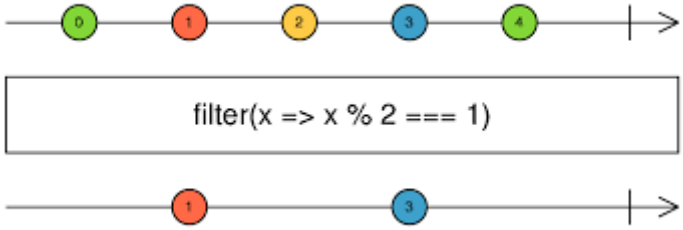
- **MapTo**

No importa el valor emitido, lo transforma en otro.



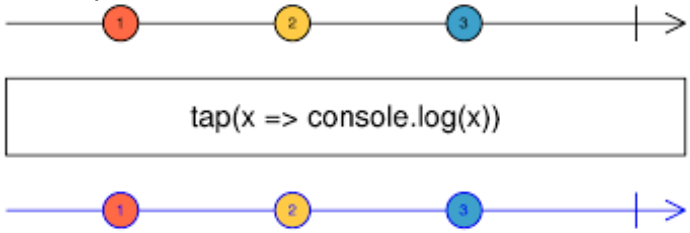
- **Filter**

A través de un callback que regrese un bool, filtra que valor pasa a la suscripción y cual no.



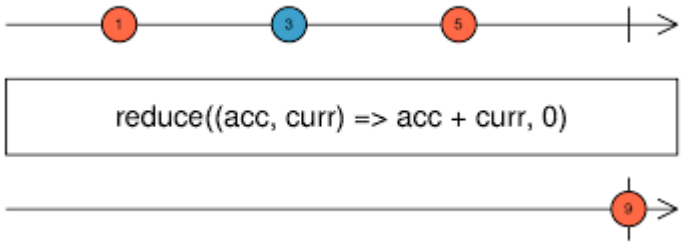
- **Tap**

No transforma la emisión, solo es un callback usado para efectos secundarios de la emisión



- **Reduce**

Hace un acumulador de los valores emitidos y solo emitirá cuando el observable se complete.

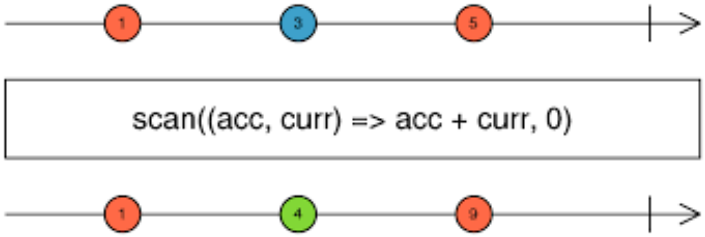


RXJS

Patron reactivo

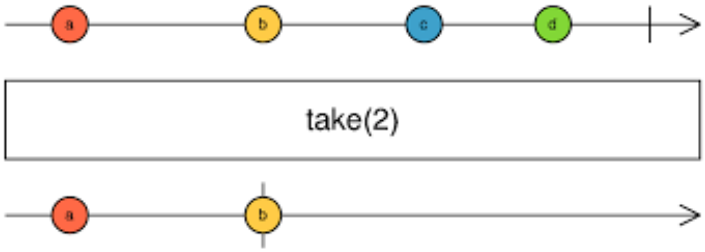
- **Scan**

Con función similar a reduce, con diferencia de que emitirá cada que se reciba un valor.



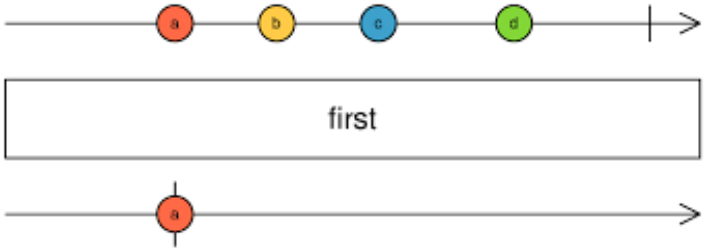
- **Take**

Toma los primeros valores mencionados, posteriormente completa el observable.



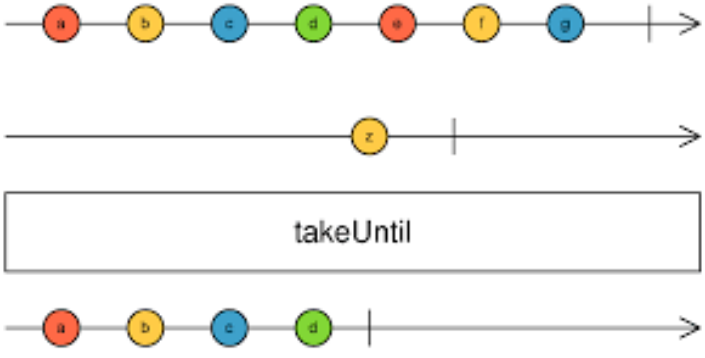
- **First**

Toma solo el primer valor y finaliza el observable.



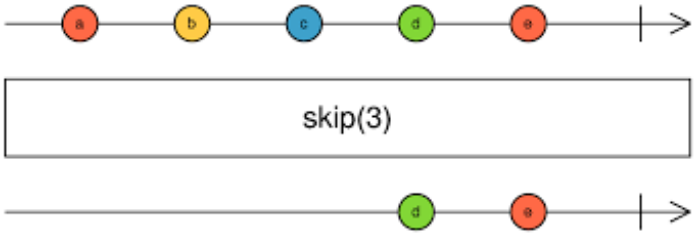
- **TakeUntil**

Permite la emisión de valores hasta que el observable del parámetro emita algo, entonces finaliza el observable local.



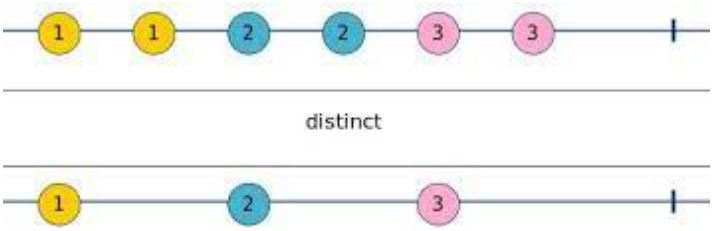
- **Skip**

Salta N cantidad de emisiones, posteriormente emite con normalidad.



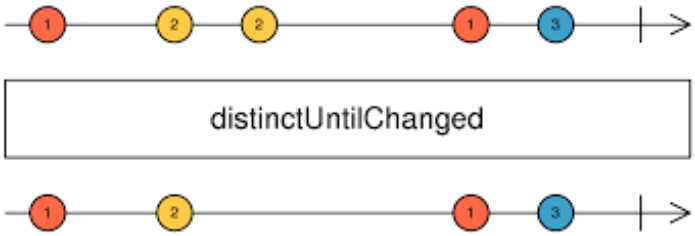
- **Distinct**

Si un valor ya fue emitido, no lo emitira.



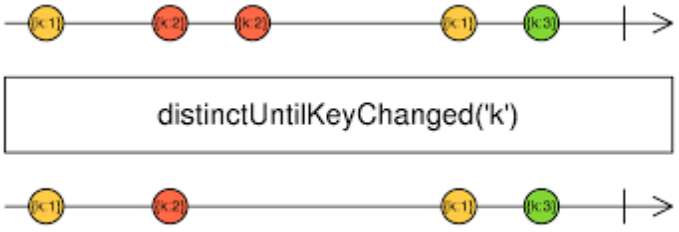
- **DistinctUntilChange**

No emitirá valores repetidos si son consecutivos.



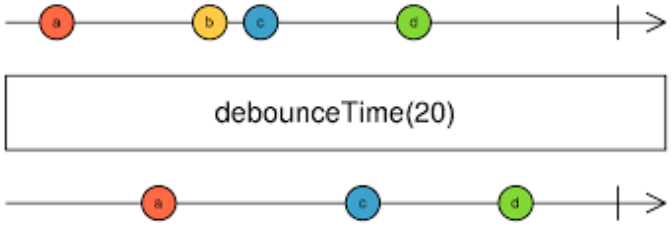
- **DistinctUntilKeyChange**

No emite valores repetidos consecutivos de una propiedad.



- **DebounceTime**

Previene emisiones masivas, espera N tiempo desde la ultima emisión, si no se emite nada durante ese lapso, emitirá lo ultimo emitido.

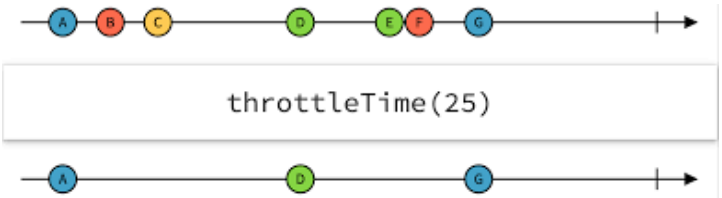


RXJS

Patron reactivo

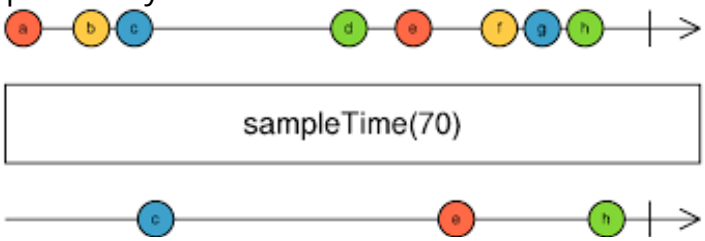
- **ThrottleTime**

Con función similar a debounce, dejara pasar la primera emisión y N cantidad de tiempo ya no emitirá nada hasta que pase dicho tiempo.



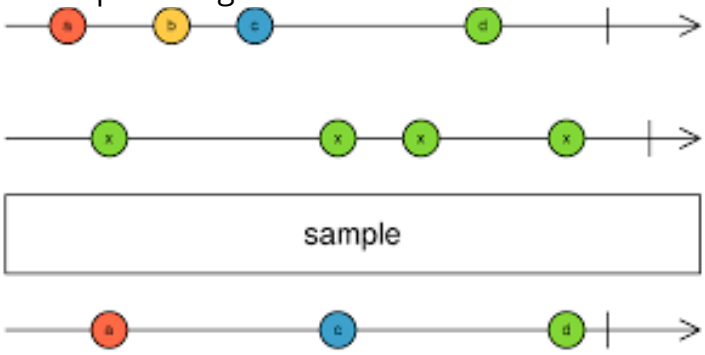
- **SampleTime**

Siempre estará en un intervalo, cada N cantidad de tiempo emitira un el ultimo valor, a menos que no haya ultima emisión.



- **Sample**

Recibe como parámetro otro observable, emitirá el ultimo valor del observable principal hasta que el segundo observable emita.

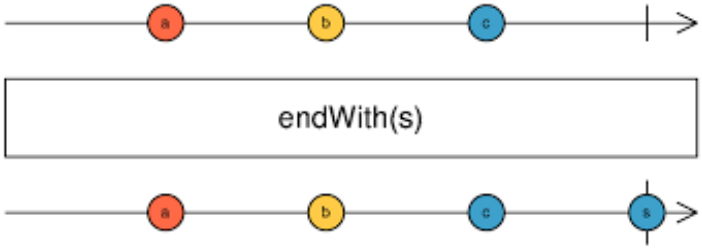


- **Startwith**

En cuanto se crea el observable hará una emisión principal o por defecto, la cual será el argumento enviado.

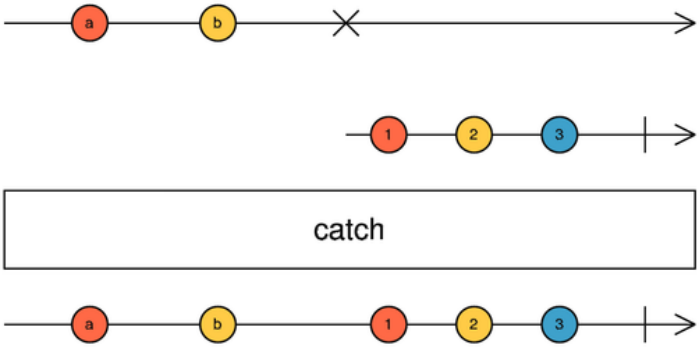
- **Endwith**

Cuando se complete el observable hará una emisión final por defecto.



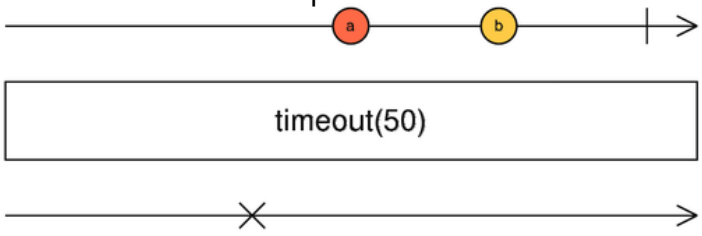
- **CatchError**

CatchError se ejecutara en el momento en que el observable emita un error, catchError recibe como callback una función que retornara un nuevo observable para continuar con la suscripción



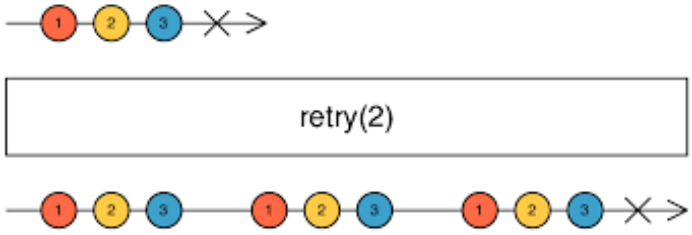
- **timeOut**

TimeOut recibe un tiempo como parámetro, será el tiempo en milisegundos como tolerancia para esperar una emisión, en caso de que no haya una emisión dentro de ese tiempo, lanzara un error de tipo timeOut



- **Retry**

Usado en peticiones HTTP, retry reintenta n veces en caso de un error



RXJS

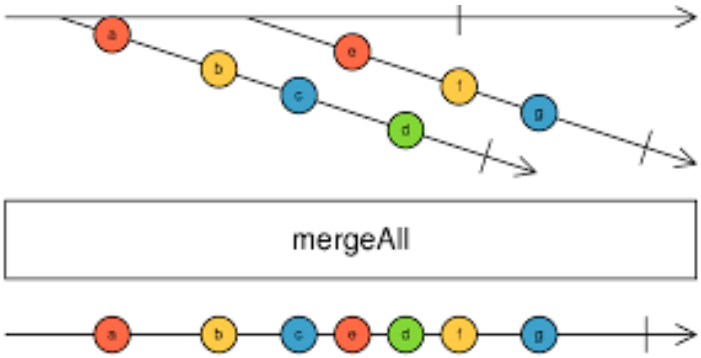
Patron reactivo

Operadores de aplanamiento

Un operador de aplanamiento de encarga de que cuando un observable raiz emita o atravez de un proceso tipo pipe se reciba o genere otro observable, se subscriba internamente a el observable generado para tener una unica subscripcion final.

- **MergeAll**

Suponiendo que un observable raíz emita otro observable, mergeall se subscribe internamente a todos los observables emitidos por la raíz y los valores emitidos por todos se verán reflejados una única suscripción.

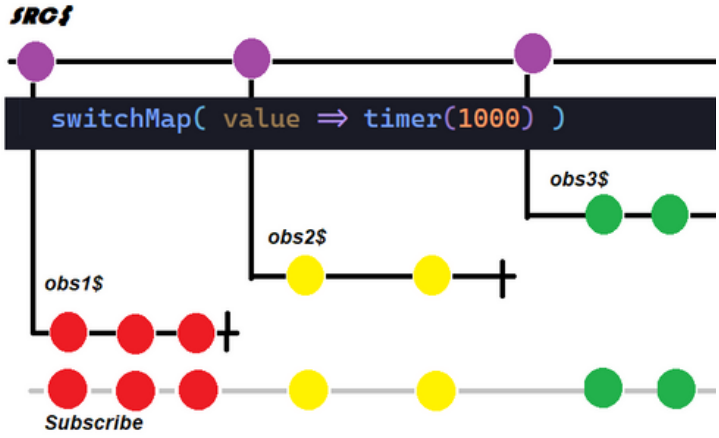


- **MergeMap**

Muy similar a mergeAll, mergeMap recibe un callback que retorna un observable y dejara fluir el producto de todos los observables que retorne mergeMap, se completara hasta que todos los observables activos se complete.

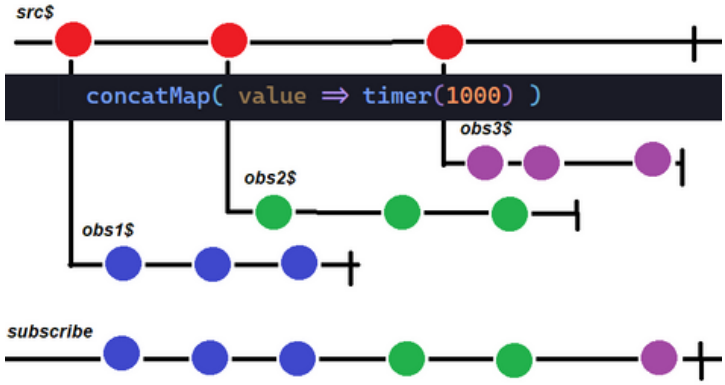
- **SwitchMap**

A diferencia de mergeAll y mergeMap, SwitchMap no deja vivos todos los observables generados, solo deja el ultimo activo y los demás los completa.



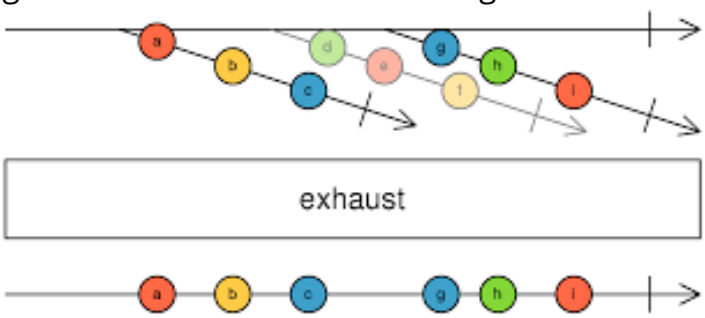
- **concatMap**

concatMap solo dejara fluir las misiones de un primer observable generado, cuando este se complete, ahora si cambiara para emitir el otro observable que quedo en cola.



- **exhaustMap**

Toma en cuenta el primer observable generado y ignorara todos los demás siempre y cuando el primero no se haya completado, cuando se complete, tomara en cuenta el siguiente e ignorara a los demás mientras siga activo



RXJS AJAX

RXJS brinda un apartado en su API para poder hacer peticiones HTTP sin necesidad de usar promesas y fetchAPI, con el fin de facilitar el manejo de respuestas en base a observables

AJAX

La función AJAX de RXJS recibe de parámetro una URL a donde hacer la petición, posteriormente retorna un observable con los datos de la petición y el response, la respuesta puede parsearse con un parámetro genérico.

```
const url = 'https://api.github.com/users?per_page=5';
ajax( url ).pipe(
    pluck('response'),
)
.subscribe( users ⇒ console.log('usuarios:', users) );
```

RXJS

Patron reactivo

GetJSON

La función GetJson también recibe una URL de parámetro y retorna un observable, con diferencia de que accede automáticamente a la response, también puede enviarse los headers de la petición como segundo parámetro en objeto.

```
const url = 'https://httpbin.org/delay/1';
const obs$ = ajax.getJSON( url, {
    'Content-Type': 'application/json',
    'mi-token': 'ABC123'
});
```

INTERFACES

AjaxRequest

La función AJAX también puede recibir un objeto que implemente la interfaz de AjaxRequest, interfaz que nos ayuda a realizar la petición con distinta configuración, como headers, url, body, method (get, post, put delete), etc.

```
export interface AjaxRequest {
    url?: string;
    body?: any;
    user?: string;
    async?: boolean;
    method?: string;
    headers?: Object;
    timeout?: number;
    password?: string;
    hasContent?: boolean;
    crossDomain?: boolean;
    withCredentials?: boolean;
    createXHR?: () => XMLHttpRequest;
    progressSubscriber?: Subscriber<any>;
    responseType?: string;
}
```

AjaxResponse

La función AJAX emite una respuesta de tipo AjaxResponse, con la informacion de status, petición y cuerpo de respuesta del servidor

```
export declare class AjaxResponse {
    originalEvent: Event;
    xhr: XMLHttpRequest;
    request: AjaxRequest;
    /** @type {number} The HTTP status code */
    status: number;
    /** @type {string|ArrayBuffer|Document|object|any} The response data */
    response: any;
    /** @type {string} The raw responseText */
    responseText: string;
    /** @type {string} The responseType (e.g. 'json', 'arraybuffer', or 'xml') */
    responseType: string;
    constructor(originalEvent: Event, xhr: XMLHttpRequest, request: AjaxRequest);
}
```

AjaxError

La función AJAX emite una error en la peticion con un objeto de interfaz AjaxError, con mensaje de error, status, etc.

```
export interface AjaxError extends Error {
    /** @type {XMLHttpRequest} The XHR instance associated with the error */
    xhr: XMLHttpRequest;
    /** @type {AjaxRequest} The AjaxRequest associated with the error */
    request: AjaxRequest;
    /** @type {number} The HTTP status code */
    status: number;
    /** @type {string} The responseType (e.g. 'json', 'arraybuffer', or 'xml') */
    responseType: string;
    /** @type {string|ArrayBuffer|Document|object|any} The response data */
    response: any;
}
```

METODOS HTTP

La clase Ajax contiene métodos especiales para cada petición, como PUT, DELETE y POST, al igual que GET, la función puede recibir url, headers y body como parámetros.

- **Get (por defecto)**

Recibe URL y headers como parámetros en la función.

```
ajax.get( url,
    { 'mi-token': 'ABC123' }
).subscribe( console.log  );
```

- **Post**

Recibe URL, body y headers como parámetros en la función.

```
ajax.post(
    url,
    { id: 2 },
    { 'mi-token': 'ABC123' }
).subscribe(console.log);
```

- **Delete**

Recibe URL y headers como parámetros en la función.

```
ajax.delete(
    url,
    { 'mi-token': 'ABC123' }
).subscribe(console.log);
```

- **Put**

Recibe URL, body y headers como parametros en la función.

```
ajax.put( url,
    { id: 1,},
    { 'mi-token': 'ABC123' }
).subscribe( console.log  );
```