Ejemplos de programación reactiva con RxJS

Programación Reactiva

Qué es la Programación Reactiva

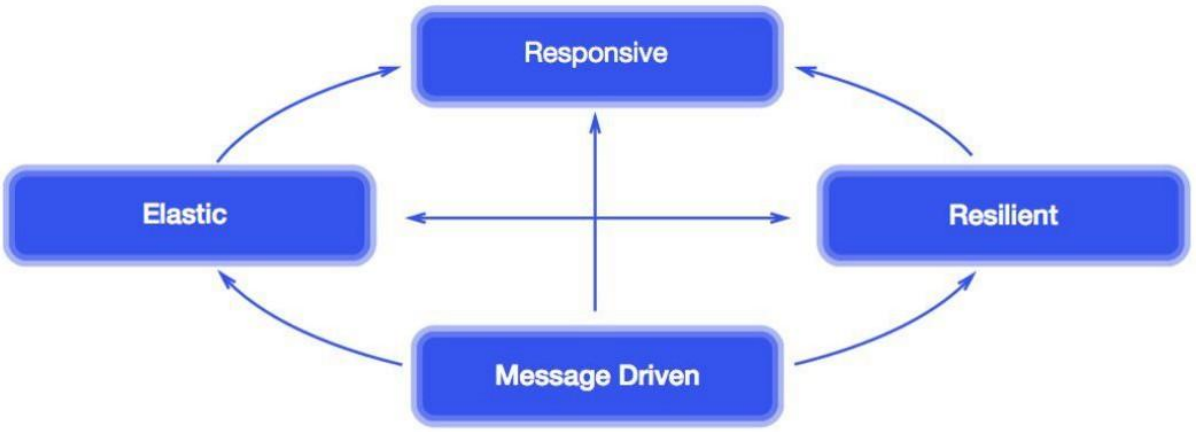
La programación reactiva es un paradigma enfocado en el trabajo con flujos de datos finitos o infinitos de manera asíncrona.

- Paradigma: define un marco de trabajo o forma de programar.
- Streams: flujo de datos.
- Datos asíncronos: no sabemos cuándo se producirán o cuándo llegarán a nuestro sistema.

Ya podemos empezar a hacernos una idea de qué es la programación reactiva: unas directrices de cómo debemos programar nuestro sistema para trabajar con datos que no sabemos cuándo se generan, pero que queremos reaccionar y actuar en consecuencia. Esto parece sencillo de enunciar, sin embargo, ¿por qué es tan difícil de entender? Esto se debe a que mezcla diversos conceptos no tan sencillos:

- Programación funcional.
- Patrón observer.
- Patrón iterator.

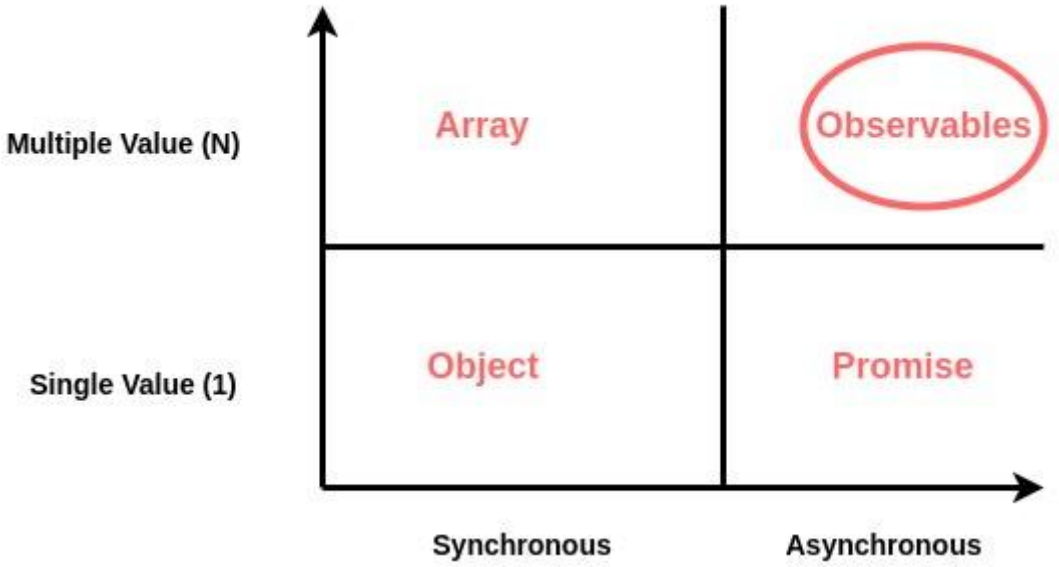
Su concepción y evolución ha ido ligada a la publicación del *Reactive Manifesto*, que establecía las bases de los sistemas reactivos, los cuales deben ser:



- Responsivos: aseguran la calidad del servicio cumpliendo unos tiempos de respuesta establecidos.
- Resilientes: se mantienen responsivos incluso cuando se enfrentan a situaciones de error.
- Elásticos: se mantienen responsivos incluso ante aumentos en la carga de trabajo.

- Orientados a mensajes: minimizan el acoplamiento entre componentes al establecer interacciones basadas en el intercambio de mensajes de manera asíncrona.

La motivación detrás de este nuevo paradigma procede de la necesidad de responder a las limitaciones de escalado presentes en los modelos de desarrollo actuales, que se caracterizan por su desaprovechamiento del uso de la CPU debido al I/O, el sobreuso de memoria (enormes thread pools) y la ineficiencia de las interacciones bloqueantes.



Los flujos de datos transportarán eventos, mensajes, llamadas e incluso fallas. La programación reactiva significa que cuando ve estos flujos, la aplicación puede reaccionar ante ellas utilizando una caja de herramientas para filtrar, crear, transformar y conectar cualquiera de esos flujos.

El código dentro de una aplicación reactiva crea flujos de datos de cualquier cosa y desde cualquier cosa, como solicitudes HTTP, mensajes, notificaciones, cambios a variables, eventos de caché, medidas de sensores y clics. Cuando esto sucede, la aplicación se vuelve intrínsecamente asincrónica.

ReactiveX

ReactiveX, *Reactive eXtensions* o *Extensiones Reactivas* es el nombre de esta tecnología. Las implementaciones más conocidas son las de JavaScript (***RxJS***), Java (*RxJava*) o Scala (*RxScala*) entre muchas otras.

La programación reactiva no despegó hasta la introducción de las extensiones reactivas. Las extensiones reactivas son una API que emplea estos conceptos:.

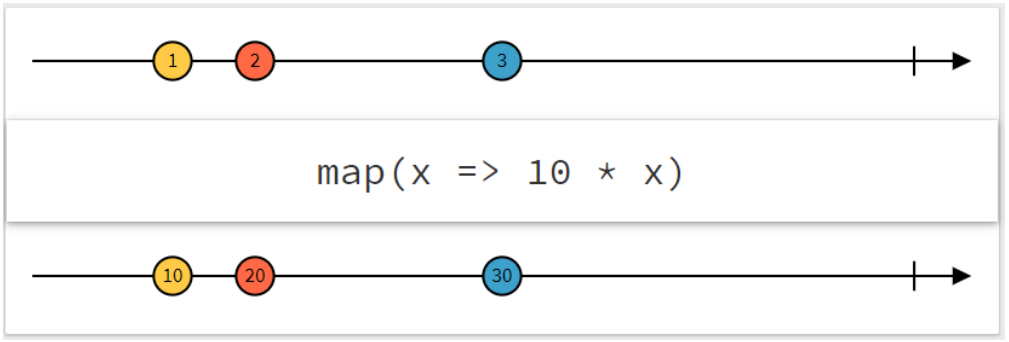
ReactiveX tiene dos clases principales: ***observables*** y ***observadores***:

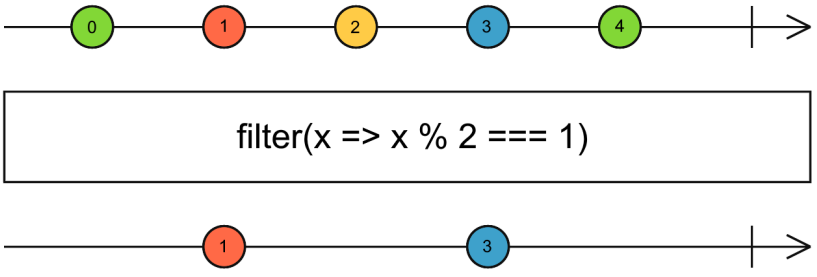
- Una clase *observable* es la fuente de datos de flujos o eventos.
- Una clase *observador* es la que usa (o reacciona) los elementos producidos.

Un observable puede tener varios observadores, por lo que cada observador recibirá cada elemento de datos producido. En ReactiveX un observador se suscribe a un observable. Un observable entonces produce flujos de datos que el observador escucha y reacciona. Esto desencadena una serie de operaciones en un flujo de datos.

Otra clase principal es la de ***sujeto***. Un *sujeto* es una extensión observable que también implementa una interfaz de observador. Esto significa que los sujetos pueden actuar como observadores y observables.

También podemos aplicar ***operadores*** a un flujo. Estos definen cómo y cuándo los observables deberían producir flujos. La mayoría de los operadores ejecutan funciones en un observable y devuelve un observable.





Usar el CDN para importar la librería

Observable con constructor from

El siguiente ejemplo emplea un ***CDN*** que almacena esta librería para hacer uso de ella, de este modo no es necesario descargarla en el servidor.

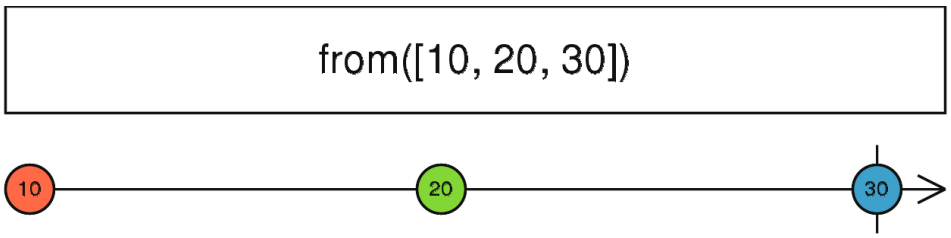
Un ***observable*** es un elemento del que se desea saber cuándo se ha generado un nuevo elemento.

El ***observer*** es un elemento que desea saber cada uno de los nuevos elementos generados por el observable.

La nomenclatura para indicar que una variable almacena un observable es añadir el símbolo de dólar ($) al final del nombre.

En este ejemplo solo se configura el observer para hacer algo cada vez que el observable genera un nuevo elemento. En el proceso de subscripción se aporta la función que se ejecutará para cada elemento.

El operador ***from*** es un operador de construcción de observables. Recibe un arreglo y genera un observable que introduce en la tubería los elementos del arreglo sucesivamente.



El programa muestra sucesivamente los números 20, 40, 60.

- Primero se genera un observable a partir de un arreglo con todos los números entre 1 y 7 empleando el constructor from, que se obtiene deconstruyendo el objeto rxjs obtenido al importar la librería.
- A continuación se filtran los números pares.
- Luego se mapean para multiplicar cada número por 10.
- Después se pone en ejecución el observable. Los observables son perezosos (lazy), y no se ejecutan hasta que no hay una subscripción.
- Finalmente se desubscribe para liberar todos los recursos generados en el proceso de subscripción.

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>RxJS</title>
</head>
<body>
    <script src=
"https://cdnjs.cloudflare.com/ajax/libs/rxjs/6.6.3/rxjs.umd.min.js">
    </script>

    <script>
        const { from } = rxjs;
        const { map, filter } = rxjs.operators;

        const numeros$ = from([1, 2, 3, 4, 5, 6, 7])
            .pipe(
                filter(x => x % 2 === 0),
                map(x => x * 10),
            );

        const subscripcion = numeros$.subscribe(
            elemento => console.log(elemento)
        );
        subscripcion.unsubscribe();

    </script>
</body>
</html>
```

Los tres métodos del observer

El *observer* tiene tres situaciones de interés que pueden ser empleados:

- next: lo que se hace con cada elemento del flujo. Se proporciona la función a ejecutar cada vez que el observable genera un nuevo elemento.
- complete: lo que se hace cuando finaliza el observable. Se proporciona la función a ejecutar cuando el observable ha finalizado.
- error: lo que se hace cuando ocurre un error. Se proporciona la función a ejecutar cuando el observable informa de un error.

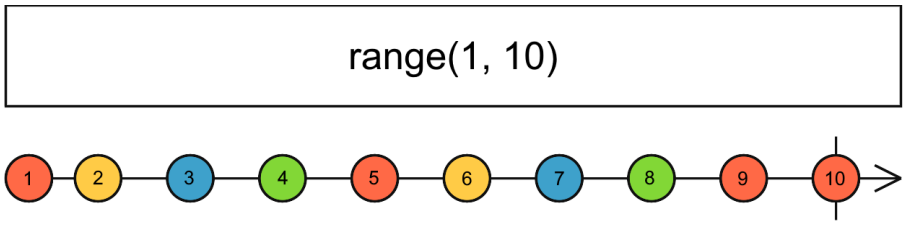
El ejemplo se modifica con el siguiente cambio:

```
        const { from } = rxjs;
        const { map, filter } = rxjs.operators;

        const numeros$ = from([1, 2, 3, 4, 5, 6, 7])
            .pipe(
                filter(x => x % 2 === 0),
```

```
            map(x => x * 10),
        );

    const subscripcion = numeros$.subscribe({
        next: console.log,
        error: console.error,
        complete: () => console.log('Stream finalizado')
    })
    subscripcion.unsubscribe();
```

Ejemplo con constructor range

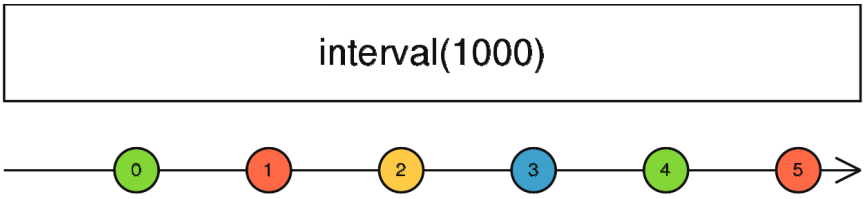
El operador ***range*** es un operador de construcción de observables. Genera todos los números entre los valores especificados.



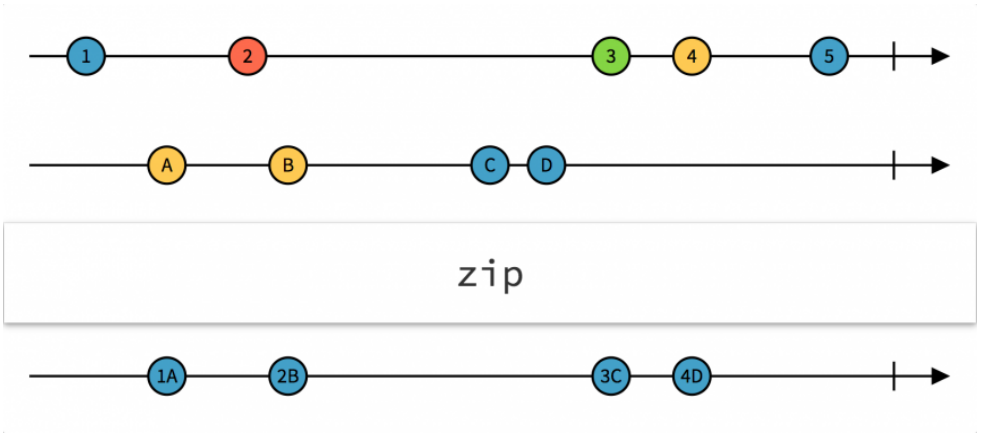

```
    const { range } = rxjs;
    const { map, filter } = rxjs.operators;

    const numeros$ = range(1, 20)
        .pipe(
            filter(x => x % 2 === 0),
            map(x => x * 10),
        );

    const subscripcion = numeros$.subscribe({
        next: console.log,
        error: console.error,
        complete: () => console.log('Stream finalizado')
    })
    subscripcion.unsubscribe();
```

Ejemplo con constructores range, interval, zip y operador delay

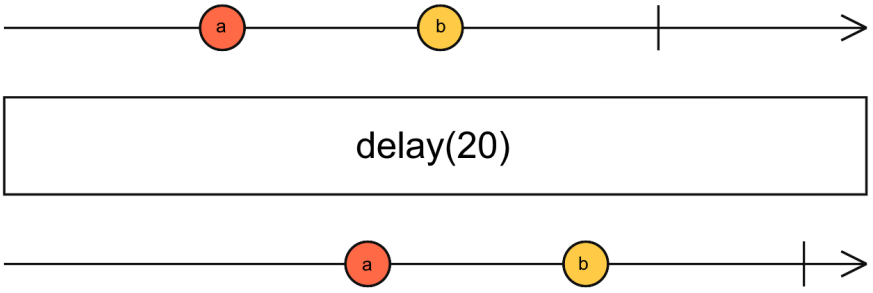
El constructor de observables ***interval*** es un operador de construcción. Genera un elemento cada periodo de tiempo indicado.



El constructor de observables ***zip*** es un operador de combinación. Combina los elementos generados por varios observables en el momento en que hay un elemento disponible en todos y cada uno de ellos.



El operador ***delay*** es un operador de utilidad. Retrasa la entrega de elementos del observable el tiempo especificado.

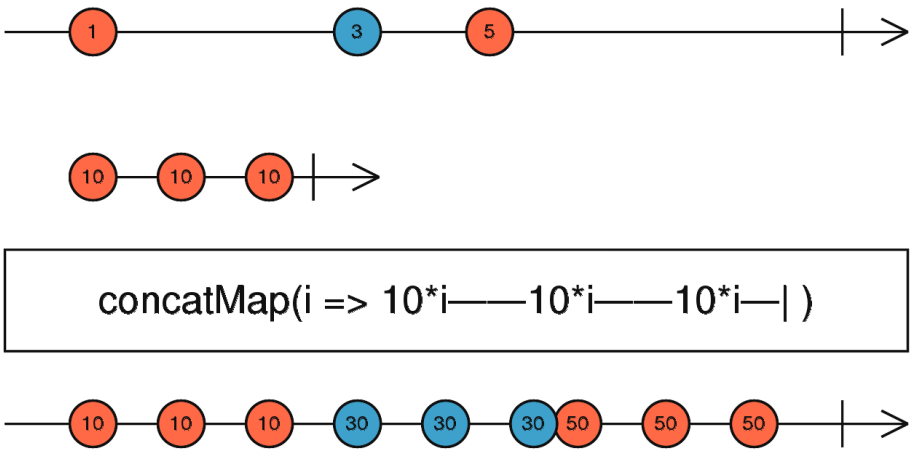



```
const { range, zip, interval } = rxjs;
const { map, filter } = rxjs.operators;

const esPar = x => x % 2 === 0;
const multiplicaPor10 = x => x * 10;

const numeros$ = range(1, 100)
    .pipe(
        filter(esPar),
        map(multiplicaPor10),
    );

const intervalo$ = interval(1000);

zip(
    numeros$,
    intervalo$
).subscribe(([elemento0]) => console.log(elemento0));
```

Ejemplo con constructores of y operadores delay y concatMap

El operador ***concatMap*** es un operador de transformación. Se aplica al elemento generado por un observable, y a partir de él genera un nuevo observable y en el flujo se abandona el observable anterior para ir al nuevo observable. Además el antiguo observable esperará a que termine el nuevo observable.

El argumento de este operador es una función que recibe el elemento actual y devuelve un nuevo observable.




```
const { range, of } = rxjs;
const { map, filter, delay, concatMap } = rxjs.operators;

const esPar = x => x % 2 === 0;
const multiplicaPor10 = x => x * 10;

const numeros$ = range(1, 20)
    .pipe(
        filter(esPar),
        map(multiplicaPor10),
    );

numeros$.pipe(
    concatMap(numero => of(numero).pipe(
        delay(1000)
    ))
).subscribe(console.log);
```

Instalar la librería RxJS con NPM en NodeJS

Los ejemplos ofrecidos a partir de este apartado usan la librería *RxJS* desde ***NodeJS***.

Creación del proyecto e instalación de la librería RxJS

Desde un terminal creamos el directorio destinado al proyecto, entramos en este directorio y creamos el proyecto. Finalmente instalamos la librería.

```
$ mkdir proyecto_hola

$ cd proyecto_hola

$ npm init -y

$ npm install rxjs
```

El último comando instala la última versión de RxJS. Si necesitamos otra versión añadimos al final del nombre el número de versión deseado.

Ejemplo con constructor of

El constructor ***of*** crea un observable que genera un stream con un único elemento y finaliza. El único elemento puede ser de cualquier tipo.

```
const { of } = require('rxjs');

const observable$ = of('HOLA');

const observer = {
    next: mensaje ⇒ console.log(mensaje),
    complete: () ⇒ console.log('Flujo finalizado'),
    error: error ⇒ console.log(error)
}

const subscription = observable$.subscribe(observer);
subscription.unsubscribe()
```

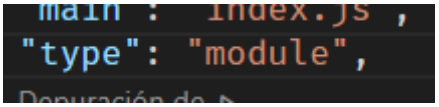
El código empieza importando el constructor *of* a la manera tradicional de *NodeJS*, llamada *CommonJS*. A continuación se crea el observable y se crea el observer. El observer se subscribe al observable para consumir los elementos del stream (flujo) y finaliza liberando los recursos del stream.

El código se ejecuta desde el terminal con node.

```
$ node hola.js
```

Mejora

Se va a cambiar el sistema de importación al estándar ES6. Para ello en el fichero package.json hay que añadir la línea



Realizamos la importación con la sintaxis de EcmaScript 6.

```
// const { of } = require('rxjs');
import { of } from 'rxjs'
```

Arreglo de error

El código funciona pero tiene un error que no se aprecia porque se ejecuta tan rápido que no tiene oportunidad de presentarse o hacerse visible, pero que no debe dejarse sin corregir.

Se pone un retardo para comprobar que el error se produce, como paso previo a solucionarlo. El operador ***delay*** establece un retardo especificado en milisegundos.

Al ejecutar el siguiente programa no aparece ningún resultado por pantalla. El motivo es que se ha desubscrito del observable antes de que este tuviera la oportunidad de mostrar el saludo.

```
import { of } from 'rxjs'
import { delay } from 'rxjs/operators';

const observable$ = of('hola').pipe(
    delay(3000)
);

const observer = {
    next: mensaje ⇒ console.log(mensaje),
    complete: () ⇒ console.log('Flujo finalizado'),
    error: error ⇒ console.log(error)
}

const subscription = observable$.subscribe(observer);
subscription.unsubscribe();
```

La solución parte de que hay que desubscribir una vez el observable haya terminado. La propiedad *complete* del observer permite indicar que se ejecute código cuando el observable informa que ha terminado.

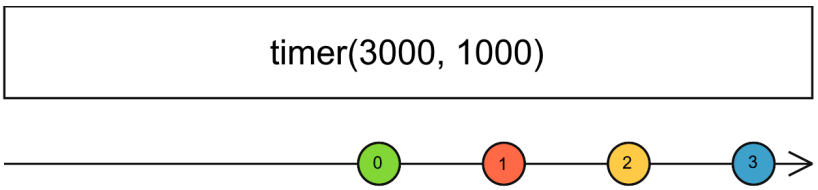
```
import { of } from 'rxjs'
import { delay } from 'rxjs/operators';

const observable$ = of('hola').pipe(
    delay(3000)
);

const observer = {
    next: mensaje ⇒ console.log(mensaje),
    complete: () ⇒ {
        console.log('Flujo finalizado');
        subscription.unsubscribe();
    },
    error: error ⇒ console.log(error)
}

const subscription = observable$.subscribe(observer);
```

Esta versión funciona como se espera, al ejecutarla aparece el saludo.

Constructores variados

Constructor timer

El operador ***timer*** es un constructor de utilidad. Genera números espaciados en el tiempo según se indica en el constructor.



El constructor acepta distintos argumentos, y en este ejemplo el primer argumento es el tiempo de espera antes de emitir números, y el segundo argumento es el intervalo de tiempo entre números.

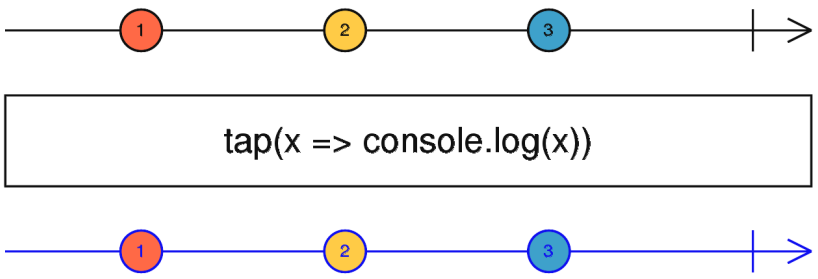
```
import { timer } from 'rxjs';

const secuencia$ = timer(3000, 1000);

secuencia$.subscribe(console.log);
```

Constructor from y operadores filter, map, tap y delay. Dragones

El operador ***from*** recibe como argumento un arreglo y genera un flujo en el que entran sucesivamente los elementos del arreglo.

El operador ***delay*** provoca un retardo en el flujo en general, no un retardo entre los distintos elementos.

El operador ***tap*** permite ejecutar funciones no puras (funciones que producen efectos laterales) de forma segura para el stream.



El operador ***filter*** permite que continúen en el flujo solo los elementos que cumplen el criterio especificado.

El operador ***map*** cambia el elemento actual en el stream por otro elemento, que normalmente es una transformación del elemento actual.

```
import { from } from 'rxjs';
import { filter, map, tap, delay } from 'rxjs/operators';

const dragones = [
    { id: 1, nombre: 'Acarion', longitud: 9, peso: 23000 },
    { id: 2, nombre: 'Besarion', longitud: 15, peso: 65000 },
    { id: 3, nombre: 'Cursarion', longitud: 7, peso: 15000 },
    { id: 4, nombre: 'Divirion', longitud: 14, peso: 54000 },
    { id: 5, nombre: 'Estocarion', longitud: 18, peso: 97000 }
];

const mostrarFin = () => console.log('\nFINALIZADO');

const esGrande = ({ peso }) => peso > 49000;

const cambiarInformacion = ({ nombre }) =>
    `Soy ${nombre} y soy grande`;

const dragones$ = from(dragones)
    .pipe(
        tap(console.log),
        filter(esGrande),
        tap(dragon => console.log(dragon.nombre)),
        map(cambiarInformacion),
        delay(5000)
    )

dragones$.subscribe({
    next: console.log,
    error: console.error,
    complete: mostrarFin
});
```

En este ejemplo:

- Se declara un arreglo de dragones.
- Se muestra con *tap* cada uno de los dragones de forma sucesiva.
- Se filtran con *filter* solo aquellos que son grandes.
- Se cambia con map el contenido del flujo de un dragón a un mensaje.

- Se indica con delay que espere tres segundos para suministrar los elementos del flujo al observer.

Constructor fromEvent

El constructor ***fromEvent*** crea un observable a partir de un evento, es decir, solo aparece un nuevo elemento en el stream cuando se produce el evento.

Este constructor recibe dos argumentos, la etiqueta que genera el evento y el tipo de evento.

En el siguiente ejemplo aparece un mensaje en la consola cada vez que se cliquea el botón.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>

<body>
    <button id="eButMas">Siguiente elemento</button>
    <p id="eParMensaje"></p>

    <script src=
        "https://cdnjs.cloudflare.com/ajax/libs/rxjs/6.6.3/rxjs.umd.min.js">
    </script>

    <script>
        const { fromEvent } = rxjs;

        const boton$ = fromEvent(eButMas, 'click');

        boton$.subscribe(console.log);
    </script>
</body>

</html>
```

Constructor fromEvent y arreglos. Dragones

El constructor ***fromEvent*** crea un observable a partir de un evento, es decir, solo aparece un nuevo elemento en el stream cuando se produce el evento.

El siguiente ejemplo muestra un dragón cada vez que se cliquea el botón. Emplea un contador para determinar cuando ha finalizado de mostrar todos los dragones. En el siguiente apartado se eliminará el contador buscando una solución más profesional.

```html
<!DOCTYPE html>
<html lang="en">
<head>···
</head>
<body>
    <button id="eButMas">Siguiente dragón</button>
    <p id="eParMensaje"></p>

    <script src=
        "https://cdnjs.cloudflare.com/ajax/libs/rxjs/6.6.3/rxjs.umd.min.js">
    </script>
    <script>
        const dragones = [
            { id: 1, nombre: 'Acarion', longitud: 9, peso: 23 },
            { id: 2, nombre: 'Besarion', longitud: 15, peso: 65 },
            { id: 3, nombre: 'Cursarion', longitud: 7, peso: 15 },
            { id: 4, nombre: 'Divirion', longitud: 14, peso: 54 },
            { id: 5, nombre: 'Estocarion', longitud: 18, peso: 97 }
        ]

        const { fromEvent } = rxjs;

        boton$ = fromEvent(eButMas, 'click');
        const subscripcion = boton$.subscribe(mostrarMensaje);
        let contador = 0;

        function mostrarMensaje(elemento) {
            console.log(elemento);
            eParMensaje.innerText = dragones[contador++].nombre;
            if (contador === dragones.length) {
                eButMas.disabled = true;
                subscripcion.unsubscribe();
            }
        }
    </script>
</body>
</html>
```

Mejora con un observable sobre el arreglo y concatMap

Se creará un observable sobre el arreglo con el operador ***from*** (aquí se le pone un alias).

Se emplea el operador ***concatMap***, que permite el cambio del elemento que está en el flujo por un nuevo flujo, es decir, abandonamos momentáneamente el observable actual y cambiamos a otro para luego regresar al observable primero.

Al probar el nuevo desarrollo comprobamos que no funciona de la forma deseada, por lo que en el siguiente apartado buscamos una nueva solución.

```
const { fromEvent, from: fromArray } = rxjs;
const { concatMap } = rxjs.operators;

boton$ = fromEvent(eButMas, 'click');
dragones$ = fromArray(dragones);

stream$ = boton$.pipe(
    concatMap( click ⇒ dragones$ )
)

const subscripcion = stream$.subscribe({
    next: console.log,
    complete: () ⇒ console.log('FINALIZADO'),
    error: console.error
});
```

Mejora con un observable sobre el arreglo y zip

El constructor zip combina varios observables en un único observable en el que cada elemento del nuevo flujo será un arreglo con un elemento de cada observable original en el orden establecido en el constructor.

```
const { fromEvent, from: fromArray, zip } = rxjs;
const { map } = rxjs.operators;

boton$ = fromEvent(eButMas, 'click');
dragones$ = fromArray(dragones);

stream$ = zip(boton$, dragones$).pipe(
    map( ([_, dragon]) ⇒ dragon )
)

const subscripcion = stream$.subscribe({
    next: console.log,
    complete: () ⇒ console.log('FINALIZADO'),
    error: console.error
});
```

En el ejemplo se emplea un mapeo para ignorar el evento y quedarse solo con el dragón. Finalmente este programa funciona correctamente.

Observable y Subject

Un ***Observable*** solo puede emitir el valor a un único observer, mientras que un ***Subject*** puede emitir el valor a varios observers.

En los siguientes ejemplos aparece el mismo código implementado mediante un observable y mediante un subject, y se podrá comprobar que el resultado no es idéntico.

Código con Observable

```
import { Observable } from 'rxjs'

const randomStream$ = new Observable(subscriber ⇒ {
    for (let i = 1; i ⩽ 10; i++) {
        const number = Math.random()
        subscriber.next(number)
    }
})

randomStream$.subscribe(console.log)
randomStream$.subscribe(console.log)
```

Código con Subject

```
import { Subject } from 'rxjs'

const randomStream$ = new Subject()

randomStream$.subscribe(console.log)
randomStream$.subscribe(console.log)

for (let i = 1; i ⩽ 10; i++) {
    const number = Math.random()
    randomStream$.next(number)
}
```

Subject. Velocímetro

El siguiente ejemplo muestra el salpicadero de un vehículo en el que la velocidad se verá simultáneamente en el velocímetro tras el volante y en la pantalla central de infoentretenimiento.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        body {
            display: flex;
        }
        .velocimeter-dashboard {
            width: 100px;
            height: 100px;
            border: solid 5px red;
            border-radius: 50%;
            display: grid;
            place-content: center;
        }
        .velocimeter-centralscreen {
            width: 150px;
            height: 200px;
            border: solid 5px darkgrey;
            display: grid;
            place-content: center;
        }
    </style>
</head>
<body>
    <div class="velocimeter-dashboard"></div>
    <div class="velocimeter-centralscreen"></div>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/7.5.1/rxjs.umd.min.js"></script>
    <script type="module">
        import VelocitySensor from './VelocitySensor.js'

        document.addEventListener('DOMContentLoaded', (e) => {
            const sensor = new VelocitySensor()
            const sensor$ = sensor.getSensor$()

            const velocimeterDashboard = document.querySelector('.velocimeter-dashboard')
            sensor$.subscribe(velocity => {
                velocimeterDashboard.innerText = velocity
            })

            const velocimeterCentralscreen = document.querySelector('.velocimeter-centralscreen')
            sensor$.subscribe(velocity => {
                velocimeterCentralscreen.innerText = velocity;
            })

            sensor.activate()
        })
    </script>
</body>
</html>
```

La velocidad es proporcionada por un *Subject*, y ambos elementos velocímetro y pantalla se suscriben al subject para recibir la misma velocidad.

```
const { Subject, from: fromArray, of } = rxjs
const { concatMap, delay } = rxjs.operators

export default class VelocitySensor {
    constructor() {
        if (!VelocitySensor._sensor$)
            VelocitySensor._sensor$ = new Subject()
    }

    getSensor$() {
        return VelocitySensor._sensor$
    }

    activate() {
        const velocities = [100, 105, 125, 110, 95, 115, 120, 90, 105, 100]
        fromArray(velocities).pipe(
            concatMap(velocity ⇒
                of(velocity).pipe(delay(3000))
            )
        ).subscribe(velocity ⇒ VelocitySensor._sensor$.next(velocity))
    }
}
```

Solicitudes AJAX

Solicitud sencilla GET. Constructor ajax y operadores mergeMap y take

La librería RxJS incluye herramientas para realizar solicitudes AJAX. El método ***ajax*** recibe como parámetros los criterios para hacer una solicitud y devuelve un observable que permite acceder a los valores devueltos por la solicitud.

```
import { ajax } from 'rxjs/ajax'
import XMLHttpRequest from 'xhr2'
import { map, mergeMap, take } from 'rxjs/operators'

obtenerPaises$().subscribe(console.log)

function obtenerPaises$() {
    const extraerPaises = ({ response: paises }) ⇒ paises
    const cogerNombreEnEspanol = ({ translations: { es: nombre } }) ⇒ nombre

    return ajax({
        url: 'https://restcountries.com/v2/all',
        createXHR: () ⇒ new XMLHttpRequest(),
        method: 'get',
        responseType: 'json',
        crossDomain: true,
        withCredentials: false,
    }).pipe(
        map(extraerPaises),
        mergeMap(paises ⇒ paises),
        take(10),
        map(cogerNombreEnEspanol),
    )
}
```

En este ejemplo se emplean estos elementos:

- Se definen unas funciones de ayuda.
- Se establece el endpoint que se consumirá.
- Se crea un elemento ***XMLHttpRequest*** que es necesario para AJAX.
- Se indican algunos parámetros de la consulta AJAX.
- Se usa ***map*** para extraer la parte de la respuesta de todo lo recibido.
- Se usa ***mergeMap*** para convertir el único arreglo recibido en un stream de países en secuencia (en programación se denomina flat, flatten, flattering).
- Se usa ***take*** para tomar solo 10 países.
- Se usa ***map*** para extraer solo el nombre.

Solicitudes GET en secuencia

El siguiente ejemplo logra realizar varias operaciones AJAX en secuencia dentro del mismo stream.

```
import { zip, of, iif } from 'rxjs'
import { ajax } from 'rxjs/ajax'
import { map, mergeMap, concatMap, take } from 'rxjs/operators'
import XMLHttpRequest from 'xhr2'

obtenerPaises$().subscribe(console.log)

//-------------------------------------------------------------------
function obtenerPaises$() {
    const extraerPaises = ({ response: paises }) => paises

    const cogerCodigoNombreEspanolFronteras =
        ({
            name: nombreIngles,
            alpha3Code: codigo,
            translations: { es: nombreEspanol }, borders: fronteras
        }) =>
        ({
            codigo,
            nombre: nombreEspanol ? nombreEspanol : nombreIngles,
            fronteras: fronteras ? fronteras : []
        })

    return ajax({
        url: 'https://restcountries.com/v2/all',
        createXHR: () => new XMLHttpRequest(),
        method: 'get',
        responseType: 'json'
    }).pipe(
        map(extraerPaises),
        mergeMap(paises => paises),
        take(10),
        map(cogerCodigoNombreEspanolFronteras),
        concatMap(pais =>
            zip(
                of(pais.nombre),
                obtenerNombresPaises$(pais.fronteras),
            )
        ),
        map(([nombre, fronteras]) => ({ nombre, fronteras }))
    )
}
//-------------------------------------------------------------------
function obtenerNombresPaises$(codigos) {
    return iif(
        () => codigos.length === 0,
        of([]),
        zip( ...codigos.map(obtenerNombrePais$))
    )
}
//-------------------------------------------------------------------
function obtenerNombrePais$(codigo) {
    const extraerPais = ({ response: pais }) => pais
    const cogerNombreEspanol = ({ translations: { es: nombre } }) => nombre

    return ajax({
        url: `https://restcountries.com/v2/alpha/${codigo}`,
        createXHR: () => new XMLHttpRequest(),
        method: 'get',
        responseType: 'json'
    })
        .pipe(
            map(extraerPais),
            map(cogerNombreEspanol),
        )
}
```

Las operaciones que dan sentido al código son:

- Siguiendo la nomenclatura de esta tecnología, las funciones cuyo nombre termina en dólar ($) devuelven un observable que envuelve el contenido descrito por el nombre.
- El operador ***concatMap*** cambia del observable actual a el nuevo observable resultado de la función suministrada como argumento, por el que seguirá el flujo. Cuando termina de trabajar con el nuevo observable retorna el flujo al anterior observable.
- El constructor ***iif*** recibe como argumentos una función que se evalúa como true o false, y dos observables, y si la función es true el constructor devuelve el primer observable y si es false el segundo.
- El constructor ***zip*** recibe como argumentos varios observables y los ejecuta todos, generando en el flujo un elemento que es un arreglo con los elementos generados por cada observable en el orden en el que están declarados los observables.
- El operador de JavaScript spread (...) que se emplea para descomponer un arreglo en sus elementos constituyentes.

Super ejemplo de países con operadores propios

```
import { from, Observable, of, zip } from 'rxjs'
import { map, take, mergeMap, concatMap, catchError, retry, pluck } from
'rxjs/operators'
import { ajax } from 'rxjs/ajax'
import XMLHttpRequest from 'xhr2'

retrieveCountries$().subscribe(console.log)


function retrieveBordersNames$(countries) {
    return from(countries).pipe(
        concatMap( code => retrieveCountryByCode$(code)),
        pluck('name'),
        unflat,
    )
}

function unflat(elements$) {
    return new Observable(subscriber => {
        const result = []
        elements$.subscribe({
            next: element => result.push(element),
            complete: () => {
                subscriber.next(result)
                subscriber.complete()
            },
            error: console.error,
        })
    })
}

function retrieveCountryByCode$(code) {
    const url = `http://127.0.0.1/api/countries/${code}`

    return ajax({
        url,
        createXHR: () => new XMLHttpRequest()
    }).pipe(
        parseJson(),
        extractCountry(),
        retry(10)
    )
}

function extractCountry() {
    return map(({country}) => country)
}

function extractCodeNameAndBorders() {
    return map(({alpha3Code: code, name, borders}) => ({
        code,
        name,
```

```
        borders: borders ? borders : []
    }))
}

function parseJson() {
    return map(({response : data}) => data)
}

function extractCountries() {
    return map(({countries}) => countries)
}

function flatCountries() {
    return mergeMap(countries => countries)
}

function retrieveCountries$() {
    const url = 'http://127.0.0.1/api/countries'

    return ajax({
        url,
        createXHR: () => new XMLHttpRequest()
    }).pipe(
        parseJson(),
        extractCountries(),
        flatCountries(),
        take(15),
        concatMap(country => retrieveCountryByCode$(country.code)),
        extractCodeNameAndBorders(),
        concatMap(({code, name, borders}) => retrieveBordersNames$(borders)
            .pipe(
                concatMap(borders => of({code, name, borders}))
            )
        ),
        catchError(error => of('Hay un error')),
    )
}
```