Introducción

RxJS es una biblioteca para componer programas asíncronos y basados en eventos, mediante secuencias observables. Proporciona un tipo *core*, el Observable, varios tipos satélite (Observer, Schedulers, Subjects) y operadores inspirados por las funciones de Array (`map`, `filter`, `reduce`, `every` etc.) para manejar eventos asíncronos como si fuesen colecciones.

> *Podemos considerar a RxJS como el Lodash para eventos.*

ReactiveX combina el patrón Observador con el patrón Iterador y la programación funcional con colecciones, constituyendo así la solución ideal para manejar secuencias de eventos.

Los conceptos esenciales de RxJS que resuelven el manejo asíncrono de eventos son los siguientes:

Observable: representa la idea de una colección invocable de valores futuros o eventos.

Observador: es una colleción de *callbacks* que sabe cómo escuchar a los valores proporcionados por el Observable.

Suscripción: representa la ejecución de un Observable; es muy útil a la hora de cancelar la ejecución.

Operadores: son funciones puras que permiten enfocar el manejo de las colecciones desde un estilo de programación funcional, con operaciones como `map`, `filter`, `concat`, `reduce` etc.

Sujeto: es el equivalente a un EventEmitter, y la única manera de multidifundir un valor o un evento a múltiples Observadores.

Planificadores: son despachadores centralizados para controlar la concurrencia, permitiendo coordinar cuándo ocurrirá la computación en `setTimeout`, `requestAnimationFrame` u otros.

Primeros Ejemplos

Normalmente, tenemos que registrar *event listeners*.

```
document.addEventListener("click", () => console.log("Clicked!"));
```

Copy

En lugar de hacerlo así, RxJS nos permite crear un Observable:

```
import { fromEvent } from "rxjs";

fromEvent(document, "click").subscribe(() => console.log("Clicked!"));
```

Pureza

Lo que hace que RxJS sea tan potente es su habilidad para producir valores mediante funciones puras. Esto equivale a un código menos propenso a errores.

Normalmente, se tendría que crear una función impura, planteando la posibilidad de que otros fragmentos del código puedan interferir con el estado.

```
let count = 0;
document.addEventListener("click", () =>
  console.log(`Clicked ${++count} times`)
);
```

Usando RxJS, se puede aislar el estado.

```
import { fromEvent } from "rxjs";
import { scan } from "rxjs/operators";

fromEvent(document, "click")
  .pipe(scan((count) => count + 1, 0))
  .subscribe((count) => console.log(`Clicked ${count} times`));
```

El operador `scan` funciona exactamente igual que el `reduce` para arrays. Recibe un valor que se le proporciona a una *callback*. El valor retornado por la *callback* se convierte en el siguiente valor que se proporcionará a la *callback*, la siguiente vez que esta sea ejecutada.

Flow

RxJS tiene una gran cantidad de operadores que permiten controlar cómo fluyen los eventos a través de los Observables.

Así es como se permitiría únicamente un click por segundo, en JavaScript 'vainilla':

```
let count = 0;
let rate = 1000;
let lastClick = Date.now() - rate;
document.addEventListener("click", () => {
  if (Date.now() - lastClick >= rate) {
    console.log(`Clicked ${++count} times`);
    lastClick = Date.now();
  }
});
```

Con RxJS:

```
import { fromEvent } from "rxjs";
import { throttleTime, scan } from "rxjs/operators";

fromEvent(document, "click")
  .pipe(
    throttleTime(1000),
    scan((count) => count + 1, 0)
  )
  .subscribe((count) => console.log(`Clicked ${count} times`));
```

Otros operadores para el control del flujo son `filter`, `delay`, `debouncetime`, `take`, `takeUntil`, `distinct`, `distinctUntilChanged` etc.

Valores

Los valores que pasan a través de un Observable se pueden transformar.

A continuación se muestra un ejemplo de cómo sumar la posición `x` del ratón por cada click, en JavaScript 'vainilla':

```
let count = 0;
const rate = 1000;
let lastClick = Date.now() - rate;
document.addEventListener("click", (event) => {
  if (Date.now() - lastClick >= rate) {
    count += event.clientX;
    console.log(count);
    lastClick = Date.now();
  }
});
```

Con RxJS:

```
import { fromEvent } from "rxjs";
import { throttleTime, map, scan } from "rxjs/operators";

fromEvent(document, "click")
  .pipe(
    throttleTime(1000),
    map((event) => event.clientX),
    scan((count, clientX) => count + clientX, 0)
  )
  .subscribe((count) => console.log(count));
```

Otros operadores que producen valores son `pluck`, `pairwise`, `sample` etc.