
# La Programación Reactiva:

## Un nuevo enfoque para trabajar con código asincrónico en la programación web

*Trabajo final presentado para obtener el grado de Especialista en Ingeniería del Software*

**Director**: Dr. Federico Balaguer

**Alumno**: Lic. Malqui Fernandez Vidal

# Índice general

# 1.Introducción

La mayoría de los sitios web actuales usan Javascript y todos los web browsers modernos, en computadoras de escritorio, notebooks, tabletas y teléfonos incluyen intérpretes de Javascript. Además Javascript puede ejecutarse también fuera de los browser mediante plataformas como Node.Todo lo anterior hace de Javascript uno de los lenguajes más usados de la actualidad[1] pero el manejo de eventos en Javascript , necesario para obtener recursos remotos y manejar complejas interacciones de usuario se realiza de manera imperativa.Esto conduce a problemas como código con alto acoplamiento y baja reusabilidad.

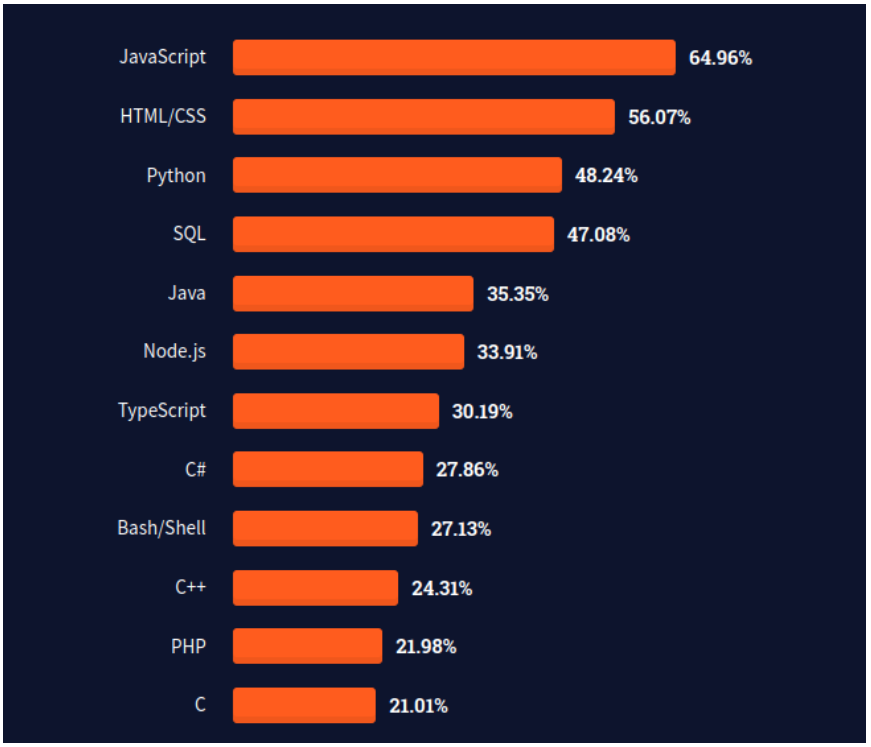


Imagen 1. Popularidad de lenguajes

El presente trabajo utiliza abstracciones de la programación reactiva, un paradigma basado en la propagación del cambio, como solución a los problemas del manejo imperativo de eventos implementado en Javascript. El análisis de la bibliografía sobre el tema permitió detectar los siguientes problemas:

---

1 https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-programming-scripting-and-markup-languages

- **Variables Condicionales**. En el desarrollo web muchas veces es necesario escuchar un evento hasta que otro se produce. Esta técnica es la base de funcionalidades como el drag and drop o las barras de progreso. Implementar estas funcionalidades mediante el manejo imperativo de eventos conduce al uso de Variables Condicionales cuya finalidad es únicamente detectar si se produce o no un evento. En el trabajo caracterizamos el problema mediante el ejemplo de un timer y presentamos la solución basada en abstracciones de la programación reactiva.
- **Backpressure**.-Las condiciones de carrera también pueden presentarse en la programación web. Un ejemplo son las cajas de búsqueda. Con cada letra que presiona el usuario se realiza una petición a un servidor remoto el cual devuelve las coincidencias con la palabra escrita en la caja de texto. Pero cada respuesta no necesariamente llega en el orden en que se envió. Es responsabilidad del programador evitar que se produzcan resultados inesperados, pero las herramientas que provee Javascript para manejar dichas condiciones de carrera son complejas y poco adecuadas. En el capítulo 3 en la sección Backpressure se estudia este problema y en el capítulo de soluciones se muestran las ventajas que tienen la programación reactiva para manejar dichas situaciones.
- **Wait for all**. Las aplicaciones web modernas consumen recursos de varios servidores. Puede ser necesario esperar la respuesta de todos los servidores antes de efectivamente trabajar con esos datos- problema Wait for all-, o por el contrario, puede ser necesario trabajar con la primera respuesta descartando las demás-problema Wait for one-. En ambos casos se requiere de variables globales y estructuras condicionales cuya única finalidad es verificar que arribaron todas las respuestas o que arribó la primera pero que no tienen que ver con la tarea que se quiere realizar con dichas respuestas. En este trabajo estudiamos el problema del Wait for all y aplicamos la solución basada en programación reactiva. Dicha solución aplica también al problema Wait for one.

**1.1 Alcance del Trabajo**. La programación reactiva puede aplicarse a diversos escenarios como las interfaces gráficas de usuario, animación, robótica, simulación o visión por computadora **(Oliveira, 2017, 8)** . En particular el presente trabajo se centra en las interfaces gráficas de usuario y manejo de eventos en el ámbito web.

**1.2 Metodología.-** El trabajo se basa en el análisis de bibliografía sobre programación web , Javascript, programación reactiva entre otros temas relacionados. En base a este análisis se identificaron por un lado los problemas que presenta el manejo de eventos en Javascript en el ámbito web y por otro lado las soluciones a los mismos desde el enfoque de la programación reactiva. Se consultaron asimismo recursos web principalmente documentación sobre frameworks y librerías relacionados con la programación reactiva. El trabajo requirió además de codificación de ejemplos en lenguaje Javascript.

**1.3 Contenido del trabajo.-** En el capítulo 2 veremos las principales definiciones y abstracciones de la Programación Reactiva como los Observables.
En el capítulo 3 se da un panorama de los problemas propios del manejo imperativo de eventos de Javascript que detectamos en el ámbito web .
En el capítulo 4 se presentan soluciones, basadas en Programación Reactiva, a los problemas planteados en el capítulo anterior
El capítulo 5 da un panorama de la adopción de Programación Reactiva en frameworks y librerías actuales de desarrollo web y mobile.
Finalmente el capítulo 6 presenta las conclusiones del trabajo.

# 2.Programación reactiva

La programación reactiva es un paradigma que gira alrededor de la propagación del cambio. En otras palabras, si un programa, módulo o sistema propaga todos los cambios que modifican sus datos a todas las partes interesadas(usuarios, otros programas, componentes y subpartes) entonces el programa módulo o sistema se puede llamar reactivo **(Tsvetinov, 2015, 2)**.

La mayoría de soluciones de programación reactiva soportan dos tipos de abstracciones :
**Event Streams**: Son las abstracciones que manejan el arribo de valores o datos provenientes de eventos discretos o discontinuos en el tiempo. Por ejemplo, los eventos de un botón de un formulario pueden ser representados como Event Streams.
**Behaviors o Signals:** Son las abstracciones que manejan el arribo de valores o datos provenientes de eventos continuos en el tiempo. Por ejemplo, un timer puede ser representado como un Behavior o Signal.**(Kambona, 2013)**

Combinando Event Streams y Behaviors se crea una red de dependencias.Los datos o valores fluyen a través de esta red desde varias fuentes de eventos como por ejemplo los eventos de un botón de formulario o las posiciones del cursor. **(van der Plas, 2017)**

Cada vez que un EventStream o Behavior cambia, un evento se dispara y se propaga a las dependencias del EventStream o Behavior.Como el evento contiene el valor del EventStream o Behavior, las dependencias pueden actualizar sus valores. A su vez las dependencias disparan un nuevo evento de modo que las subsecuentes dependencias también se actualizan **(Parker, 2013 )**.

## 2.1 Programación reactiva: comparación con otros paradigmas

Programación Imperativa. La programación imperativa está formada por enunciados que ayudan a cambiar el estado de un programa **(Noring, 2018, 98)**. El programador se encarga de establecer los pasos que la computadora debe realizar para llevar a a cabo una tarea.Es decir se centra en los detalles de implementación. **(Mezzalira, 2018, 7)**.
En comparación la programación reactiva nos da construcciones para propagar el cambio de modo que podemos concentrarnos en qué hacer en lugar de en cómo hacerlo **(Oliveira, 2017, 7)**. Es decir con Programación Reactiva el código es declarativo. **(Doglio, 2016, 11)**

Data Flow : La programación Data Flow trabaja con el movimiento de datos o información.Los elementos principales son los bloques-contenedores de código-y enlaces que llevan datos de un bloque a otro.El principal punto a entender es que es el dato arribando al bloque lo que causa que se ejecute. El programador no necesita explícitamente sondear el input para ver si el dato está disponible. Por esto se ha usado el término "Reactive programming" para definir dataflow. **(Carkci, 2014, 1)**.
En comparación La programación reactiva puede verse como Data Flow mejorado debido que ganamos el concepto de streams, una fuente de datos a la que nos podemos suscribir aplicar transformaciones y obtener nuevos streams. **(Doglio, 2016, 3)**

Programación Funcional: La Programación Funcional es un subconjunto de la Programación Declarativa **(Van Roy & Haridi, 2004, 29**). Es un paradigma basado en funciones, en el sentido matemático de la palabra.
Deliberadamente evita compartir estado mutable, esto implica el uso de estructuras de datos inmutables y el énfasis en la composicionalidad. **(Blackheath & Jones,2016,4)**
En comparación muchas implementaciones de Programación Reactiva hacen uso de nombres y conceptos de la Programación funcional para efectivizar la transformación de los streams. Por ejemplo map en la librería Sodium **(Blackheath & Jones, 2016, 30)** ,map en Bacon.js **(Oliveira, 2017, 10)** o map en RxJava. **(Tsvetinov, 2015, 52)**
En la literatura sobre programación reactiva muchas veces se habla de programación funcional reactiva para referirse a estas implementaciones **(Doglio, 2016, 8)** .Otros autores consideran que se ha hecho un abuso del término. **(Borges, 2015, 15)**

## 2.2 Streams y el lugar de la programación reactiva

Tradicionalmente el término stream se ha usado en programación como un objeto relacionado a operaciones de entrada salida (I/O) tales como leer un archivo, leer un socket, o hacer una request http **(Daniels & Atencio, 2017, 17)** . Los streams pueden ser **(Peelen, 2016)**:

**Sincrónicos**.- Son aquellos que entregan información a una tasa constante hasta que se detiene su consumo. Es el cliente quien va solicitando información al stream.

**Asincrónicos**.- Por lo general no entregan información a una tasa constante. El cliente no sabe cuando va a arribar el próximo valor**(Peelen, 2016)**. Es el propio stream quien hace la notificación.

Con esta idea podemos ver los eventos que se producen en una aplicación web como un stream. Por ejemplo podemos ver los clicks sobre un botón de un formulario como una secuencia de eventos que se producen en tiempo real a medida que el usuario hace click **(Meijer, 2012)**. El lugar de la Programación Reactiva es justamente el manejo de estas secuencias asincrónicas de eventos o datos, considerando que la mayoría de lenguajes de programación no están bien equipados para manejar estas computaciones asíncronas que resultan en múltiples valores **(Bertoluzzo, 2017)**.

## 2.3 Soporte en Javascript para programación reactiva

El soporte en Javascript para programación reactiva puede clasificarse en lenguajes y extensiones. Los lenguajes reactivos requieren que el programador desarrolle sus aplicaciones en dicho lenguaje que posteriormente debe ser compilado a Javascript. Como ejemplo tenemos a Flapjax **(Meyerovich, 2009, 1-20)** y Elm **(Czaplicki & Chong, 2013, 411-422)**.

Las extensiones o librerías agregan a Javascript construcciones reactivas dentro del mismo lenguaje. Por ejemplo las librerías Rxjs[2],BaconJs[3],Kefir[4],Mobx[5].

[2] https://rxjs-dev.firebaseapp.com/

[3] https://baconjs.github.io/
[4] https://kefirjs.github.io/kefir/
[5] https://mobx.js.org/README.html

## 2.4 Una implementación de programación reactiva

En este trabajo utilizaremos la librería Rxjs[6] para implementar funcionalidad reactiva a Javascript. Rxjs es una implementación en Javascript de las Extensiones Reactivas. Estas extensiones brindan un Api para el manejo de streams asincrónicos[7]. Rxjs es una de las librerías de programación reactiva más ampliamente usadas **(Zimmerle & Gama, 2018, 69-72)**
La principal abstracción de esta Api es la noción de Observable, que además es usada en otras librerías reactivas **(Koutnik, 2019, 10)** como las anteriormente mencionadas BaconJs, Kefir y Mobx. Un Observable es un objeto en el que podemos suscribirnos para escuchar eventos, por ejemplo podemos crear un Observable para los eventos click de un botón de formulario y entonces escuchar y actuar cuando el click sucede **(Oliveira, 2017,18)** además estos Observables pueden ser transformados usando Operadores.**(Oliveira, 2017,92)**
El observable como abstracción, es decir como una interface que puede ser implementada por diversos lenguajes de programación,se basa en los Patrones de Diseño Observer, Iterator y en ideas de Programación Funcional **(Daniels & Atencio, 2017, 3)** como por ejemplo la noción de Functores.

### 2.4.1 Observables y Patrones de Diseño

El Observable toma ideas de dos Patrones de Diseño , el Iterador y el Observer.
Como vemos en la imagen 2 , extraída del libro GOF **(Gamma, 2002, 11)** no se documenta una relación entre estos dos patrones de diseño. Sin embargo ambos son útiles para manejar Streams.

[6] https://rxjs-dev.firebaseapp.com/

[7] http://reactivex.io/

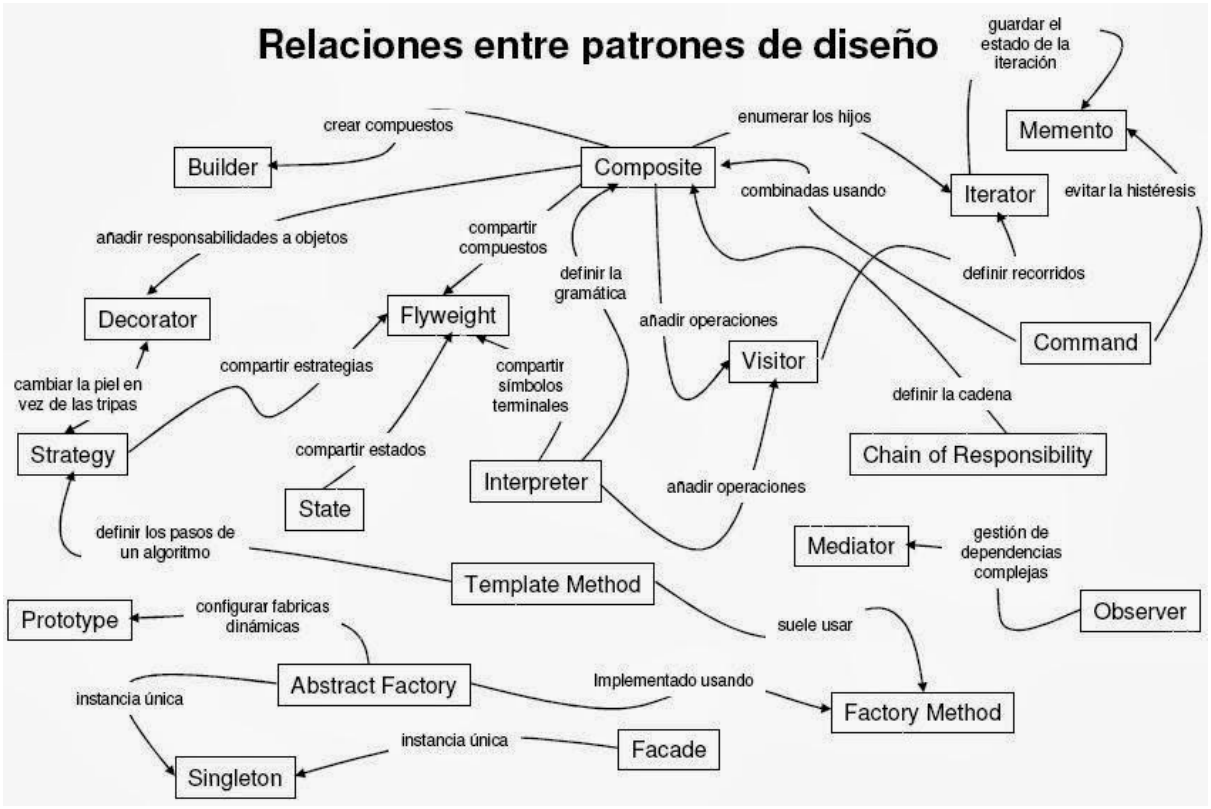


Imagen 2 Relaciones Entre Patrones de Diseño

Sabemos que el Patrón Iterator nos permite acceder a los elementos de una lista o una colección extrayendo uno a uno los elementos de la colección o de la lista **(Tsvetinov, 2015, 6)**. Recordemos además que el Patrón Iterator proporciona métodos para saber si hay items en la colección y para extraer el siguiente ítem **(Gamma, 2002, 238)**. Pero nótese que en este modelo es el consumidor de los datos quién va solicitando uno a uno los datos **(Daniels & Atencio, 2017, 50)**, esto corresponde a lo que se conoce como Iterator externo**(Gamma, 2002, 238)**.

Dado que es el consumidor quien está a cargo ,el Patrón Iterator puede usarse para manejar Streams Sincrónicos. Pero en escenarios como escuchar el click de un botón donde el consumidor no tiene manera de saber cuándo la próxima pieza de información va a estar disponible, no podemos usar el Iterator **(Daniels & Atencio, 2017, 51)**.

En este caso el productor es el responsable por crear el próximo ítem mientras que el consumidor solo escucha por nuevos eventos **(Daniels & Atencio, 2017, 51)**. Si necesitamos de un productor de datos que notifique a los consumidores de los mismos entonces podemos usar el Patrón de Diseño Observer para manejar estos Streams Asincrónicos.

Desde el punto de vista del manejo de Streams El Observable es una extensión del Patrón Observer. En la Imagen 3 vemos la interfaz IObservable de las Extensiones Reactivas[8] que se corresponde con el Subject del Patrón Observer

```
interface IObservable<T>{
    0 references
    IDisposable Subscribe(IObserver<T> observer);
}
```

Imagen 3 Interfaz IObservable

El método Subscribe se utiliza para registrar IObservers que corresponden a los Observers del Patrón Observer. En la Imagen 4 vemos la interfaz del IObsever.

Es en esta interfaz donde está la diferencia. Así como Patron Iterator puede indicar a sus clientes que no hay más ítems o que se produjo un error. El Observable puede notificar a sus IObservers no solo el siguiente dato sino también que no hay más datos( OnCompleted) o que se produjo un error(OnError).

```
interface IObserver<T>{
    0 references
    void OnNext(T value);
    0 references
    void OnError(Exception error);
    0 references
    void OnCompleted();
}
```

Imagen 4 Interfaz IObserver

### 2.4.2 Functores y Observables

Una de las capacidades de los Observables es soportar diversos operadores **(Willems, 2016, 8)** que se usan para filtrar, transformar y combinar los Observables.Estos operadores se aplican a los observables y devuelven otro Observable. Por ejemplo un Observable puede contener una lista de números del 1 al 100, le podemos aplicar un operador filter para obtener los números mayores que 10 y obtenemos un Observable con solo esos números.

Es en este sentido que se dice que los Observables preservan la estructura, como los functores de la programación funcional. **(Daniels & Atencio, 2017, 35) (Stewart et al., 2009, 249) (Daniels & Atencio, 2017, 44).**

[8] http://reactivex.io/

## 2.5 Diagramas Marble

Hemos visto que a los Observables podemos aplicar distintos operadores. Para comprender las soluciones planteadas a partir de los Observables y sus operadores necesitamos entender qué transformaciones generan los operadores en los Observables a los que se aplican.
Para ello usamos una representación visual estándar de las secuencias ,denominadas Diagramas Marble. Estos diagramas representan secuencias(streams) asincrónicas de datos **(Mansilla, 2015, 17).**
Un Diagrama Marble(ver imagen 5) consiste de una flecha que representa el paso del tiempo de izquierda a derecha **(Noring, 2018, 147)**. Hay círculos en esa flecha que representan los valores emitidos. Los marbles tienen dentro un valor y la distancia entre ellos puede también describir lo que está sucediendo en el tiempo. Un Diagrama Marble consiste de por lo menos dos flechas con marbles en ellas como también un operador. La idea es describir qué le sucede al stream **(Tsvetinov, 2015, 52)**.
La pequeña línea vertical representa que el stream completó .

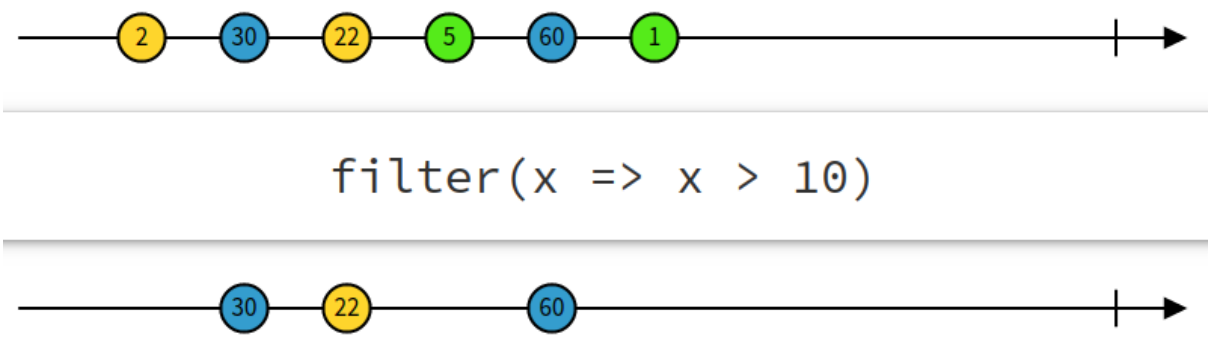


Imagen 5 Diagrama Marble[9] del operador filter

Para finalizar presentamos un ejemplo que documentamos con diagramas Marble.
***Requerimientos***
Tenemos una página html con un botón que requiere la siguiente funcionalidad:
-Al hacer click en el botón se escribirá un texto en una posición aleatoria de la página.
-El botón deberá ejecutarse no más de dos veces, es decir solo se escribirá un máximo de dos veces en la página .En la imagen 6 un ejemplo de ejecución.

[9] https://rxmarbles.com/#filter

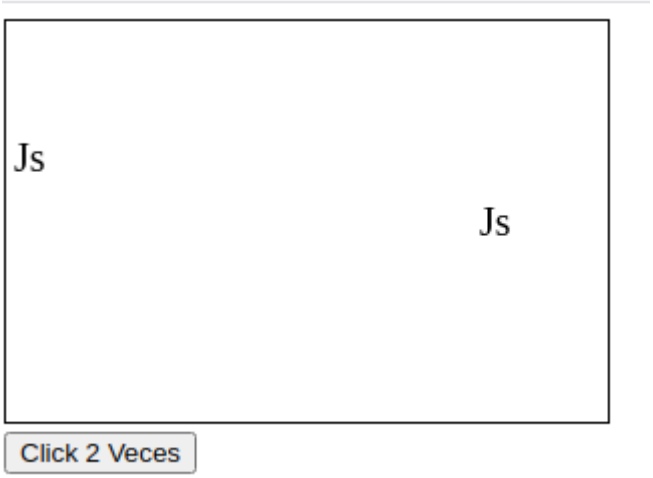


Imagen 6 Click dos veces

En el  el código 1 vemos la forma de  cumplir con el requerimiento

```
1//referencia a las librerias rxjs
2 const { fromEvent } = require("rxjs");
3 const { take, map } = require("rxjs/operators");
4 //obtiene referencia al boton
5 let btn = document.getElementById("btn");
6
7 let observableDosClicks = fromEvent(btn, "click").pipe(
8 take(2),
9 map((eve) => generarPosicion())
10);
11
12 let observador = {
13  next: (pos) => generarTexto(pos),
14  error: (err) => console.log(err),
15 complete: () => console.log("No hay mas clicks"),
16 };
17 //subscripcion al observable
18 observableDosClicks.subscribe(observador);
```

Código 1 Click dos veces

En las primeras líneas importamos la librería rxjs para agregar funcionalidad reactiva a Javascript.

En la línea 5 obtenemos una referencia al botón cuyos clicks queremos escuchar.

En la línea 7 creamos un Observable para escuchar  los clicks del botón.

Para cumplir con el requerimiento necesitamos escuchar solo en dos clicks del botón.

El operador take, cuyo diagrama marble se muestra en la imagen 7, sirve para este propósito

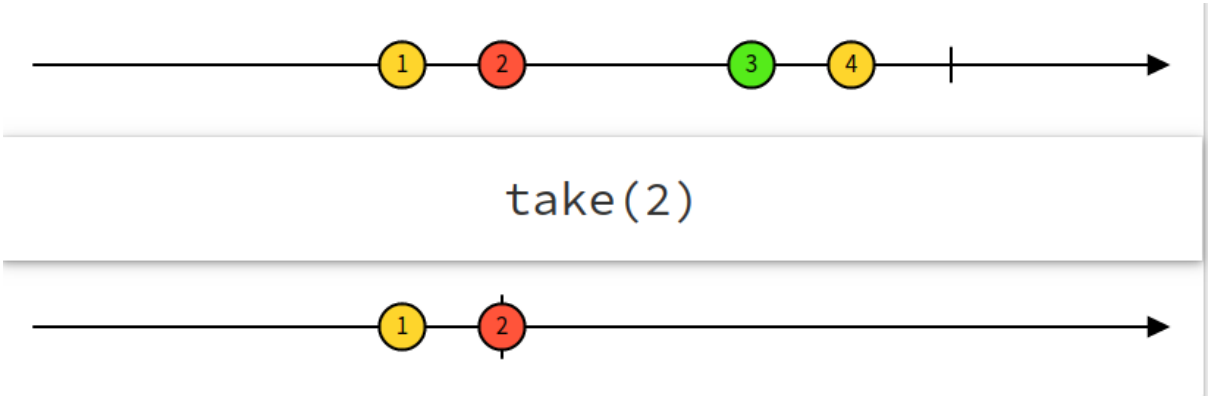


Imagen 7 Operador Take

El operador take devuelve un Observable con solo los primeros n valores del Observable original. En este caso n=2.

Además por cada click necesitamos obtener una posición aleatoria para escribir el texto. El operador map cuyo diagrama marble vemos en la imagen 8 permite aplicar una función a cada valor del Observable original y obtener un nuevo Observable con los resultados.

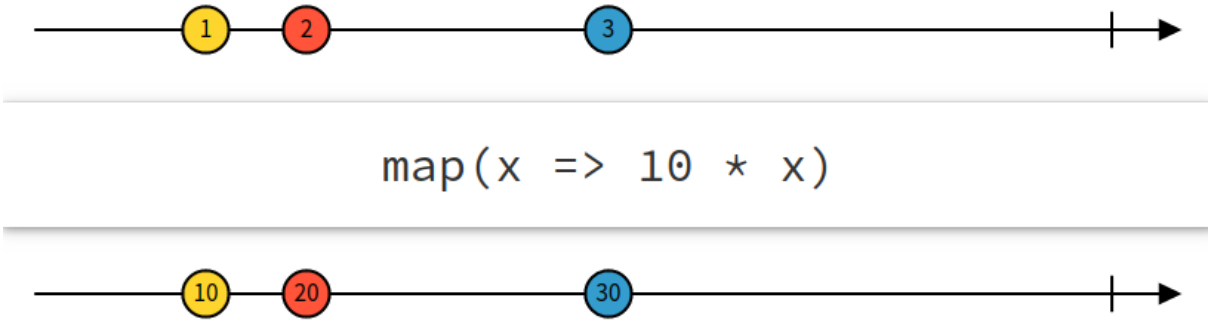


Imagen 8 Operador Map

En las líneas 7 a 9 aplicamos los operadores take y map al Observable original.Como resultado tenemos un nuevo Observable que solo permite escuchar los dos primeros clicks del botón y que por cada click nos devuelve una posición aleatoria en la pantalla.

En la línea 12 creamos un objeto que implementa la interfaz IObserver , la interfaz adecuada para suscribirse al Observable. En la línea 18 suscribimos el IObserver . Cada vez que se produce un click en el botón el IObserver recibe en su método next una nueva posición aleatoria de la pantalla , llama a la función generarTexto(), que escribe un texto en dicha posición.

Cuando el Observable se completa,porque ya se emitieron dos clicks, El Observable notifica al Observer mediante el método complete(). Si hubiera un error se notificará mediante el método error()

# 3. Code Smells en el manejo de eventos en  Javascript

El enfoque incorporado en JavaScript para manejar eventos , ya sean eventos de elementos de formulario como click de botones o eventos Ajax se basan en la utilización de Callbacks que se ejecutan cuando se produce el evento.

Cuando la funcionalidad a programar requiere la interacción de dos o  más eventos dicho enfoque obliga al programador a una o ambas de las siguientes alternativas:

- El uso de variables globales.
- el uso de estructuras condicionales que se usan dentro de los callbacks para coordinar la interacción entre los eventos

Este esquema es impuesto al programador debido al enfoque imperativo del manejo de eventos y debe utilizarlo independientemente de la lógica de la funcionalidad a implementar. Veremos cómo este esquema se repite en funcionalidades diferentes.

## 3.1 Variables condicionales.

En desarrollo web hay una  técnica muy utilizada que consiste en lanzar un evento y escucharlo hasta que otro se produce **(Koutnik, 2019, 11)**. Este es un caso en que el programador necesita utilizar el estado de variables globales para decidir cuándo habilitar o deshabilitar un evento. Veamos un ejemplo  adaptado de **(Meyerovich, 2009, 2)**


Imagen 9  Timer

En la imagen 9 vemos un timer . Cada vez que se da click al botón  Inicio se ejecuta un contador basado en la función javascript setInterval[10]. Esta función lanza un evento cada determinada  cantidad  de tiempo. El timer se basa en escuchar estos eventos hasta que ocurre el click del botón fin. Veamos la solución imperativa en JavaScript (código 2)

[10] https://developer.mozilla.org/es/docs/Web/API/WindowOrWorkerGlobalScope/setInterval

```
let botonInicio1 = document.getElementById("btnInicio1");
let botonFin1 = document.getElementById("btnFin1");
let resultado1 = document.getElementById("lbl1");

let intervalo1 = [];
let contador1 = 0;

const limpiarVariablesGlobales = (contador, intervalo) => {
  contador = 0;
  intervalo.forEach((ele) => clearInterval(ele));
  intervalo = [];
};

//inicia conteo usando setInterval
const iniciarConteo = (contador, intervalo, resultado) => {
  limpiarVariablesGlobales(contador, intervalo); //manejo de
  //estado
  inter = setInterval(function () {
        contador++; //manejo del estado
        let res = contador / 10; //transformacion datos
        resultado.innerHTML = res.toString(); //objeto  interesado
//se actualiza.
  }, 100);
  intervalo.push(inter); //manejo de estado
};
//callback boton Inicio
botonInicio1.addEventListener("click", function () {
  iniciarConteo(contador1, intervalo1, resultado1);
});

//callback boton Fin
botonFin1.addEventListener("click", function () {
  limpiarVariablesGlobales(contador1, intervalo1); //manejo de
  //estado
});
```

código 2 timer imperativo

La implementación imperativa presenta los siguientes inconvenientes

- El uso de variables globales **(Meyerovich, 2009, 2)**. Como los callback retornan void necesitamos las variables globales contador1 e intervalo1(líneas 5 y 6). Es el programador quien debe manipular estas variables para pasar información entre los eventos.

- Se violan importantes principios de la Ingeniería del software como : Encapsulamiento y Separación de Responsabilidades **(Maier & Odersky, 2012, 2).** Se rompe el Encapsulamiento porque el estado es almacenado en variables (como las variables intervalo y contador) que necesitan ser conocidas por ambos callbacks.Estas variables tienen un alcance global y podrían ser usadas incorrectamente por código no relacionado.

- A medida que crece la cantidad de eventos manejados ,crecen también la cantidad de variables globales , con lo cual los inconvenientes anteriores se amplifican.Si en lugar de un timer necesitaríamos dos timers pero que funcionen independientemente (ver imagen 10), se debe duplicar el número de variables globales (intervalo y contador)

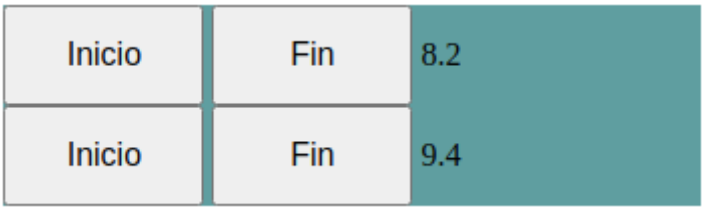


Imagen 10 Timer Doble

Como podemos ver en el siguiente código (código 3) se necesita duplicar el número de variables globales.

```
let botonInicio1 = document.getElementById("btnInicio1");
let botonFin1 = document.getElementById("btnFin1");
let resultado1 = document.getElementById("lbl1");
let botonInicio2 = document.getElementById("btnInicio2");
let botonFin2 = document.getElementById("btnFin2");
let resultado2 = document.getElementById("lbl2");
let intervalo1 = [];
let contador1 = 0;
let intervalo2 = [];
let contador2 = 0;

const limpiarVariablesGlobales = (contador, intervalo) => {
  contador = 0;
  intervalo.forEach((ele) => clearInterval(ele));
  intervalo = [];
};
const iniciarConteo = (contador, intervalo, resultado) => {
  limpiarVariablesGlobales(contador, intervalo);
  inter = setInterval(function () {
          contador++;
          let res = contador / 10;
          resultado.innerHTML = res.toString();
  }, 100);
  intervalo.push(inter);
};
botonInicio1.addEventListener("click", function () {
  iniciarConteo(contador1, intervalo1, resultado1);
});

botonFin1.addEventListener("click", function () {
  limpiarVariablesGlobales(contador1, intervalo1);
});
botonInicio2.addEventListener("click", function () {
  iniciarConteo(contador2, intervalo2, resultado2);
});
botonFin2.addEventListener("click", function () {
  limpiarVariablesGlobales(contador2, intervalo2);
});
```

código 3 timer doble imperativo

## 3.2 Backpressure

El problema del backpressure se da cuando hay más eventos o datos de los que se pueden manejar **(Davis, 2018, 16)**.

Un caso particular en la programación web es la implementación de un typeahead, es decir la funcionalidad en la que el usuario introduce texto en un input, y medida que lo hace la aplicación le sugiere resultados(ver imagen 11) **(Koutnik, 2019, 27)**.

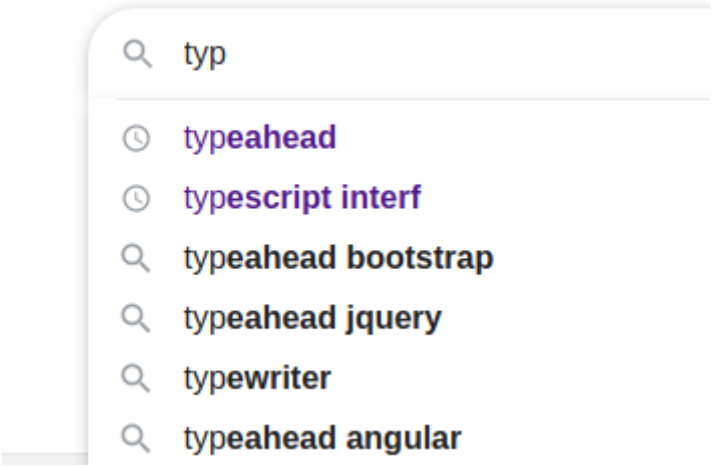


Imagen 11 Typeahead

En este tipo de funcionalidades la aplicación debería esperar un breve periodo de tiempo antes de hacer una búsqueda, para evitar hacer una request cuyo resultados no van a ser vistos. **(Daniels & Atencio, 2017, 103)** Como ejemplo vamos a ver un typeahead que devuelva el nombre de países a partir de los caracteres ingresados . Se basa en hacer request a un webservice que devuelven las coincidencias con lo escrito por el usuario.


Imagen 12 Typeahead Países

El desafío para el programador al implementar el typeahead es que pueden producirse condiciones de carrera **(Silberschatz et al., 2009, 194)**. Una condición de carrera se manifiesta de la siguiente manera **(Koutnik, 2019, 44).**

1. El usuario escribe a

2. Se obtiene y muestra la respuesta para a
3. El usuario escribe ab
4. El usuario escribe abc
5. Se obtiene y muestra la respuesta para abc
6. Se obtiene y muestra la respuesta para ab

Es decir, si la respuesta a una request anterior es más lenta que la respuesta a una request posterior pueden producirse resultados inesperados para el usuario.

Veamos la solución imperativa(código 4) basada en **(Daniels & Atencio, 2017, 105)** y **(Koutnik, 2019, 44)**.

```
let inputpaises = document.getElementById('paises');
let timeoutId = null;
let urlBase = 'http://restcountries.eu/rest/v2/name/';
let resultados = document.getElementById('resultados');
let ultimaQuery;
inputpaises.addEventListener('keyup',function(event){
  let query = ultimaQuery = event.target.value;
  clearTimeout(timeoutId);
  timeoutId = setTimeout(function(query){
      if(query && query.length > 0){
          resultados.innerHTML = '';
          let url =urlBase +query;
          fetch(url).then(function(v){
              return v.json();
          }).then(function(data){
              if(ultimaQuery == query){
                  procesarData(data);
              }
          });
      }

  },1000,event.target.value)
});
function procesarData(data){
  data.forEach(element => {
      resultados.innerHTML+=element.name + '<br/>';
  });
}
```

Código 4 Typeahead imperativo

La solución imperativa se basa en el uso de la function JavaScript setTimeOut[11]. En el código anterior cada vez que se presiona y suelta una tecla se produce el evento keyup. La función setTimeOut se utiliza para ejecutar la función que se le pasa como parámetro luego de un periodo de tiempo, un segundo en este caso.La función clearTimeout se utiliza para cancelar la función pasada a setTimeout, si no se ejecutó aun. Para lograr que las request se envie al webservice cada segundo , cada vez que se presiona una tecla se utiliza clearTimeout para cancelar la función pasada a setTimeOut si esta aun no se ejecutó, de no hacer esto se lanzaría una request por cada tecla pulsada. Para tener una referencia a las funciones que se deben cancelar se utiliza la variable global timeoutId. Nótese también que usamos una sentencia condicional para verificar que no enviamos una request con una cadena vacía.
Para evitar la condición de carrera se utiliza la variable global ultimaQuery . Esta variable guarda el último texto ingresado por el usuario , de modo que si query y ultimaQuery no coinciden significa que hay una request mas reciente en proceso.
La solución imperativa basada para el desarrollo del typeahead presenta el siguiente inconveniente:

**El uso de variables globales y estructuras condicionales para coordinar las request y detectar la request actual.**

[11] https://developer.mozilla.org/es/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout

## 3.3 Wait for all

En JavaScript las request hechas a servidores web se realizan de manera asincrónica mediante la utilización de callbacks **(Flanagan, 2020, 606).**
Cuando se necesita hacer varias request  independientes pero que sin embargo están relacionadas para lograr cierta funcionalidad el programador asume la responsabilidad de coordinar y secuenciar las respuestas. Esto es análogo al uso de Monitores  en los Sistemas Operativos.
Como ejemplo consideremos un script que al ejecutarse debe buscar el menor precio de un producto en varios web service y luego presentar al usuario el sitio con el menor precio.

---

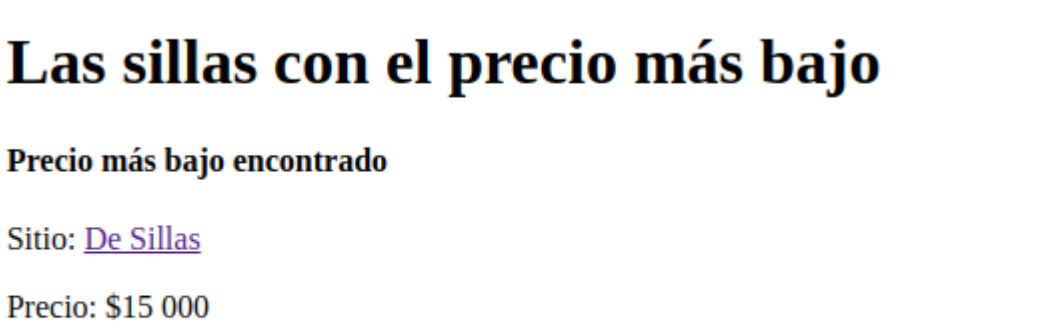


Imagen 13 wait for all

En este caso el programador debe asegurarse de obtener todas las respuestas antes de ejecutar el algoritmo que busque el menor precio. Veamos la solución imperativa(código 5)

```
1 obtenerPrecioAjax = (url,onSuccess)=>{..
2 }
3 let buscaSitioPrecioMenor = function(sitio1, sitio2){..
4 }
5 let mostrarPrecioMasBajo = function(sitio){..
6 }
7 let sitio1 = {
8    url:"http://SillaSite/getPrecio",
9    precio=undefined,
10    nombre:"SillaSite"
11 };
12 let sitio2 = {
13    url:"http://DeSillas/getPrecio",
14    precio=undefined,
15    nombre="DeSillas",
16};
17 obtenerPrecioAjax(sitio1.url, function(resultado){
18    sitio1.precio  = resultado;
19    if(sitio2.precio){
20        let sitioPrecioMenor = buscarPrecioMenor(sitio1,sitio2);
21        mostrarPrecioMasBajo(sitioPrecioMenor);
22    }
23 });
24
25 obtenerPrecioAjax(sitio2.url, function(resultado){
26    sitio2.precio  = resultado;
27    if(sitio1.precio){
28        let precioMenor = buscarPrecioMenor(sitio1,sitio2);
29    }
30});
```

Código 5 wait for all imperativo

La solución imperativa presenta los siguientes inconvenientes;

- La solución se basa en el uso de las variables globales sitio1.precio y sitio2.precio(líneas 9 y 14) para detectar que se cuenta con todas las respuestas. Es responsabilidad del programador verificar que estas variables son diferentes de undefined(líneas 19 y 27) antes de aplicar el algoritmo que busca el menor precio y lo muestra.
- El código para coordinar la llegada de las respuestas utiliza estructuras condicionales que se mezclan con el código propio de la funcionalidad. Se valida que las variables globales mencionadas no sean undefined pero esta validación no está relacionada con la lógica propia de la funcionalidad si no que viene impuesta por el enfoque imperativo del manejo de eventos.
- La solución imperativa se vuelve complicada si en lugar de consultar únicamente dos sitios web se consultan más. En este caso aumentan el número de variables globales y el número de verificaciones que se tiene que hacer para asegurar que se reciben todas las respuestas.

# 4.Soluciones Reactivas a los problemas del manejo de eventos en JavaScript

Como vimos en el capítulo anterior el enfoque imperativo incorporado en JavaScript para el manejo de interacciones entre eventos obliga al programador al uso de variables globales o al uso de estructuras condicionales no relacionadas con la funcionalidad sino con la coordinación de los eventos. A continuación se presentan los tres casos del capítulo 2 de manera consecutiva pero solucionados aplicando abstracciones de la Programación Reactiva. Utilizaremos la libreria Rxjs[12] para agregar abstracciones reactivas sobre JavaScript además de funciones lambda propias del lenguaje. En la tabla 1 se presentan los operadores y funciones que usaremos.

Tabla 1 Operadores y Funciones utilizados en las soluciones

| **Función u Operador** | **Descripción** |
|---|---|
| funciones lambda[13] | Una manera compacta de escribir las funciones en JavaScript. |
| función fromEvent[14] | Permite crear un Observable para un evento específico como por ejemplo eventos click de un botón |
| función interval[15] | Crea un Observable que emite números secuenciales cada periodo de tiempo especificado por parámetro |
| función merge[16] | Permite combinar varios Observables en uno solo |
| función ajax[17] | Crea un Observable para una Request Ajax |
| operador takeUntil[18] | Es un operador que se aplica a un Observable. Recibe por parámetro otro Observable. El Observable original continúa emitiendo hasta que el observable pasado por parámetro hace su primera emisión. |

[12] https://github.com/ReactiveX/rxjs
[13] https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/Arrow_functions
[14] https://rxjs.dev/api/index/function/fromEvent
[15] https://rxjs.dev/api/index/function/interval
[16] https://www.learnrxjs.io/learn-rxjs/operators/combination/merge
[17] https://www.learnrxjs.io/learn-rxjs/operators/creation/ajax
[18] https://www.learnrxjs.io/learn-rxjs/operators/filtering/takeuntil

| | |
|---|---|
| operador map[19] | Aplica una función a cada valor de un Observable y emite los valores resultantes como un Observable |
| operador switchMap[20] | Se aplica a un observable y por parámetro toma un función que devuelve un Observable. Cada vez que el Observable original emite se ejecuta la función pasada por parámetro con lo cual se emite otro Observable pero con la particularidad que los observables anteriores se cancelan. |
| operador tap[21] | permite ejecutar acciones como por ejemplo escribir por consola. |
| operador filter[22] | Se aplica a un Observable y permite emitir otro Observable con los valores del Observable inicial que cumplen una determinada condición. |
| operador pluck[23] | Es como un map que permite extraer determinadas propiedades de los valores de un Observable. |
| operador reduce[24] | Reduce los valores de un Observable a un único valor el cual es emitido en otro Observable |
| operador debounceTime[25] | Descarta valores de un Observable que se emiten antes que pase un determinado tiempo desde la última emisión |

## 4.1 Variables Condicionales

Como mencionamos anteriormente(sección 3 .1), en desarrollo web hay una  técnica muy utilizada que consiste en lanzar un evento y escucharlo hasta que otro evento se produzca **(Koutnik, 2019, 11)**. Este es un caso en que el programador necesita utilizar el estado de variables

[19] https://rxjs.dev/api/operators/map
[20] https://www.learnrxjs.io/learn-rxjs/operators/transformation/switchmap
[21] https://rxjs.dev/api/operators/tap
[22] https://rxjs.dev/api/operators/filter
[23] https://rxjs.dev/api/operators/pluck
[24] https://www.learnrxjs.io/learn-rxjs/operators/transformation/reduce?q=
[25] https://www.learnrxjs.io/learn-rxjs/operators/filtering/debouncetime

globales para decidir cuándo habilitar o deshabilitar un evento. Como ejemplo planteamos un timer(ver Imagen 14)

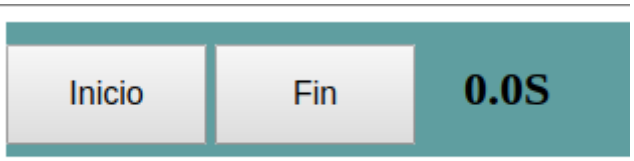


Imagen 14 Timer reactivo

La solución reactiva es la siguiente(código 6).

```
1 const { fromEvent, interval } = require("rxjs");
2 const { takeUntil, map, switchMap } = require("rxjs/operators");
3 const getObservableFromClick = (elemento) => fromEvent(elemento, "click");
4 const getObservableFromInterval = (observableParaConteo) =>
5  interval(100).pipe(
6    map((x) => x / 10), //transformacion de datos
7    takeUntil(observableParaConteo) //coordinacion declarativa de eventos
  );
8 const getObserver = (elemento) => {
9  observer = {
10    next: (segundos) => (elemento.innerHTML = segundos.toString()),
11    error: () => console.log("se produjo un error"),
12    complete: () => console.log("No hay mas eventos"),
13  };
14  return observer;
15};
16 const getConteoObservable = (inicioConteoBoton, finConteoBoton) 17 => {
18  let iniciarConteoClick =
19 getObservableFromClick(inicioConteoBoton);
20   let detenerConteoClick =  getObservableFromClick(finConteoBoton);
21  let iniciarConteoObservable = iniciarConteoClick.pipe(
    //coordinacion declarativa de eventos.
22    switchMap(() => getObservableFromInterval(detenerConteoClick))
23  );
24  return iniciarConteoObservable;
25 };
26 let contadorUno = getConteoObservable(
27  document.getElementById("btnInicio1"),
28  document.getElementById("btnFin1"));
29 contadorUno.subscribe(
getObserver(document.getElementById("lbl1")));
```

Código 6 timer reactivo

El código anterior puede leerse  de la siguiente manera : Emite un evento cada décima de segundo(línea 5) aplica una transformación (línea 6) y continúa haciendo esto hasta que otro evento se produzca (línea 7). El operador takeUntil hace esto último. En la imagen 15 vemos el diagrama marble del operador takeUntil.

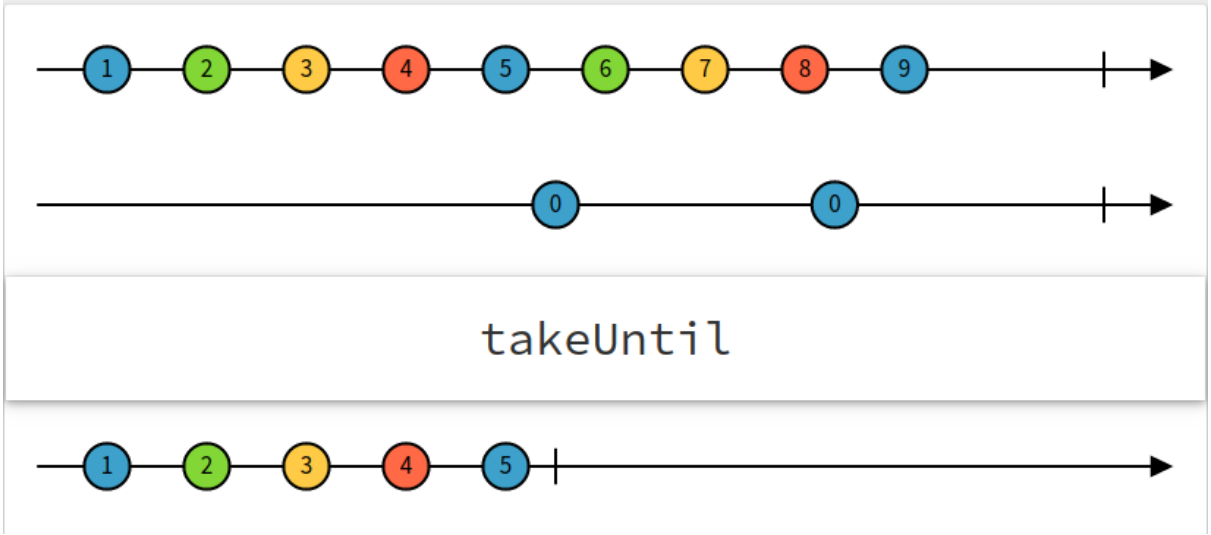


Imagen 15 Operador TakeUntil

La funcionalidad de emitir un evento hasta que otro se produzca mediante el operador takeUntil se encapsula en la funcion getObservableFromInterval( línea 4).
En la línea 3 se define una función que toma un elemento html como por ejemplo un botón y genera un observable para su evento click.Hace uso de la función fromEvent incorporada en Rxjs.La función getConteoObservable(línea 16) toma dos referencias a dos botones y haciendo uso de las funciones anteriores genera el observable encargado de la funcionalidad del timer.Para ello utiliza el operador switchMap.Este operador permite que cada vez que el observable iniciarConteoClick emite( es decir cada click) se emita un observable intervalo a partir de la funcion getObservableFromInterval. De esta manera se puede reiniciar el contador cada vez que se hace click en el botón inicio.
En la línea 8 se define la función getObserver , se utiliza para generar un observer que pueda suscribirse a los eventos emitidos(línea 29)

Consecuencias del Enfoque Reactivo

- El programador no necesita utilizar variables globales para coordinar los eventos. Específicamente no necesita utilizar variables globales para lograr la funcionalidad de escuchar un evento hasta que otro se produce.En su lugar utiliza un enfoque declarativo mediante el uso de abstracciones de la librería reactiva como takeUntil y switchMap.
- Como vimos en la sección 3.1 uno de los inconvenientes del enfoque imperativo en el desarrollo del timer es que si se requiere agregar otro timer independiente del primero(ver imagen 16) es necesario coordinar más eventos . Esto implica agregar

más variables globales.Con el enfoque reactivo No se necesitan agregar variables globales si crecen el número de eventos a manejar. Si necesitamos otro timer solo necesitamos otra llamada a la función getConteoObservable pasandole las referencias a los botones inicio y fin  y finalmente subscribirnos al observable como se muestra en el siguiente código(código 7)

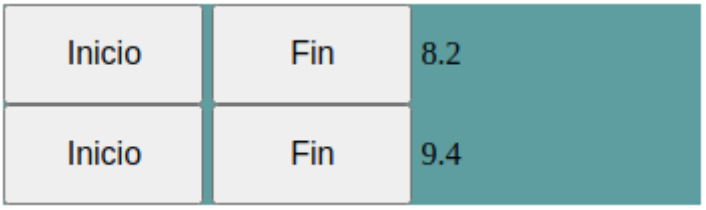


Imagen 16 Timer doble reactivo

```
//si quiero agregar otro contador simplemente genero el observable correspondiente, no es necesario variables globales adicionales
let contadorDos = getConteoObservable(
  document.getElementById("btnInicio2"),
  document.getElementById("btnFin2")
);
contadorDos.subscribe(getObserver(document.getElementById("lbl2")));
```

Código 7 Timer doble reactivo

## 4.2 Backpressure

Como vimos en la sección 3.2 el problema del backpressure se da cuando hay más eventos o datos de los que se pueden manejar **(Davis, 2018,16 ).**

Un caso particular en la programación web es la implementación de un typeahead, es decir la funcionalidad en la que el usuario introduce texto en un input, y medida que lo hace la aplicación le sugiere resultados **(Koutnik, 2019, 27)**-ver imagen 17-

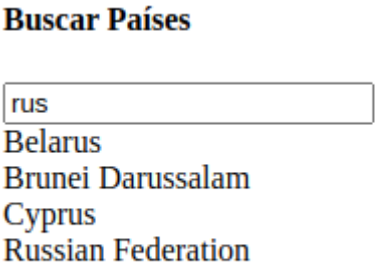


Imagen 17 Backpressure

La implementación mínima[26]mediante abstracciones de la Programación Reactiva es la siguiente **(Koutnik, 2019, 53)**. (código 8).

[26] La implementación completa debería considerar el manejo de errores.

```
1 const { fromEvent } = require("rxjs");
2 const { ajax } = require("rxjs/ajax");
3 const {debounceTime,map,switchMap,tap,filter,pluck,} = require("rxjs/operators");
4
5 const notEmpty = (input) => !!input && input.trim().length > 0;
6
7 let urlBase = "http://restcountries.eu/rest/v2/name/";
8 let resultados = document.getElementById("resultados");
9 let inputpaises = document.getElementById("paises")
10
11 let getData = (val) =>
12  ajax(urlBase + val).pipe(
13    tap(() => (resultados.innerHTML = "")),
14    map((data) => data.response)
15  );
16
17 let procesarData = (data, res) =>
18  data.forEach((val) => (res.innerHTML += val.name + "<br/>"));
19
20 fromEvent(inputpaises, "keyup")
21  .pipe(
22    debounceTime(1000),
23    pluck("target", "value"),
24    filter(notEmpty),
25    switchMap((val) => getData(val))
26  )
27  .subscribe((data) => procesarData(data, resultados));
```

Código 8 Backpressure reactivo

En las primeras tres líneas importamos las funciones y operadores que vamos a usar. Las líneas de la 7 a la 9 contienen referencias a los elementos HTML y la definición de la url del web service utilizado. La función getData toma un valor y genera un observable con el resultado de una llamada ajax. La function ajax toma una url y hace una llamada ajax, encapsula el resultado de la llamada en un Observable.Al Observable devuelto por ajax le encadenamos el operador tap para limpiar el elemento html donde se actualizan los países y el operador map para obtener solo los datos de la llamada ajax.

Luego utilizamos la función fromEvent. Esta función genera un observable a partir de los eventos keyup del input donde se van a escribir los caracteres de búsqueda. A este observable le encadenamos el operador debounceTime,este operador emite un valor si el observable fuente no ha emitido un valor luego de un determinado tiempo que se le pasa por parámetro(ver tabla 1 e imagen 18). El efecto es que solo se lee el valor del input cuando ha pasado un segundo sin que se presione una tecla, ***de esta manera se puede manejar de manera declarativa el backpressure.***

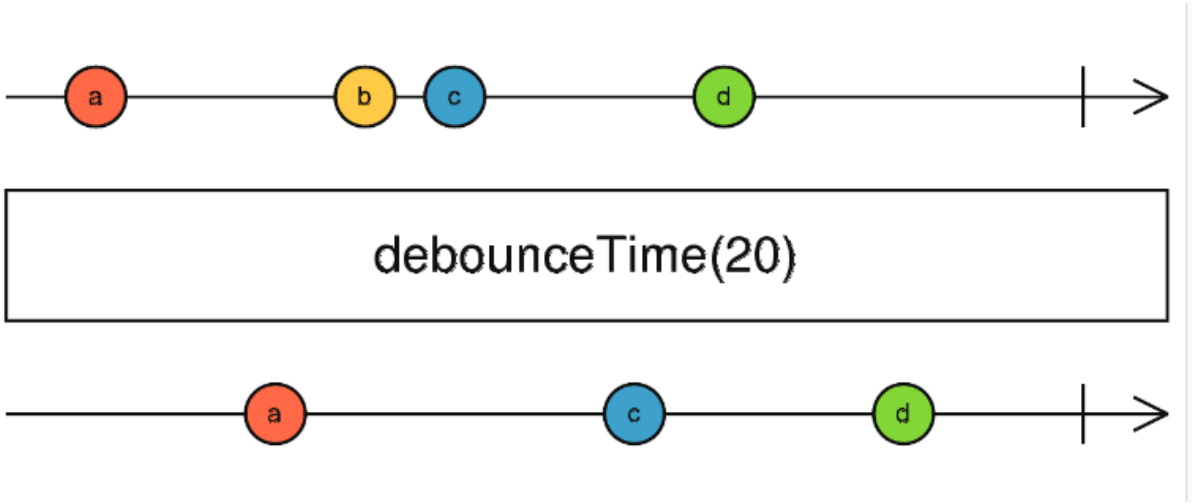


Imagen 18 diagrama Marble Operador debounceTime

Luego usamos el operador pluck para extraer  los caracteres escritos en el input. Luego usamos el operador filter para evitar hacer una request con una cadena vacía. Por último el valor devuelto por filter es pasado al operador switchMap, este operador va usar la función que le pasamos por parámetro para generar una llamada ajax asociada al valor que le pasamos. Lo importante del switchMap es que cada vez que recibe un valor si el observable que genera a partir de la función que le pasamos no ha entregado su resultado, se cancela ese observable y se genera uno nuevo partir del nuevo valor obtenido, como resultado podemos manejar el problema de la condición de carrera de manera declarativa, sin tener  que usar variables globales externas. Por último el observer pasado a subscribe se encarga de actualizar la pantalla.

Consecuencias del Enfoque Reactivo

- Con el enfoque reactivo podemos manejar el backpressure de manera declarativa. El programador no necesita usar variables globales para asegurarse que no se envían request innecesarias.
- Tampoco es necesario usar variables globales y estructuras condicionales para evitar condiciones de carrera, en este caso podemos confiar en el operador switchMap.
- Otra ventaja es que podemos confiar al operador filter que no se envíen request cuando no hay ninguna cadena ingresada.
- Además todas estas diferentes responsabilidades permanecen separadas, manejadas declarativamente por un operador específico y no de manera imperativa por el programador.

## 4.3 Wait For All

En JavaScript las request hechas a webs server se realizan de manera asincrónica mediante la utilización de callbacks **(Flanagan, 2020, 606)**. Como vimos en la sección 3.3 cuando se necesita hacer varias request  independientes pero que sin embargo están relacionadas para lograr cierta funcionalidad el programador asume la responsabilidad de coordinar y secuenciar las respuestas utilizando variables globales y estructuras condicionales.

Como ejemplo consideramos un script que al ejecutarse debe buscar el menor precio de un producto en varios web service y luego presentar al usuario el sitio con el menor precio(ver imagen 19).


Imagen 19 Wait for all

Veamos el enfoque Reactivo(código 9):

```
1 const { fromEvent, merge } = require("rxjs");
2 const { ajax } = require("rxjs/ajax");
3 const { tap, map, switchMap, reduce } = require("rxjs/operators");
4
5 let btn = document.getElementById("btn");
6 let res = document.getElementById("res");
7
```

```
sitio1 = {
 url: "http://localhost:5500/sitio1.json",
 nombre: "DeSillas",
 getPrecio: () =>
   ajax.getJSON(sitio1.url).pipe(
     map((data) => {
       return { precio: data[0].precio, nombre: sitio1.nombre };
     })
   ),
};
sitio2 = {
 url: "http://localhost:5500/sitio2.json",
 nombre: "SillasOutlet",
 getPrecio: () =>
   ajax.getJSON(sitio2.url).pipe(
     map((data) => {
       return { precio: data.p, nombre: sitio2.nombre };
     })
   ),
};
let coordinaYReduce$ = merge(
  sitio1.getPrecio(),
 sitio2.getPrecio(),
).pipe(
 reduce((acc, val) => {
   if (acc.precio < val.precio) {
     return acc;
   }
   return val;
 }),
 map((s) => "<h1>" + s.nombre + "</h1><br/><p>" + s.precio + "</p>")
);
fromEvent(btn, "click")
  .pipe(
    tap((x) => (res.innerHTML = "")),
    switchMap((ev) => coordinaYReduce$)
  )
  .subscribe((resAct) => (res.innerHTML = resAct));
```

Código 9 Wait for all reactivo

Son varios los operadores y/o funciones que podemos usar para resolver el problema de esperar varias llamadas ajax de manera declarativa. En este caso vamos a usar la función merge(ver imagen 20 , diagrama marble del operador merge).

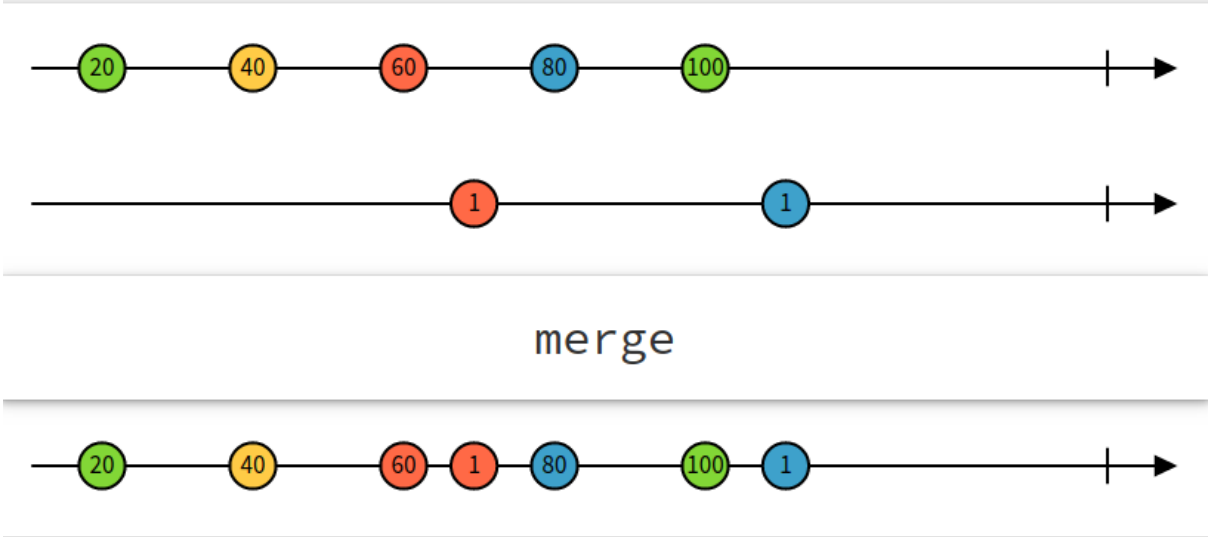


Imagen 20 Operador Merge

Esta función toma dos o más Observables y va a devolver un Observable que va devolviendo los resultados de los observables que le pasamos a medida que llegan **(Mansilla, 2015, 18)**.
Esto es útil porque podemos combinar merge con el operador reduce. Reduce toma una función que permite acumular los resultados de un Observable para retornar solo un valor(el valor acumulado) **(Oliveira, 2017, 87)**

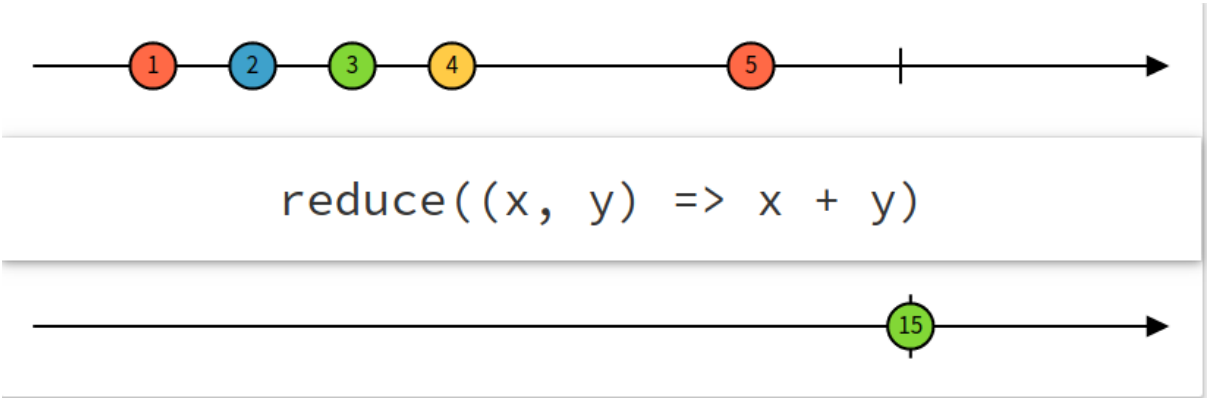


Imagen 21 Operador Reduce]

Creamos el Observable coordinayReduce$ usando la función merge, esta función toma los observables que encapsulan los resultados de las llamadas ajax a cada sitio. Reduce va acumulando los diferentes resultados y va retornando el resultado con el menor precio. Al final reduce devolverá una Observable que encapsula el valor con el menor precio. Luego usamos el operador map para mapear el resultado a un string con el que vamos a actualizar la pantalla. Nótese que el observable coordinayReduce$ como todo Observable no ejecuta nada hasta que recibe una suscripción **(Oliveira, 2017, 42)**.

Consecuencias de Enfoque Reactivo

- El programador no necesita verificar los valores de variables globales mediante estructuras condicionales para poder determinar el arribo de todas las respuestas de los diferentes servicios web. La verificación se realiza declarativamente mediante la abstracción merge.
- En el enfoque imperativo(sección 3.3) si se necesitan consultar más sitios aumentan el número de variables globales y de estructuras condicionales para verificar el correcto arribo de las respuestas. En el enfoque reactivo simplemente agregamos un nuevo observable al operador merge(ver código 10).

```
let coordinaYReduce$ = merge(
  sitio1.getPrecio(),
  sitio2.getPrecio(),
  sitio3.getPrecio() //sitio nuevo a consultar
).pipe(
  reduce((acc, val) => {
    if (acc.precio < val.precio) {
      return acc;
    }
    return val;
  }),
  map((s) => "<h1>" + s.nombre + "</h1><br/><p>" + s.precio + "</p>")
);
```

Código 10 Wait for all con más sitios de consulta

# 5.Panorama de la Programación Reactiva en frameworks y librerías web actuales.

Desde el punto de vista de la Programación Reactiva todo lo que sucede dentro de una aplicación web puede verse como un stream de datos o eventos que suceden en el tiempo **(Mezzalira, 2018, 67)**. La noción de Observables(que discutimos en el capítulo 2) es una abstracción para el manejo de dichos streams. En este capítulo veremos cómo estas ideas de streams ,Observables y sus implementaciones se están usando en frameworks o librerías actuales dedicadas a la programación web. Específicamente veremos el empleo de Observables en librerías Javascript para el manejo del estado de la aplicación, el uso de Observables en Angular y la incorporación de Observables en lenguajes como Dart

## 5.1 Librerías Javascript para el manejo de estado

Para entender la utilidad de estas librerías necesitamos entender el concepto de Single Page Application(SPA). Plataformas como Angular, y frameworks como Vue o React permiten crear Aplicaciones de página única. Una SPA se caracteriza por que toda la aplicación corre en una única página html. Las diferentes partes de la aplicación se conforman de componentes web, es decir fragmentos de html, css y javascript que se manejan dentro del navegador. Estos diferentes componentes se hacen visibles como respuesta a un evento o por cambios en la url **(Scott, 2015, 3-8)** .

Cada componente tiene variables internas que cambian su estado . Muchas veces estos cambios deben ser comunicados a otros componentes . Manejar y controlar el estado de los diferentes componentes se vuelve complicado para aplicaciones no triviales. La situación se complica porque se manejan diferentes tipos de estados: variables guardadas localmente, respuestas de llamadas ajax etc y porque múltiples actores tanto asíncronos como síncronos pueden cambiar el estado de la aplicación. Las librerías de manejo de estado de la aplicación se encargan de resolver este problema.

**Mobx**[27] .- Se presenta como una librería que hace el manejo del estado de la aplicación simple y transparente mediante el uso de Programación Funcional Reactiva. Con Mobx se

[27] https://mobx.js.org/README.html

utilizan Observables para almacenar valores que definen el estado de la aplicación. Mediante la suscripción a estos Observables se comunican los cambios de manera automática. **(Mezzalira, 2018, 130)**

**Ngrx**[28] Es la librería preferida para el manejo de estado en aplicaciones Angular. Se basa en el uso de Observables de la librería Rxjs para notificar de manera automática los cambios en el estado de la aplicación.

## 5.2 Angular.[29]

Angular es una plataforma de desarrollo construida sobre TypeScript[30]. Como plataforma Angular incluye:

- Un framework basado en componentes[31] para construir aplicaciones web.
- Una colección de librerías que cubre una amplia variedad de características , incluyendo ruteo, manejo de formularios, comunicación cliente servidor y más.
- Un conjunto de herramientas de desarrollo que ayudan a desarrollar, construir , testear y actualizar el código.

Angular adopta el enfoque reactivo para manejar eventos, tanto eventos de formulario como eventos Ajax.Cuando se crea un proyecto Angular se incluye la librería Rxjs-que es la que usamos para construir las soluciones reactivas en el capítulo anterior-de este modo Angular no solo permite al programador usar la librería cuando lo crea conveniente sino que también dos de las herramientas incorporadas en angular , los formularios reactivos y la comunicación cliente servidor están basados en Observables **(Moiseev & Fain, 2018, 134)**

**Formularios Reactivos** Angular ofrece una herramienta para el manejo de formularios, angular reactive forms[32] , que agrega propiedades implementadas como Observables a los objetos que representan formularios web o a los controles de los formularios. Como por ejemplo **(Moiseev & Fain, 2018, 138)**:

- **valueChanges**: Esta propiedad es un Observable que emite cuando el valor del control de formulario(como por ejemplo una caja de texto) cambia

[28] https://ngrx.io/
[29] https://angular.io/
[30] https://www.typescriptlang.org/
[31] https://www.webcomponents.org/introduction
[32] https://angular.io/guide/reactive-forms

- **statusChange**: En html los formularios y los controles de formulario tienen una propiedad que indica si son válidos . Angular agrega la propiedad statusChange a los objetos que representan formularios u objetos de formulario. Esta propiedad es un Observable.

Como ejemplo veamos una aplicación angular que toma el nombre de una empresa y muestra el precio de su acción financiera correspondiente. Ver imagen 22

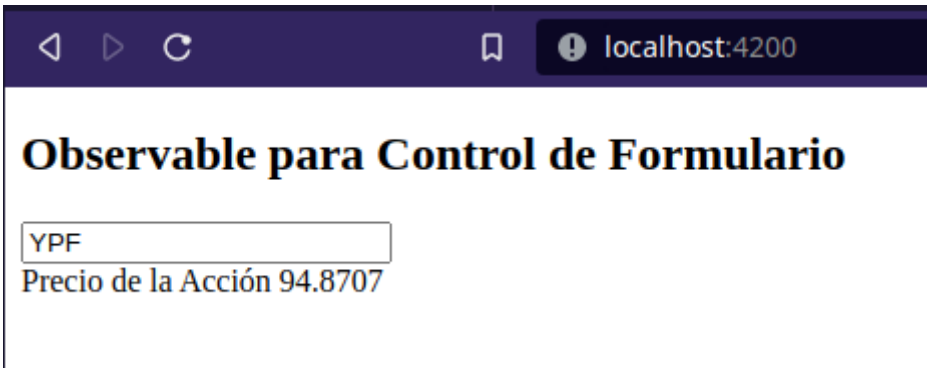


Imagen 22 Value Changes

El código de Angular se muestra a continuación(código 11):

```
1 import { FormControl } from '@angular/forms';
2 import { debounceTime } from 'rxjs/operators';
3 @Component({
4 selector: 'app-rx',
5 templateUrl: './rx.component.html',
6 styleUrls: ['./rx.component.css']
7 })
8 export class RxComponent implements OnInit {
9
10 searchInput = new FormControl('');
11 res = "";
12
13 constructor() {
14
15   this.searchInput.valueChanges
16       .pipe(debounceTime(1000))
17       .subscribe(stock => this.res = this.getStockQuoteFromServer(stock));
18
19 }
20 getStockQuoteFromServer(stock: string) {
21   return (100 * Math.random()).toFixed(4);
22
23 }
24 ngOnInit(): void {
25 }
26
}
```

Código 11 Value Changes

En la línea 10 creamos un objeto control del formulario que se va a mostrar como una caja de texto. En la línea 15 utilizamos el Observable provisto por la propiedad valueChanges para aplicar el operador debounceTime, el cual permite esperar un segundo antes de leer los datos ingresados por el usuario.

**Comunicación Cliente Servidor** En Angular la librería encargada de hacer request Http devuelve Observables **(Moiseev & Fain, 2018, 140)**. Como ejemplo veamos una aplicación angular

toma el nombre de un país y muestra los datos del clima obtenidos mediante una llamada ajax a un web server.

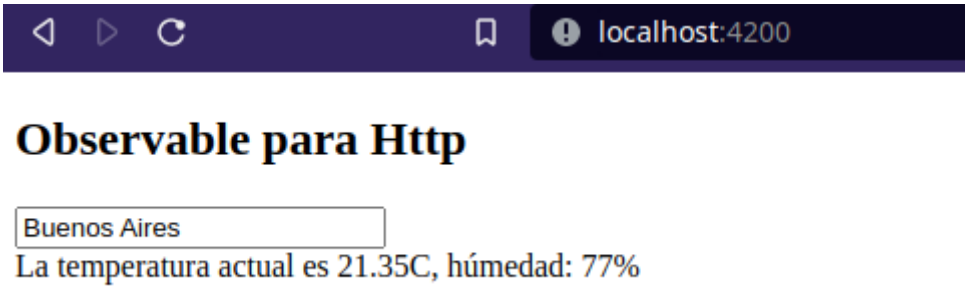


Imagen 23 Observable Http

Veamos el código Angular(código 12)

```
1 import { Component, OnInit } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3 import { FormControl } from '@angular/forms';
4 import { switchMap, catchError } from 'rxjs/operators';
5 import { Observable, EMPTY } from 'rxjs';
6
7 @Component({
8  selector: 'app-httprx',
9  templateUrl: './httprx.component.html',
10  styleUrls: ['./httprx.component.css'],
11 })
12 export class HttprxComponent implements OnInit {
13  searchInput = new FormControl();
14  clima: string = '';
15  baseURL: string = 'http://api.openweathermap.org/data/2.5/weather?q=';
16  private urlSufijo: string =
17       '&units=metric&appid=71f5c1f4b0c9f9f9b';
18  constructor(private http: HttpClient) {}

19  getWeather(city: string): Observable<any> {
20       return this.http.get(this.baseURL + city + this.urlSufijo).pipe(
21       catchError((err) => {
22       if (err.status === 404) {
23       console.log('Ciudad no encontrada');
24       }
25       return EMPTY;
26       })
27       );
28  }
29  ngOnInit(): void {
30       this.searchInput.valueChanges
31       .pipe(switchMap((city) => this.getWeather(city)))
32       .subscribe((res) => {
33       this.clima =
34       `La temperatura actual  es ${res['main'].temp}C, ` +
35       `húmedad: ${res['main'].humidity}%`;
         });
  }
}
```

Código 12 Observable Http

En este caso usamos un observable para detectar los cambios en la caja de texto (línea 30) . En la línea 20 al observable devuelto por el objeto http le agregamos un operador para manejar los errores. En la línea 30 el uso del operador switchMap nos permite trabajar únicamente con el último valor ingresado por el usuario.

## 5.3 Dart-Flutter-Streams

Flutter[33] es un framework para construir aplicaciones tanto móviles como web. Flutter se implementa sobre el lenguaje de programación Dart[34].

Hasta ahora hemos visto que el Observable es una abstracción que permite manejar streams de datos o eventos. Dart tiene soporte nativo para Streams, que cumplen la misma función que los Observables. **(Windmill, 2019, 242) (Bracha, 2016, 178).**

Según la documentación de Dart[35] la clase Stream<T> es una fuente de eventos asíncronos . Un stream provee una manera de recibir una secuencia de eventos , cada evento es una evento de datos-llamado elemento del stream-, o un evento de error ,que es una notificación de que algo falló.  Cuando un stream emite todos sus eventos , un único evento "done" es emitido para notificar que se alcanzó el fin del stream.

Se tiene que definir "listeners" para recibir los eventos del stream. Al escuchar un stream se recibe un objeto StreamSubscription que permite cancelar la suscripción al stream.

Véase la tabla 2  para ver la  similitud de interfaces entre los Observables Rxjs y los  Streams de Dart

[33] https://flutter.dev/
[34] https://dart.dev/
[35] https://api.dart.dev/stable/2.14.2/dart-async/Stream-class.html

| Dart Streams | Rxjs Observable |
| --- | --- |
| Iterable<int> data = [1, 2, 3];<br>// creamos el stream<br>Stream<num> stream =<br>Stream<num>.fromIterable(data);<br>// aplicamos transformación<br>Stream<num> stream2 = stream.map((x) {<br>return x * x;<br>});<br>// comenzamos a escuchar stream<br>var subscriber = stream2.listen((data) {<br>print(data);<br>}, onError: (error) {<br>print(error);},<br>onDone:(){print('done');}); | var data = [1, 2, 3];<br>//creamos observable<br>var observable = of(...data);<br><br>//aplicamos transformación<br>var observable2 = observable.pipe(map((x)<br>=> x * x));<br><br>// subscripción<br>var subscriber = observable2.subscribe({<br>next: (x) => console.log(x),<br>error: (error) => console.log(error), |

Tabla 2 : Comparación de las interfaces de Dart Stream y Rxjs Observable

Uso de Streams en Dart-Flutter. Como vimos en la tabla podemos usar los Streams de Dart manualmente sin embargo en Flutter una forma de usar los streams es a partir de la clase StreamBuilder[36]. Esta clase automáticamente escucha un stream que le pasamos y construye o actualiza la UI cuando se produce una emisión del stream **(Katz et al., 2020, 459) (Windmill, 2019, 254)**.

También sobre la base de los Streams se construyen librerías para manejar el estado en aplicaciones Flutter como por ejemplo BLoC[37].

[36] https://api.flutter.dev/flutter/widgets/StreamBuilder-class.html
[37] https://bloclibrary.dev/#/coreconcepts

## 5.4 Adopción Tecnológica Actual

Angular , que como hemos visto adopta el enfoque reactivo, es una plataforma web ya establecida según la encuesta 2021 del conocido sitio Stackoverflow[38].

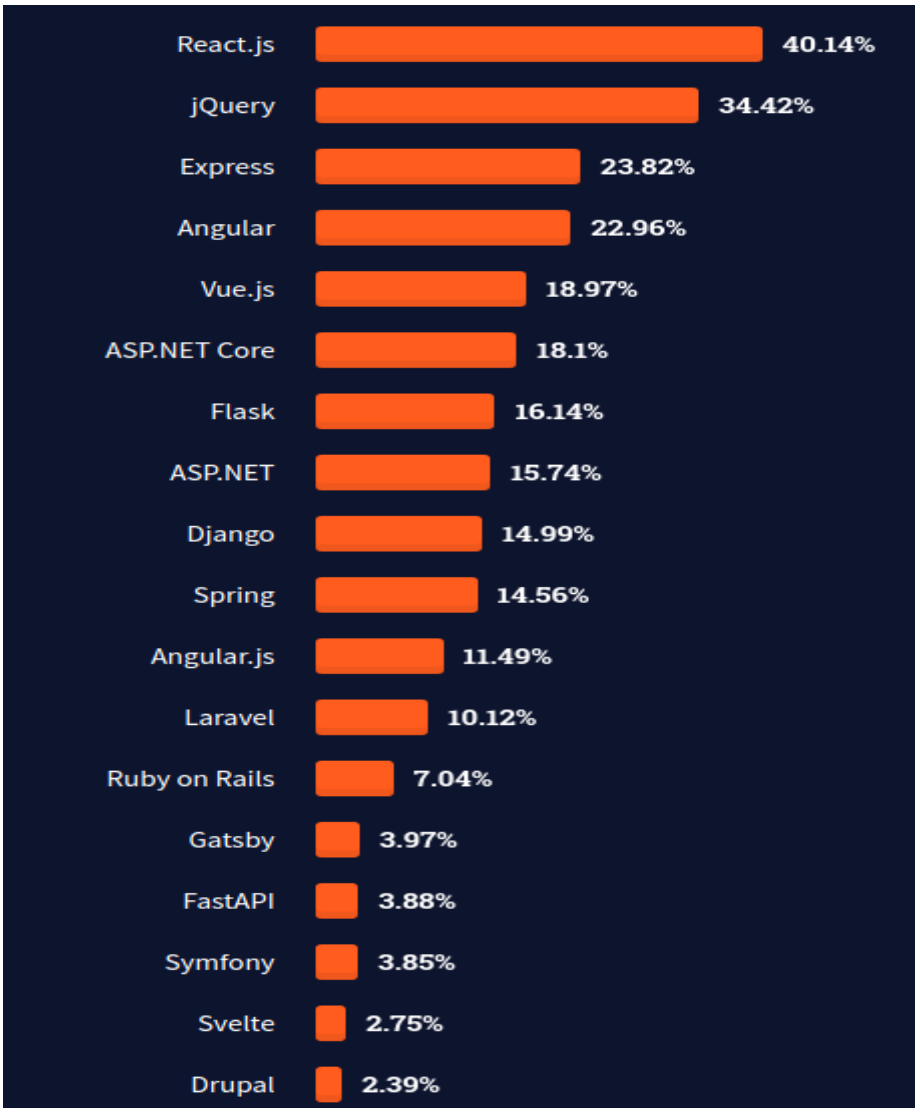


Imagen 24 Web Frameworks mas populares 2021(StackOverflow)

En el caso de las aplicaciones móviles, Flutter es uno de los frameworks que más creció desde el 2019 según el sitio Statista[39] . Además según la anteriormente mencionada encuesta de Stackoverlow el 68% de los encuestados ha trabajado o pretende trabajar con Flutter[40]

[38] https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks
[39] https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/
[40] https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted-misc-tech-love-dread

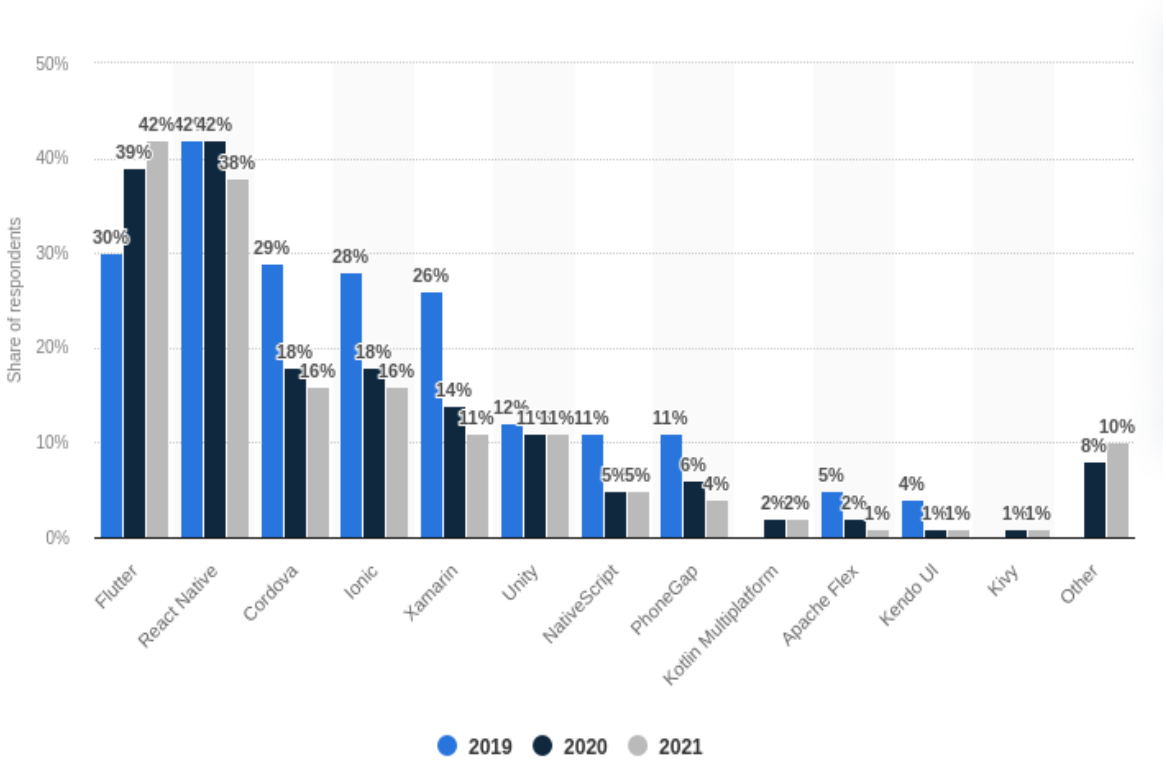


Imagen 25 Frameworks que más crecieron desde 2019

Por lo anterior podemos decir que las ideas de la Programación Reactiva ya comienzan a ser advertidas dentro de la programación web y de móviles.

# 6.Conclusiones

Las aplicaciones web tratan con eventos de diferentes fuentes. Desde eventos de interfaz gráfica como por ejemplo los clicks de un botón en un formulario hasta eventos ajax como los utilizados para refrescar cierta área de la página web con datos provenientes del servidor. Como vimos en el capítulo 3 muchas veces es necesario esperar que un evento termine antes que otro se dispare o se necesita que un evento se ejecute un número determinado de veces, es decir se necesita coordinar los eventos para lograr la funcionalidad pedida.

El enfoque imperativo para el manejo de eventos provisto por JavaScript requiere que el programador escriba código para coordinar explícitamente los eventos. Como consecuencia se escribe código repetitivo que sigue el esquema de utilizar estructuras condicionales para verificar el estado de variables globales.

En este trabajo presentamos abstracciones de la Programación Reactiva como los Observables y sus diversos operadores. Estas abstracciones ofrecen una alternativa al manejo de eventos imperativo proporcionado por JavaScript. Permiten tratar a las secuencias de eventos como si fueran arrays o listas y aplicarles diversos operadores como map, filter, reduce , takeUntil, switchMap etc.

Una consecuencia positiva de utilizar estas abstracciones es que se libera al programador de la responsabilidad de escribir el código que coordina los eventos . Como se libera de estos detalles de implementación del manejo de eventos puede centrarse en la funcionalidad que está desarrollando.

Otra consecuencia positiva de utilizar Programación Reactiva se debe a su enfoque declarativo. En nuestra opinión y la de otros autores **(Salvaneschi, 2014)** el programador entrenado puede entender más rápida y fácilmente lo que se pretende hacer en el código al usar operadores cuyo nombre declaran su intención como first o take , en comparación con su contraparte imperativa, más genérica, como los bucles y condicionales.
Por otro lado, se debe tener en cuenta que el uso de estas librerías reactivas añaden tiempo de carga a la página web por lo que no deberían ser usadas en aplicaciones sencillas.

Las abstracciones Reactivas y las soluciones que presentamos en el capítulo 4 se aplican no solo a la programación web o de móviles. Son útiles en cualquier aplicación que necesite manejar código asincrónico y tratar con secuencias ya sean eventos o de otro tipo. Nurkiewicz y Christensen muestran el uso de Observables en la implementación de clientes que se conectan a bases de datos de manera asincrónica. **(Nurkiewicz & Christensen, 2016, 306)** mientras que herramientas como Vert.x **(Ponge, 2020)** se utilizan para comunicar distintas partes de una aplicación mediante eventos.

Este trabajo puede servir como guía para implementar herramientas de refactoring y deuda técnica.

## 6.1 Líneas de Trabajo Futuro.

**Refatoring to Reactive Programming**

Refactoring es el proceso de cambiar un sistema de software de tal manera que no altera su comportamiento externo pero mejora su estructura interna.Es una manera disciplinada de limpiar el código que minimiza las posibilidades de introducir errores **(Fowler, 2019)**

Existen herramientas automáticas de refactoring de código sincrónico a código asincrónico como las propuestas por Dany Dig **(Dig, 2015)**y que se implementan en los Ides más populares como Netbeans y Visual Studio.

Una línea de trabajo futuro consiste en la creación de herramientas de análisis estático que soporten refactoring desde código imperativo que maneja eventos en JavaScript a su contraparte Reactiva. Estas herramientas permitirían la transformación de callbacks, estructuras condicionales y variables globales utilizados para manejar eventos de formulario o eventos Ajax en Observables y sus operadores.

**Análisis de Deuda Técnica**

La Deuda Técnica es una metáfora, tomada del dominio económico, que describe la decisión de no realizar ahora determinadas tareas de desarrollo de software pero que al no hacerla pueden causar mayor trabajo en el futuro e incluso poner en riesgo el proyecto **(Alves et al.**,

**2014,)**.Es una consecuencia de decisiones tomadas con la finalidad de asegurar entregas más rápidas del producto. **(Codabux & Williams, 2013)**

Una de las dimensiones en que la deuda técnica puede manifestarse es mediante código repetitivo , duplicado o complejo **(Tom et al., 2013, 1498-1516)**.Esto es justamente el caso cuando se utiliza el enfoque imperativo para coordinar eventos en JavaScript.Usando la información del código fuente(scripts) se pueden desarrollar herramientas de identificación automática de este tipo de defectos con foco en la identificación de estructuras condicionales y variables globales utilizados con la finalidad de coordinar eventos.

Para abordar ambas líneas de trabajo es necesario profundizar en el Análisis de Programas y específicamente en herramientas que analicen el código sin ejecutarlo(Análisis Estático de Código) **(Rival & Yi, 2020)**

# Bibliografía.


Alves, N. S., Ribeiro, L. F., Caires, V., Mendes, T. S., & Spínola, R. O. (2014, September). Towards an ontology of terms on technical debt. In *2014 Sixth International Workshop on Managing Technical Debt* (pp. 1-7). IEEE.

Bertoluzzo, E. (2017). The essence of reactive programming: A theoretical approach.

Blackheath, S., & Jones, A. (2016). *Functional Reactive Programming*. Manning.

Borges, L. (2015). *Clojure Reactive Programming*. Packt Publishing.

Bracha, G. (2016). *The Dart Programming Language*. Addison-Wesley.

Carkci, Mark. *Tpl Dataflow By Example. Dataflow and Reactive in .Net by example*. Lean Pub, 2014

Codabux, Z., & Williams, B. (2013, May). Managing technical debt: An industrial case study. In *2013 4th International Workshop on Managing Technical Debt (MTD)* (pp. 8-15). IEEE.

Czaplicki, E., & Chong, S. (2013). Asynchronous functional reactive programming for GUIs. *ACM SIGPLAN Notices*, *48*(6), 411-422 *DOI*: https://doi.org/10.1145/2499370.2462161

Daniels, P. P., & Atencio, L. (2017). *RxJS in Action*. Manning.

DIG, Danny. Refactoring for asynchronous execution on mobile devices. *IEEE Software*, 2015, vol. 32, no 6, p. 52-61.doi: 10.1109/MS.2015.133.

Doglio, F. (2016). *Reactive Programming with Node.js*. Apress.

Flanagan, D. (2020). *JavaScript: The Definitive Guide : Master the World's Most-used Programming Language*. O'Reilly Media, Incorporated.

Fowler, M. (2019). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gamma, E. (2002). *Patrones de diseño*. Pearson Educación.

Kambona, K., Boix, E. G., & De Meuter, W. (2013, July). An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications* (pp. 1-9).

Katz, M., Moore, K., Ngo, V., & raywenderlich Tutorial Team. (2020). *Flutter Apprentice (First Edition): Learn to Build Cross-Platform Apps*. Amazon Digital Services LLC - KDP Print US.

Koutnik, R. (2019). *Build Reactive Websites with RxJS: Master Observables and Wrangle Events*. Pragmatic Bookshelf.

Maier, I., & Odersky, M. (2012). *Deprecating the observer pattern with Scala. react* (No. REP_WORK). https://infoscience.epfl.ch/record/176887

Mansilla, S. (2015). *Reactive Programming with RxJS: Untangle Your Asynchronous JavaScript Code*. Pragmatic Bookshelf.

Meijer, E. (2012). Your Mouse is a Database: Web and mobile applications are increasingly composed of asynchronous and realtime streaming services and push notifications. *Queue*, *10*(3), 20-33 DOI: https://doi.org/10.1145/2160718.2160735.

Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., & Krishnamurthi, S. (2009, October). Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (pp. 1-20). DOI: https://doi.org/10.1145/1639949.1640091 .

Mezzalira, L. (2018). *Front-End Reactive Architectures: Explore the Future of the Front-End Using Reactive JavaScript Frameworks and Libraries*. Apress.

Moiseev, A., & Fain, Y. (2018). *Angular Development with TypeScript*. Manning.

Noring, C. (2018). *Architecting Angular Applications with Redux, RxJS, and NgRx: Learn to Build Redux Style High-performing Applications with Angular 6*. Packt Publishing.

Nurkiewicz, T., & Christensen, B. (2016). *Reactive Programming with RxJava: Creating Asynchronous, Event-based Applications*. O'Reilly Media, Incorporated.

Oliveira, E. d. S. (2017). *Mastering Reactive JavaScript*. Packt Publishing.

Parker, J. (2013). Froid: Functional Reactive Android. *Undergraduate Honours Thesis, University of Maryland*.

Peelen, D. (2016). Working with Unreliable Observers Using Reactive Extensions. *Nov*, *10*, 1-46.https://www.ru.nl/publish/pages/769526/dorus_peelen.pdf

Ponge, J. (2020). *Vert.x in Action: Asynchronous and Reactive Java*. Manning.

Alves, N. S., Ribeiro, L. F., Caires, V., Mendes, T. S., & Spinola, R. =. (2014). *Towards an ontology of terms on technical debt.*

Bertoluzzo, E. (2017). *The essence of reactive programming: A theoretical approach.*

Blackheath, S., & Jones, A. (2016). *Functional Reactive Programming*. Manning.

Borges, L. (2015). *Clojure Reactive Programming*. Packt Publishing.

Bracha, G. (2016). *The Dart Programming Language*. Addison-Wesley.

Carkci, M. (2014). *Tpl Dataflow By Example. Dataflow and Reactive in .Net by example*.

Codabux, Z., & Williams, W. (2013). *Managing technical debt: An industrial case study.*

Czaplicki, E., & Chong, S. (2013). *Asynchronous functional reactive programming for GUIs*.

Daniels, P. P., & Atencio, L. (2017). *RxJS in Action*. Manning.

Davis, A. L. (2018). *Reactive Streams in Java: Concurrency with RxJava, Reactor, and Akka Streams*. Apress.

Doglio, F. (2016). *Reactive Programming with Node.js*. Apress.

Flanagan, D. (2020). *JavaScript: The Definitive Guide : Master the World's Most-used Programming Language*. O'Reilly Media, Incorporated.

Fowler, M. (2019). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gamma, E. (2002). *Patrones de diseño*. Pearson Educación.

Kambona, K. (n.d.). An evaluation of reactive programming and promises for structuring collaborative web applications.

Katz, M., Moore, K., Ngo, V., & raywenderlich Tutorial Team. (2020). *Flutter Apprentice (First Edition): Learn to Build Cross-Platform Apps*. Amazon Digital Services LLC - KDP Print US.Dig, D. (2015). *Refactoring for asynchronous execution on mobile devices*.

Koutnik, R. (2019). *Build Reactive Websites with RxJS: Master Observables and Wrangle Events*. Pragmatic Bookshelf.

Salvaneschi, G., Amann, S., Proksch, S., & Mezini, M. (2014, November). An empirical study on program comprehension with reactive programming. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 564-575).

Maier, I., & Odersky, M. (2012). *Deprecating the observer pattern with Scala.*

Mansilla, S. (2015). *Reactive Programming with RxJS: Untangle Your Asynchronous JavaScript Code*. Pragmatic Bookshelf.

Meijer, E. (2012). *your mouse is a database*.

Meyerovich, L. (2009). *Flapjax: a programming language for Ajax applications*.

Mezzalira, L. (2018). *Front-End Reactive Architectures: Explore the Future of the Front-End Using Reactive JavaScript Frameworks and Libraries*. Apress.

Moiseev, A., & Fain, Y. (2018). *Angular Development with TypeScript*. Manning.

Noring, C. (2018). *Architecting Angular Applications with Redux, RxJS, and NgRx: Learn to Build Redux Style High-performing Applications with Angular 6*. Packt Publishing.

Nurkiewicz, T., & Christensen, B. (2016). *Reactive Programming with RxJava: Creating Asynchronous, Event-based Applications*. O'Reilly Media, Incorporated.

Oliveira, E. d. S. (2017). *Mastering Reactive JavaScript*. Packt Publishing.

Ponge, J. (2020). *Vert.x in Action: Asynchronous and Reactive Java*. Manning.

Rival, X., & Yi, K. (2020). *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press.

Scott, E. (2015). *SPA Design and Architecture: Understanding Single Page Web Applications*. Manning.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2009). *Operating system concepts*. Wiley.

Stewart, D., O'Sullivan, B., & Goerzen, J. (2009). *Real world Haskell*. O'Reilly Media.

Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, *86*(6), 1498-1516.

Tsvetinov, N. (2015). *Learning Reactive Programming with Java 8: Learn how to Use RxJava and Its Reactive Observables to Build Fast, Concurrent, and Powerful Applications Through Detailed Examples*. Packt Publishing.

Van der Plas, J. K. (2016). *Slim: functional reactive user interface programming* (Master's thesis).

Van Roy, P., & Haridi, S. (2004). *Concepts, Techniques, and Models of Computer Programming*. Prentice-Hall.

Willems, L. (2016). Data streams in spreadsheets with reactive extensions. http://resolver.tudelft.nl/uuid:5c84b35a-39d0-4c19-a905-a7bf60239138

Windmill, E. (2019). *Flutter in Action*. Manning.

Zimmerle, C., & Gama, K. (2018, October). Reactive CEP: Integrating Complex Event Processing into Web Reactive Languages. In *Proceedings of the 24th Brazilian Symposium on Multimedia and the Web* (pp. 69-72).