



Aprende
RxJS
desde cero

Aprende RxJS desde cero – parte II

Published on 10 abril, 2019

Hace unos días, te explicaba algunos casos de uso interesantes de RxJS. Hoy te voy a explicar los fundamentos en los que se basa esta librería de programación reactiva.

Resumiendo mi artículo anterior: **la gracia de RxJS es que permite trabajar con secuencias de eventos de forma simple**. Y lo consigue gracias a 3 conceptos muy poderosos:

- El patrón Observador (Observer pattern)
- El patrón Iterador (Iterator pattern)
- y la programación funcional

El patrón Observador

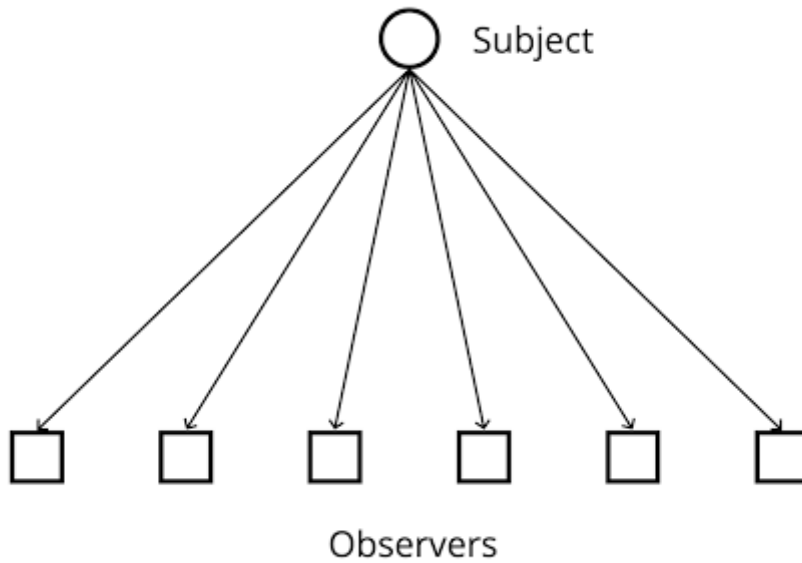
El patrón observador es muy conocido dentro de los patrones de diseño de SW. Es un **patrón de comportamiento que define una dependencia *one-to-many* entre objetos**.

En el patrón Observer hay un objeto, el Sujeto (o subject), que mantiene una lista de

X

Aprende a hacer apps móviles con **ionic 3** [Ver curso](#)

cualquier cambio de su estado.



El objetivo principal de este patrón es el de evitar bucles de actualización (o *polling*). Es decir, se suele utilizar cuando un elemento quiere estar pendiente de otro, sin tener que comprobar continuamente si ha cambiado o no.

Además, el patrón Observer, reduce el acoplamiento entre elementos (Subject y Observer apenas necesitan conocerse entre sí). Esto se consigue gracias a una API y comportamiento claramente definidos.

Curso

OFERTA

RxJS Nivel PRO

Entiende qué es y cómo usar RxJS: Aprende infinidad de Operadores RxJS y Domina laProgramación Reactiva.

Idioma: Español

X

Aprende a hacer apps móviles con **ionic 3**

[Ver curso](#)

Como funciona el patrón Observer

En el patrón Observer, el **Subject** dispone de una API con 3 métodos:

- **subscribe** : para que los Observers se suscriban
- **unsubscribe** : para que los Observers cancelen la suscripción
- **notify** : lo llama internamente cuando detecta cambios en su estado.

Por otro lado, el **Observer** expone el método **update** .

El mecanismo es bien simple. Cuando el estado interno de un **Subject** cambia, éste llama a **notify** . Este método recorre la lista con todos los Observers que tiene suscritos, y para cada uno de ellos llama a su método **update** .

Aquí tienes una implementación sencilla de la clase Subject en el patrón Observer:

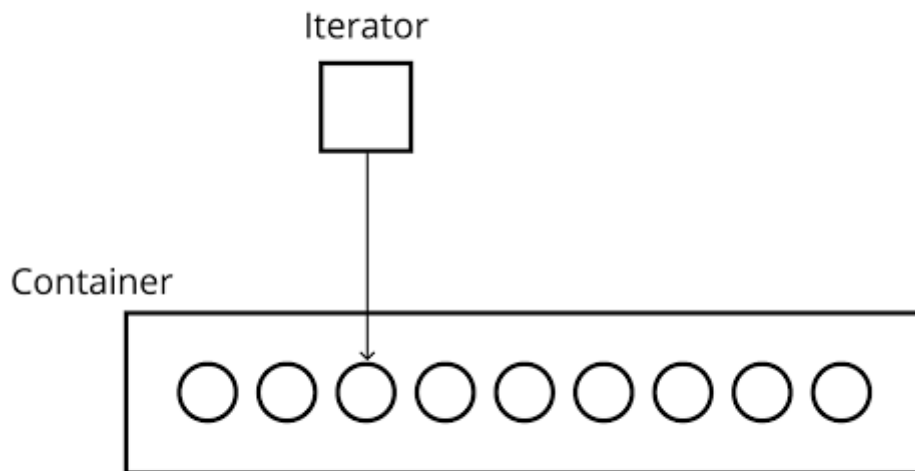
```
class Subject{
  constructor(){
    this.observers = [];
  }
  subscribe(observer){
    this.observers.push(observer);
  }
  unsubscribe(observer){
    this.observers = this.observers.filter(item => item !== observer);
  }
  notify(event){
    this.observers.forEach(observer => observer.update(event));
  }
}
```

El patrón Iterador

El patrón Iterador es otro patrón de diseño muy conocido.

X

En este caso, se utiliza un objeto (el Iterador), como mecanismo para atravesar una colección de elementos (o contenedor) de forma secuencial, para acceder a su contenido.



La gracia del Patrón Iterador, es que te permite iterar la colección sin necesidad de conocer la estructura del contenedor, gracias a una API bien definida.

Como funciona el patrón Iterador

La API de un Iterador, expone típicamente 2 métodos:

- **hasNext()** para saber si todavía quedan elementos en la colección
- **next()** para acceder al siguiente elemento de la colección

Por tanto te da igual como esté implementada la lista que contiene los datos, lo único que necesitas es saber que implementa el patrón iterador y que por tanto puedes usar estos dos métodos, así, por ejemplo:

```
let myArray = new IterableList ( 1, 2, 3, 4, 5 );

let iterator = myArray.iterator();

while( iterator.hasNext( ) ){
    console.log( iterator.next( ) );
}
```

X

La programación funcional

La programación funcional es un paradigma de programación clásico, alejado de la programación imperativa a la que estás acostumbrado, aunque en los últimos años está cogiendo fuerza de nuevo.

La idea de la programación funcional es crear código a partir de funciones en el sentido más matemático de la palabra. En matemáticas, una función se suele expresar del siguiente modo:

$$y = f(x)$$

Lo que significa que siempre que la entrada de la función sea el valor X, el resultado será Y.

- Sin efectos colaterales
- Sin estado compartido entre distintas funciones
- Sin mutaciones de datos, es decir, la función no altera la variable de entrada. X seguirá siendo X.

Sin entrar en detalles, te diré que estas tres características tienen ciertos beneficios que facilitan que tu código sea más fácil de entender y predecible.

Tampoco se necesitan ejemplos muy complicados para explicar lo que es la programación funcional. La clase Array de Javascript, sin ir más lejos, tiene algunos ejemplos, como los métodos **filter**, **map** o **reduce**.

Por ejemplo, yo podría sumar los números pares del 1 al 10 mediante programación funcional, del siguiente modo:

```
const numbers = [1,2,3,4,5,6,7,8,9,10];  
let even = numbers.filter(item => item %2 == 0);
```

X

Como **filter** y **reduce** están implementados siguiendo una aproximación funcional, se garantiza que **filter** no modifica a la variable **numbers**, o **reduce** a la variable **even**.

Además, el resultado de estas funciones no depende de ninguna variable externa, sino únicamente de los datos sobre los que se aplican (las variables **numbers** e **even**, respectivamente).

Podría alargarme bastante en los puntos positivos de la programación funcional, que es lo que la han puesto tan de moda últimamente, pero sobra material sobre este tema en internet. Si te hablo de la programación reactiva, es porque **RxJS** la utiliza de forma consistente, a través de sus **operadores**.

RxJS dispone de más de un centenar de funciones reactivas para manipular los flujos de datos. A estas funciones, les llama “operadores”, y te hablaré de ellos en detalle en el próximo artículo.

Los actores principales de RxJS

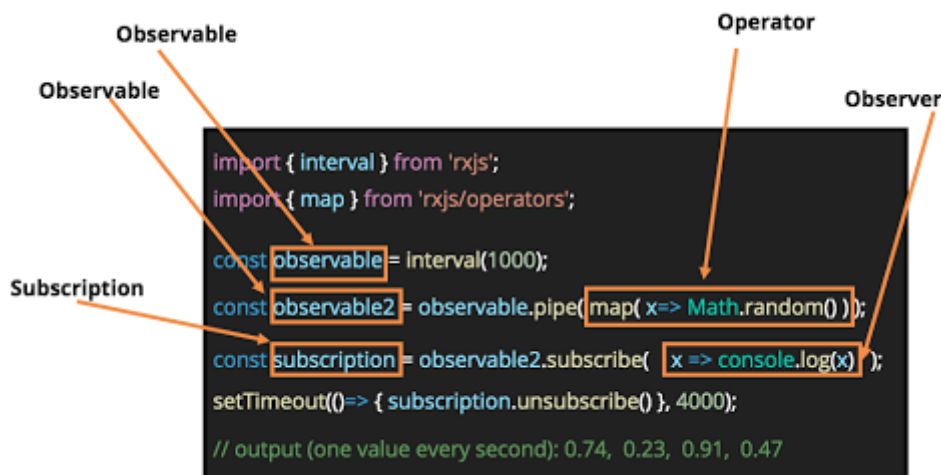
Así que RxJS está fuertemente inspirado en estos 3 aspectos. ¿Pero como los usa? Para explicarlo mejor, déjame enseñarte primero las clases principales de RxJS:

- **Observable:** El observable representa un flujo de datos, una colección de eventos que se pueden emitir en algún momento futuro.
- **Observer:** Los observers son objetos que están escuchando el flujo de datos y actúan sobre los valores que éste emite.
- **Subscription:** Una suscripción representa la ejecución de un observable y también sirve para cancelar la ejecución en un momento dado.

X

- **Subject**: Similar al *Subject* del patrón Observer. En RxJS sirven para distribuir un Observable hacia varios Observers simultáneamente.
- **Schedulers**: Los schedulers sirven para controlar el orden de las suscripciones y el orden y velocidad de emisión de eventos. En otras librerías de ReactiveX, permiten además definir el thread de ejecución, pero eso no pasa en Javascript, que es *single-threaded*, así que en RxJS no suele hablarse mucho de ellos.

A continuación puedes ver los 4 primeros elementos en acción:




RxJS y sus fundamentos

La parte de programación funcional está clara. Hay multitud de operadores para manipular los datos. ¿Que hay de los otros dos?

De los actores principales puedes ver similitudes con el **patrón Observer** (comunicar cambios de forma desacoplada mediante una API de suscripción), pero en RxJS todo gira alrededor de los **Observables**.

Haciendo la analogía con el **patrón Iterador**, un **Observable** sería un contenedor (de eventos) y el objeto **Observer** se asemeja más a su iterador, ya que implementa el método **next**.

X



```
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';

const observable = interval ( 1000 ).pipe ( take ( 4 ) );

const observer = {
  next: x => console.log ( x ),
  error: err => console.error ( err ),
  complete: () => console.log ( 'done' )
};

observable.subscribe( observer );

//output: 0, 1, 2, 3, done
```

Así que, de la misma forma en que en el **patrón Observer**, el Subject envía los cambios a través del método **update** de sus Observers, **en RxJS**, el **Observable** empuja los eventos usando el método **next** de sus **Observers**.

Hasta aquí las similitudes. Pero también tienen sus diferencias:

En el **patrón Iterador**, el método **next** del iterador simplemente devuelve un valor, y fuera de ese iterador haces lo que quieras con el valor recibido. En cambio, los **Observables** de **RxJS** funcionan en sentido contrario, pasando el evento como argumento al método **next** de sus **Observers**. Es dentro del método **next** donde el **Observer** decide que hacer con ese valor.

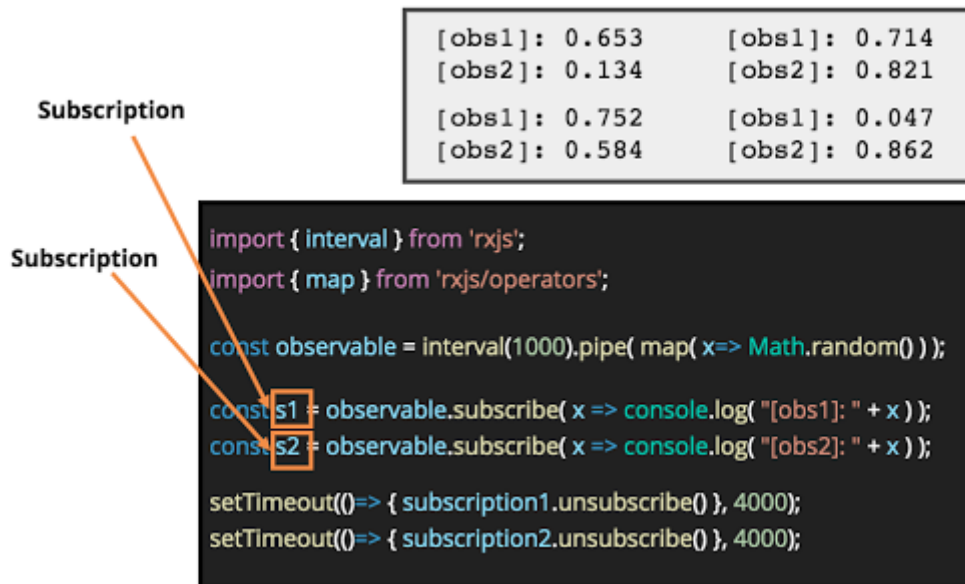
Por otro lado, mientras que en el **patrón Observer** un Subject hace *broadcast* de cualquier cambio a todos sus Observers, en **RxJS** el **Observable** **instanci**a un **nuevo flujo de datos** por cada **Observer** que se suscribe.

Por ejemplo: Imagina varias suscripciones con el **patrón Observer** donde el Subject emite un valor aleatorio. Aquí, todos los observadores recibirían el mismo valor. Esto es lo que se conoce como **comportamiento hot**.

En cambio, **con RxJS**, si un **Observable** genera eventos con un valor aleatorio, cada uno de sus observadores recibirá un valor distinto: ¡Cada suscripción es una ejecución distinta del flujo de datos! Esto es lo que se conoce como **comportamiento cold**.

Lo puedes ver en la imagen siguiente:

X



Además, en el **patrón Observer** los cambios suceden, tanto si hay objetos suscritos al Subject como si no. El Subject sencillamente notifica a los suscritos. Con los **Observables** de **RxJS** la cosa cambia. **Si no hay una suscripción, el flujo asíncrono no se ejecuta.**

El ejemplo de la imagen anterior mostraba un **Observable** que, a cada segundo, generaba un valor aleatorio. Bueno, pues si ese **Observable** no tuviera ninguna suscripción, en realidad nunca se estaría calculando ningún valor aleatorio.

Un Observable de RxJS no se ejecuta a menos que exista una suscripción al mismo.

Por cierto, la clase **Subject** de RxJS permite entre otras cosas, pasar esos *cold Observables* a *hot Observables*, para hacer *broadcast* a todos los suscritos igual que con el **patrón Observer**. Pero los **Subjects** de RxJS son otro tema, que merecen su propio artículo, así que te hablaré de ellos en próximos posts.

Si no puedes esperar a saber más, te recomiendo que le des un vistazo a mi curso **RxJS Nivel PRO**, con un espectacular rating 4.9 sobre 5 😊

RxJS Nivel PRO

Entiende qué es y cómo usar RxJS: Aprende infinidad de Operadores RxJS y Domina la Programación Reactiva.

Idioma: **Español**

23,9 €

125 €

Conclusiones

RxJS no está ganando popularidad por casualidad. Sus sólidos fundamentos en patrones que han demostrado ser muy útiles a lo largo del tiempo, son garantía de un buen diseño, y eso se nota a la hora de utilizarlo.

En cualquier caso, todavía no he entrado en materia de verdad. Casi no has visto código, te falta trastear un poco y experimentar con RxJS para entender realmente su potencial. Eso lo dejo para el próximo post...

¿Te ha gustado este artículo? No te cortes, déjame un comentario y ayúdame a compartirlo 😊

Compártelo:



Published in [Angular](#) [Javascript](#) [RxJS](#)

X

Aprende RxJS desde cero – parte I

No Newer Posts

[Return to Blog](#)

Be First to Comment

Deja un comentario

Introduce aquí tu comentario...

Author Theme modified by Enrique Oriol

Enrique Oriol

X

Aprende a hacer apps móviles con **ionic 3** [Ver curso](#)