actibyti a netmind social project

Librería RxJS

© 2017, ACTIBYTI PROJECT SLU, Barcelona
Autor: Ricardo Ahumada

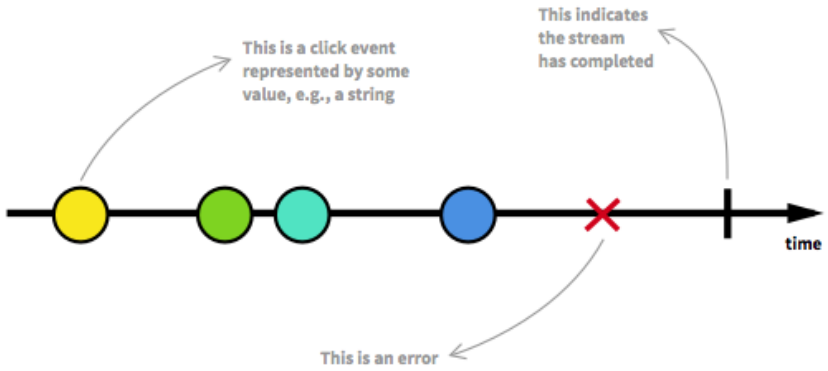
GOBIERNO DE ESPAÑA
MINISTERIO DE ENERGÍA, TURISMO Y AGENDA DIGITAL
red.es
ESTRATEGIA DE EMPRENDIMIENTO Y EMPLEO JOVEN garantía juvenil
Agenda Digital para España
UNIÓN EUROPEA
Fondo Social Europeo
"El FSE invierte en tu futuro"

ÍNDICE DE CONTENIDOS

1 Introducción

¿Qué es la programación reactiva?

- Un creciente uso de datos es el reto actual de las aplicaciones JavaScript.
- A medida que se reciben más y más datos, una aplicación debe escalar para mantenerse operativa Además existe el problema de la latencia de datos deben remotos.
- La programación reactiva es la **programación con flujos de datos concurrentes** y la **propagación del cambio**.
- En muchos casos, nos encontraremos con la palabra concurrente reemplazada por **asíncrona**, sin embargo, los flujos no tiene por que ser siempre asíncronos.



Todo es un flujo

- Es fácil ver como podemos aplicar el enfoque de "**todo es un flujo de datos**" a nuestros problemas de programación. Después de todo, una CPU no es mas que un dispositivo que procesa un flujo de información que consta de instrucciones y datos. Nuestro objetivo es observar ese flujo y transformarlo en datos particulares.
- Los principios de la programación reactiva no son totalmente nuevos para JavaScript. Ya tenemos elementos como binding de propiedades, el patrón *EventEmitter* o Node.js.
- A veces, la elegancia de estos métodos viene con una disminución de rendimiento, abstracciones demasiado complicadas o problemas con la depuración.
- Por lo general, estos inconvenientes son mínimos en comparación con las ventajas de la nueva capa de abstracción.

RxJS

- **RxJS** o **Reactive Extensions para JavaScript**, es una librería para transformar, componer y consumir flujos de datos.
  - http://reactivex.io/rxjs/
  - https://github.com/Reactive-Extensions/RxJS

- RxJS combina la programación reactiva y funcional para ofrecer un sistema de gestión de eventos asincrónico extensible para JavaScript.
- Es ideal para aplicaciones con funcionalidades que imlican flujos de datos continuos que tienen que buscar y combinar múltiples piezas de datos remotos, autocompletar cuadros de texto, arrastrar y soltar, procesar la entrada de usuarios y mucho más.
- En general, se pueden distinguir dos tipos de flujos de datos:
  - **Internos**: se puede considerar un flujo artificial y dentro de nuestro control
  - **Externo**: proviene de fuentes fuera de nuestro control y puede ser desencadenado (directa o indirectamente) de nuestro código.

Introducción a Librería RxJS

- Por lo general, las flujos no esperan por nosotros, simplemente suceden. El flujo pasa independiente de si observamos o no.
- En Rx se denomina a esto un **observable en caliente (hot observables)**.
- Rx también introduce **observables en frio (cold observables)**, que se comportan mas como **iteradores** estándar.
- En la siguiente imagen vemos algunos tipos de flujos externos:

Ejemplo – Worker Números primos

- **prime.js:** intenta encontrar los números primos de 2 a 10^{10}. una vez que se encuentra un numero, se informa del resultado.

```
(function (start, end) {
    var n = start - 1;

    while (n++ < end) {
        var k = Math.sqrt(n);
        var found = false;

        for (var i = 2; !found &&t i <= k; ++i) {
            found = n % i === 0;
        }

        if (!found) {
            postMessage(n.toString());
        }
    }
})(2, 1e10);
```

Ejemplo – Worker Números primos

- Llamada al worker:

```
var worker = new Worker('prime.js');
worker.addEventListener('message', function (ev)
{
    var primeNumber = ev.data * 1;
    console.log(primeNumber);
}, false);
```

- Se sabe que los números primos siguen una distribución asintótica entre los números enteros positivos.
- Para X a ∞ se obtiene una distribución de X/log(X). Esto significa que vamos a ver mas números al principio.
- Esto se puede ilustrar con un simple eje de tiempo y gotas para los resultados:

2 Instalación

Instalación

ES6 via npm

- En la consola

```
npm install rxjs-es
```

- Para importar todo el conjunto básico de funcionalidades:

```
import Rx from 'rxjs/Rx';

Rx.Observable.of(1,2,3)
```

- Para importar solo lo que necesitamos:

```
import { Observable} from 'rxjs/Observable';
import 'rxjs/add/observable/of';
import 'rxjs/add/operator/map';

Observable.of(1,2,3).map(x => x + '!!!'); //
etc
```

Instalación

CommonJS via npm

- ejecutamos en la línea de comandos:

```
npm install rxjs
```

- Para importar toda la funcionalidad básica:

```
var Rx = require('rxjs/Rx');

Rx.Observable.of(1,2,3); // etc
```

- Para importar solo lo que necesitamos:

```
var Observable = require('rxjs/Observable').Observable;
// patch Observable with appropriate methods
require('rxjs/add/observable/of');
require('rxjs/add/operator/map');

Observable.of(1,2,3).map(function (x) { return x + '!!!'; }); // etc
```

- Si se requieren los operadores de importación:

```
var of = require('rxjs/observable/of').of;
var map = require('rxjs/operator/map').map;

map.call(of(1,2,3), function (x) { return x + '!!!'; });
```

- También pode ar nuestro propio observable y exportarlo desde nuestro modulo.

Instalación

CDN

- Podemos utilizar unpkg. Simplemente reemplazamos “versión” con la versión actual de los siguientes enlaces:
  - Para RxJS 5.0.0-beta.1 a través de beta.11:

    https://unpkg.com/@reactivex/rxjs@version/dist/global/Rx.umd.js
  - Para RxJS 5.0.0-beta.12 y superior:

    https://unpkg.com/@reactivex/rxjs@version/dist/global/Rx.js

2 Observables

Observables y Observers

- Rx es una librería para componer programas asíncronos y basados en eventos, utilizando colecciones **observables**.
- Los bloques de construcción básicos de RxJS son:
  - **Observables**: Productores de *observables.* A su vez se dividen en dos tipos:
    - **Hot Observables :** hacen push de datos incluso cuando no estamos suscritos a ellos (por ejemplo, eventos de interfaz de usuario).
    - **Cold Observables:** estos empiezan a hacer push solo cuando nos suscribimos a ellos. Comienzan de nuevo si nos volvemos a suscribir.
  - **Observadores** (Observer): consumen observables

Cold Observables

- Por lo general se refieren a matrices o valores individuales que han sido convertidos para poder ser usados en RxJS.
- Veamos un ejemplo donde el código crea un observable en frio que solo produce un valor único antes de terminar:

```
var observable = Rx.Observable.create(function (observer) {
  observer.onNext(42);
  observer.onCompleted();
});
```

Suscripción

- La suscripción a los observables es independiente del tipo de observable y se realiza mediante el método

  suscribe(onNext, onError y onCompleted).

- Para ambos tipos podemos proporcionar tres funciones que cumplen el requisito básico de la gramática que consiste en la notificación **onNext**, **onError** y **onCompleted**.
- La devolución de llamada *onNext* es obligatoria.

```
var subscription = observable.subscribe(
  function (value) {// para onNext
      console.log('Next: %s.', value);
  },
  function (ev) {// para onErro
      console.log('Error: %s!', ev);
  },
  function () {// para onCompleted
      console.log('Completed!');
  }
);

subscription.dispose(); // limpia recursos
```

Suscripción

- Como mejor practica, deberíamos terminar la suscripción mediante el uso del método **dispose()**.
- Este llevara a cabo los pasos de limpieza requeridos. De lo contrario, se podría evitar la recolección de basura de recursos no usados.

```
subscription.dispose();
```

- Sin **suscribe()** el observable contenido en la variable **observable** es solo un observable en frio.
- Sin embargo, también es posible convertirlo en una secuencia caliente (es decir, realizar una pseudo suscripción) usando el método **publish**.

```
var hotObservable = observable.publish();
```

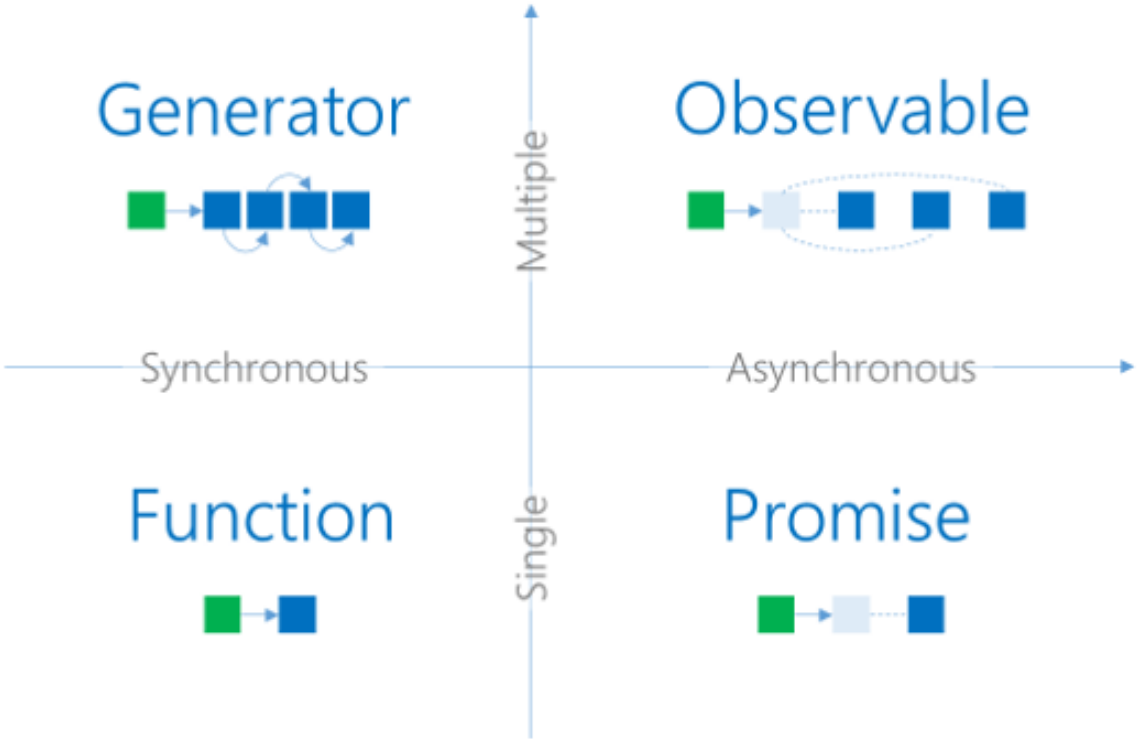
Observables y Helpers

- Los helpers son funciones de apoyan el proceso datos.
- Algunos helpers contenidos en RxJS solo tratan con la conversión de estructuras de datos existentes.
- En JavaScript, podemos distinguir entre tres de ellos:
  - **Promises**: para el retorno de resultados asíncronos individuales.
  - **Funtions**: para los resultados síncronos individuales.
  - **Generators**: para proveer iteradores (síncronos múktiples).

- Este ultimo es nuevo en ES6 y puede ser reemplazado por matrices (aunque es un mal sustituto y debe tratarse como un valor único) para ES5 o mas antiguo.

Observables y Helpers

- Con Observables, RxJS proporciona un tipo de datos que da soporte a valores (de retorno) múltiples asíncronos.
- Por lo tanto, los cuatro cuadrantes de posibilidades (síncrono-asíncrono/único-múltiple) estarían completos.



Observables y Helpers

- Mientras que los iteradores necesitan que se haga **pull**, sobre los valores de observables se hace **push**.
- Un ejemplo seria un flujo de eventos, donde no podemos forzar el próximo evento, solo podemos esperar para ser notificados por el bucle de eventos

```
var array = [1,2,3,4,5];
var source = Rx.Observable.from(array);
```

- La mayoría de los helpers que crean o manejan observables también aceptan un planificador, que controla cuando comienza una suscripción y cuando se publican las notificaciones.

- Muchos operadores en RxJS introducen concurrencia, tales como *throttle*, *Interval* o *delay*.

Consumiendo números primos

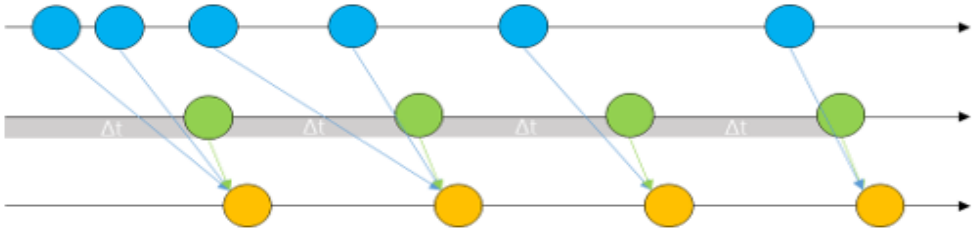
- Para el ejemplo anterior del generador de números primos; queremos agregar los resultados en un tiempo dado, tratando de que la interfaz de usuario (sobre todo al principio) no lidie con muchas actualizaciones.

Suscripción al worker

- Podemos utilizar la función *buffer* de RxJS, en conjunto con el helper *interval*.
- *Map* nos ayuda a transformar los datos.

```
var worker = new Worker('prime.js');
var observable = Rx.Observable.fromEvent(worker, 'message')
                 .map(function (ev) { return ev.data * 1; })
                 .buffer(Rx.Observable.interval(500))
                 .where(function (x) { return x.length > 0; })
                 .map(function (x) { return x.length; });
```

- El resultado se representa en el siguiente diagrama:
  - las gotas verdes después de un intervalo de tiempo especificado (dado por el tiempo utilizado para construir el *interval*).
  - Un buffer agregara todas las gotas azules visto durante ese intervalo.



Suscripción al worker

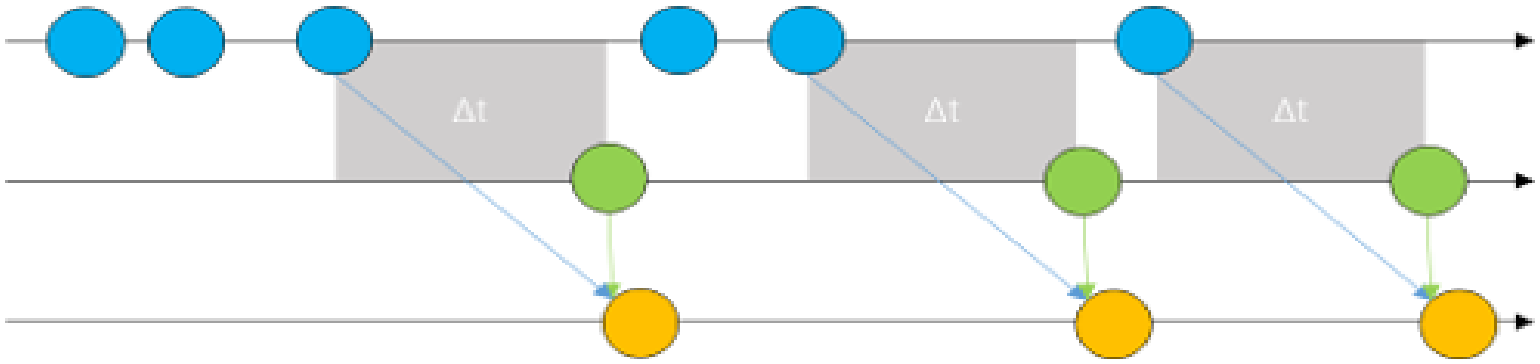
- La función *fromEvent* nos permite construir un observable a partir de cualquier objeto utilizando el patrón de emisor de sucesos estándar.
- El *buffer* también devolverá matrices con longitud cero, por lo que introducimos la función *where* para reducir el flujo a matrices no vacías.
- Específicamente en este ejemplo solo nos interesa el número de números primos generados, por lo tanto mapeamos el buffer para obtener su longitud.

Ejemplo: cuadro de búsqueda

- Un cuadro de búsqueda que hace sugerencias al usuario (realizando peticiones a una API)
- Queremos que inicie las peticiones solo después un tiempo de inactividad del usuario (no después de cada pulsación de teclas).

Posibles funciones a usar

- Hay dos funciones que nos pueden ser de utilidad en un escenario de este tipo:
  - La función ***throttle*** genera la primera entrada en un tiempo especificado.
  - La función ***debounce*** produce la ultima entrada vista dentro de una ventana de tiempo especificada.
- En consecuencia, las ventanas de tiempo también se desplazarían (en relación con el primer/ultimo elemento)
- Queremos lograr un comportamiento que se refleje en el siguiente diagrama, por lo que utilizaremos ***debounce***.



Código de suscripción

- Queremos descartar todos los resultados anteriores y solo obtener el ultimo de ellos antes de que se agote la ventana de tiempo (300ms).
- Suponiendo que el campo de entrada tiene el id *#query*, podemos utilizar el siguiente código:

```
var q = document.querySelector('#query');
var observable = Rx.Observable.fromEvent(q, 'keyup')
                .debounce(300)
                .map(function (ev) { return ev.target.value; })
                .where(function (text) { return text.length >= 3; })
                .distinctUntilChanged()
                .map(searchFor)
                .switch()
                .where(function (obj) { return obj !== undefined; });
```

Código de suscripción

- En el código anterior la ventana se establece en 300ms, y restringimos las consultas de valores con al menos 3 caracteres escritos, siempre que sean distintos de las consultas anteriores.
- Con esto eliminaríamos las solicitudes innecesarias de entradas que se acaban de corregir escribiendo algo y borrándolo.

- Hay dos partes cruciales en la expresión de suscripción:
  - Una de ellas es la transformación del texto de la consulta a una petición usando *searchFor*,
  - La otra es la función switch (). Esta toma cualquier función que devuelve observables anidados y produce valores solo de la secuencia observable mas reciente.

Función de solicitud

- La función para crear las solicitudes se podría establecer de la siguiente manera:

```
function searchFor(text) {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', apibaseUrl + '?q=' + text, true);
    xhr.send();
    return Rx.Observable.fromEvent(xhr, 'load').map(function (ev) {
        var request = ev.currentTarget;

        if (request.status === 200) {
            var response = request.responseText;
            return JSON.parse(response);
        }
    });
}
```

- Debemos tener en cuenta el observable anidado, que podría originar *undefined* a solicitudes invalidas, y es por ello que concatenamos *switch ()* y *where ().*

3 Creación y uso de Observables

Convertir a Observables

- RxJs da varias opciones para crear Observable a partir de varios tipos de datos

```
// From one or multiple values
Rx.Observable.of('foo', 'bar');

// From array of values
Rx.Observable.from([1,2,3]);

// From an event
Rx.Observable.fromEvent(document.querySelector('button'),
'click');

// From a Promise
Rx.Observable.fromPromise(fetch('/users'));
```

Convertir a Observables

```
// From a callback (last argument is a callback)
// fs.exists = (path, cb(exists))
var exists = Rx.Observable.bindCallback(fs.exists);
exists('file.txt').subscribe(exists => console.log('Does file exist?', exists));


// From a callback (last argument is a callback)
// fs.rename = (pathA, pathB, cb(err, result))
var rename = Rx.Observable.bindNodeCallback(fs.rename);
rename('file.txt', 'else.txt').subscribe(() => console.log('Renamed!'));
```

Creación de Observables

- Producir nuevos eventos externos

```
var myObservable = new Rx.Subject();
myObservable.subscribe(value => console.log(value));
myObservable.next('foo');
```

- Producir nuevos eventos internos

```
var myObservable = Rx.Observable.create(observer => {
  observer.next('foo');
  setTimeout(() => observer.next('bar'), 1000);
});
myObservable.subscribe(value => console.log(value));
```

- El **observable** normal es bueno cuando se quiere envolver una funcionalidad que produce valores en el tiempo.
  - Un ejemplo seria un **websocket**.
  - Con **subjetc** podemos activar nuevos eventos desde cualquier parte y podemos conectar observables existentes.

Control del Flujo

- Asimismo, RxJs da varias opciones para control de flujo de datos

```
// typing "hello world"
var input = Rx.Observable.fromEvent(document.querySelector('input'), 'input');

// Filter out target values less than 3 characters long
input.filter(event => event.target.value.length > 2)
  .map(event => event.target.value)
  .subscribe(value => console.log(value)); // "hel"

// Delay the events
input.delay(200)
  .map(event => event.target.value)
  .subscribe(value => console.log(value)); // "h" -200ms-> "e" -200ms-> "l" ...

// Only let through an event every 200 ms
input.throttleTime(200)
  .map(event => event.target.value)
  .subscribe(value => console.log(value)); // "h" -200ms-> "w"
```

Control del Flujo

```
// Let through latest event after 200 ms
input.debounceTime(200)
  .map(event => event.target.value)
  .subscribe(value => console.log(value)); // "o" -200ms-> "d"

// Stop the stream of events after 3 events
input.take(3)
  .map(event => event.target.value)
  .subscribe(value => console.log(value)); // "hel"

// Passes through events until other observable triggers an event
var stopStream = Rx.Observable.fromEvent(document.querySelector('button'), 'click');
input.takeUntil(stopStream)
  .map(event => event.target.value)
  .subscribe(value => console.log(value)); // "hello" (click)
```

Producir Valores

- Ofrece también opciones para generar valores

```
// typing "hello world"
var input = Rx.Observable.fromEvent(document.querySelector('input'), 'input');

// Pass on a new value
input.map(event => event.target.value)
  .subscribe(value => console.log(value)); // "h"

// Pass on a new value by plucking it
input.pluck('target', 'value')
  .subscribe(value => console.log(value)); // "h“

// Pass the two previous values
input.pluck('target', 'value').pairwise()
  .subscribe(value => console.log(value)); // ["h", "e"]

// Only pass unique values through
input.pluck('target', 'value').distinct()
  .subscribe(value => console.log(value)); // "helo wrd"

// Do not pass repeating values through
input.pluck('target', 'value').distinctUntilChanged()
  .subscribe(value => console.log(value)); // "helo world"
```

Creando un almacén de estado

- RxJS es una gran herramienta para mantener nuestro código menos propenso a errores.
- Hace posible esto usando funciones puras y sin estado.

Almacén de estado

- Las aplicaciones utilizan los **almacenes de estado** para mantener el estado de los datos de la misma.
- Los almacenes tienen nombres diferentes en diferentes framework, como **store**, **reducer** y **modelo**, pero en el core son solo un objeto simple que mantiene el estado.
- También necesitaremos manejar múltiples observables que podrán actualizar el almacén de estado ante diferentes eventos.
- Como base, creamos un almacén de estado simple: Por cada clic aumentará el valor de una variable de estado.

```
var increaseButton = document.querySelector('#increase');
var increase = Rx.Observable.fromEvent(increaseButton, 'click')
  // We map to a function that will change our state
  .map(() => state => Object.assign({}, state, {count: state.count + 1}));
```

Almacén de estado y función de estado

- En lugar de asignar un valor inicial al estado, podemos asignar una función (changeFn).
- Esta cambiará el estado del almacén de estado.

```
var increaseButton = document.querySelector('#increase');
var increase = Rx.Observable.fromEvent(increaseButton, 'click')
  .map(() => state => Object.assign({}, state, {count: state.count + 1}));

// We create an object with our initial state. Whenever a new state change function
// is received we call it and pass the state. The new state is returned and
// ready to be changed again on the next click
var state = increase.scan((state, changeFn) => changeFn(state), {count: 0});
```

Múltiples observables

- Ahora podemos añadir un par de observables más, que también cambiaran el almacén de estado (incrementar, decrementar o definir un valor).

```
var increaseButton = document.querySelector('#increase');
var increase = Rx.Observable.fromEvent(increaseButton, 'click')
  // Again we map to a function the will increase the count
  .map(() => state => Object.assign({}, state, {count: state.count + 1}));

var decreaseButton = document.querySelector('#decrease');
var decrease = Rx.Observable.fromEvent(decreaseButton, 'click')
  // We also map to a function that will decrease the count
  .map(() => state => Object.assign({}, state, {count: state.count - 1}));

var inputElement = document.querySelector('#input');
var input = Rx.Observable.fromEvent(inputElement, 'keypress')
  // Let us also map the keypress events to produce an inputValue state
  .map(event => state => Object.assign({}, state, {inputValue: event.target.value}));
```

Múltiples observables

- Mezclamos los productores de observables en el almacén y nos suscribimos a los cambios de estado

```
// We merge the three state change producing observables
var state = Rx.Observable.merge(
  increase,
  decrease,
  input
).scan((state, changeFn) => changeFn(state), {
  count: 0,
  inputValue: ''
});

// We subscribe to state changes and update the DOM
state.subscribe((state) => {
  document.querySelector('#count').innerHTML = state.count;
  document.querySelector('#hello').innerHTML = 'Hello ' + state.inputValue;
});
```

Optimización

- Podemos optimizar el código para verificar qué estado realmente ha cambiado

```
// To optimize our rendering we can check what state
// has actually changed
var prevState = {};
state.subscribe((state) => {
  if (state.count !== prevState.count) {
    document.querySelector('#count').innerHTML = state.count;
  }
  if (state.inputValue !== prevState.inputValue) {
    document.querySelector('#hello').innerHTML = 'Hello ' + state.inputValue;
  }
  prevState = state;
});
```

- Podemos tomar este enfoque de **almacén de estado** y usarlo con muchos frameworks y librerías diferentes.

netmind

WeKnowIT

Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

GOBIERNO DE ESPAÑA
MINISTERIO DE ENERGÍA, TURISMO Y AGENDA DIGITAL
red.es
ESTRATEGIA DE EMPRENDIMIENTO Y EMPLEO JOVEN garantía juvenil
Agenda Digital para España
UNIÓN EUROPEA
Fondo Social Europeo
"El FSE invierte en tu futuro"