My gentle introduction to RxJS

About me

Mattia Occhiuto

Front-end Software Engineer @ Tuotempo s.r.l.

Twitter @MattiaOch

Github mattiaocchiuto

Reactive Systems

" Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback.

from www.reactivemanifesto.org

So…

Reactive Systems are:

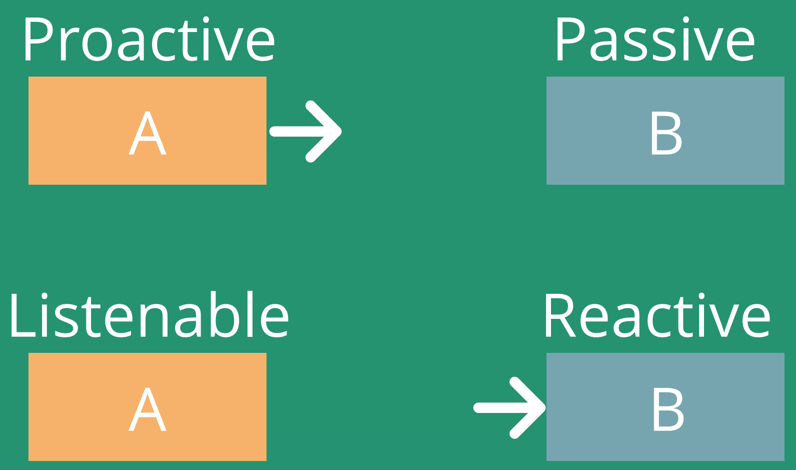
- **Responsive** - system respond in a timely manner where possible
- **Resilient** - system stays responsive in case of failure
- **Elastic** - scalable
- **Message driven** - based on async messages

Reactive Programming

" reactive programming is a programming paradigm ***oriented around data flows*** *and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow.*

from wikipedia

graphically ...



From Cycle.js documentation

- **using asynchronous data streams**
- more declarative
- more reusability
- more testability
- increase separation on concerns

Why we need it ?

Nowadays Modern Web Applications are complex **systems** composed of asynchronous data coming from many different sources:

- user interaction (DOM-events)
- AJAX call
- device event (GPS)
- web socket
- web workers
- reactive user interface (Model/View sync)
- etc...

RxJS - The Observables way

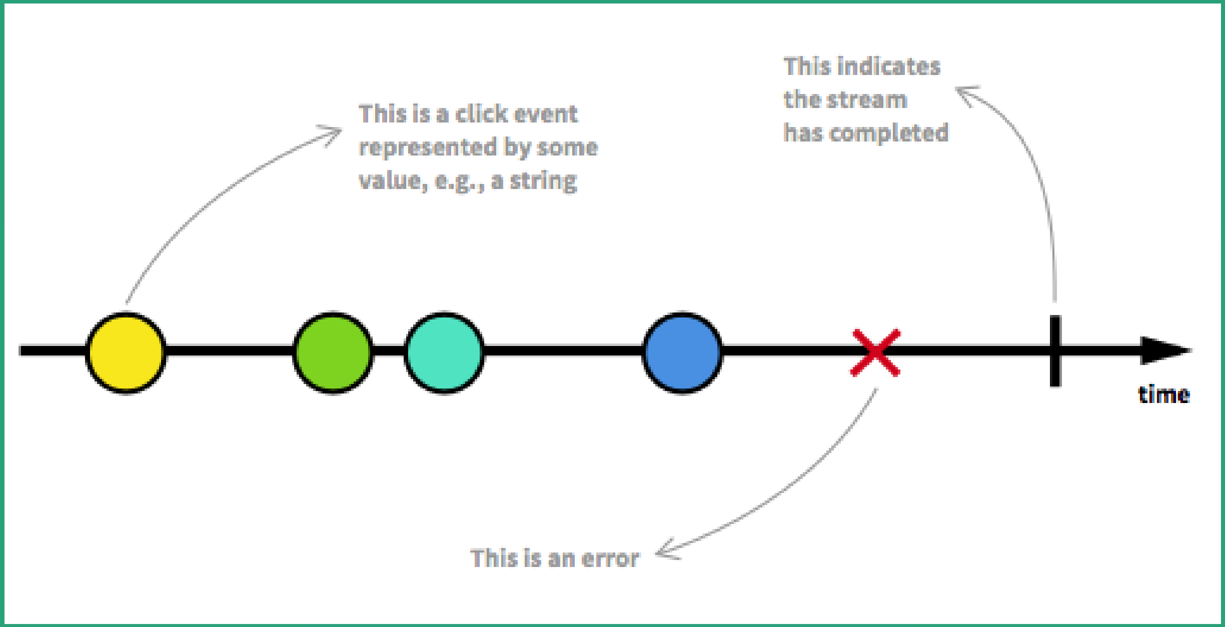
Iterator + Observer

```
var arr = ['a', 'b', 'c'];
var iter = arr[Symbol.iterator]();

iter.next()
>   { value: 'a', done: false }
iter.next()
>   { value: 'b', done: false }
iter.next()
>   { value: 'c', done: false }
iter.next()
>   { value: undefined, done: true }
```

```
var el = document.getElementById("button");

el.addEventListener("click", doSomething);
```

RxJS - The Observables way

In short, an **Observable** is an event stream which can emit zero or more events, and may or may not finish. If it finishes, then it does so by either emitting an error or a special “complete” event

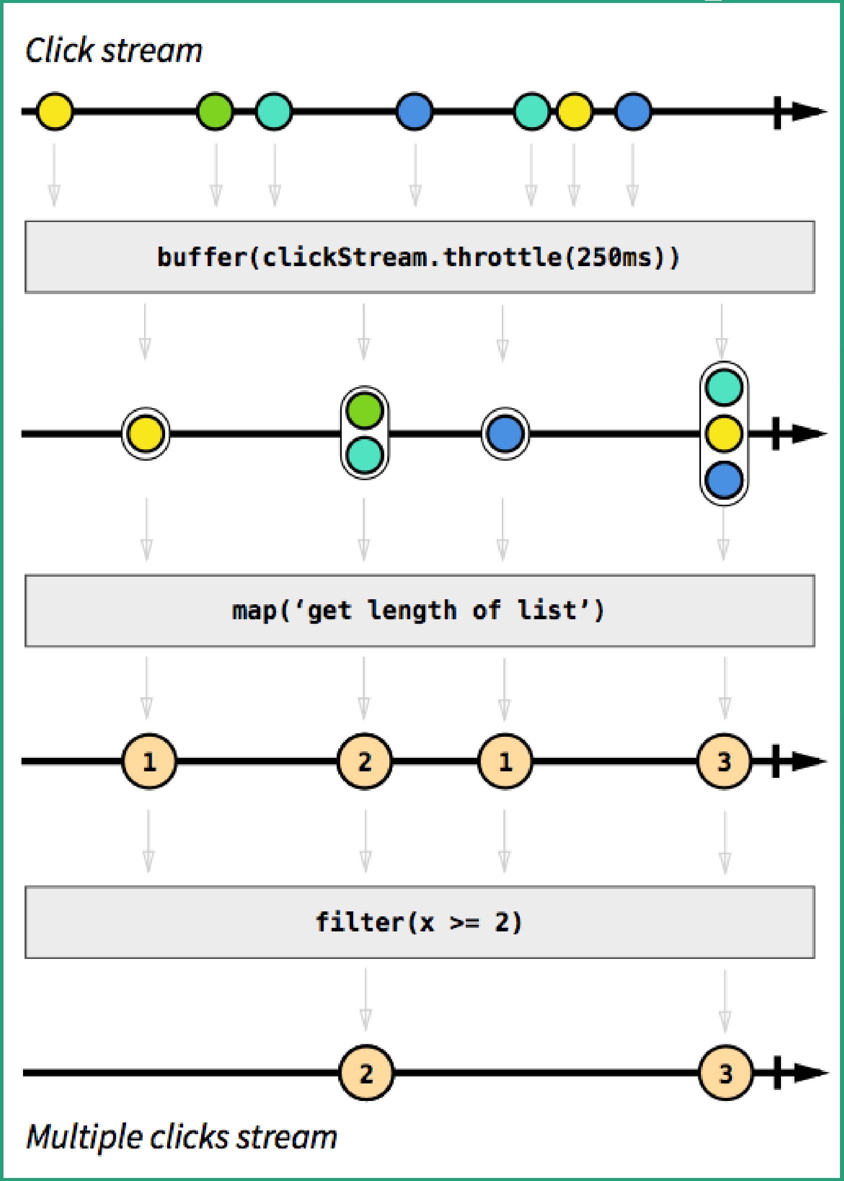

This is a click event represented by some value, e.g., a string
This indicates the stream has completed
time
This is an error

And What about operators?

Both are collection and both of course has
powerful operators!

Array [1, 2, 3, 4, 5]	Stream {…1..2…..3…4……5..}
.map	.map
.filter	.filter
.reduce	.reduce
.each	.each
…	…(and > 120 other operators)

And What about operators?



Click stream
buffer(clickStream.throttle(250ms))
map('get length of list')
1
2
1
3
filter(x >= 2)
2
3
Multiple clicks stream

How they looks like

```
var source = Rx.Observable.fromEvent($input, 'keyup')

var subscription = source.subscribe(
  function (x) {
    console.log('Next: ' + x);
  },
  function (e) {
    console.log('Error: ' + e);
  },
  function () {
    console.log('Completed');
  }
);

subscription.dispose();
```

- data pushed
- centralised error handling
- notification on complete
- disposable

Who is the right source?

Easily: everything

More precisely (?): every data structure or async data source

RxJS here we are!

- Reactive programming
- Events as collection of data
- Manipulation over Array-like "operators"

Let's code!

https://github.com/mattiaocchiuto/talks/blob/master/RxJS/

Conslusion

Cons:

- too many operators that can lead to the same results and to remember!
- learning curve
- not enough examples and tutorial (at least for me)

Conslusion

Pros:

- great abstraction for async and events data/operations
- easy way to compose async data stream
- support for many languages: [Java, .Net, Scala, C#, Clojure, C++, Ruby, Python, Swift, etc..]
- performance => for big array processing, it remove intermediate array allocation
- super powerful operators

Questions?

Thanks!