



## The introduction to Reactive Programming you've been missing

introrx.md

## The introduction to Reactive Programming you've been missing

(by [@andrealtz](#))

### This tutorial as a series of videos

If you prefer to watch video tutorials with live-coding, then check out this series I recorded with the same contents as in this article: [Egghead.io - Introduction to Reactive Programming](#).

So you're curious in learning this new thing called Reactive Programming, particularly its variant comprising of Rx, Bacon.js, RAC, and others.

Learning it is hard, even harder by the lack of good material. When I started, I tried looking for tutorials. I found only a handful of practical guides, but they just scratched the surface and never tackled the challenge of building the whole architecture around it. Library documentations often don't help when you're trying to understand some function. I mean, honestly, look at this:

```
Rx.Observable.prototype.flatMapLatest(selector, [thisArg])
```

Projects each element of an observable sequence into a new sequence of observable sequences by incorporating the element's index and then transforms an observable sequence of observable sequences into an observable sequence producing values only from the most recent observable sequence.

Holy cow.

I've read two books, one just painted the big picture, while the other dived into how to use the Reactive library. I ended up learning Reactive Programming the hard way: figuring it out while building with it. At my work in [Futurice](#) I got to use it in a real project, and had the [support of some colleagues](#) when I ran into troubles.

The hardest part of the learning journey is **thinking in Reactive**. It's a lot about letting go of old imperative and stateful habits of typical programming, and forcing your brain to work in a different paradigm. I haven't found any guide on the internet in this aspect, and I think the world deserves a practical tutorial on how to think in Reactive, so that you can get started. Library documentation can light your way after that. I hope this helps you.

## "What is Reactive Programming?"

There are plenty of bad explanations and definitions out there on the internet. [Wikipedia](#) is too generic and theoretical as usual. [Stackoverflow](#)'s canonical answer is obviously not suitable for newcomers. [Reactive Manifesto](#) sounds like the kind of thing you show to your project manager or the businessmen at your company. Microsoft's [Rx terminology](#) "Rx = Observables + LINQ + Schedulers" is so heavy and Microsoftish that most of us are left confused. Terms like "reactive" and "propagation of change" don't convey anything specifically different to what your typical MV\* and favorite language already does. Of course my framework views react to the models. Of course change is propagated. If it wouldn't, nothing would be rendered.

So let's cut the bullshit.

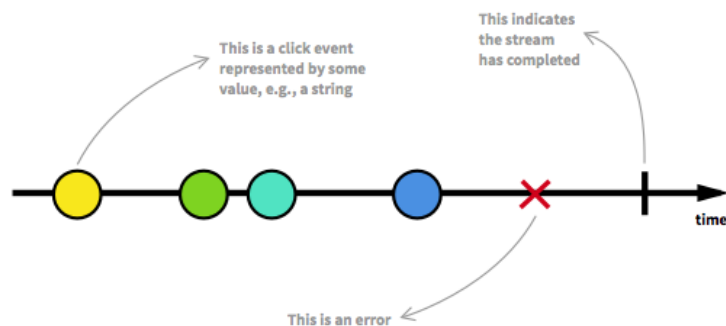
**Reactive programming is programming with asynchronous data streams.**

In a way, this isn't anything new. Event buses or your typical click events are really an asynchronous event stream, on which you can observe and do some side effects. Reactive is that idea on steroids. You are able to create data streams of anything, not just from click and hover events. Streams are cheap and ubiquitous, anything can be a stream: variables, user inputs, properties, caches, data structures, etc. For example, imagine your Twitter feed would be a data stream in the same fashion that click events are. You can listen to that stream and react accordingly.

**On top of that, you are given an amazing toolbox of functions to combine, create and filter any of those streams.**

That's where the "functional" magic kicks in. A stream can be used as an input to another one. Even multiple streams can be used as inputs to another stream. You can *merge* two streams. You can *filter* a stream to get another one that has only those events you are interested in. You can *map* data values from one stream to another new one.

If streams are so central to Reactive, let's take a careful look at them, starting with our familiar "clicks on a button" event stream.



A stream is a sequence of **ongoing events ordered in time**. It can emit three different things: a value (of some type), an error, or a "completed" signal. Consider that the "completed" takes place, for instance, when the current window or view containing that button is closed.

We capture these emitted events only **asynchronously**, by defining a function that will execute when a value is emitted, another function when an error is emitted, and another function when 'completed' is emitted. Sometimes these last two can be omitted and you can just focus on defining the function for values. The "listening" to the stream is called **subscribing**. The functions we are defining are observers. The stream is the subject (or "observable") being observed. This is precisely the [Observer Design Pattern](#).

An alternative way of drawing that diagram is with ASCII, which we will use in some parts of this tutorial:

```
--a---b-c---d---X---|-->
```

```
a, b, c, d are emitted values
X is an error
| is the 'completed' signal
---> is the timeline
```

Since this feels so familiar already, and I don't want you to get bored, let's do something new: we are going to create new click event streams transformed out of the original click event stream.

First, let's make a counter stream that indicates how many times a button was clicked. In common Reactive libraries, each stream has many functions attached to it, such as `map`, `filter`, `scan`, etc. When you call one of these functions, such as `clickStream.map(f)`, it returns a **new stream** based on the click stream. It does not modify the original click stream in any way. This is a property called **immutability**, and it goes together with Reactive streams just like pancakes are good with syrup. That allows us to chain functions like `clickStream.map(f).scan(g)`:

```
clickStream: ---c---c---c---c---c---c--->
              vvvvv map(c becomes 1) vvvv
              ---1---1---1---1---1---1--->
```

```

vvvvvvvvv scan(+) vvvvvvvvv
counterStream: ---1---2---3---4-----5-->

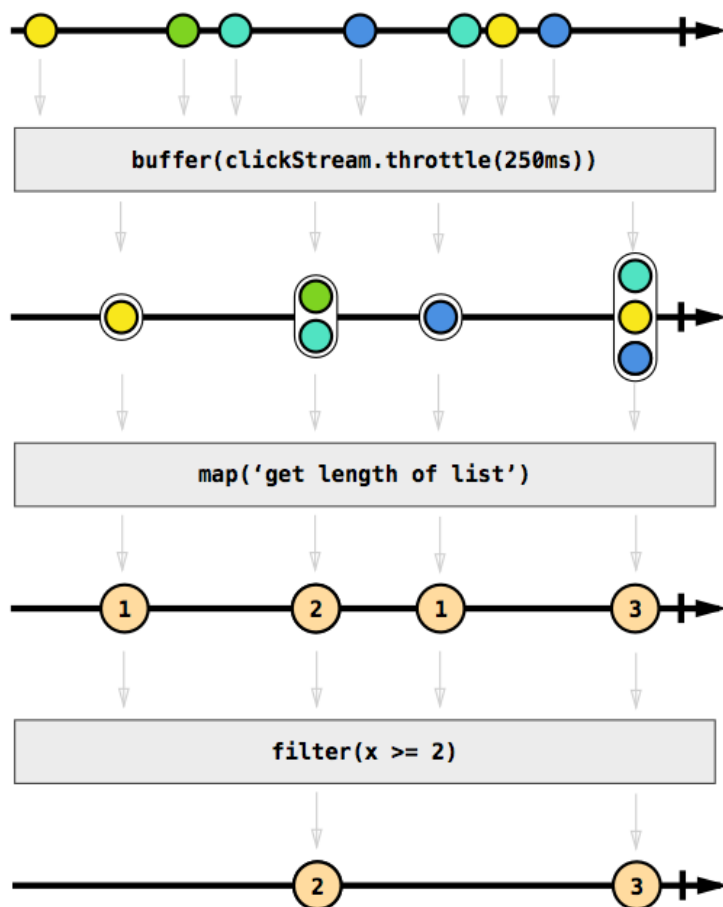
```

The `map(f)` function replaces (into the new stream) each emitted value according to a function `f` you provide. In our case, we mapped to the number 1 on each click. The `scan(g)` function aggregates all previous values on the stream, producing value  $x = g(\text{accumulated}, \text{current})$ , where `g` was simply the add function in this example. Then, `counterStream` emits the total number of clicks whenever a click happens.

To show the real power of Reactive, let's just say that you want to have a stream of "double click" events. To make it even more interesting, let's say we want the new stream to consider triple clicks as double clicks, or in general, multiple clicks (two or more). Take a deep breath and imagine how you would do that in a traditional imperative and stateful fashion. I bet it sounds fairly nasty and involves some variables to keep state and some fiddling with time intervals.

Well, in Reactive it's pretty simple. In fact, the logic is just [4 lines of code](#). But let's ignore code for now. Thinking in diagrams is the best way to understand and build streams, whether you're a beginner or an expert.

### Click stream



### Multiple clicks stream

Grey boxes are functions transforming one stream into another. First we accumulate clicks in lists, whenever 250 milliseconds of "event silence" has happened (that's what `buffer(stream.throttle(250ms))` does, in a nutshell. Don't worry about understanding the details at this point, we are just demoing Reactive for now). The result is a stream of lists, from which we apply `map()` to map each list to an integer matching the length of that list. Finally, we ignore 1 integers using the `filter(x >= 2)` function. That's it: 3 operations to produce our intended stream. We can then subscribe ("listen") to it to react accordingly how we wish.

I hope you enjoy the beauty of this approach. This example is just the tip of the iceberg: you can apply the same operations on different kinds of streams, for instance, on a stream of API responses; on the other hand, there are many other functions available.

## "Why should I consider adopting RP?"

Reactive Programming raises the level of abstraction of your code so you can focus on the interdependence of events that define the business logic, rather than having to constantly fiddle with a large amount of implementation details. Code in RP will likely be more concise.

The benefit is more evident in modern webapps and mobile apps that are highly interactive with a multitude of UI events related to data events. 10 years ago, interaction with web pages was basically about submitting a long form to the backend and performing simple rendering to the frontend. Apps have evolved to be more real-time: modifying a single form field can automatically trigger a save to the backend, "likes" to some content can be reflected in real time to other connected users, and so forth.

Apps nowadays have an abundance of real-time events of every kind that enable a highly interactive experience to the user. We need tools for properly dealing with that, and Reactive Programming is an answer.

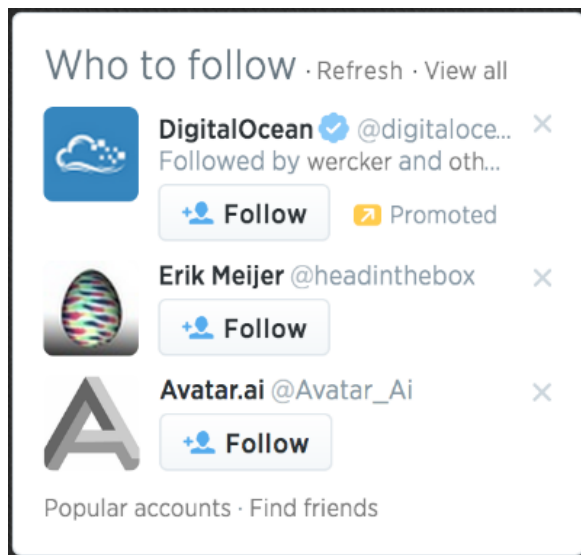
## Thinking in RP, with examples

Let's dive into the real stuff. A real-world example with a step-by-step guide on how to think in RP. No synthetic examples, no half-explained concepts. By the end of this tutorial we will have produced real functioning code, while knowing why we did each thing.

I picked **JavaScript** and **RxJS** as the tools for this, for a reason: JavaScript is the most familiar language out there at the moment, and the **Rx\* library family** is widely available for many languages and platforms ([.NET](#), [Java](#), [Scala](#), [Clojure](#), [JavaScript](#), [Ruby](#), [Python](#), [C++](#), [Objective-C/Cocoa](#), [Groovy](#), etc). So whatever your tools are, you can concretely benefit by following this tutorial.

## Implementing a "Who to follow" suggestions box

In Twitter there is this UI element that suggests other accounts you could follow:



We are going to focus on imitating its core features, which are:

- On startup, load accounts data from the API and display 3 suggestions
- On clicking "Refresh", load 3 other account suggestions into the 3 rows
- On click 'x' button on an account row, clear only that current account and display another
- Each row displays the account's avatar and links to their page

We can leave out the other features and buttons because they are minor. And, instead of Twitter, which recently closed its API to the unauthorized public, let's build that UI for following people on Github. There's a [Github API for getting users](#).

The complete code for this is ready at <http://jsfiddle.net/staltz/8jFJH/48/> in case you want to take a peak already.

## Request and response

**How do you approach this problem with Rx?** Well, to start with, (almost) *everything can be a stream*. That's the Rx mantra. Let's start with the easiest feature: "on startup, load 3 accounts data from the API". There is nothing special here, this is simply about (1) doing a request, (2) getting a response, (3) rendering the response. So let's go ahead and represent our requests as a stream. At first this will feel like overkill, but we need to start from the basics, right?

On startup we need to do only one request, so if we model it as a data stream, it will be a stream with only one emitted value. Later, we know we will have many requests happening, but for now, it is just one.

```
--a-----|-->
```

Where `a` is the string `'https://api.github.com/users'`

This is a stream of URLs that we want to request. Whenever a request event happens, it tells us two things: when and what. "When" the request should be executed is when the event is emitted. And "what" should be requested is the value emitted: a string containing the URL.

To create such stream with a single value is very simple in Rx\*. The official terminology for a stream is "Observable", for the fact that it can be observed, but I find it to be a silly name, so I call it *stream*.

```
var requestStream = Rx.Observable.just('https://api.github.com/users');
```

But now, that is just a stream of strings, doing no other operation, so we need to somehow make something happen when that value is emitted. That's done by **subscribing** to the stream.

```
requestStream.subscribe(function(requestUrl) {
  // execute the request
  jQuery.getJSON(requestUrl, function(responseData) {
    // ...
  });
})
```

Notice we are using a jQuery Ajax callback (which we assume you **should know already**) to handle the asynchronicity of the request operation. But wait a moment, Rx is for dealing with **asynchronous** data streams. Couldn't the response for that request be a stream containing the data arriving at some time in the future? Well, at a conceptual level, it sure looks like it, so let's try that.

```
requestStream.subscribe(function(requestUrl) {
  // execute the request
  var responseStream = Rx.Observable.create(function (observer) {
    jQuery.getJSON(requestUrl)
      .done(function(response) { observer.onNext(response); })
      .fail(function(jqXHR, status, error) { observer.onError(error); })
      .always(function() { observer.onCompleted(); });
  });

  responseStream.subscribe(function(response) {
    // do something with the response
  });
})
```

What `Rx.Observable.create()` does is create your own custom stream by explicitly informing each observer (or in other words, a "subscriber") about data events ( `onNext()` ) or errors ( `onError()` ). What we did was just wrap that jQuery Ajax Promise. **Excuse me, does this mean that a Promise is an Observable?**

Yes.

Observable is Promise++. In Rx you can easily convert a Promise to an Observable by doing `var stream = Rx.Observable.fromPromise(promise)`, so let's use that. The only difference is that Observables are not [Promises/A+](#) compliant, but conceptually there is no clash. A Promise is simply an Observable with one single emitted value. Rx streams go beyond promises by allowing many returned values.

This is pretty nice, and shows how Observables are at least as powerful as Promises. So if you believe the Promises hype, keep an eye on what Rx Observables are capable of.

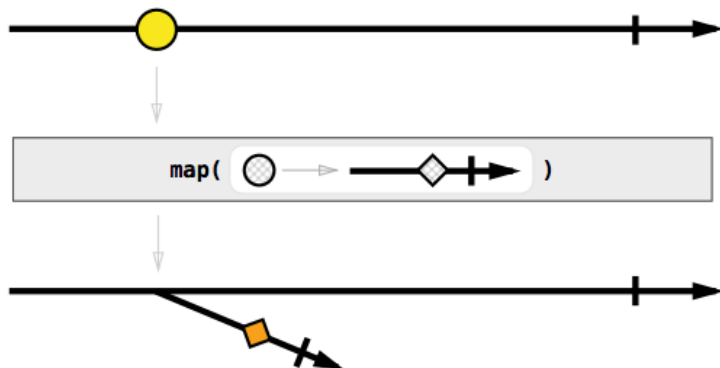
Now back to our example, if you were quick to notice, we have one `subscribe()` call inside another, which is somewhat akin to callback hell. Also, the creation of `responseStream` is dependent on `requestStream`. As you heard before, in Rx there are simple mechanisms for transforming and creating new streams out of others, so we should be doing that.

The one basic function that you should know by now is `map(f)`, which takes each value of stream A, applies `f()` on it, and produces a value on stream B. If we do that to our request and response streams, we can map request URLs to response Promises (disguised as streams).

```
var responseMetastream = requestStream
  .map(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });
```

Then we will have created a beast called "*metastream*": a stream of streams. Don't panic yet. A metastream is a stream where each emitted value is yet another stream. You can think of it as [pointers](#): each emitted value is a *pointer* to another stream. In our example, each request URL is mapped to a pointer to the promise stream containing the corresponding response.

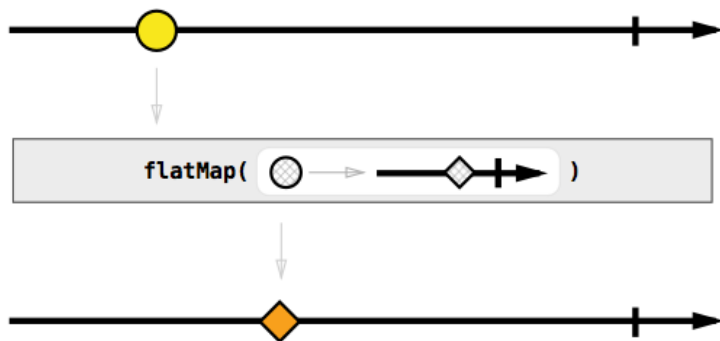
### Request stream



### Response metastream

A metastream for responses looks confusing, and doesn't seem to help us at all. We just want a simple stream of responses, where each emitted value is a JSON object, not a 'Promise' of a JSON object. Say hi to [Mr. Flatmap](#): a version of `map()` that "flattens" a metastream, by emitting on the "trunk" stream everything that will be emitted on "branch" streams. Flatmap is not a "fix" and metastreams are not a bug, these are really the tools for dealing with asynchronous responses in Rx.

```
var responseStream = requestStream
  .flatMap(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });
```

*Request stream**Response stream*

Nice. And because the response stream is defined according to request stream, if we have later on more events happening on request stream, we will have the corresponding response events happening on response stream, as expected:

```
requestStream: --a-----b--c-----|->
responseStream: -----A-----B-----C---|->

(lowercase is a request, uppercase is its response)
```

Now that we finally have a response stream, we can render the data we receive:

```
responseStream.subscribe(function(response) {
  // render `response` to the DOM however you wish
});
```

Joining all the code until now, we have:

```
var requestStream = Rx.Observable.just('https://api.github.com/users');

var responseStream = requestStream
  .flatMap(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });

responseStream.subscribe(function(response) {
  // render `response` to the DOM however you wish
});
```

## The refresh button

I did not yet mention that the JSON in the response is a list with 100 users. The API only allows us to specify the page offset, and not the page size, so we're using just 3 data objects and wasting 97 others. We can ignore that problem for now, since later on we will see how to cache the responses.

Everytime the refresh button is clicked, the request stream should emit a new URL, so that we can get a new response. We need two things: a stream of click events on the refresh button (mantra: anything can be a stream), and we need to change the request stream to depend on the refresh click stream. Gladly, RxJS comes with tools to make Observables from event listeners.

```
var refreshButton = document.querySelector('.refresh');
var refreshClickStream = Rx.Observable.fromEvent(refreshButton, 'click');
```

Since the refresh click event doesn't itself carry any API URL, we need to map each click to an actual URL. Now we change the request stream to be the refresh click stream mapped to the API endpoint with a random offset parameter each time.

```
var requestStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });
```

Because I'm dumb and I don't have automated tests, I just broke one of our previously built features. A request doesn't happen anymore on startup, it happens only when the refresh is clicked. Urgh. I need both behaviors: a request when *either* a refresh is clicked *or* the webpage was just opened.

We know how to make a separate stream for each one of those cases:

```
var requestOnRefreshStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });

var startupRequestStream = Rx.Observable.just('https://api.github.com/users');
```

But how can we "merge" these two into one? Well, there's `merge()`. Explained in the diagram dialect, this is what it does:

```
stream A: ---a-----e-----o----->
stream B: -----B---C-----D----->
          vvvvvvvvv merge vvvvvvvvv
          ---a-B---C--e--D--o----->
```

It should be easy now:

```
var requestOnRefreshStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });

var startupRequestStream = Rx.Observable.just('https://api.github.com/users');

var requestStream = Rx.Observable.merge(
  requestOnRefreshStream, startupRequestStream
);
```

There is an alternative and cleaner way of writing that, without the intermediate streams.

```
var requestStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  })
  .merge(Rx.Observable.just('https://api.github.com/users'));
```

Even shorter, even more readable:

```
var requestStream = refreshClickStream
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  })
  .startWith('https://api.github.com/users');
```



The `startWith()` function does exactly what you think it does. No matter how your input stream looks like, the output stream resulting of `startWith(x)` will have `x` at the beginning. But I'm not **DRY** enough, I'm repeating the API endpoint string. One way to fix this is by moving the `startWith()` close to the `refreshClickStream`, to essentially "emulate" a refresh click on startup.

```
var requestStream = refreshClickStream.startWith('startup click')
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });
```

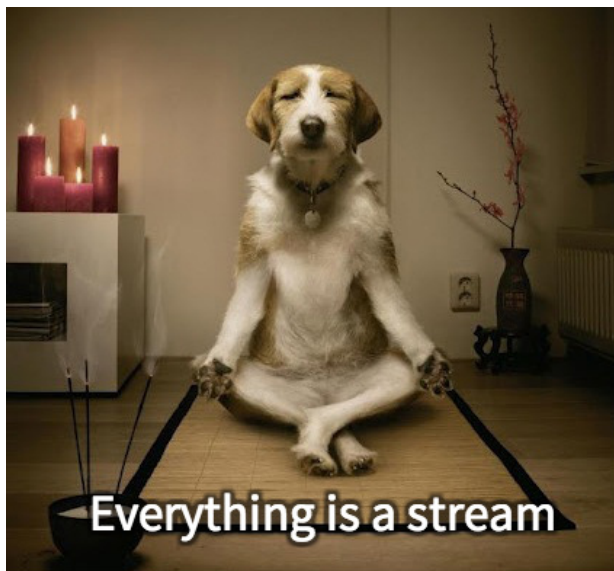
Nice. If you go back to the point where I "broke the automated tests", you should see that the only difference with this last approach is that I added the `startWith()`.

## Modelling the 3 suggestions with streams

Until now, we have only touched a *suggestion* UI element on the rendering step that happens in the `responseStream`'s `subscribe()`. Now with the refresh button, we have a problem: as soon as you click 'refresh', the current 3 suggestions are not cleared. New suggestions come in only after a response has arrived, but to make the UI look nice, we need to clean out the current suggestions when clicks happen on the refresh.

```
refreshClickStream.subscribe(function() {
  // clear the 3 suggestion DOM elements
});
```

No, not so fast, pal. This is bad, because we now have **two** subscribers that affect the suggestion DOM elements (the other one being `responseStream.subscribe()`), and that doesn't really sound like [Separation of concerns](#). Remember the Reactive mantra?



So let's model a suggestion as a stream, where each emitted value is the JSON object containing the suggestion data. We will do this separately for each of the 3 suggestions. This is how the stream for suggestion #1 could look like:

```
var suggestion1Stream = responseStream
  .map(function(listUsers) {
    // get one random user from the list
    return listUsers[Math.floor(Math.random()*listUsers.length)];
  });
```

The others, `suggestion2Stream` and `suggestion3Stream` can be simply copy pasted from `suggestion1Stream`. This is not DRY, but it will keep our example simple for this tutorial, plus I think it's a good exercise to think how to avoid repetition in this case.

Instead of having the rendering happen in `responseStream`'s `subscribe()`, we do that here:

```
suggestion1Stream.subscribe(function(suggestion) {
  // render the 1st suggestion to the DOM
});
```

Back to the "on refresh, clear the suggestions", we can simply map refresh clicks to `null` suggestion data, and include that in the `suggestion1Stream`, as such:

```
var suggestion1Stream = responseStream
  .map(function(listUsers) {
    // get one random user from the list
    return listUsers[Math.floor(Math.random()*listUsers.length)];
  })
  .merge(
    refreshClickStream.map(function(){ return null; })
  );
```

And when rendering, we interpret `null` as "no data", hence hiding its UI element.

```
suggestion1Stream.subscribe(function(suggestion) {
  if (suggestion === null) {
    // hide the first suggestion DOM element
  }
  else {
    // show the first suggestion DOM element
    // and render the data
  }
});
```

The big picture is now:

```
refreshClickStream: -----o-----o---->
  requestStream:  -r-----r-----r---->
  responseStream:  ----R-----R-----R-->
  suggestion1Stream: ----s-----N---s-----N-s-->
  suggestion2Stream: ----q-----N---q-----N-q-->
  suggestion3Stream: ----t-----N---t-----N-t-->
```

Where `N` stands for `null`.

As a bonus, we can also render "empty" suggestions on startup. That is done by adding `startWith(null)` to the suggestion streams:

```
var suggestion1Stream = responseStream
  .map(function(listUsers) {
    // get one random user from the list
    return listUsers[Math.floor(Math.random()*listUsers.length)];
  })
  .merge(
    refreshClickStream.map(function(){ return null; })
  )
  .startWith(null);
```

Which results in:

```
refreshClickStream: -----o-----o---->
  requestStream: -r-----r-----r---->
  responseStream: ----R-----R-----R-->
  suggestion1Stream: -N--s-----N---s---N-s-->
  suggestion2Stream: -N--q-----N---q---N-q-->
  suggestion3Stream: -N--t-----N---t---N-t-->
```

## Closing a suggestion and using cached responses

There is one feature remaining to implement. Each suggestion should have its own 'x' button for closing it, and loading another in its place. At first thought, you could say it's enough to make a new request when any close button is clicked:

```
var close1Button = document.querySelector('.close1');
var close1ClickStream = Rx.Observable.fromEvent(close1Button, 'click');
// and the same for close2Button and close3Button

var requestStream = refreshClickStream.startWith('startup click')
  .merge(close1ClickStream) // we added this
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });
```

That does not work. It will close and reload *all* suggestions, rather than just only the one we clicked on. There are a couple of different ways of solving this, and to keep it interesting, we will solve it by reusing previous responses. The API's response page size is 100 users while we were using just 3 of those, so there is plenty of fresh data available. No need to request more.

Again, let's think in streams. When a 'close1' click event happens, we want to use the *most recently emitted* response on `responseStream` to get one random user from the list in the response. As such:

```
requestStream: --r----->
responseStream: -----R----->
close1ClickStream: -----c----->
suggestion1Stream: -----s----->
```

In Rx\* there is a combinator function called `combineLatest` that seems to do what we need. It takes two streams A and B as inputs, and whenever either stream emits a value, `combineLatest` joins the two most recently emitted values `a` and `b` from both streams and outputs a value `c = f(x,y)`, where `f` is a function you define. It is better explained with a diagram:

```
stream A: --a-----e-----i----->
stream B: -----b-----c-----d-----q----->
          vvvvvvvv combineLatest(f) vvvvvvvv
          ----AB---AC--EC---ED--ID--IQ----->
```

where `f` is the uppercase function

We can apply `combineLatest()` on `close1ClickStream` and `responseStream`, so that whenever the close 1 button is clicked, we get the latest response emitted and produce a new value on `suggestion1Stream`. On the other hand, `combineLatest()` is symmetric: whenever a new response is emitted on `responseStream`, it will combine with the latest 'close 1' click to produce a new suggestion. That is interesting, because it allows us to simplify our previous code for `suggestion1Stream`, like this:

```
var suggestion1Stream = close1ClickStream
  .combineLatest(responseStream,
    function(click, listUsers) {
      return listUsers[Math.floor(Math.random()*listUsers.length)];
    }
  )
```

```
.merge(
  refreshClickStream.map(function(){ return null; })
)
.startWith(null);
```

One piece is still missing in the puzzle. The `combineLatest()` uses the most recent of the two sources, but if one of those sources hasn't emitted anything yet, `combineLatest()` cannot produce a data event on the output stream. If you look at the ASCII diagram above, you will see that the output has nothing when the first stream emitted value `a`. Only when the second stream emitted value `b` could it produce an output value.

There are different ways of solving this, and we will stay with the simplest one, which is simulating a click to the 'close 1' button on startup:

```
var suggestion1Stream = close1ClickStream.startWith('startup click') // we added this
.combineLatest(responseStream,
  function(click, listUsers) {
    return listUsers[Math.floor(Math.random()*listUsers.length)];
  }
)
.merge(
  refreshClickStream.map(function(){ return null; })
)
.startWith(null);
```

## Wrapping up

And we're done. The complete code for all this was:

```
var refreshButton = document.querySelector('.refresh');
var refreshClickStream = Rx.Observable.fromEvent(refreshButton, 'click');

var closeButton1 = document.querySelector('.close1');
var close1ClickStream = Rx.Observable.fromEvent(closeButton1, 'click');
// and the same logic for close2 and close3

var requestStream = refreshClickStream.startWith('startup click')
  .map(function() {
    var randomOffset = Math.floor(Math.random()*500);
    return 'https://api.github.com/users?since=' + randomOffset;
  });

var responseStream = requestStream
  .flatMap(function (requestUrl) {
    return Rx.Observable.fromPromise($.ajax({url: requestUrl}));
  });

var suggestion1Stream = close1ClickStream.startWith('startup click')
  .combineLatest(responseStream,
    function(click, listUsers) {
      return listUsers[Math.floor(Math.random()*listUsers.length)];
    }
  )
  .merge(
    refreshClickStream.map(function(){ return null; })
  )
  .startWith(null);
// and the same logic for suggestion2Stream and suggestion3Stream

suggestion1Stream.subscribe(function(suggestion) {
  if (suggestion === null) {
    // hide the first suggestion DOM element
  }
  else {
    // show the first suggestion DOM element
    // and render the data
  }
});
```

```
}  
});
```

You can see this working example at <http://jsfiddle.net/staltz/8jFJH/48/>

That piece of code is small but dense: it features management of multiple events with proper separation of concerns, and even caching of responses. The functional style made the code look more declarative than imperative: we are not giving a sequence of instructions to execute, we are just **telling what something is** by defining relationships between streams. For instance, with Rx we told the computer that *suggestionStream* **is** the 'close 1' stream combined with one user from the latest response, besides being *null* when a refresh happens or program startup happened.

Notice also the impressive absence of control flow elements such as `if`, `for`, `while`, and the typical callback-based control flow that you expect from a JavaScript application. You can even get rid of the `if` and `else` in the `subscribe()` above by using `filter()` if you want (I'll leave the implementation details to you as an exercise). In Rx, we have stream functions such as `map`, `filter`, `scan`, `merge`, `combineLatest`, `startWith`, and many more to control the flow of an event-driven program. This toolset of functions gives you more power in less code.

## What comes next

If you think Rx\* will be your preferred library for Reactive Programming, take a while to get acquainted with the [big list of functions](#) for transforming, combining, and creating Observables. If you want to understand those functions in diagrams of streams, take a look at [RxJava's very useful documentation with marble diagrams](#). Whenever you get stuck trying to do something, draw those diagrams, think on them, look at the long list of functions, and think more. This workflow has been effective in my experience.

Once you start getting the hang of programming with Rx\*, it is absolutely required to understand the concept of [Cold vs Hot Observables](#). If you ignore this, it will come back and bite you brutally. You have been warned. Sharpen your skills further by learning real functional programming, and getting acquainted with issues such as side effects that affect Rx\*.

But Reactive Programming is not just Rx\*. There is [Bacon.js](#) which is intuitive to work with, without the quirks you sometimes encounter in Rx\*. The [Elm Language](#) lives in its own category: it's a Functional Reactive Programming **language** that compiles to JavaScript + HTML + CSS, and features a [time travelling debugger](#). Pretty awesome.

Rx works great for event-heavy frontends and apps. But it is not just a client-side thing, it works great also in the backend and close to databases. In fact, [RxJava is a key component for enabling server-side concurrency in Netflix's API](#). Rx is not a framework restricted to one specific type of application or language. It really is a paradigm that you can use when programming any event-driven software.

If this tutorial helped you, [tweet it forward](#).

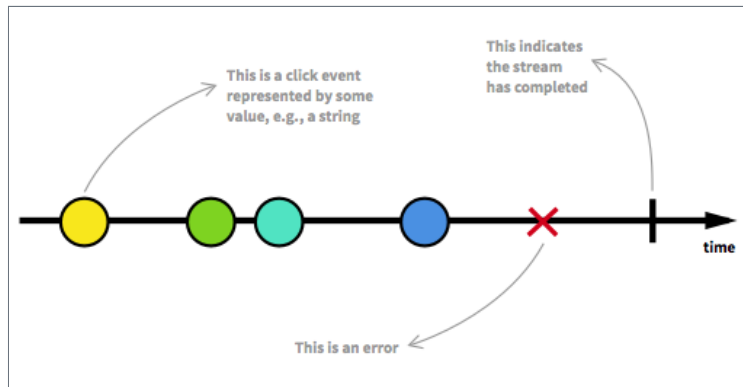
## Legal


© Andre Cesar de Souza Medeiros (alias "Andre Staltz"), 2014. Unauthorized use and/or duplication of this material without express and written permission from this site's author and/or owner is strictly prohibited. Excerpts and links may be used, provided that full and clear credit is given to Andre Medeiros and <http://andre.staltz.com> with appropriate and specific direction to the original content.

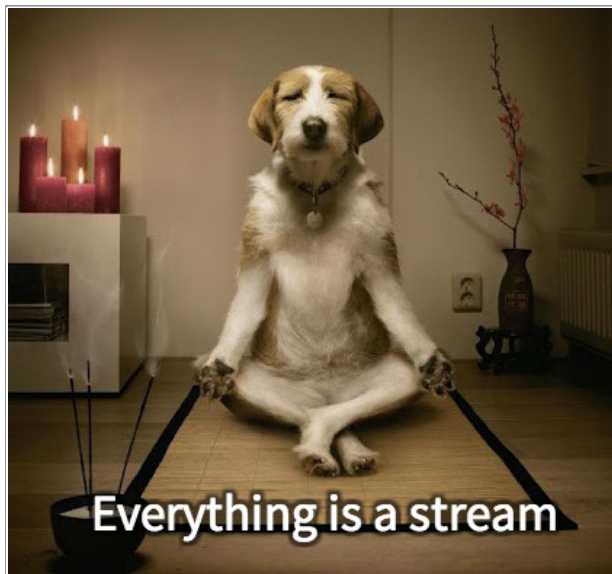


"Introduction to Reactive Programming you've been missing" by [Andre Staltz](#) is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](#).

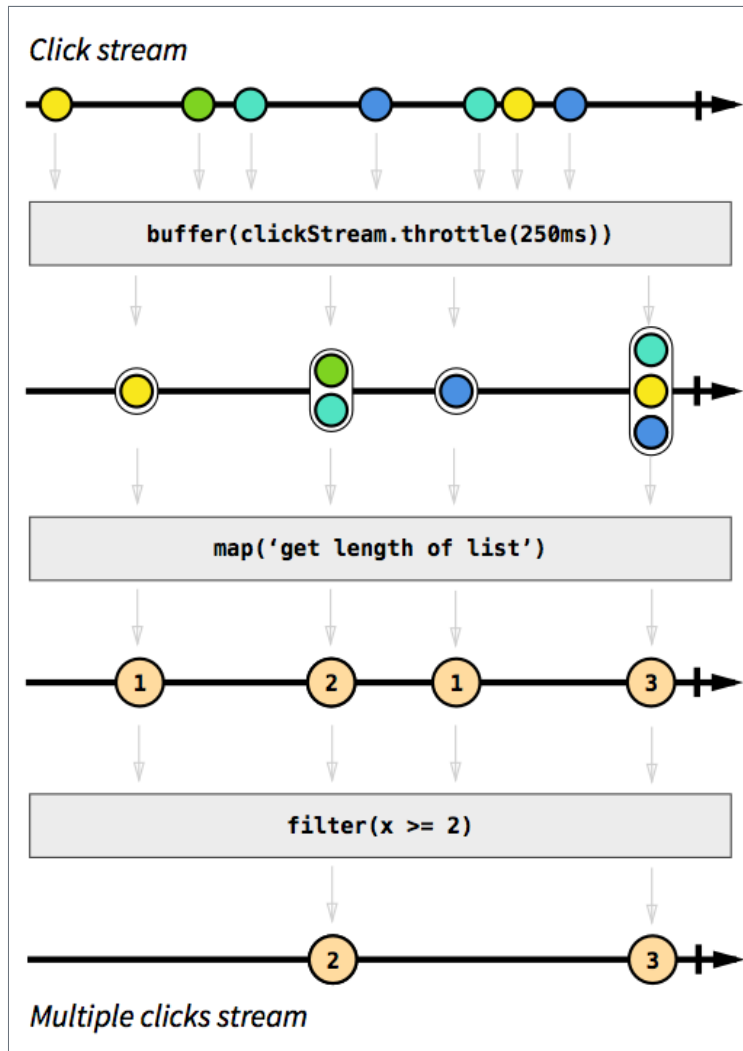
Based on a work at <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>.



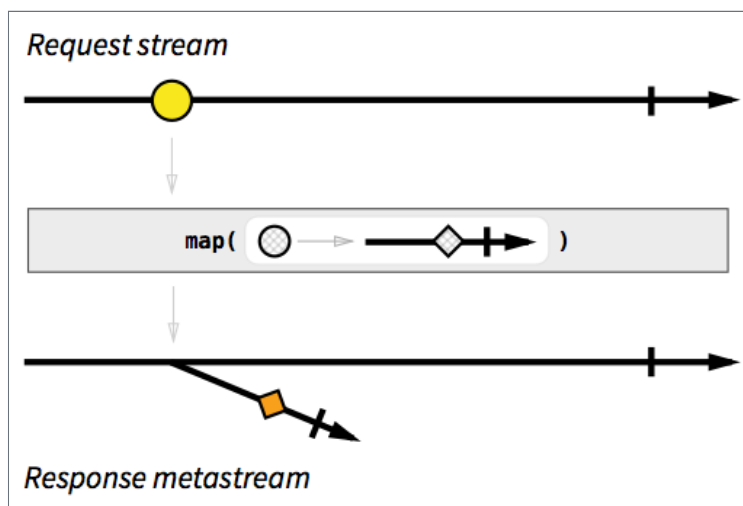
 [zmantra.jpg](#)



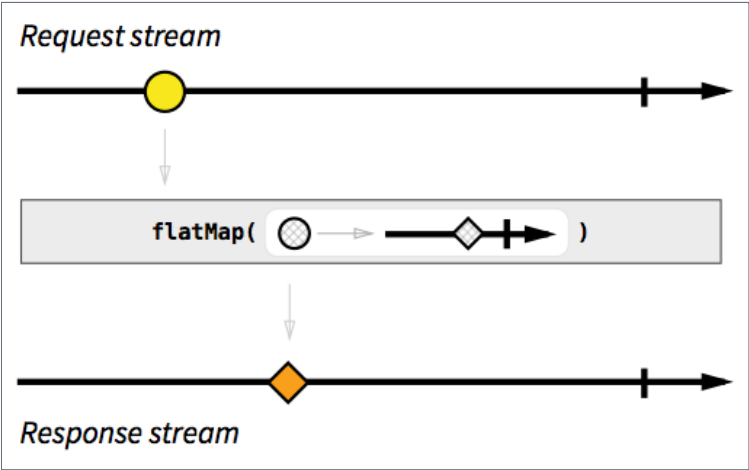
 [zmulticlickstream.png](#)



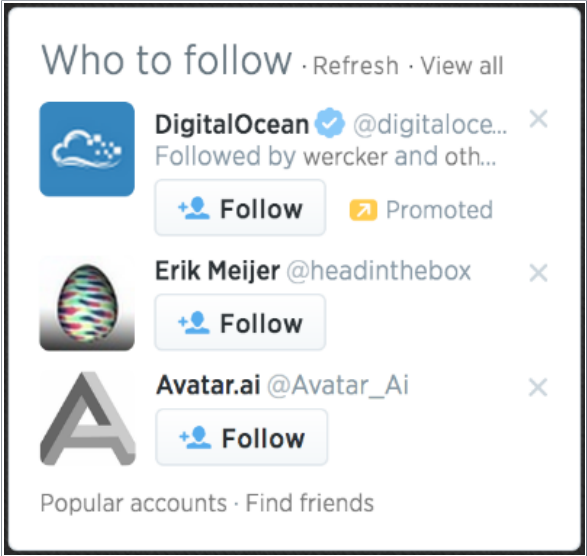
 [zresponsemetastream.png](#)



 [zresponsestream.png](#)



ztwitterbox.png



ryana commented on Jun 30, 2014

This is incredibly well-written. @staltz do you mind talking a little bit about your process for writing this, how long it took, etc... Very impressive stuff.



staltz commented on Jun 30, 2014

Owner

UPDATE: there's been a lot of confusion around the terms *Functional Reactive Programming* and *Reactive Programming* [1] [2].

Sorry, my bad. I guess this sort of confusion happens easily with new paradigms in computing. Replace all the occurrences of "FRP" with "RP" in the tutorial. Functional Reactive Programming is a variant of Reactive Programming that follows Functional Programming principles such as referential transparency, and seeks to be purely functional. Other people are better at explaining this than I am. [3] [4] [5]



trkrameshkumar commented on Jun 30, 2014



Very informative , well documented



se77en commented on Jun 30, 2014

I love this post



ghost commented on Jun 30, 2014

great read!



swanson commented on Jun 30, 2014

I've struggled to grok FRP in the past - I hear lots of smart people trumpeting it, but I haven't been able to wrap my head around it. This tutorial was great and I was following along right until it got to the part about modeling the suggestions as streams (<https://gist.github.com/staltz/868e7e9bc2a7b8c1f754#modellin...>).

That was where everything went haywire for me - my brain doesn't want to think about those UI elements as streams and I instantly tune out and think "this doesn't make sense". I'm curious if anyone else also got stuck at that same part, as well as if anyone has suggestions/ideas for breaking past this mental barrier? Is there a different way of solving the same problem (that might make more sense to me)?

I can see value in having a `Suggestion` emit some kind of event ("I need to be refreshed" or "My X was clicked") but I'm struggling to see how that connects to the network/API stream. Do they have to be so tightly linked?



staltz commented on Jun 30, 2014

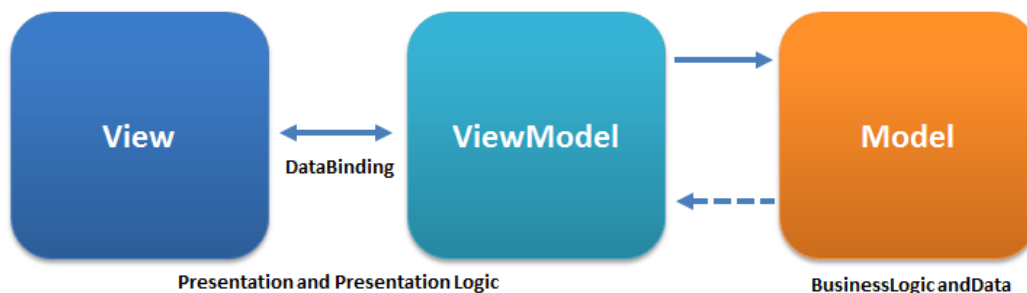
Owner

Hi @swanson, sorry for that section, I might have gone a bit too fast near the end.

Try not to think of the suggestion stream as "UI element stream". It is a data stream in the sense that everything it emits is simply data. Although, as you noticed, it is tightly linked to the close button click stream, and also tightly linked to the response stream. In a way, it lives between Model and View, if you think about an MVC framework.

In fact, suggestion stream is best understood as a ViewModel in the [MVVM architecture](#). MVVM happens to be a good fit for Rx\*. Quoting Wikipedia:

The view model of MVVM is a value converter meaning that the view model is responsible for exposing the data objects from the model in such a way that those objects are easily managed and consumed. In this respect, the view model is more model than view, and handles most if not all of the view's display logic.



Suggestion stream prepares data coming from the model (response stream) in such a way that the data it emits can be immediately consumed by the View (the subscriber which renders to the DOM). It frees the View from having to do any kind of logic.

If we had followed MVVM strictly in this example, we would need also a `close1ClickViewModelStream` just to detach the suggestion stream from `close1ClickStream`, and it would be defined as such: `close1ClickViewModelStream = close1ClickStream.startWith('startup click')`.

Programming with Rx\* doesn't require MVVM, but I've found it to be a good architecture.



swanson commented on Jun 30, 2014

Is there a concept of some independent thing emitting an event to hook into the stream? I guess I am more used to a traditional event bus, where a `SuggestionViewModel` might emit `SuggestionRemoved` event which would trigger the API stream to go get some new data. And the view model might want to subscribe to a `NewSuggestion` event to render the new data. My natural feeling is that a view/view model should not care about the internals of how the data was loaded, that I am using a stream, etc and the approach you outlined seems like this is all mashed together because all the streams must be merged/connected directly.



staltz commented on Jun 30, 2014

Owner

@swanson Nothing stops you from doing that kind of approach. For instance, you could do `suggestionRemovedStream = close1ClickStream.startWith('startup click')`.

However, one hint for "thinking in the right way": avoid saying that the `SuggestionRemoved` event would trigger the API stream. Think the other way around: the API stream is *triggered by* the `SuggestionRemoved`. It's an important distinction because Rx is "Push" while imperative is "Pull". <https://github.com/Reactive-Extensions/RxJS/tree/master/doc#rxjs-v22> It makes a difference because `SuggestionRemoved` will have no knowledge of the API stream. It's the API stream that can listen to ("subscribe") events from `SuggestionRemoved`.

The mashing together happens in the `ViewModel`, which is typically a class/module with a couple of streams, assuming two roles: Observer, and Observed. Events move in both directions (from Model to View and vice-versa) in the `ViewModel`. The benefit is that Model streams and View streams don't need to care about any other streams, they just do their thing by defining what they emit. The `ViewModel` glues it together.

View streams: `refreshClickStream`, `closeClickStream`

Model streams: `requestStream`, `responseStream`



staltz commented on Jun 30, 2014

Owner

@swanson That said, I recommend reading about [Subjects](#) which are streams that you can explicitly feed in new events, imperative-style:

```
subject.onNext("suggestion removed!");
```



grant commented on Jun 30, 2014



Love it.



queimadus commented on Jun 30, 2014

Great read! After fiddling around a little with the demo I got some unexpected behavior. Clicking the refresh button, makes not one, but three JSON calls to the service, each time. Why is that?

Seems like it is the behavior exemplified in the [Rx.Observable.prototype.publish\(\[selector\]\)](#) documentation.



ajhino commented on Jun 30, 2014

This is one of the best FRP breakdowns I've seen. Nice work!



staltz commented on Jun 30, 2014

Owner

Very good catch @queimadus! We were just bitten by the Cold vs Hot Observable problem, that I hoped wouldn't show up in this demo, but it did.

It happened because `responseStream` was a cold observable (observables are cold by default), and it had 3 observers `suggestion1Stream`, `suggestion2Stream`, and `suggestion3Stream`. Essentially, these 3 create their own internal response stream, and that's what causes the 3 API requests.

To fix it, we made `responseStream` become a hot observable:

```
var responseStream = requestStream
    .flatMap(function (requestUrl) {
        return Rx.Observable.fromPromise($.getJSON(requestUrl));
    })
    .publish().refCount();
```

Fixed here: <http://jsfiddle.net/staltz/8jFJH/59/>

As far as I've seen, this is the best example-based explanation of Cold vs Hot: <http://leecampbell.blogspot.co.uk/2010/08/rx-part-7-hot-and-cold-observables.html>

Other good references: [\[1\]](#) [\[2\]](#) [\[3\]](#)



**aeikenberry** commented on Jun 30, 2014

Thanks for doing this! Very well done.



**tieTYT** commented on Jun 30, 2014

I'm really confused by the appearance of 'startup click'. Is that a string with special meaning? I think you should explain that part because you lost me there.



**peterkhayes** commented on Jun 30, 2014

The 'startup click' is passed to a stream that takes the same action regardless of what the input is, so it really could be anything.

```
var requestStream = refreshClickStream.startWith('startup click')
    .map(function() {
        var randomOffset = Math.floor(Math.random()*500);
        return 'https://api.github.com/users?since=' + randomOffset;
    });
```

notice that the `requestStream` is mapped to the link to github, without reference to any input. This is different from the usual use of `map`, where the iterator function uses the input in some way, like `map(function(number) {return 2*number})`.

I guess he's using the phrase 'startup click' here to be more descriptive. You have to pass something (or else the stream won't emit anything), so why not use something that explains why you're doing it?



**paulirotta** commented on Jul 1, 2014

Excellent, thanks!



**prassu21** commented on Jul 1, 2014

Brilliant article.. Thanks for sharing..



**ledbutter** commented on Jul 1, 2014

So I'll bite on the exercise to avoid copy/pasting for the different suggestion streams, would that involve using the `repeat()` method?



**staltz** commented on Jul 1, 2014

Owner

Hi @ledbutter, actually `repeat()` doesn't help for that, and the DRY problem is already solved in the JSFiddle:  
<http://jsfiddle.net/staltz/8jFJH/48/>

```
function createSuggestionStream(closeClickStream) {
  return closeClickStream.startWith('startup click')
    .combineLatest(responseStream,
      function(click, listUsers) {
        return listUsers[Math.floor(Math.random()*listUsers.length)];
      }
    )
    .merge(
      refreshClickStream.map(function(){
        return null;
      })
    )
    .startWith(null);
}
```

`repeat()` just "echoes" forward some events that happened in the past. See [1] [2]



domenic commented on Jul 1, 2014

Observable is Promise++. In Rx you can easily convert a Promise to an Observable by doing `var stream = Rx.Observable.fromPromise(promise)`, so let's use that. The only difference is that Observables are not Promises/A+ compliant, but conceptually there is no clash. A Promise is simply an Observable with one single emitted value. FRP streams go beyond promises by allowing many returned values.

Array is value++. In JS you can easily convert a value to an Array by doing `var array = [value]`, so let's use that. The only difference is Arrays are not ECMAScript Basic Values compliant, but conceptually there is no clash. An Array is simply a value with one single entry. OOP arrays go beyond values by allowing many contained values.



ChubV commented on Jul 1, 2014

don't try to play this tabs

```
-----o-----o---->
-r-----r-----r---->
---R-----R-----R-->
---s-----N---s-----N-s-->
---q-----N---q-----N-q-->
---t-----N---t-----N-t-->
```



spion commented on Jul 1, 2014

This is pretty nice, and shows how FRP is at least as powerful as Promises. So if you believe the Promises hype, keep an eye on what FRP is capable of.

This is probably incorrect. Because promises deal with single events, they're also able to memoize and accept new subscribers after the event. AFAIK Observables are incapable of retaining values: to get the same functionality one would have to use both `repeat()`, and `.take(1)` before each attached consumer.

Promises abstract immutable, asynchronously available values. FRP abstracts values that change over time. They're different abstractions and as such useful in different circumstances.



staltz commented on Jul 1, 2014

Owner

Array is value++. In JS you can easily convert a value to an Array by doing `var array = [value]`, so let's use that. The only difference is Arrays are not ECMAScript Basic Values compliant, but conceptually there is no clash. An Array is simply a value with one single entry. OOP arrays go beyond values by allowing many contained values.

I actually genuinely laughed with that kind of humor. 😊

This is probably incorrect. Because promises deal with single events, they're also able to memoize and accept new subscribers after the event. AFAIK Observables are incapable of retaining values: to get the same functionality one would have to use both `repeat()`, and `.take(1)` before each attached consumer.

Actually it was correct. You don't need `repeat() + take(1)`. Cold Observables can "memoize". You can also achieve similar behavior with `BehaviorSubject` or `ReplaySubject`.



spion commented on Jul 1, 2014

@staltz I see. In that case, a much bigger distinction should be made between these in the API, I think. (Perhaps that's one of the reasons why it comes back and bites you brutally? :P)

So promises are cold single-value observables. Fair enough :) Does RxJS have something similar to bluebird's long stack traces?

`BehaviorSubject` seems close, but it looks mutable to me - a promise's value cannot be changed once its resolved or rejected.



staltz commented on Jul 1, 2014

Owner

@staltz I see. In that case, a much bigger distinction should be made between these in the API, I think.

Yes, Rx is extensively documented, but seems to have many quirks and gotchas. Bacon.js [was born out of dissatisfaction of RxJS](#) because of these quirks and other reasons.

So promises are cold single-value observables. Fair enough :) Does RxJS have something similar to bluebird's long stack traces?

I'm not sure as I'm not familiar with Bluebird. I have less experience with RxJS than I do with RxJava ("RxJVM"). In JVM you get stack traces. :/

`BehaviorSubject` seems close, but it looks mutable to me - a promise's value cannot be changed once its resolved or rejected.

I could elaborate on this later on, maybe it deserves its own well-thought blog post. My comparison Observables / Promises in this tutorial was playful. Conceptually they are very close and Observables go beyond a bit. That is not to say "everywhere you have a Promise you can put an Observable", or "throw away your Promises library". I'm actually a hard core [fan of Promises](#) in JS. My point in the article is that you can wrap a Promise in an Observable and continue working from that point onwards.



spion commented on Jul 1, 2014

@staltz, i did an experiment and it turns out that promises are not cold single-value observables after all.

Promises execute the side-effect (e.g. `xhr.get` or `console.log` in my example) once and memoize the value.

Cold observables will execute side-effects multiple times rather than memoize: <http://plnkr.co/edit/pB7dURuPUBMo0kTWy2lY?p=preview>

My comment was not really criticism - I'm genuinely interested in combining the capabilities of Promises and Observables and I'm trying to figure out what would be the best way to do it. (we replaced callbacks with Promises; looking into replacing the imperative node streams and event emitters with Observables).

It's quite cool that Rx observables can wrap Promises - now if only Promises could also consume observables :)

e.g. if we assume a connection can multiplex multiple subscriptions:

```
// connectionPromise is the memoized connection
var eventStream = connectionPromise
  .toObservable(c => c.getSubscription("someEvent"));
// eventStream is now a proper observable and can be "piped" back to a client
```

that would be really cool. I suppose this will do:

```
var eventStream = Rx.Observable.fromPromise(connectionPromise)
  .flatMap(c => conn.getSubscription("someEvent"));
```



**staltz** commented on Jul 1, 2014

Owner

What about BehaviorSubject, ReplaySubject, `defer()` ? If you really figured out Observables absolutely cannot do what Promises can, then that would be quite good knowledge to share. I'd be interested in seeing how that would affect, e.g., `Rx.Observable.fromPromise()` .



**spion** commented on Jul 1, 2014

No, I haven't figured it out yet. I'm still learning about Observables (there is a lot to learn, really). Will definitely share a comparison as soon as I figure things out :)



**jmandel** commented on Jul 1, 2014

Lovely Tutorial!

Quick correction: In the snippet below, which motivates `fromPromise` but isn't used directly in your final demo app, the use of `.then` is an error:

```
var responseStream = Rx.Observable.create(function (observer) {  
  jQuery.getJSON(requestUrl)  
    .then(function(response) { observer.onNext(response); })  
    .fail(function(jqXHR, status, error) { observer.onError(error); })  
    .done(function() { observer.onCompleted(); });  
});
```

The issue is that `.then()` returns a new Promise, so the `.fail` and `.done` handlers are attached to the wrong object.



**warmwaffles** commented on Jul 1, 2014

@staltz, this should honestly be in a git repo so that PRs can be done and discussions could sprawl out in the issues section.



**staltz** commented on Jul 2, 2014

Owner

Thanks @jmandel, it should have been `.done( ).fail( ).always( )` . I haven't touched jQuery in a while...



**alexblanquart** commented on Jul 2, 2014

Indeed, very impressive tutorial, well explained with no bullshit!



**karianna** commented on Jul 2, 2014

Wonderful post - really allows folks new to FRP to just get it.



**ledbutter** commented on Jul 2, 2014

Thanks for the explanation @staltz regarding the suggestions and the great tutorial. I just got hung up on the "exercise" wording and didn't notice that you solved it later without explicitly calling it out (or at least I didn't read it as such).



**bbirand** commented on Jul 3, 2014

Is there a way to also send data or parameters "upstream"?

I'm trying to understand the case where the stream is a continuously changing quantity, like "current\_time" or a sensor reading. I want to sample it at regular intervals, creating a regular "stream" of values. But for instance the sampling rate of this can depend on who the consumer is at the very end of the chain. Do you know if there are any notions in FRP that deals with these types of ideas?



**staltz** commented on Jul 6, 2014

Owner

@**bbirand**, if you want to handle streams of continuous (mathematical continuous) nature, then you're looking for "Behaviors" in [Classical Functional Reactive Programming](#). As far as I know, Rx and other reactive libraries don't support continuous 'behaviors'.



**bbirand** commented on Jul 9, 2014

@**staltz** Thanks for the pointer (and the excellent write-up!). I will definitely look into it!



**lenage** commented on Jul 10, 2014

Great read 👍



**juanfezero** commented on Jul 11, 2014

Thank's for this great introduction.  
From this I can learn more.



**gekosurf** commented on Jul 16, 2014

Great introduction.  
I'm just looking at the Java Play framework for Reactive Programming.  
Your article certainly helps  
Thanks



**rajeshpv** commented on Jul 17, 2014

wish I could send infinite thanks in a stream to this NICE , BEST and CLEAR write up,  
(btw, stream supports infinite size) I loved the real example of twitter on jsFiddle, it is almost like putting logic in where logic gets executed for those events.

Thanks again Stalz, brilliant work



**sebhitas** commented on Jul 23, 2014

This was very informative. Thanks for this introduction to Reactive Programming!



**jmig** commented on Jul 23, 2014

Have played with ReactiveCocoa in the past, mostly to handle my KVO stuff in a block based API, but nothing more.  
I just joined a new project where they use Rx and fell completely lost. I cannot tell you how much your introduction is simple yet incredibly clear and powerful. The code base looks totally logic and clear to me now. Thanks!



**razmig** commented on Jul 28, 2014

Great tutorial, thanks!



**kwbr** commented on Aug 7, 2014

I like your graphics. How are they done?



**staltz** commented on Aug 8, 2014

Owner

Hi @kwbr, you're not the first asking this. I made them with Sketch for OS X. Actually now I'm preparing an interactive diagrams webapp, still under construction: <http://staltz.github.io/rxmarbles/>  
It'll be part of the upcoming ReactiveX documentation site: [ReactiveX/reactivex.github.io#1](https://github.com/ReactiveX/reactivex.github.io#1)



**ondrek** commented on Aug 11, 2014

👍 This is really nice reading, dude! .. I was trying to find something about FRP, but most of articles are really heavy and kinda difficult to read. If you are more interested in FRP check also this tutorial (<http://conal.net/fran/tutorial.htm>) and this answer (<http://stackoverflow.com/questions/1028250/what-is-functional-reactive-programming/1028642#1028642>) 🐉



**vamdt** commented on Aug 13, 2014

This help me a lot, thanks!



**qesuto** commented on Aug 21, 2014

Best introduction on the subject I've read so far. Great work!



**Hardwareman** commented on Oct 7, 2014

We use our formative years learning how best to navigate in a parallel-concurrent world. Programmers must learn—and keep on learning—how to navigate in the strictly linear-sequential world of computation and make the LS-ness of the machine essentially transparent to the user. A tough task and the effort too often fails.

I would like to share some of my work on a fundamentally parallel-concurrent method of building process controllers. It has a special logic and operators that are much better for creating systems that react to real-world events.

My solution is hardware, as my UN suggests. A key realization is that, lo and behold, we think in P-C mode all the time. We just don't have the P-C machines w/w to build our reactive systems and must do it with the ubiquitous LS systems that are fundamentally unsuitable for those jobs.



**jainprash** commented on Oct 8, 2014

Great share... Just came across it today.....Hopefully a quick start for me in FRP



**tommix000** commented on Oct 16, 2014

An eye-opener and a great Tutorial, thank you very much.



**benjycui** commented on Oct 18, 2014

@staltz Excuse me. I spent several days in translating this tutorial into Chinese. So, could you add an reference to it? For not all of Chinese programmers are good at English and RP is a difficult concept to understand.

Chinese edition: <https://github.com/benjycui/intro-rx-chinese-edition>





**simpzan** commented on Oct 19, 2014

The 3 suggestions are independently selected from the response, so it is possible that the 3 suggestions shown are the same one. I think it is an issue.

Great tutorial anyway.



**kzar** commented on Nov 2, 2014

Wouldn't it make more sense to provide a GitHub user stream by flattening each response array down? Something like this:

```
var userStream = Rx.Observable.range(0)
  .flatMap(function(i) {
    return Rx.Observable.fromPromise($.getJSON('https://api.github.com/users?since=' + i));
  })
  .flatMap(function(response) {
    return Rx.Observable.fromArray(response);
  });
```

(I've never looked at RxJS and this code is untested but hopefully conveys what I mean. I realise this user stream would be in order which isn't what we want as well.)



**LeeCampbell** commented on Nov 5, 2014

Hey @bbirand, if I understand you correctly, I think you have something that you observe changes in values at an *unknown cadence*, and you want to map that to an observable sequence that published values at a *known cadence*. An example might be a GPS sensor that pushes events at various intervals (maybe as the device moves) and you want to only get information at most every second, and only if the value has actually changed. In this case Rx is great at dealing with this.

```
gpsValues.bufferWithTime(1000)
  .where(function(gpsPoints){ return gpsPoints.length >0; })
  .subscribe(...);
```



**jxblum** commented on Nov 6, 2014

+1



**snowmantw** commented on Nov 11, 2014

I've learned some how Yampa, a Haskell FRP framework, handles futures things like other FRP tools do, and it's good to see JavaScript community adopt FRP approach to avoid nasty things like callback hell, especially when we need to handle some complex UI use cases. However, maybe I'm not enough familiar with these JS tools and their approaches, from them I haven't found things like switches in Yampa to handle the discrete cases to determinate when does the future handling logic would be replaced with a different one.

For example, if we have a single page RPG game with different stages, these stages need to handle very different events (conditions from story lines, maybe). If we want to transfer from stage to stage, we need to change all handlers. In Yampa I think (according to what I've learned) we can use switchers to switch our logic circuit partially or entirely when the conditions get satisfied; hence we keep the same FRP structure without complex `if...else`, or get help from external world that is not so FRP. I don't know how this could go smooth with FRP libraries without like, since tools to `reduce` future streams seems not including this part. Or, maybe beyond this tutorial, the RxJS already provides a solution?



**paprikka** commented on Nov 17, 2014

+1 Great job, shame I didn't read it couple years ago, this would've saved me couple days of banging my head on the keyboard. Sometimes I feel these articles are written using a fork of "Postmodernist Essay Generator", when it comes to the language.

Created a Bacon.js fork in case someone was interested in a comparison: <http://jsfiddle.net/paprikka/dmgnhmxo/3/>



justinjmoses commented on Nov 23, 2014

Just a small note @staltz, might be worth renaming `startup click` to `startup-click` to prevent confusion with the jQuery convention to separate multiple events with a space.



justinjmoses commented on Nov 23, 2014

@paprikka appreciate the complementary example in bacon. The juxtaposition of the two is also quite informative on illustrating the differences between Hot & Cold observables lazy evaluation in bacon.js (slated for removal in 0.8) - the RxJS example does one ajax call for each subscriber where as the bacon example only lazily performs one ajax call for all subscribers.



mbernats commented on Nov 28, 2014

FRP is programming with asynchronous data streams.

No it's not. FRP is programming with Behaviours (pure continuous-time dependent values); please read the links you yourself provide, especially what Conal Elliott writes (since he's the one who coined the term).

If you only have discrete stuff, i.e. events, streams, or whatever you want to call it, you have stream-based/data-flow/reactive programming. Just because it's now fashionable to call everything that's not standard imperative programming FRP, doesn't mean we should do it. So please don't.



staltz commented on Dec 4, 2014

Owner

@mbernats, relax, calm down, and scroll up to the 2nd comment in this gist.



itoshkov commented on Dec 21, 2014

@staltz, great article! Thank you.

I haven't tried the code, but it looks like there's nothing stopping the three suggestion to have duplicates. Or have I missed the piece where this is solved?



gautampriya commented on Dec 23, 2014

S.U.P.E.R and thank you very much.



mcraiveiro commented on Dec 30, 2014

Great article, thanks a lot.



AshkanImmortal commented on Jan 5, 2015

Awesome! That's it.



dpc commented on Jan 17, 2015

Thank you.



**stejskal** commented on Jan 21, 2015

fantastic article!



**hydrotik** commented on Feb 21, 2015

This is great. Thank you for taking the time to write this!



**pm771** commented on Feb 23, 2015

Very helpful. Thank you.



**christoph-daehne** commented on Feb 26, 2015

This article is awesome, tanks :).



**geoand** commented on Feb 27, 2015

Wonderful!



**simonewebdesign** commented on Mar 2, 2015

@**snowmantw** I think maybe the RxJS solution to this is `Rx.Observable.case` : <https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/case.md>



**jaredly** commented on Mar 4, 2015

This was exactly what I was looking for -- very thorough!  
To help get my mind around things, I made an interactive visualization of the streams, and thought I'd share it.  
[reactive streams](#)  
So far it's pretty rough, but I hope to make it into a full debugger.



**crisberrios** commented on Mar 6, 2015

Hello, I read the 4 lines example and I think it can be refactored a little for less DRY and more DOT. Does this impact the performance by adding an extra step to the chain? btw, great intro, I'm mostly new to JS and totally new to reactive programming, thanks!

<http://jsfiddle.net/a6a4vaLc/2/>



**noushi** commented on Mar 10, 2015

@**staltz** beautiful article, thank you!



**johnwook** commented on Mar 11, 2015

Great article! Thank you @**staltz**! May I translate this article to Korean and share it? Of course, I will show the origin of the article.



**PhiLhoSoft** commented on Mar 13, 2015

Indeed a good article, with a good, concrete example to back it up.  
What is funny is that after reading, and looking again, I think I can now understand the mysterious description of `Rx.Observable.prototype.flatMapLates` you quote at the start... 😊

I am interested to see what you will do with `cycle.js`. Looks very promising.

Note: I take the "everything is a stream" mantra with a grain of salt: on one hand, it looks like an interesting point of view, at least challenging the way I use to think (coming from imperative languages, C and ASM, then from OOP languages, C++ and Java, going to functional side, Scala and Ceylon). On the other hand, it feels a bit like "everything is a nail when you have a golden hammer"... 😊



**wongcain** commented on Mar 23, 2015

Great explanation. Thank you!



**ewintory** commented on Mar 26, 2015

Awesome intro, thanks a lot!



**timjacobi** commented on Apr 1, 2015

Great article! Slowly starting to get what this is all about.

Unfortunately:

```
var requestStream = Rx.Observable.just('https://api.github.com/users');
```

Does not work for me. It yields:

Uncaught TypeError: undefined is not a function

What's the correct syntax please? I couldn't find anything in the docs.



**vire** commented on Apr 4, 2015

@timjacobi what do you try to achieve?

<http://xgrommx.github.io/rx-book/content/helpers/just.html>



**timjacobi** commented on Apr 4, 2015

I've taken that code from the post. Just want to start with a string stream.



**yeshr** commented on Apr 16, 2015

Great read. Helped me wrap my head around Rx. Thank you!



**elcioabrahao** commented on Apr 19, 2015

Very clear and effective tutorial ! Thanks !



**kenaiX** commented on Apr 21, 2015

very very thanks for your article

but i meet some trouble

how rxjava work in touch event ?

this is my code :

```
public class RxJavaActivity extends Activity{

    @Override
    public boolean onTouchEvent(MotionEvent event) {
        Observable.just(event)
            .buffer(2)
            .map(List::size)
            .subscribe(integer -> Log.e("test", "" + integer));
        return true;
    }
}
```

it not work ...

every time it return "1"



**chibicode** commented on Apr 21, 2015

Extremely well written :)



**hangtwenty** commented on Apr 22, 2015

This post may have changed my professional life. Thank you thank you thank you.



**markchagers** commented on Apr 30, 2015

Very nice tutorial, thank you. However the example code on <http://jsfiddle.net/staltz/8jFJH/48/> suddenly stopped working when I was half-way through the tutorial.

This happened after I tried modifying the fiddle to experiment with your code. Going back to the original fiddle doesn't make a difference though.

What's up?

Edit: solved:

```
{
  "message": "API rate limit exceeded for (my IP#). (But here's the good news: Authenticated requests get a higher rate limit. Check out the documentation for more details.)",
  "documentation_url": "https://developer.github.com/v3/#rate-limiting"
}
```

I leave my comment here in case anyone else runs into this limit.



**gyetvan-andras** commented on Apr 30, 2015

Thank you!



**MohamadAtieh** commented on Apr 30, 2015

props to you!



**owenoak** commented on Apr 30, 2015

Awesome introduction, thank you very much!



**miguelmota** commented on May 1, 2015

good stuff man.



**SimplGy** commented on May 1, 2015

This helped me write my first non-trivial thing in FRP/bacon: <https://gist.github.com/SimplGy/d6362369ac4ebf27f3ec> Thank you!



**milaneuo** commented on May 10, 2015

@staltz. What's the best practice to integrate with reactive and interactive(such as alert for user input validation)? Thanks very much.



**oldboyxx** commented on May 21, 2015

Holy cow! OP you're awesome.



**mbernat** commented on May 26, 2015

@staltz sorry if I came across as too aggressive. This is an amazing post, it just bothered me a lot you claimed it was about FRP next to a link to actual FRP. Kudos to you for fixing this, now the article is perfect. Best introduction to streams hands down.



**tjwudi** commented on May 27, 2015

Great work! While I do appreciate your work, here is the buggy part: `suggestion1Stream`, `suggestion2Stream` and `suggestion3Stream` will produce duplicates. :)



**krinkere** commented on May 27, 2015

Very nice work. Thank you



**benoptimus** commented on May 29, 2015

thanks for share!! excellent tuto.



**ravindranathakila** commented on Jun 1, 2015

Just, THANK YOU! 🙏



**malukenho** commented on Jun 3, 2015

Great article, Thank you 😊



**lenage** commented on Jun 10, 2015

Awesome article for RP, Thank you



**hueitan** commented on Jun 11, 2015

Awesome explanation, Thank you 👍 Thanks for sharing.



**billyct** commented on Jun 13, 2015

very nice~!!



**atolmachev** commented on Jun 15, 2015

@staltz, thanks for great post.  
What tool do you use for drawing diagrams?



**revelfire** commented on Jun 19, 2015

Oh. Finally. I think I get it. Thank you.



**cyounkins** commented on Jun 19, 2015

If you use streams for processing ajax requests and responses, won't you run into race conditions if the first URL takes longer to respond than the second?



**fpipita** commented on Jun 24, 2015

Thank you, awesome content!



**Maamar** commented on Jun 24, 2015

Thank you, good Introduction



**linfongi** commented on Jun 25, 2015

great work!!!



**willrster** commented on Jun 30, 2015

Great article! The concepts are communicated very well.  
Your fiddle is making triple ajax calls for every refresh, but still...great read.



**daniel-williams** commented on Jun 30, 2015

Really enjoyed this. Thank you!



**the-fine** commented on Jul 5, 2015

Awesome article!



**wonjungk** commented on Jul 8, 2015

Super well written introduction. Thank you a lot.



**viniciusbo** commented on Jul 21, 2015

Very nice! Thank you!



**GermanoGiudici** commented on Jul 23, 2015

great read! thank you!



**killbirds** commented on Jul 23, 2015

Fun! thank you!



**daden** commented on Jul 26, 2015

Great article, very nicely explained and demonstrated. I wish more articles like this existed to define basic terms and provide concrete examples without assuming the reader knows the full subject already.

I have one minor question about something in which I am not an expert so apply salt appropriately. Where you describe asynchronous events which call a method on success, error or completed you said this is precisely the Observer Design Pattern. But, when I pulled out my GOF reference, it looks like the Observer design pattern has the subject invoking a "notify()" method to the observers which then themselves invoke a `getState()` method on the subject to get the current state: state is not passed to the method called on the observer. The result is the same but it seems the pattern is a bit different.



**gajus** commented on Jul 28, 2015

┆ a version of `map()` than "flattens" a metastream, by emitting  
"that" not "than".



**gajus** commented on Jul 28, 2015

┆ One way to fix this is by moving the `startWith()` close to the `refreshClickStream`, to essentially "emulate" a refresh click on startup.

Should be a separate paragraph. It would read better if you said:

┆ You can "emulate" a refresh click on a startup using `startWith()`. The value that you pass to `startWith()` is irrelevant, because it is not used to make the request. Here `startWith()` is used to "trigger a new event":

@staltz Lack of the second part is what got me confused. I could not make out which stream/condition is responsible for recognising "startup click" input.



**gajus** commented on Jul 28, 2015

@staltz You have mentioned that:

┆ it is absolutely required to understand the concept of Cold vs Hot Observables. If you ignore this, it will come back and bite you brutally.

Understanding the difference between cold vs hot observables is not complex. Please provide an example of where it can "come back and bite [me] brutally". I cannot think of an example where I would implement cold observable mistaken for hot observable and vice-versa.





**alonecuzzo** commented on Aug 5, 2015

awesome! 👍



**Vintharas** commented on Aug 7, 2015

Beautiful article Andre! 👍



**chrisber** commented on Aug 8, 2015



**overture8** commented on Aug 15, 2015

I've read two books, one just painted the big picture, while the other dived into how to use the Reactive library.

@staltz What books do you recommend?

ps. 👍



**LamCiuLoeng** commented on Aug 19, 2015

Good post



**SeungJinCho** commented on Aug 19, 2015

Good post. two thumbs up.



**aikomastboom** commented on Aug 28, 2015

great article, however I don't understand why `suggestion1Stream` at the end needs two `startwith` calls?



**Shubhampatni86** commented on Sep 2, 2015

I'm very new here and trying to adapt terminology first...but I like representation here <http://staltz.com/rxmarbles/>



**sevko** commented on Sep 2, 2015

Amazing writeup! There are a couple of parallels to Node.js's streams here.



**chenkie** commented on Sep 4, 2015

Very, very good.



**xieweizhi** commented on Sep 5, 2015

Great post, thanks.



**xclouder** commented on Sep 7, 2015

great article! thank you!



**aronwoost** commented on Sep 8, 2015

Great read. Thanks!



**Qaaj** commented on Sep 10, 2015

Wonderful article. Probably the most intense IT-related article I've read in the last couple of years - It really challenged my deeply embedded way of thinking about handling of events. The drawing of diagrams made it so much easier. Thanks so much for putting this together!



**fedek6** commented on Sep 14, 2015

Dude! Awesome tutorial! I finally (almost) understand what's what with reactive js.



**oracleaide** commented on Sep 15, 2015

Nice doc. But this is what database developers have been doing for years (data pipelines / data transformations using views with function-based or virtual columns / functional programming with dynamic sql / pipelined functions/ messaging-queues-as-tables). Finally front-end people are getting on board.



**sshlyk** commented on Sep 18, 2015

Reminds me scala and more specifically scalding :)

Nice tutorial.



**veggimonk** commented on Sep 27, 2015

Good job making things simple for others!

Thank you.



**StarpTech** commented on Sep 30, 2015

Very well written. Thank you!



**playground** commented on Oct 5, 2015

This is awesome. Thanks. Can you talk a bit about how to deal with responsiveness with rxjs?



**kennetpostigo** commented on Oct 6, 2015

@staltz You should make this a Gitbook!



**kennetpostigo** commented on Oct 7, 2015

@staltz check out <https://www.gitbook.com> to publish it as a GitBook. You can copy and paste your Markdown into GitBook but separate it into organized sections or chapters, people can download epub of it and read it offline. Hope you do it :) !



**jfbosch** commented on Oct 18, 2015

Brilliantly written! Well done.



**expalmer** commented on Nov 5, 2015

Thanks for sharing! I'm starting with RP and it's awesome.



**jemshit** commented on Nov 10, 2015

Would be better if could understand javascript...



**bigfish** commented on Nov 11, 2015

Anyone else get 401 on the github API call? This due to cross-domain XHR, getJSON() call. I could get it fine if my browser which was logged into GH.. just copied to local file.



**IgorVieira** commented on Nov 21, 2015

Great !!!  
And thanks for Sharing!



**konurbaev-e** commented on Nov 29, 2015

The best article about Reactive Programming! Thanks!



**jhbsk** commented on Nov 30, 2015

+1



**nicegoing** commented on Nov 30, 2015

+1



**lightSky** commented on Dec 2, 2015

Great tutorial , Thanks,I am studying Rx recently, and I have translated it to chinese : [translate-introduction-to-reactive](#) ,hope more chinese developers can read this awesome article and benefit it, Thanks again !



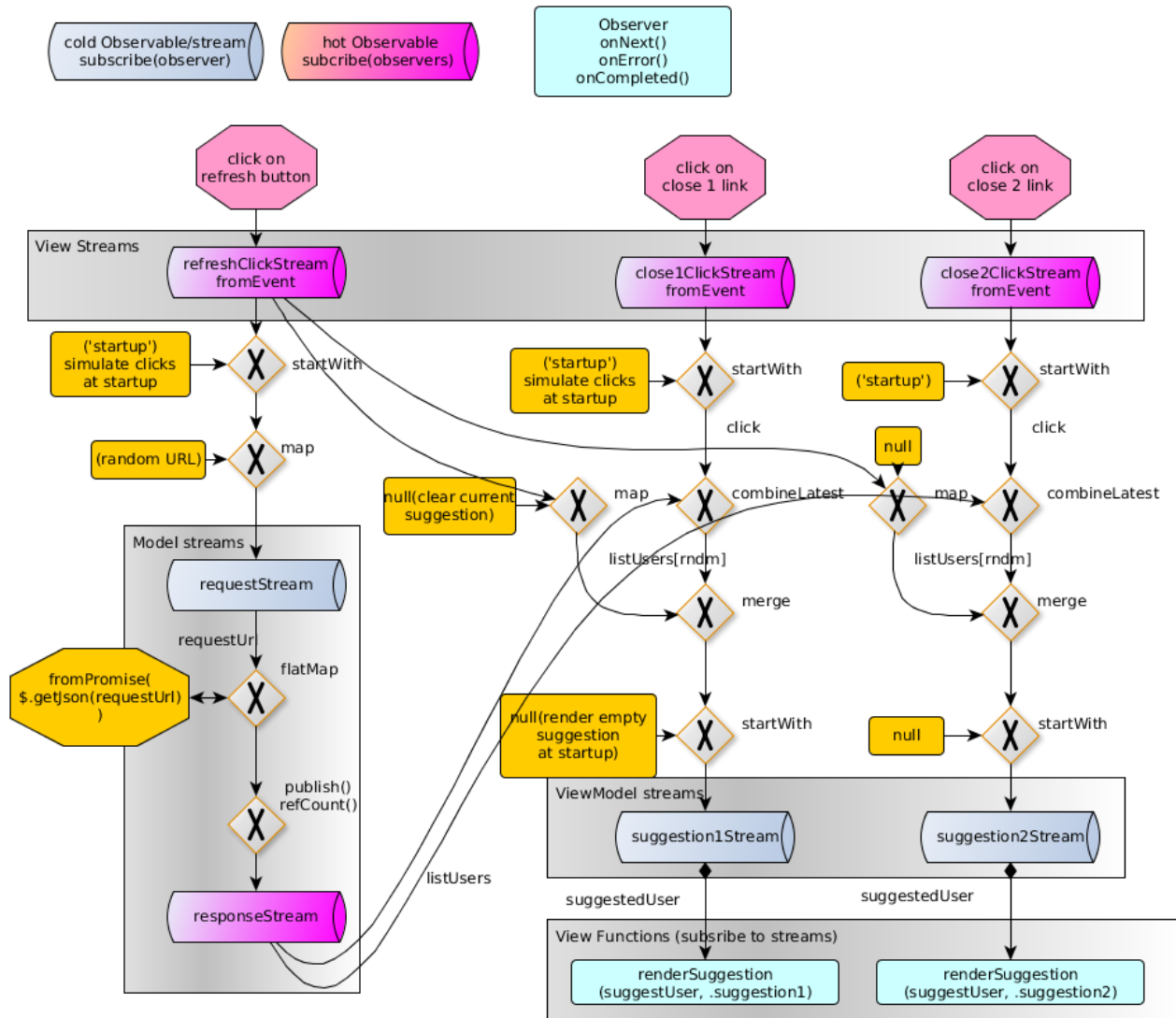
**SteffenSorensen** commented on Dec 9, 2015

You made it very understandable, thanks!

**stephan271** commented on Dec 13, 2015



Great tutorial - thanks. I tried to visualize the overall stream logic for me, see appended drawing. May be its helpful for others as well...



mujiatong commented on Dec 13, 2015

+1



NathanLi commented on Dec 15, 2015

+1

How to draw these pictures?



ianha commented on Dec 17, 2015

Not sure why this:

```
var responseStream = requestStream
  .flatMap(function(requestUrl) {
    return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
  });
```

is not written like this, without the `flatMap` :

```
var responseStream = requestStream
  .map(function(requestUrl) {
    return jQuery.getJSON(requestUrl);
  });
```



**zldrq** commented on Dec 23, 2015

too bad examples are java. but anyway, nice article



**haithngnbak** commented on Dec 27, 2015

thanks, 🍷



**frontend-3** commented on Dec 29, 2015

Thanks! Nice article and thanks stephan271 for the visual stream logic helps so much!



**stephan271** commented on Jan 4, 2016

The drawing was done "manually" using the program <https://www.yworks.com/products/yed>. The source file is here <https://www.dropbox.com/s/st5pf8ehmez7mt2/WhoToFollowSuggestionBox.graphml?dl=0>



**junjiah** commented on Jan 7, 2016

Great article! I use the same logic and implemented [a command line toy](#) with Node.js and [RxJS 5 \(beta\)](#) for fun :)



**darekzak** commented on Jan 28, 2016

Why do we need to merge `close1ClickStream` with `refreshClickStream.map(function(){ return null; })`?  
Why do we need to listen for null in `suggestion1Stream.subscribe`? From what i understand if some new value is emitted by `suggestion1Stream` we know that first we have to remove some html data and after that insert some new html.

```
var suggestion1Stream = close1ClickStream.startWith('startup click') .combineLatest(responseStream, function(click, listUsers) { return listUsers[Math.floor(Math.random()*listUsers.length)]; } ) .merge( refreshClickStream.map(function(){ return null; }) ) .startWith(null);
```



**ignusin** commented on Feb 3, 2016

Wow, that article has blown my brain! Thank you!!



**gamebox** commented on Feb 4, 2016

Like everything I've ever read of yours, well done! Thanks for all you do, [@staltz](#)



**djindjic** commented on Feb 14, 2016

By the end of this tutorial we will have produced real functioning code, while knowing why we did each thing

ufff, a lot of grateful comments, @staltz, but I'm still frustrated by not understanding the magic how did you move startupWith and change the url string with 'startup click'? but it's just a second reading... 😊



**djindjic** commented on Feb 15, 2016

[DAY 2]: after 24 hours fully concentrated to catch it I've made a minimal progress in understanding this article. it looks like there need to be listed some pre requested materials to pickup before.



**eyakcn** commented on Feb 17, 2016

Great work! Great article! Thanks a lot ^\_^



**MatrixHero** commented on Feb 25, 2016

aaaaasome!



**adriantawse** commented on Feb 27, 2016

There is one toolkit you seem to have missed. Have a look at [ConnectiveLogic.co.uk](http://ConnectiveLogic.co.uk). I first used this some eight years ago and found it just intuitively obvious. It maintains the strict separation of infrastructure (when and where some code is executed) and the actual transforms themselves. You need absolutely no programming skills in any language to understand the dependency graph. It helps if you have done some electronics because the dependency graph is specified graphically and has a passing resemblance to a electronic circuit.



**trungdq88** commented on Feb 27, 2016

@staltz: Although I find this article very helpful, I don't like the way you write/organize your code. Your fiddle is not very readable, not many people can understand it without reading this article.

So I've been looking through and made a fork here: <http://jsfiddle.net/trungdq88/qzsLzskz/4/>. I tried to restructure the code in MVVM architecture, comment and DRY it up, I also remove the hard-coded 3 items limit.

If you have time please take a look and feedback, thank you!



**deathmood** commented on Mar 2, 2016

First 4-line example (double click stream) somehow not working with rx.js v3.1.2 and higher (with v3.1.1 and lower it works). Does anybody know why?



**blackhawk389** commented on Mar 3, 2016

Very informative!



**juszzz** commented on Mar 10, 2016

Very helpful, thanks!



**r3wt** commented on Mar 16, 2016

i fail to see the usefulness of these patterns.



**JaimeBeneytez** commented on Mar 22, 2016

Thanks a lot ! very nice work and super helpful contribution



**miguelmota** commented on Mar 25, 2016

Super cool, a whole different way of thinking. Got me interested in FRP.



**beepilin** commented on Apr 3, 2016

greaaaaaaat! Thanks!



**davidgish760** commented on Apr 12, 2016

Fantastic article, thanks!

Is there some advantage to treating a constant string as a stream, or were you just demonstrating that it can be done? Also, since events occur in time and are "pushed", the order in which streams and observers are created would seem to matter, at least from a conceptual standpoint. Take that constant string, for example. Presumably it "emits" its value at launch (or compile time)? When observers subscribe to it, does it re-emit the value? Isn't that essentially a "pull"?



**davidgish760** commented on Apr 13, 2016

Just realized **@staltz** hasn't commented for well over a year. For anyone who cares, the answer to my question above is that a constant string is an example of a "cold" sequence that pushes a private (to the new subscriber only), unique copy of its value, along with an end-of-stream, each time it is subscribed to.



**yyscamper** commented on Apr 22, 2016

Very helpful!! many thanks



**danday74** commented on May 2, 2016 • edited ▼

The final stream diagram and code to prove it ...

<https://embed.plnkr.co/mqR9jE/preview>

Many thanks

(includes the Hot Observable fix provided in a comment above BUT that is not in the tutorial itself)



**lucianoks** commented on May 3, 2016

Thanks **@staltz**, that's indeed the best guide I've found so far (seached a lot!!) to get the FRP core concepts! =)



**HilmiDEV** commented on May 12, 2016

Thanks, great article



**csrcondeiro** commented on May 13, 2016

Thanks for the article!



**mocovenwitch** commented on May 13, 2016

Have read a lot of articles talking about RX, this one is the best ever. Thanks!



**afattahi54** commented on May 16, 2016

Thanks for the article!



**cadam11** commented on May 21, 2016

Thanks for writing this article @staltz. I found a tiny word error and fixed in a fork for you:

<https://gist.github.com/cadam11/fcf0827019558a83936b1ca478247870>

I'd submit a pull request, but that's not a thing for gists.



**scorfishyu** commented on May 27, 2016

Very helpful, thanks



**chawthet** commented on May 28, 2016

Very nice review, thx so much



**channgo2203** commented on May 30, 2016

Thanks, however there is an introduction about general reactive programming and/or synchronous programming, in which the very basic constructor is stream. In fact, the two paradigms were designed to program reactive systems (synchronous data flow graph used in logic board, electronic, simulink models).

<http://www-sop.inria.fr/mimosa/rp/generalPresentation/index.html>



**kotojo** commented on Jun 24, 2016

So I had the same question as @darekzak, but I think I figured it out. Someone correct me if I'm wrong. The use of the line

```
.merge(  
    refreshClickStream.map(function(){  
        return null;  
    })  
)
```

in the `createSuggestionStream` is to hide the html elements while waiting on data to return from the api. When refresh is clicked it will cause both the `$.getJSON` call to be fired as well as the `null` to be returned, but since null is an instant response it causes the users to be cleared out while waiting on new api data?



**Xunnamius** commented on Jun 26, 2016

... wow. Excellent. Just excellent. Thank you.



**roydellclarke** commented on Jul 11, 2016 • edited ▼



I was new to Javascript programming 1 year ago when I read this post. I did not know how great this post was then but now I do. Great Post!  
Side Note: Learning functional javascript and generators before tackling RFP, would make Rx. 100 times easier to grasp.



**liorma** commented on Jul 14, 2016

It is by far, the best Reactive programming tutorial i have read till now, it is really the first one that made me change the way of thinking.  
thank you



**shwetakhimpur** commented on Jul 27, 2016

Thank You so much. I have read only a part of this yet, however I can definitely say, after seeing sudden and random mentions of Rx in tutorials, I really needed to understand the concept and topic from an architect point of view. Really appreciate your work on doing this.



**zckevin** commented on Aug 4, 2016

Excellent write-up!



**fernandoacorreia** commented on Aug 6, 2016 • edited ▼

Excellent tutorial, but it lacks a clean-slate starting point. Using the dependencies at the provided JSFiddle, the very first line of code in the tutorial (`var requestStream = Rx.Observable.just('https://api.github.com/users');`) will produce the error `Rx.Observable.just is not a function`.

I worked around that by updating the dependencies. [This commit](#) contains working code for the very first step in the tutorial (Request and response). It also has instructions about how to avoid the rate-limiting GitHub API errors.

Here's my final version of the tutorial code: <https://github.com/fernandoacorreia/introRx>



**ebegoli** commented on Aug 15, 2016

What a fantastic tutorial. Thank you so much.

Here is my favorite line:

Library documentations often don't help when you're trying to understand some function. I mean, honestly, look at this:

```
Rx.Observable.prototype.flatMapLatest(selector, [thisArg])
```

Projects each element of an observable sequence into a new sequence of observable sequences by incorporating the element's index and then transforms an observable sequence of observable sequences into an observable sequence producing values > only from the most recent observable sequence.

Holy cow.

and of course:

So let's cut the bu..s..t.



**logavanig** commented on Aug 16, 2016

hi welcome to this blog.really you have post an informative blog.thank you for sharing this blog.it will be really helpful to many peoples.  
[c programming training](#)



**logavanig** commented on Aug 24, 2016

hi welcome to this blog.really you have post an informative blog.it will be really helpful to many peoples.thank you for sharing this blog.  
[c programming training](#)



**eduardnanu** commented on Aug 25, 2016

Excellent write-up! Thank you!



**kevguy** commented on Sep 7, 2016

Am I drawing the marble diagram correctly?

```
refreshClickStream: -'o'-----o-----o-----o-----o----->
requestStream:    --r-----r-----r-----r-----r----->
responseStream:   ---R-----R-----R-----R-----R----->
close1ClickStream: -----'c'-----c-----c-----c----->
close2ClickStream: -----'c'-----c-----c-----c----->
close3ClickStream: -----'c'-----c-----c-----c----->
suggestion1Stream: --N-----s1-----N-----s2-----N-----s3-----s3'-----s3'-----N-----s4----->
suggestion2Stream: --N-----q1-----N-----q2-----N-----q3-----q2'-----N-----q4----->
suggestion3Stream: --N-----t1-----N-----t2-----N-----t3-----t3'-----N-----t4----->
```



**john-bonachon** commented on Sep 14, 2016 • edited ▼

Hey this is great but I got dissapointed when splitting each suggestion making variables suggestion1Stream, suggestion2Stream, suggestion3Stream. I mean, if we had 10 suggestions would you have done the same approach?

I'm starting with Rx, and the tutorial starts really really good. However, this last part of the code looks like stateful old code!

```
...
.merge(
  refreshClickStream.map(function(){ return null; })
)
.startWith(null);
// and the same logic for suggestion2Stream and suggestion3Stream

suggestion1Stream.subscribe(function(suggestion) {
  if (suggestion === null) {
    // hide the first suggestion DOM element
  }
  else {
    // show the first suggestion DOM element
    // and render the data
  }
});
```



**damon-wang** commented on Sep 26, 2016

I love this article!



**alwayrun** commented on Sep 27, 2016

+1



**margielm** commented on Oct 3, 2016

Hey Andre,  
Thank you for this excellent tutorial! That helped a lot.  
Frankly, it is so excellent that some guy decided to make a presentation out of this tutorial <https://www.youtube.com/watch?v=1abiJ9VBsDc>. He is even using the same comments as you do (like "holy cow" for the rxJs documentation).  
I hope that he got your approval to do this, but I think it is worth checking.



**U007D** commented on Oct 12, 2016

You're probably tired of hearing this (nah, you're not!), but this is a fantastic article. Exactly what I needed to onboard myself to a new programming paradigm. Thank you!



**cshalinid** commented on Oct 17, 2016

Thanks it is a very good explanation for a newbie



**oshalygin** commented on Oct 17, 2016

Great read, thanks for sharing man!



**grundyoso** commented on Oct 29, 2016



**peternoordijk** commented on Nov 1, 2016

Hi, great read! I have one small request: can you please replace the occurrences of `observable.just` with `observable.of` in the document? The `just` method is deprecated in newer versions of RxJS. Thanks!



**palcalde** commented on Nov 3, 2016

This article es amazing, event after two years it's still the best explanation I've found of Reactive Programming concepts. Thank you!



**lawkai** commented on Nov 6, 2016

This is amazing!. Thanks for this, now I have a basic understanding of what Rx\*/Reactive programming is about!



**ksco** commented on Nov 8, 2016



**rodrigm** commented on Nov 9, 2016

Very nice article! Thanks



**whimsycwd** commented on Nov 17, 2016

Great post! Thanks.



**bbaia** commented on Nov 24, 2016 • edited ▼

Thanks! Great introduction! You can start writing a 'Reactive programming for dummies' book :)

A question about the example: how can you avoid a same user to be suggested twice ?

```
refreshClickStream: -----o-----o---->
  requestStream:  -r-----r-----r---->
  responseStream:  ---R-----R-----R-->
  suggestion1Stream: ---s-----N---s-----N-s-->
  suggestion2Stream: ---q-----N---q-----N-q-->
  suggestion3Stream: ---s-----N---s-----N-s--> // same than suggestion1
```



**tusharbihani** commented on Nov 28, 2016

Very well written. Thanks for the post.



**muaazsalagar** commented on Dec 6, 2016

Nice Article !  
Thanks



**rohmanhakim** commented on Dec 10, 2016 • edited ▼

Hey @staltz thanks for making this great article. This article gave me enlightenment.

I have a plan to made blog post to explain Reactive Programming too to my android engineer peer who have difficulties reading fully-english article using my native language (Indonesia).

So i'd like to ask you, can i have a permission to quote your definition about Reactive Programming (*Reactive programming is programming with asynchronous data streams*) and the way you use diagram to explain it?

It will be different though since i will explain it with android example instead of JS.



**UgglarpTom** commented on Dec 12, 2016

Well done, and thank you.



**jledere3-ford** commented on Dec 13, 2016

Very well written, thanks!



**jays1204** commented on Dec 13, 2016

@staltz very nice article. Thanks for your article. May I translate this document to korean and share it?



**wilburt** commented on Dec 19, 2016

Wow! Thanks man! Have been struggling to understand this.



**dkwong-ch** commented on Dec 22, 2016

Quick question . When refresh button is clicked , it actually trigger 2 streams , one set null to the suggestion1,2,3 .. , the other fetch randomly from the response data . So . it there any sequence promising btw these two actions (set null & fetch ) ?  
BTW , nice article and forgive my poor english .



NicoXcc commented on Jan 2

This may be the most user friendly explanation on RX ..... the Reactive Way :)  
Thanks a mill...



simarpreetsingharora commented on Jan 5

@ketanbhatt



karosLi commented on Jan 5

Very nice, I love this post



adrianmcli commented on Jan 6

I've refactored the JSBin example to ES6 syntax and collected some of the logic into helper functions to make the program logic more clear:

```
// Helper Functions -----
const makeRequestUrl = () => {
  const randomOffset = Math.floor(Math.random() * 500);
  return `https://api.github.com/users?since=${randomOffset}`;
};

const getRandomUser = users =>
  users[Math.floor(Math.random() * users.length)];

// Program Logic -----
const requestStream = refreshClickStream
  .startWith('startup click')
  .map(makeRequestUrl);

const responseStream = requestStream
  .flatMap(url => Rx.Observable.fromPromise($.getJSON(url)));

const createSuggestionStream = closeClickStream =>
  closeClickStream
    .startWith('startupclick')
    .combineLatest(responseStream, (click, users) => getRandomUser(users))
    .merge(refreshClickStream.map(() => null))
    .startWith(null);

const suggestion1Stream = createSuggestionStream(close1ClickStream);
const suggestion2Stream = createSuggestionStream(close2ClickStream);
const suggestion3Stream = createSuggestionStream(close3ClickStream);
```

The full source (pastable into JSBin) is here: <https://gist.github.com/adrianmcli/31732d17268c762132c7ca053f3d7cb2>



csc-bnguyen43 commented on Jan 9

Thanks, nice explanation!



jdjuan commented on Jan 9

Awesome!



**Tømmertom** commented on Jan 12

This is the best read in a long time, which made learn and laugh!

Just to let it sink in a bit more..read about hot/cold, when not to use Observables... how to design for Observables .....



**DevinXian** commented on Jan 16

I love the code, so dry and pretty!



**lalitjeevanrao** commented on Jan 16

Fantastic article. After days of searching articles on RX over the web, i found this and am so delighted.. ! Well done **@staltz**



**knee-cola** commented on Jan 23 • edited ▼

I was interested in how reactive programming could be put to use and found this really dense i focused text!  
Thanks for showing real-life example and not wasting time on theoretical and abstract lamenting!

As I understand it, the given example is a showcase of what can be done by using reactive programming and not a general guideline on how such this class of problems should be solved - frameworks are the ones which deal with that!

Anyways keep up the good work!



**ultimate1352** commented on Feb 1

Great article :) 👍



**naveenraj Kumar** commented on Feb 2

Well written..Great !!



**marioscience** commented on Feb 3 • edited ▼

Hello! I have a question. In this [jsfiddle](#) I do a map in stream and do another map to it. The map in the first stream is happening twice. I don't really understand why.

The way that I see it is this

1. I create an event stream.
2. I map a url from that event stream.
3. I subscribe to it to see what the stream is so far.
4. I map the stream again to see if the url has the number five. At this point I was assuming that the same stream was going to be used, but that's not the case and I don't understand why.
5. I subscribe to that last stream to see how the stream has changed.

Can someone please help me understand what's wrong with my logic?

Thanks for a great tutorial by the way :D



**acutmore** commented on Feb 3 • edited ▼

@marioscience Rx.Observable.fromEvent(refreshBtn, 'click') returns a shared observable (i.e only one eventListener will be added) but the observable returned from the other operators (.map etc) are not shared so will be repeated separately for each subscribe. The simple fix is to put .share() after the startWith('https://api.github.com/users')



1984weed commented on Feb 4

Good article. TY



marioscience commented on Feb 4

That explains it clearly. Thanks @acutmore



asmodehn commented on Feb 15

This doc might actually deserve its own repo...



ruwustark commented on Feb 17

Great article :) This may be the most user friendly explanation on RX ..... wallpaper the Reactive Way :) Thanks a mill...



e-cloud commented on Feb 18

@Saltz, it would be great to update the tutorial to fit with rxjs@5.x



e-cloud commented on Feb 19 • edited ▼

Here comes the examples in the document with rxjs@5.x

- double clicks <https://jsfiddle.net/clouda/jddwfmk/2/>
- the full user feed <http://jsfiddle.net/clouda/s9hsa86g/>



rudiedirkx commented on Mar 3

@Saltz <http://jsfiddle.net/staltz/8jFJH/59/> doesn't ever do a second Ajax request to fetch more users. It just keeps reusing the data from the initial response. Is that .publish()'s fault, or .refCount()'s?



amirpiri commented on Mar 27

I'm still don't understand why you change the URL with "startup click" in startWith() function?



horrido commented on Apr 4

I picked JavaScript and RxJS as the tools for this, for a reason: **JavaScript is the most familiar language out there at the moment...**

I'm not sure that's true. Have the majority of the world's 18 million programmers done web development? And even if they've done web development, are they necessarily familiar enough with JavaScript to understand this article? I'm not, even though I've been writing web applications for 10 years.

I know just enough JavaScript to interface with jQuery. That's it. I've never had to write more than a hundred lines of JavaScript for every web application (using PHP, Smalltalk, Python, Go, and *Amber Smalltalk for client-side applications*). In fact, I find the code snippets in this article a bit foreign.

Understandably, it's hard to find a language that everyone knows. I'm just not sure that JavaScript is the best choice. Maybe Java or Python?



**billrancho** commented on Apr 9

great post, it does transform generic things and vague stuff into something like touchable and imaginable where your mind can reach out.



**casamia918** commented on Apr 20 • edited ▼

Very nice tutorial! I've translated this tutorial to korean. <https://gist.github.com/casamia918/93b8db69beb9ee06b92a96b2a234d48e> . Hope to be a great starting point to learning Reactive Programming like me. I translated almost as same as the original text, but added or edited some parts to be more detailed description.



**hugadams** commented on Apr 25

Thank you, this was the first guide that clicked for me. In fact, I think the example about capturing double and triple clicks gave me some deep insights to functional vs. stateful programming that never fully sunk in (and I've been programming a while...)



**danielsGit** commented on May 6

Very Very Thanks. I've struggled for about 3 days to understand Observable and Action stream in React Native App development. But i was not able to find clear tutorial anywhere. Here! I found here. Thanks again! Best luck of you.



**herbae** commented on May 18

Excellent post! Thank you very much for taking the time to write this.



**bidu** commented on May 18

Super helpful, thank you!



**yangnianbing** commented on May 22

i have a question.

// and the same logic for suggestion2Stream and suggestion3Stream  
it's mean we should copy the code twice?  
if there are ten close menu?i must copy ten times?



**MatthewSickler** commented on May 23 • edited ▼

Great tutorial, really helped me get my head around how to think with reactive.

@**yangnianbing** He left it as an exercise for people to solve themselves. Though if you look at the fiddle he provides, he solves this using a helper function to keep the code DRY.

<http://jsfiddle.net/staltz/8jFJH/48/>



I personally only used 1 observable for all the x buttons and passed in the index from ngFor since i'm using Angular4. I'm disabling the buttons on refresh with a flag and using that to determine if all suggestions should get replaced. Here's my code at the time of writing this, can probably follow it without knowing angular since all the Observable function names are the same.

[controller](#)  
[html](#)



**Mrodent** commented on Jun 23

This looks fabulous. I just started running the code... from the start hereof. Your explanations are put in plain language, which inspires a lot of confidence.

However, as a lower intermediate JS/JQ person, I encountered one of those problems which almost defines the world we live in: having obtained a file helpfully called "Rx.min.js" and then trying your very first bit of Rx code, I got

```
/*
```

- i. just is not a function --> of
  - ii. onNext is not a function --> next
  - iii. onCompleted is not a function --> complete
  - iv. onError is not a function --> error
- ```
*/
```

You can see that I found the RxJS 5 "Rosetta Stone" at <https://github.com/ReactiveX/rxjs/blob/master/MIGRATION.md#subscription-dispose-is-now-unsubscribe> ... without losing too much of the few strands of hair I still have left.

Not only that, but your very first bit of RxJS code, starting "requestStream.subscribe(function(requestUrl) {..." actually has the closing ");" missing. Fine: if you can't work that out you almost certainly shouldn't be on this page. But, to make your page a little less challenging, do you think you might possibly update things to version 5?



**penzandrea** commented on Jul 5

Very helpful, thank a lot!



**moreJs** commented on Jul 6

good job.



**deenjohn** commented on Jul 7 • edited ▼

Awesome article..I read it and later watched the same egghead course..I loved this style of teaching "problem first then solution".You have explained all the "Why do we need this" in your course.Also using rxjs marbles while explaining "merge" , "startswith" , "map", "flatMap" etc which was very helpful.Fantastic !! Thank you !!



**konkola** commented on Jul 11

Hmm, confusing. Things can be done this way, but why? I can't see the benefit.



**alakra** commented on Jul 19

I think it's important to note that Elm is no longer considered a FRP language and that reference should probably be removed.

See <http://elm-lang.org/blog/farewell-to-frp>

**hosein2398** commented on Jul 20



In this snippet:

```
// execute the request
var responseStream = Rx.Observable.create(function (observer) {
  jQuery.getJSON(requestUrl)
    .done(function(response) { observer.onNext(response); })
    .fail(function(jqXHR, status, error) { observer.onError(error); })
    .always(function() { observer.onCompleted(); });
});

responseStream.subscribe(function(response) {
  // do something with the response
});
}
```

Seems you're messing a ) at the end.



**agdiaz** commented on Jul 25

Great article, thanks for it 🙌



**LiJinyao** commented on Aug 2

Great job!



**marcusalmeyda2017** commented on Aug 9

Really Awesome! Thanks for it.



**gamunax** commented on Aug 15 • edited ▼

thanks, great post.



**feitingshadow** commented on Aug 16

```
void onClick(MouseClick * Event) {
  static uint countedClicks = 0;
  countedClicks += (event.type != eventType.singleClick) 1 ? 0;
}
```

You lost me when you said it takes 4 lines in React to do something any good OOP programmer can in 2.



**semiversus** commented on Sep 14

The ONE essential introduction to reactive programming! Well done!



**sambeyene** commented on Oct 18

One of the best, well written tutorials I have seen.  
Thanks



**rima-smart** commented on Nov 3

2017 and still this is the best introduction to reactive programming on the web ...Thank you so much !



**deomsj** commented on Nov 4 • edited ▼

this has been incredibly helpful. thanks for sharing such a clear and well thought out intro :)



**iwejay** commented on Nov 6

*I think the world deserves a practical tutorial on how to think in Reactive*

:) Yes, Thank you!! 👍



**lwpro2** commented 24 days ago

The best Rx intro course i have found so far.



**PEZO19** commented 23 days ago

Ty for this! :)



**MarcSteven** commented 20 days ago

ok



**hendismail** commented 8 days ago • edited ▼

Been so long since I enjoyed a tutorial that much. Thank you!



**omadoyeabraham** commented 6 days ago

Thanks a lot for the great tutorial. So clear and helpful 👍