

Extracted from:

# Reactive Programming with RxJS

Untangle Your Asynchronous JavaScript Code

This PDF file contains pages extracted from *Reactive Programming with RxJS*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2015 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The  
Pragmatic  
Programmers

# Reactive Programming with RxJS

Untangle Your  
Asynchronous  
JavaScript Code



Sergi Mansilla

edited by Rebecca Gulick

# Reactive Programming with RxJS

Untangle Your Asynchronous JavaScript Code

Sergi Mansilla

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at <https://pragprog.com>.

The team that produced this book includes:

Rebecca Gulick (editor)  
Potomac Indexing, LLC (index)  
Candace Cunningham (copyedit)  
Dave Thomas (layout)  
Janet Furlow (producer)  
Ellie Callahan (support)

For sales, volume licensing, and support, please contact [support@pragprog.com](mailto:support@pragprog.com).

For international rights, please contact [rights@pragprog.com](mailto:rights@pragprog.com).

Copyright © 2015 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-13: 978-1-68050-129-2

Encoded using the finest acid-free high-entropy binary digits.

Book version: P2.0—August 2016

# The Reactive Way

---

The real world is pretty messy: events happen in random order, applications crash, and networks fail. Few applications are completely synchronous, and writing asynchronous code is necessary to keep applications responsive. Most of the time it's downright painful, but it really doesn't have to be.

Modern applications need super-fast responses and the ability to process data from different sources at the same time without missing a beat. Current techniques won't get us there because they don't scale—code becomes exponentially more complex as we add concurrency and application state. They get the job done only at the expense of a considerable mental load on the developer, and that leads to bugs and complexity in our code.

This chapter introduces you to reactive programming, a natural, easier way to think about asynchronous code. I'll show you how streams of events—which we call *Observables*—are a beautiful way to handle asynchronous code. Then we'll create an Observable and see how reactive thinking and RxJS dramatically improve on existing techniques and make you a happier, more productive programmer.

## What's Reactive?

Let's start by looking at a little reactive RxJS program. This program needs to retrieve data from different sources with the click of a button, and it has the following requirements:

- It must unify data from two different locations that use different JSON structures.
- The final result should not contain any duplicates.
- To avoid requesting data too many times, the user should not be able to click the button more than once per second.

Using RxJS, we would write something like this:

```
var button = document.getElementById('retrieveDataBtn');
var source1 = Rx.DOM.getJSON('/resource1').pluck('name');
var source2 = Rx.DOM.getJSON('/resource2').pluck('props', 'name');

function getResults(amount) {
  return source1.merge(source2)
    .pluck('names')
    .flatMap(function(array) { return Rx.Observable.from(array); })
    .distinct()
    .take(amount);
}

var clicks = Rx.Observable.fromEvent(button, 'click');
clicks.debounce(1000)
  .flatMap(getResults(5))
  .subscribe(
    function(value) { console.log('Received value', value); },
    function(err) { console.error(err); },
    function() { console.log('All values retrieved!'); }
  );
```

Don't worry about understanding what's going on here; let's focus on the 10,000-foot view for now. The first thing you see is that we express more with fewer lines of code. We accomplish this by using Observables.

An Observable represents a stream of data. Programs can be expressed largely as streams of data. In the preceding example, both remote sources are Observables, and so are the mouse clicks from the user. In fact, our program is essentially a single Observable made from a button's click event that we transform to get the results we want.

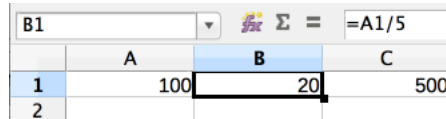
Reactive programming is expressive. Take, for instance, throttling mouse clicks in our example. Imagine how complex it would be to do that using callbacks or promises: we'd need to reset a timer every second and keep state of whether a second has passed since the last time the user clicked the button. It's a lot of complexity for so little functionality, and the code for it is not even related to your program's actual functionality. In bigger applications, these little complexities add up very quickly to make for a tangled code base.

With the reactive approach, we use the method `debounce` to throttle the stream of clicks. This ensures that there is at least a second between each click, and discards any clicks in between. We don't care how this happens internally; we just express *what* we want our code to do, not *how* to do it.

It gets much more interesting. Next you'll see how reactive programming can help us make our programs more efficient and expressive.

## Spreadsheets Are Reactive

Let's start by considering the quintessential example of a reactive system: the spreadsheet. We all have used them, but we rarely stop and think how shockingly intuitive they are. Let's say we have a value in cell A1 of the spreadsheet. We can then reference it in other cells in the spreadsheet, and whenever we change A1, every cell depending on A1 will automatically update its own value.



	A	B	C
1	100	20	500
2			

That behavior feels natural to us. We didn't have to tell the computer to update cells that depend on A1 or how to do it; these cells just *reacted* to the change. In a spreadsheet, we simply *declare* our problem, and we don't worry about how the computer calculates the results.

This is what reactive programming aims for. We declare relationships between players, and the program evolves as these entities change or come up with new values.

## The Mouse as a Stream of Values

To understand how to see events as streams of values, let's think of the program from the beginning of this chapter. There we used mouse clicks as an infinite sequence of events generated in real time as the user clicks. This is an idea by Erik Meijer—the inventor of RxJS—proposed in his paper “Your Mouse Is a Database.”<sup>1</sup>

In reactive programming, we see mouse clicks as a continuous stream of events that we can query and manipulate. Thinking of streams instead of isolated values opens up a whole new way to program, one in which we can manipulate entire sequences of values that haven't been created yet.

Let that thought sink in for a moment. This is different from what we're used to, which is having values stored somewhere such as a database or an array and waiting for them to be available before we use them. If they are not available yet (for instance, a network request), we wait for them and use them only when they become available.

1. <http://queue.acm.org/detail.cfm?id=2169076>



We can think of our streaming sequence as an array in which elements are separated by *time* instead of by memory. With either time or memory, we have sequences of elements:



Seeing your program as flowing sequences of data is key to understanding RxJS programming. It takes a bit of practice, but it is not hard. In fact, most data we use in any application can be expressed as a sequence. We'll look at sequences more in depth in [Chapter 2, Deep in the Sequence, on page ?](#).

## Querying the Sequence

Let's implement a simple version of that mouse stream using traditional event listeners in JavaScript. To log the x- and y-coordinates of mouse clicks, we could write something like this:

`ch1/thinking_sequences.js`

```
document.body.addEventListener('click', function(e) {
  console.log(e.clientX, e.clientY);
});
```

This code will print the x- and y-coordinates of every mouse click in order. The output looks like this:

```
< 252 183
  211 232
  153 323
  ...
```

Looks like a sequence, doesn't it? The problem, of course, is that manipulating events is not as easy as manipulating arrays. For example, if we want to change the preceding code so it logs only the first 10 clicks on the right side of the screen (quite a random goal, but bear with me here), we would write something like this:



```

var clicks = 0;
document.addEventListener('click', function registerClicks(e) {
  if (clicks < 10) {
    if (e.clientX > window.innerWidth / 2) {
      console.log(e.clientX, e.clientY);
      clicks += 1;
    }
  } else {
    document.removeEventListener('click', registerClicks);
  }
});

```

To meet our requirements, we introduced external state through a global variable `clicks` that counts clicks made so far. We also need to check for two different conditions and use nested conditional blocks. And when we're done, we have to tidy up and unregister the event to not leak memory.

### Side Effects and External State

If an action has impact outside of the scope where it happens, we call this a *side effect*. Changing a variable external to our function, printing to the console, or updating a value in a database are examples of side effects.

For example, changing the value of a variable that exists *inside* our function is safe. But if that variable is *outside* the scope of our function then other functions can change its value. That means our function is not in control anymore and it can't assume that external variable contains the value we expect. We'd need to track it and add checks to ensure its value is what we expect. At that point we'd be adding code that is not relevant to our program, making it more complex and error prone.

Although side effects are necessary to build any interesting program, we should strive for having as few as possible in our code. That's especially important in reactive programs, where we have many moving pieces that change over time. Throughout this book, we'll pursue an approach that avoids external state and side effects. In fact, in [Chapter 3, Building Concurrent Programs, on page ?](#), we'll build an entire video game with no side effects.

We managed to meet our easy requirements, but ended up with pretty complicated code for such a simple goal. It's difficult code to maintain and not obvious for a developer who looks at it for the first time. More importantly, we made it prone to develop subtle bugs in the future because we need to keep state.

All we want in that situation is to query the “database” of clicks. If we were dealing with a relational database, we'd use the declarative language SQL:

```

SELECT x, y FROM clicks LIMIT 10

```

What if we treated that stream of click events as a data source that can be queried and transformed? After all, it's no different from a database, one that emits values in real time. All we need is a data type that abstracts the concept for us.

Enter RxJS and its Observable data type:

```
Rx.Observable.fromEvent(document, 'click')
  .filter(function(c) { return c.clientX > window.innerWidth / 2; })
  .take(10)
  .subscribe(function(c) { console.log(c.clientX, c.clientY) })
```

This code does the same as the [code on page 5](#), and it reads like this:

Create an Observable of click events and filter out the clicks that happen on the left side of the screen. Then print the coordinates of only the first 10 clicks to the console as they happen.

Notice how the code is easy to read even if you're not familiar with it. Also, there's no need to create external variables to keep state, which makes the code self-contained and makes it harder to introduce bugs. There's no need to clean up after yourself either, so no chance of introducing memory leaks by forgetting about unregistering event handlers.

In the preceding code we created an Observable from a DOM event. An Observable provides us with a sequence or stream of events that we can manipulate as a whole instead of a single isolated event each time. Dealing with sequences gives us enormous power; we can merge, transform, or pass around Observables easily. We've turned events we can't get a handle on into a tangible data structure that's as easy to use as an array, but much more flexible.

In the next section we'll see the principles that make Observables such a great tool.