Extracted from:

Build Reactive Websites with RxJS

Master Observables and Wrangle Events

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Build Reactive Websites with RxJS

Master Observables and Wrangle Events

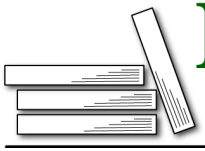
Randall Koutnik
edited by Brian MacDonald

Build Reactive Websites with RxJS

Master Observables and Wrangle Events

Randall Koutnik

The Pragmatic Bookshelf
Raleigh, North Carolina



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:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Brian MacDonald
Copy Editor: Paula Robertson
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-295-4
Book version: P1.0—December 2018

Loading with Progress Bar

That was the theory—now to build something practical. If you've ever implemented a loading bar that pulled together many different bits, you know just how irritating it can be to wrangle all those requests together. Common pre-observable asynchronous patterns plan for only one listener for each event. This results in ridiculous loading bar hacks, like adding a function call to every load event or monkey patching XMLHttpRequest. Using RxJS, our software never leaves our users waiting at 99% (not that I'm bitter).

In the following example, the progress bar represents mutiple requests. It's also possible to use the same strategies to represent a single large request by listening in to the progress event of an XMLHttpRequest.

Let's start out with 100 requests from the ajax constructor, all collected together in an array. Load up vanilla/managingAsync/mosaic.ts and code along.

vanilla/managingAsync/mosaic-complete.ts

```
let requests = [];
for (let x = 0; x < 10; x++) {
  for (let y = 0; y < 10; y++) {
    let endpoint =
      `http://localhost:3000/api/managingAsync/assets/coverpart-${x}-${y}.png`;
    let request$ = ajax({
      url: endpoint,
      responseType: 'blob'
    })
    .pipe(
      map(res => ({
        blob: res.response,
        x,
        y
      }))
    );
    requests.push(request$);
  }
}
```

At any time, there will always be a large number of requests to track, even in a singleplayer game. In *Reactive Game Development*, you'll build out an entire game based on a RxJS backbone. For now you'll just build the loading bar. (If it feels a bit strange to build the loading bar before the game, remember that this is a chance to catch unexpected bugs.) To track the overall state of the game load, all of these AJAX observables need to be combined into a single observable. There's a merge constructor that takes any number of

parameters (as long as they're all observables) and returns a single observable that will emit a value whenever any of the source observables emit. This example uses ES6's spread operator to transform the array into a series of individual parameters:

vanilla/managingAsync/mosaic-complete.ts

```
merge(...requests)
.subscribe(
  val => drawToPage(val),
  err => alert(err)
);
```

This single subscribe to the merged observables kicks off all of the requests in one fell swoop. Every request is centrally handled, and the user is notified when something goes wrong. Write out this example in mosaic.js and refresh the page.

If everything worked, the image comes together on the page as each individual request is loaded as shown in the screenshot on page 7.

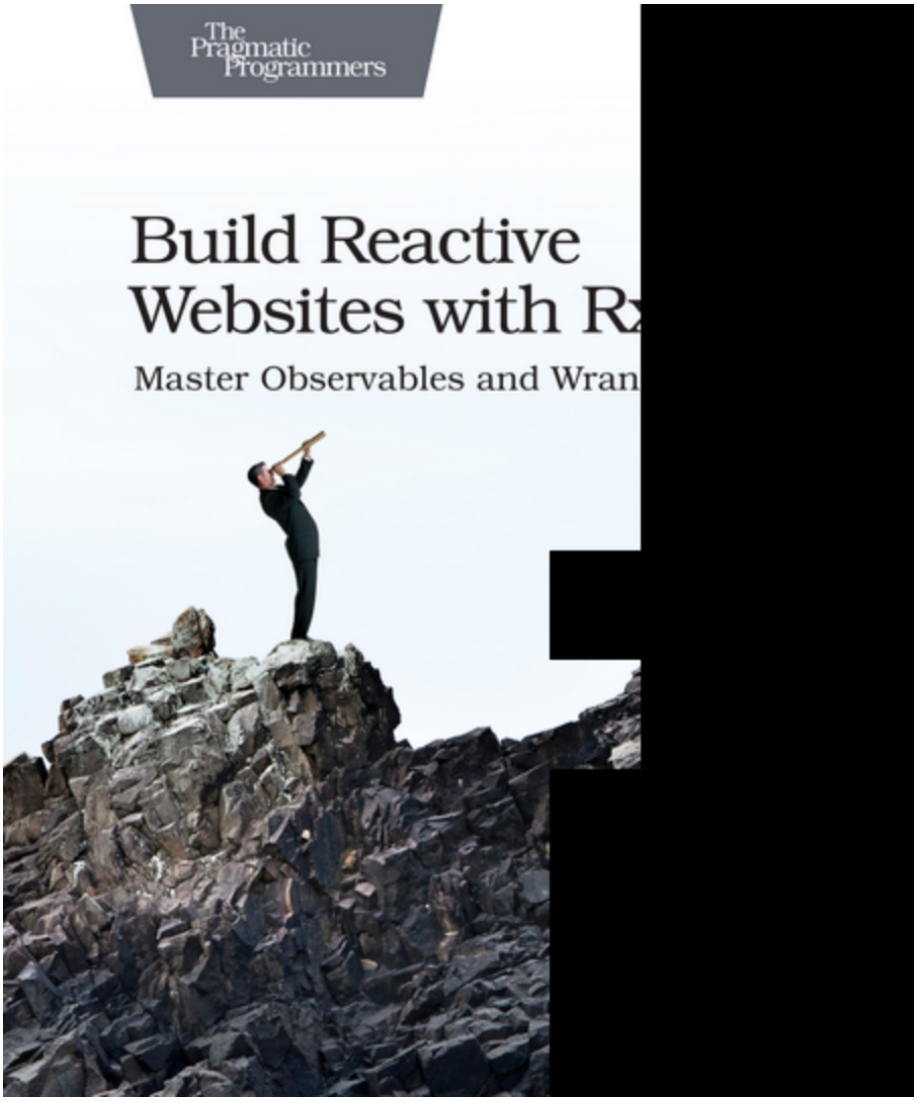
The Two Merges

In the above example, merge is used as a *constructor* (a way to create a new observable). However, it's also an *operator*:

```
let obsOne$ = interval(1000);
let obsTwo$ = interval(1500);

// Piping through the merge operator
obsOne$
.pipe(
  merge(obsTwo$)
)
.subscribe(console.log);

// Is the same as using the merge constructor

merge(obsOne$, obsTwo$)
.subscribe(console.log);
```

In cases where everything starts at the same time (like the loading bar), merge used as a constructor is simpler than the operator form. The operator form of merge comes in handy when you're in the middle of an observable chain and want to add in more data.

In this particular example, the tiles of the mosaic provide an abstract loading bar. In the auteur video game designer life, there are no such affordances. To build an award-winning game, the loading bar needs to be notified as each request completes. Previously, the reduce operator collected each item and only emitted the resulting collection after the original observable *completed*. Instead, we want the data collecting ability of reduce, but we want the operator to emit



the latest value on every new item. Digging deeper into the RxJS toolbox, you find scan. scan is an impatient reduce. Instead of politely waiting for the orignal stream to complete, scan blurts out the latest result on every event.

Here's scan in action, tracking how many requests have finished and emitting the total percentage on every event (arrayOfRequests is declared outside this snippet, see the loading-complete.ts file for the full details):

vanilla/managingAsync/loading-complete.ts

```
import { merge } from 'rxjs';
```

```
import { scan } from 'rxjs/operators';

merge(...arrayOfRequests)
.pipe(
  scan((prev) => prev + (100 / arrayOfRequests.length), 0)
)
.subscribe(percentDone => {
  progressBar.style.width = percentDone + '%';
  progressBar.innerText = Math.round(percentDone) + '%';
});
```

Like `reduce`, `scan` has two parameters: a reducer function and an initial value. `scan`'s function also takes two values—the current internal state and the latest item to be passed down the stream. This example throws away the latest value, because the loading bar doesn't care about *what* information came back, just that it *successfully* came back. `scan` then increments the internal counter by one unit (a unit is defined as 100 divided by the number of requests, so this results in the percent of the total that each request represents). If you've been lucky, you haven't hit any errors so far. Time to change that.