Extracted from:

Reactive Programming with RxJS 5

Untangle Your Asynchronous JavaScript Code

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Reactive Programming with RxJS 5

Untangle Your Asynchronous JavaScript Code

Sergi Mansilla

edited by Brian MacDonald

Reactive Programming with RxJS 5

Untangle Your Asynchronous JavaScript Code

Sergi Mansilla

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Indexing: Potomac Indexing, LLC
Copy Editor: Jasmine Kwityn
Layout: Gilson Graphics

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

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

Printed in the United States of America.
ISBN-13: 978-1-68050-247-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2018

CHAPTER 5

Bending Time with Schedulers

As soon as I discovered RxJS, I started using it in my projects. For a while I thought I knew how to use it effectively, but there was a nagging question: how do I know whether the operator I'm using is synchronous or asynchronous? In other words, when exactly do operators emit notifications? This seemed a crucial part of using RxJS correctly, but it felt a bit blurry to me.

The interval operator, I thought, is clearly asynchronous, so it must use something like setTimeout internally to emit items. But what if I'm using range? Does it emit asynchronously as well? Does it block the event loop? What about from? I was using these operators everywhere, but I didn't know much about their internal concurrency model.

Then I learned about Schedulers.

Schedulers are a powerful mechanism to precisely manage concurrency in your applications. They give you fine-grained control over how an Observable emits notifications by allowing you to change their concurrency model as you go. In this chapter you'll learn how to use Schedulers and apply them in common scenarios. We'll focus on testing, where Schedulers are especially useful, and you'll learn how to make your own Schedulers.

Using Schedulers

A Scheduler is a mechanism to "schedule" an action to happen in the future. Each operator in RxJS uses one Scheduler internally, selected to provide the best performance in the most likely scenario.

Let's see how we can change the Scheduler in operators and the consequences of doing so. First let's create an array with 1,000 integers in it:

```
const itemArray = [];
for (let i = 0; i < 1000; i++) {
  itemArray.push(i);
}
```

Then, we create an Observable from arr and force it to emit all the notifications by subscribing to it. In the code we also measure the amount of time it takes to emit all the notifications:

```
const timeStart = Date.now();
Observable.from(itemArray).subscribe(null, null, () => {
  console.log(`Total time: ${Date.now() - timeStart}ms`);
});
```

```
❮ "Total time: 1ms"
```

One millisecond—not bad! Unlike RxJS 4, RxJS 5 doesn't use any Scheduler by default, so this code processes all the notifications synchronously.

Now let's change the Scheduler to Rx.Scheduler.asap:

```
const timeStart = Date.now();
Observable.from(itemArray, Scheduler.asap).subscribe(null, null, () => {
  console.log(`Total time: ${Date.now() - timeStart}ms`);
});
```

```
❮ "Total time: 169ms"
```

Wow, our code runs more than a hundred times slower than with no Scheduler. That's because the asap Scheduler runs each notification asynchronously. We can verify this by adding a simple log statement after the subscription.

Using no Scheduler:

```
Rx.Observable.from(arr).subscribe( ... );
console.log('Hi there!');
```

```
❮ "Total time: 1ms"
  "Hi there!"
```

Using the asap Scheduler:

```
Rx.Observable.from(arr, Rx.Scheduler.asap).subscribe( ... );
console.log('Hi there!');
```

```
❮ "Hi there!"
  "Total time: 169ms"
```

When using no Scheduler, the console.log statement happens only when the Observable has emitted all of its notifications, because they happen synchronously. But when Rx.Scheduler.asap is used, console.log runs first, whereas our Observer's notifications run asynchronously, so they appear after the console.log statement.

So, Schedulers have a big impact on how our Observables work. In our case here, performance suffered from asynchronously processing a big, already-available array. But we can use Schedulers to improve performance. For example, we can switch the Scheduler on the fly before doing expensive operations on an Observable:

```
  Observable.from(itemArray)
    .groupBy(value => value % 2 === 0)
    .map(value => value.observeOn(Scheduler.asap))
➤   .map(groupedObservable => expensiveOperation(groupedObservable));
```

In the preceding code we group all the values in the array into two groups: even and uneven values. groupBy returns an Observable that emits an Observable for each group created. And here's the cool part: just before running an expensive operation on the items in each grouped Observable, we use observeOn to switch the Scheduler to the asap one, so that the expensive operation will be executed asynchronously, not blocking the event loop.

observeOn and subscribeOn

In the previous section, we used the observeOn operator to change the Scheduler in some Observables. observeOn and subscribeOn are instance operators that return a copy of the Observable instance, but that use the Scheduler we pass as a parameter.

observeOn takes a Scheduler and returns a new Observable that uses that Scheduler. It will make every next call run in the new Scheduler.

subscribeOn forces the subscription and un-subscription work (not the notifications) of an Observable to run on a particular Scheduler. Like observeOn, it accepts a Scheduler as a parameter. subscribeOn is useful when, for example, we're running in the browser and doing significant work in the subscribe call but we don't want to block the UI thread with it.

Basic Rx Schedulers

Let's look a bit more in depth at the Schedulers that we just used. The ones RxJS's operators use the most are asap and queue. There are other, more specialized Schedulers like the animationFrame scheduler, which we wll see later in the chapter.

The asap Scheduler

The asap Scheduler runs actions asynchronously. You can think of it as a rough equivalent of setTimeout with zero milliseconds delay that keeps the order in the sequence. It uses the most efficient asynchronous implementation available on the platform it runs (for example, process.nextTick in Node.js or setTimeout in the browser).

Let's take the previous example with range and make it run on the asap Scheduler. For this, we'll use the observeOn operator:

```
console.log("Before subscription");
Observable.range(1, 5)
  .do(value => {
    console.log("Processing value", value);
  })
  .observeOn(Scheduler.asap)
  .map(value => value * value)
  .subscribe(value => {
    console.log("Emitted", value);
  });
console.log("After subscription");
```

```
❮ Before subscription
Processing value 1
Processing value 2
Processing value 3
Processing value 4
Processing value 5
After subscription
Emitted 1
Emitted 4
Emitted 9
Emitted 16
Emitted 25
```

There are significant differences in the output this time. Our console.log statement runs immediately for every value, but we make the Observable run on the asap Scheduler, which yields each value asynchronously. That means our log statements in the do operator are processed before the squared values.

When to Use It

The asap Scheduler never blocks the event loop, so it's ideal for operations that involve time, like asynchronous requests. It can also be used in Observables that never complete, because it doesn't block the program while waiting for new notifications that may never happen.

The queue Scheduler

The queue Scheduler is synchronous like the immediate Scheduler. The difference is that if we use recursive operators, it enqueues the actions to execute instead of executing them right away. A recursive operator is an operator that itself schedules another operator. A good example is repeat. The repeat operator —if given no parameters—keeps repeating the previous Observable sequence in the chain indefinitely.

When to Use It

As a rule of thumb, the queue Scheduler should be used for large sequences and operations that involve recursive operators like repeat, and in general for iterations that contain nested operators.