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

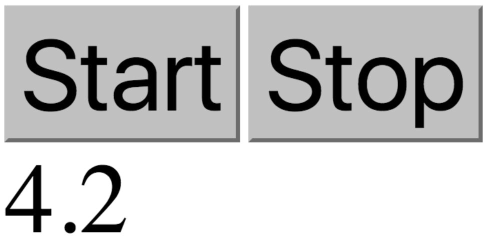
The team that produced this book includes:

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

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

Building a Stopwatch

Enough theory—you're probably itching to start building something. The first project you'll take on in this book is a stopwatch that contains three observables. The stopwatch will have two buttons, one for starting and one for stopping, with an observable monitoring each. Behind the scenes will be a third observable, ticking away the seconds since the start button was pressed in increments of 1/10th of a second. This observable will be hooked up to a counter on the page. You'll learn how to create observables that take input from the user, as well as observables that interact with the DOM to display the latest state of your app.



Before we get to the code, take a second to think about how you'd implement this without Rx. There'd be a couple of click handlers for the start and stop buttons. At some point, the program would create an interval to count the seconds. Sketch out the program structure—what order do these events happen in? Did you remember to clear the interval after the stop button was pressed? Is business logic clearly separated from the view? Typically, these aren't concerns for an app of this size; I'm specifically calling them out now, so you can see how Rx handles them in a simple stopwatch. Later on, you'll use the same techniques on much larger projects, without losing clarity.

This project has two different categories of observables. The interval timer has its own internal state and outputs to the document object model (DOM). The two-click streams will be attached to the buttons and won't have any kind of internal state. Let's tackle the hardest part first—the interval timer behind the scenes that needs to maintain state.

Running a Timer

This timer will need to track the total number of seconds elapsed and emit the latest value every 1/10th of a second. When the stop button is pressed, the interval should be cancelled. We'll need an observable for this, leading to the question: "How on earth do I build an observable?"

Good question—read through this example (don't worry about knowing everything that's going on, but take a few guesses as you go through it).

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

let tenthSecond$ = new Observable(observer => {
  let counter = 0;
  observer.next(counter);
  let interv = setInterval(() => {
    counter++;
    observer.next(counter);
  }, 100);

  return function unsubscribe() { clearInterval(interv); };
});
```

Let's walk through that line-by-line. As you read through each snippet of code, add it to the stopwatch.ts file in vanilla/creatingObservables.

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

The first thing is to use import to bring in Observable from the RxJS library. All of the projects in this book start off by bringing in the components needed to run the project. If your editor is TypeScript-aware (I recommend Visual Studio Code[1]), you probably have the option to automatically import things as you type. Most examples in this book skip the import statement for brevity's sake.

```
let tenthSecond$ = new Observable(observer => {
```

There's that dollar sign again, indicating the variable contains an observable. On the other side of the equals sign is the standard Rx constructor for observables, which takes a single argument: a function with a single parameter, an observer. Technically, an *observer* is any object that has the following methods: next(someItem) (called to pass the latest value to the observable stream), error(someError) (called when something goes wrong), and complete() (called once the data source has no more information to pass on). In the case of the observable constructor function, Rx creates the observer for you and passes it to the inner function. Later on, we'll see some other places you can use observers and even create new ones.

```
let counter = 0;
observer.next(counter);
let interv = setInterval(() => {
  counter++;
  observer.next(counter);
}, 100);
```

1. https://code.visualstudio.com/

While setInterval isn't perfect at keeping exact time, it suffices for this example. You'll learn about more detailed methods of tracking time in *Advanced Angular*.

Inside the constructor function, things get interesting. There's an internal state in the counter variable that tracks the number of tenths-of-a-second since the start. Immediately, observer.next is called with the initial value of 0. Then there's an interval that fires every 100 ms, incrementing the counter and calling observer.next(counter). This .next method on the observer is how an observable announces to the subscriber that it has a new value available for consumption. The practical upshot is that this observable emits an integer every 100 ms representing how many deciseconds have elapsed since…

…well, when exactly does this function *run*? Throw some console.log statements in and run the above snippet. What happens?

Nothing appears in the console—the constructor appears to never actually run. This is the lazy observable at work. In Rx land, this constructor function will only run when someone subscribes to it. Not only that, but if there's a *second* subscriber, all of this will run a second time, creating an entirely separate stream (this means that each subscriber gets its own timer)! You can learn more about how all of this works in *Multiplexing Observables*, but for now, just remember that each subscription creates a new stream.

Finally, the inner function returns yet another function (called an *unsubscribe* function):

```
return function unsubscribe() { clearInterval(interv); };
```

If the constructor function returns another function, this inner function runs whenever a listener unsubscribes from the source observable. In this case, the interval is no longer needed, so we clear it. This saves CPU cycles, which keeps fans from spinning up on the desktop, and mobile users will thank us for sparing their batteries. Remember, each subscriber gets their own instance of the constructor, and so, has their own cleanup function. All of the setup and teardown logic is located in the same place, so it requires less mental overhead to remember to clean up all the objects that get created.

Speaking of mental overhead, that was a lot of information in just a few lines of code. There are a lot of new concepts here, and it might get tedious writing this every time we want an interval. Fortunately, all of this work has already been implemented in the Rx library in the form of a *creation operator*:

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

let tenthSecond$ = interval(100);
```

Rx ships with a whole bunch of these creation operators for common tasks. You can find the complete list under the "Static Method Summary" heading at the official RxJS site.[2] interval(100) is similar to the big constructor function we had above. Now, to actually run this code, subscribe:

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

let tenthSecond$ = interval(100);
tenthSecond$.subscribe(console.log);
```

When there's a subscribe call, numbers start being logged to the console. The numbers that are logged are *slightly* off from what we want. The current implementation counts the number of tenths-of-a-second since the subscription, not the number of seconds. One way to fix that is to modify the constructor function, but stuffing all the logic into the constructor function gets unwieldy. Instead, an observable stream modifies data after a root observable emits it using a tool called an *operator*.

2. http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html