Chapter 9: Functional Reactive Programming

Overview

This chapter discusses functional reactive programming (FRP), with a focus on basic `RxJS` code samples that can help you develop simple Web applications (i.e., even without Angular). Starting from Chapter 4, you have seen various Angular code samples that use `Observables`. However, this chapter delves into other aspects of `RxJS` that were not covered in previous chapters.

As you will soon see, most of the code samples in this chapter are simple HTML Web pages. This approach significantly reduces development time because you simply launch the HTML Web pages in a browser without having to create applications from the command line (and also there is no need for the `node_modules` subdirectory). Thus, this chapter enables you to focus on quickly learning various aspects of RxJS, after which you can use the features that you need in your Angular applications.

The first part of the chapter contains a high-level introduction to FRP, along with a list of some popular JavaScript toolkits for FRP.

The second part of this chapter discusses intermediate operators and terminal operators in `RxJS`, including code samples that illustrate how to invoke multiple operators via method chaining. This section also discusses the difference between a cold `Observable` and a hot `Observable`, as well as how you can't "convert" the former into the latter. Almost all the code samples in this section are complete and self-contained, so you can launch them in a browser and view their output in Chrome Web Inspector.

The third part of this chapter contains examples of using `RxJS` with scalable vector graphics (SVG) to generate graphics and animation effects. You will also see an example of "reactifying" some HTML elements in an HTML Web page.

The final part of this chapter provides a very brief section regarding version 5 of `RxJS`, along with some differences between version 5 and version 4 of `RxJS`.

What Is Functional Reactive Programming (FRP)?

Various definitions of FRP are available on the Web. For our purposes, FRP is based on a combination of the `Observer` pattern, the `Iterator` pattern, and functional programming. The home page is located here:

http://reactivex.io/

Reactive programming was introduced in 1997, and can be summarized as follows:

- It is programming with asynchronous data streams.
- It is event-driven instead of proactive.
- Multiple toolkits and libraries are available.
- It supports languages such as JS, Java, Scala, Android, and Swift.

Conal Elliott is credited with creating FRP, and you can find his very specific definition of FRP here:

https://stackoverflow.com/questions/1028250/what-is- functional-reactive-programming

Another definition of FRP involves a combination of two other concepts: reactive programming (asynchronous data streams) and functional programming (pure functions, immutability, and minimal use of variables and state).

Reactive programming supports a number of operators that provide powerful functionality when working with asynchronous streams. The reactive programming paradigm avoids the "callback hell" that can occur in other environments. Moreover, `Observables` provide greater flexibility than working with `Promise`-based toolkits and libraries.

FRP is partially based on functional programming, which has gained popularity because it can reduce the amount of state in a program, which can in turn help reduce the number of code bugs. However, FRP is more declarative than functional programming, usually has more abstraction, and can involve higher-order functions. Consequently, the combination of reactive programming and functional programming enables you to write more succinct yet powerful code.

According to the Reactive home page, FRP handles errors properly in asynchronous streams and avoids the necessity of writing custom code to deal with threads, synchronization, and concurrency. From another perspective, FRP is the "culmination" of the path from `Collections`, then to `Streams`, and finally to asynchronous `Streams`.

Several toolkits for FRP in JavaScript can be found at the following sites:

RxJS: https://github.com/Reactive-Extensions/RxJS

Bacon.js: https://baconjs.github.io/

Kefir.js: https://rpominov.github.io/kefir/

most.js: https://github.com/cujojs/most

The preceding toolkits have different strengths and are typically more lightweight than RxJS. After you have completed this chapter, you will be in a better position to evaluate these alternatives to RxJS, and whether you want to use them instead of RxJS.

Now let's take a brief look at the `Observer` pattern that is fundamental to FRP.

The Observer Pattern

The `Observer` pattern is a powerful pattern that is implemented in many programming languages. In simplified terms, the `Observer` pattern involves an `Observable` (i.e., something that is observed or "watched") and one or more `Observer` objects. An `Observer` (also called a subscriber) "watches" for changes in data or the occurrence of events in another object. In languages such as Java, an `Observable` contains a collection of `Observer` objects that have registered themselves with the `Observable`. When a state change or an event occurs in the `Observable`, the `Observable` notifies the registered `Observer` objects.

The details of defining `Observables` are discussed later in this chapter, but the key idea involves combining ("chaining") operators (such as `map()` and `filter()`) and then invoking the `subscribe()` method to "make it happen."

Handling Asynchronous Events

RxJS is sometimes described as "LoDash for asynchronous events." Various types of asynchronous events include the following:

- Ajax
- User events (including mouse-related events)
- Animation
- Sockets and server-sent events (SSEs)
- Workers

By way of comparison, the following code snippet illustrates the ECMA5 style for handling an asynchronous event:

```
getDataFromSomewhere(function(result) {
    console.log("result = "+result)
});
```

The equivalent ES6 version of the preceding code snippet is shown here:

```
getDataFromSomewhere((result) => {
    console.log("result = "+result)
});
```

Promises and Asynchronous Events

`Promises` are well-suited for asynchronous operations (such as Ajax-based requests), provided that the expected behavior has one value and is then completed. The following list describes some of the properties of `Promises`:

- Guaranteed future (not always desirable in Web applications)
- Immutable
- Single value (not always desirable in Web applications)
- Caching
- Invoked immediately
- Cannot be canceled
- Cannot be reused

Hence, Promises are suitable when a future result is guaranteed and returns a single value. The following is a simple example of using a `Promise`:

```
getDataFromSomewhere(input)
   .then(data => {
       doSomethingHere(data);
           return getMoreData(data.id);
       })
      .then(data => {
          doSomethingHere(data);
          return getMoreData(data.id);
       })
```

In case you didn't already know, some of the operators that are available for `Observables` are also available as methods in ECMA5. Before we delve into FRP code samples, let's look at how to use some of those methods in the next section.

Using Operators without FRP

The following code block shows you how to chain the `filter()` and `map()` methods to process a set of integers:

```
var source = [0,1,2,3,4,5,6,7,8,9,10];

var result1 = source.map(x => x*x)
                    .filter(x => x % 5 == 0);
console.log("result1: "+result1);
// output=?

var result2 = source.filter(x => x % 5 == 0)
                      .map(x => x*x)
// output=?
```

Note that the output for `result1` and `result2` is the same. If possible, specify `filter()` methods before the `map()` methods to perform an "up-front" reduction in data (but see the caveat below). In the preceding example, the performance difference is probably undetectable, but if you change the source to include the first million positive integers, you will probably see a difference in performance.

As mentioned earlier, there is an important caveat regarding the order of operations: The `filter()` method and the `map()` method sometimes produce *different* results when they are invoked in the opposite order. For example, the following code block is a modified version of the preceding code block (modifications are shown in bold) that illustrates this point:

```
var source = [0,1,2,3,4,5,6,7,8,9,10];

var result3 = source.map(x =>  2*x)
                    .filter(x => x % 4 == 0);
console.log("result3: "+result3);  // [0,4,8,12,16,20]
var result4 = source.filter(x => x % 4 == 0)
                      .map(x => 2*x) // [0,8,16]
```

The variable `result3` has the value `[0,4,8,12,16,20]`, whereas the variable `result4` has the value `[0,8,16]`.

An Analogy Regarding `Observables`

If you are new to `Observables`, or find yourself struggling with code samples that contain `Observables`, this section provides a humorous analogy by Venkat Subramanian with a clever insight into the world of `Observables`.

First, `Observables` involve some of the key concepts: chaining intermediate operators (such as `map()`, `filter()`, and so forth) and then (possibly later) invoking a terminal operator (such as `subscribe()` or `forEach`) in order to "make stuff happen..

Skipping the syntax-related details, consider the following pair of `Observables` in JavaScript that involve the intermediate operators `map()` and `filter()`:

```
var source = [0,1,2,3,4,5,6];

var result1 = source.map(x => 3*x)
                    .filter(x => x % 4 == 0);

console.log("result1: "+result1);

var result2 = source.map(x => 3*x)
                    .filter(x => x % 4 == 0)
                    .subscribe();
console.log("result2: "+result2);
```

Question: What is the difference between `result1` and `result2`?

Answer: Only `result2` contains the terminal operator `subscribe().`

Result: The first `console.log()` displays nothing, and the second displays the numbers `0` and `12`.

Now let's read an entertaining (yet meaningful) analogy from Venkat Subramanian, who explains `Observables` by recounting a story of his wife and two teenaged sons, all of whom are watching television in their living room:

```
Mother: "It's time to switch off the TV".
```

```
Sons: [No response.]
Mother: "It's time to take out the trash."
Sons: [Nobody moves.]
Mother: "You need to start working on your homework."
Sons: [Still nothing.]
Some time passes…
Mother: "I'm going to get your father."
Sons: [Leaping into action…]
```

Hopefully you realize that the first three requests by the mother are similar to intermediate operators, and her final statement is analogous to a terminal operator, which starts the execution of the first three requests.

If this analogy has triggered a "lightbulb moment" for you regarding `Observables`, intermediate operators, and terminal operators, the good news is that many of the code samples in this chapter will be simpler for you to understand.

JavaScript Files for RxJS

The JavaScript files for `RxJS` are available via a content delivery network (CDN), and they consist of roughly 10 different files. The "core" JavaScript files that you need to include in an HTML Web page are shown here:

```
<script src="http://cdnjs.cloudflare.com/ajax/libs/rxjs/4.1.0/
                                                          rx.js">
</script>
<script src="http://cdnjs.cloudflare.com/ajax/libs/rxjs/4.1.0/
                                                    rx.async.js">
</script>
<script src="http://cdnjs.cloudflare.com/ajax/libs/rxjs/4.1.0/
                                                  rx.binding.js">
</script>
```

The version numbers may be different as this book goes to print. In addition, keep in mind that `RxJS` v5 is currently in beta. Additional `RxJS` files are available here (the first one is for animation effects):

```
<script src="http://cdnjs.cloudflare.com/ajax/libs/rxjs/4.1.0/
                                                     rx.time.js">
</script>
<script src="http://cdnjs.cloudflare.com/ajax/libs/rxjs/4.1.0/
                                                rx.coincidence.js">
</script>
<script src="http://cdnjs.cloudflare.com/ajax/libs/rxjs-
                                              dom/2.0.7/rx.dom.js">
</script>
```

With all of the preliminary details discussed, let's delve into various intermediate operators and terminal operators in the next section.

Intermediate and Terminal Operators

Think of `Observables` as streams or sets of data that can comprise any number of items (arbitrary time). `Observables` generate values when they are "subscribed," and can then be canceled and restarted (which is not the case for `Promises`). `Observables` help you avoid the "callback hell" that can occur in asynchronous code that does not use `Observables` or `Promises`.

Operators

Operators are methods in `Observables`. Operators can be intermediate or terminal (discussed later), and they allow you to compose new `Observables`. Common operators include `filter()`, `map()`, `reduce`, and `merge()`.

In the event that you need to create a custom operator in RxJS, you can do so with the following syntax:

```
Rx.Observable.prototype.myCustomOperator = // define
something here
```

You can use method chaining with operators with the following general syntax:

```
let obs = Rx.Observable
          .firstOperator()
          .secondOperator()
          .evenMoreOperatorsIfYouWant()
          .subscribe(....); // now stuff happens
```

Result: `obs` is an `Observable` that is "connected" to a source.

The subscribe() and unsubscribe() Operators

The `subscribe()` method must be invoked to generate data. By way of illustration, consider the following code block:

```
var source1 = Rx.Observable
                .range(0, 20)
                .filter(x => x < 4)

var source2 = Rx.Observable
                .range(0, 20)
                .filter(x => x < 4)
                .subscribe(x => console.log("x = "+x))
```

The first `Observable` does not generate output because there is no `subscribe()` method, whereas the second Observable displays the integers between `0` and 3 inclusive.

On the other hand, the `unsubscribe()` will "tear down" a producer, which means that the producer will stop producing data. The following code block shows you how to invoke the `unsubscribe()` method:

```
let x = Rx.Observable.(...)
let result = x.subscribe(...)
// do something here...
result.unsubscribe();
```

After invoking the `unsubscribe()` operator, you can restart an `Observable` and "resubscribe" to that `Observable`, which is not possible with `Promises`. (A proposal was submitted to the TC39 committee to add support for canceling `Promises` and was later withdrawn.)

The `subscribe()` and `forEach()` Operators

RxJS 4.0 follows the ES6 specification (but not the ES7 specification), in which `subscribe()` and `forEach()` are the same. However, RxJS 5.0 follows the ES7 specification, in which `subscribe()` and `forEach()` are different, as explained in the following paragraphs.

The syntax for the `subscribe()` method in RxJS 5.0 is shown here:

```
public subscribe(observerOrNext: Observer | Function, error:
Function, complete: Function): Subscription
```

`Observable.subscribe` returns a *subscription* token that enables you to cancel your subscription. This functionality is useful when the duration of the subscribed event is unknown, or if you need to perform an early termination.

The syntax for the `forEach()` method in RxJS 5.0 is shown here:

```
public forEach(next: Function, PromiseCtor?:
PromiseConstructor): Promise
```

`Observable.forEach` returns a *promise* that either resolves (or rejects) based on whether or not the `Observable` completes (or fails). Once again, remember that a `Promise` cannot be canceled.

Keep in mind that the vast majority of online code samples currently use RxJS 4.0, so it's probably worth your while to learn RxJS 4.0 as well as RxJS 5.0.

Converting Data Sources to `Observables`

You can "convert" other sources of data into an `Observable` with several methods, as shown here:

```
Observable.of(...)
Observable.from(promise/iterable/observable);
Observable.fromEvent(...)
```

The next set of subsections contains code samples that illustrate how to use variable operators in `RxJS`.

Using `range()` and `filter()` Operators

Listing 9.1 displays the contents of the `ObservableRangeFilter1.html` that illustrates how to define `Observables` using different syntax styles.

Listing 9.1: `ObservableRangeFilter1`.html

```
<html>
 <head>
  <meta charset="utf-8">
  <title>Working with Observables</title>
 </head>
 <body>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                            rxjs/4.1.0/rx.js">
  </script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                      rxjs/4.1.0/rx.async.js">
  </script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                    rxjs/4.1.0/rx.binding.js">
  </script>

  <script>
    var source1 = Rx.Observable
                    .range(0,  20)
                    .filter(x => x < 4)
    var source2 = Rx.Observable
                    .range(0, 20)
                    .filter(x => x < 4)
                    .subscribe(x => console.log("#2 x = "+x))
    var source3 = Rx.Observable
                    .range(1,  5)
                    .subscribe(
                      x => console.log('onNext:  %s', x),
                      e => console.log('onError: %s', e),
                      ()=> console.log('onCompleted'));
  </script>
 </body>
</html>
```

Listing 9.1 contains three `Observables`, the first of which does not generate any output because there is no `subscribe()` method. The second `Observable` filters the integers in the range `(0,20)` to those that are less than `4` and then displays their value via `console.log()`. The third observable iterates over the integers in the range `(1,5)` and displays their value via a `console.log()` method.

Launch the code in Listing 9.1 and you will see the following output:

```
#2 x = 0
#2 x = 1
#2 x = 2
#2 x = 3
onNext: 1
onNext: 2
onNext: 3
onNext: 4
onNext: 5
onCompleted
```

The next section contains an example that shows you how to chain the `from()` and `map()` intermediate operators.

Using `from()` and `map()` Operators

Listing 9.2 displays the contents of `ObservableMapUpper1.html` that illustrates how to use an `Observable` to "reactify" an HTML Web page.

Listing 9.2: ObservableMapUpper1.html

```
<html>
 <head>
  <meta charset="utf-8">
  <title>Working with Observables</title>
 </head>

 <body>
  <script  src="http://cdnjs.cloudflare.com/ajax/libs/
                                            rxjs/4.1.0/rx.js">
  </script>
  <script  src="http://cdnjs.cloudflare.com/ajax/libs/
                                      rxjs/4.1.0/rx.async.js">
  </script>
  <script  src="http://cdnjs.cloudflare.com/ajax/libs/
                                           rxjs/4.1.0/rx.binding.js">
  </script>

  <script>
    Rx.Observable.from(['a1','a2','a3'])
                 .map((item) => {
                    item = item.toUpperCase()+item;
                    return item;
                 })
                 .subscribe(str => console.log("item: "+str));
  </script>
 </body>
</html>
```

Listing 9.2 defines an `Observable` from an array of strings. The `map()` method converts each array item to its uppercase form and then appends the initial array item. The `subscribe()` method simply displays the value of each string that is created inside the `map()` method. Launch the code in Listing 9.2 in a browser and you will see the following output:

```
A1a1
A2a2
A3a3
```

The next section contains an example that shows you how to chain the `interval()`, `take()`, and `map()` intermediate operators.

Using the `interval()`, `take()`, and `map()` Operators

Listing 9.3 displays the contents of the `ObservableTake.html` that illustrates how to "interleave" the output from two `Observables` in an HTML Web page.

Listing 9.3: ObservableTake.html

```
<html>
 <head>
  <meta charset="utf-8">
  <title>Working with Observables</title>
 </head>

 <body>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                                  rxjs/4.1.0/rx.js">
  </script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                            rxjs/4.1.0/rx.async.js">
  </script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                          rxjs/4.1.0/rx.binding.js">
  </script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                             rxjs/4.1.0/rx.time.js">
  </script>

  <script>
    var source1 = Rx.Observable
                    .interval(1000)
                    .take(4)
                    .map(i => ['1','2','3','4','5'][i]);

    var result1 = source1.subscribe(x => console.log
                                                ("x = "+x));
    var source2 = Rx.Observable
                    .interval(500)
                    .take(4)
                    .map(i => ['1','2','3','4','5'][i]);

    var subscription = source2.subscribe(
      x => console.log('source2 onNext:  %s', x),
      e => console.log('source2 onError: %s', e),
      ()=> console.log('source2 onCompleted'));
    </script>
  </body>
</html>
```

Listing 9.3 defines two Observables that invoke the interval(), take(), and map() operators. The first Observable emits data every 1000 milliseconds whereas the second Observable emits data every 500 milliseconds. The first Observable invokes the subscribe() method that contains a console.log() statement for displaying data items. The second observable also invokes the subscribe() method with a different syntax: Three functions are specified that handle the next, error, and completed events, respectively.

Launch the code in Listing 9.3 and you will see the following output:

```
source2 onNext: 1
x = 1
source2 onNext: 2
source2 onNext: 3
x = 2
source2 onNext: 4
source2 onCompleted
x = 3
x = 4
```

At this point you have an understanding of how to combine some intermediate operators. RxJS supports many other operators, and the next section contains a high-level and rapid introduction to some of those operators.

Other Intermediate Operators

In addition to the filter() and map() operators that you saw earlier in this chapter, Observables support the following useful operators, most of which have intuitive names:

- reduce()
- first()
- last()
- skip()
- toArray()
- isEmpty()
- retry()
- startWith()

Even if you have not seen the preceding operators, you can probably surmise the result of this Observable:

```
var source = Rx.Observable
               .return(8)
               .startWith(1,  2,  3)
               .subscribe(x  => console.log("x = "+x));
```

The retry() Operator

The retry() operator enables you to make multiple attempts to access data from an external website. Two examples of this syntax are shown here:

```
myObservable.retry(3);
myObservable.retryWhen(errors => errors.delay(3000));
```

The next section provides a list of some intermediate operators for merging and joining data streams in Observables.

A List of Merge/Join Operators

In Chapter 4 you learned about the forkJoin() intermediate operator that merges the data returned from HTTP requests from multiple endpoints. The following list contains various intermediate operators that perform merging or joining operations on streams of data:

- merge()
- mergeMap()
- concat()

- concatMap()
- switch()
- switchMap()
- zip()
- forkJoin()
- withLatestFrom()
- combineLatest()

Some of these are intuitively named (such as `concat()` for concatenating output), yet there are some subtle differences. For example, the `merge()` operator combines multiple `Observables` into one `Observable`, with the *possibility of interleaving* data values. On the other hand, `concat()` combines multiple `Observables` sequentially into one `Observable`, *without interleaving* any data.

Read the online documentation for the intermediate operators that interest you (or learn about all of them if you have the time!).

A List of Map-Related Operators

In Chapter 4 you saw several examples that use the `map()` intermediate operator, usually to convert an input stream into JavaScript Object Notation (JSON)-based data. A list of map-related operators is shown below:

- map()
- flatMap()
- flatMapLatest()
- mergeMap()
- concatMap()
- switchMap()
- flatten()

Recall from Chapter 4 how you used the `map()` operator to convert a string into JSON-based data. A more powerful operator is `concatMap()`, which uses `concat()` to ensure that intermediate results are not interleaved, and then the map() operator is applied.

As you learned in the previous section, intermediate results can be interleaved with the `merge()` operator. Because `flatMap()`uses the `merge()` operator, intermediate results can be interleaved with `flatMap()` as well (but not with the `concatMap()` operator).

Read the online documentation for the intermediate operators that interest you, and in particular, learn about the difference between the `flatten()` and `flatMap()` operators.

The `timeout()` Operator

The `timeout()` operator is useful for detecting if an `Observable` has not produced a value after a specified time period:

```
Rx.Observable
  .fromEvent(document, 'dragover')
  .throttle(350)
  .map(true)
  .timeout(1000, Rx.Observable.just(false))
  .distinctUntilChanged();
```

Keep in mind that `timeout()` will unsubscribe from the source `Observable` and subscribe with the `Observable` that was supplied as a parameter to `timeout()`; hence, you must resubscribe to that `Observable` to receive further results. The other point to remember is that the `map()` operation is placed after the `throttle()` operator to avoid unnecessary mapping operations of values that are not used.

Cold versus Hot Observables

A *cold* observable is comparable to watching a recorded movie (e.g., viewed in a browser). Although users navigate to the same URL at different times, all of them see the entire contents of the movie. In the case of cold observables, a new producer (movie instance) is created for each consumer (which is analogous to a person watching the movie).

By contrast, a *hot* observable is comparable to watching a live online presentation. Users navigate to a website at different times, and instead of seeing the entire presentation, they see only the portion from the point in time that they launched the presentation. In the case of cold observables, the same producer (streaming presentation) is used for each consumer (person watching the presentation).

Listing 9.4 displays the contents of the Web page `ColdObservables1.html` that illustrates a sequence of cold `Observables`, and how to convert them to hot observables.

Listing 9.4: ColdObservables1.html

```
<html>
 <head>
  <meta charset="utf-8">
  <title>Working with Cold Observables</title>
 </head>
 <body>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                            rxjs/4.1.0/rx.js">
  </script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                    rxjs/4.1.0/rx.binding.js">
  </script>

  <script>
    // this is a cold observable:
    let obs = Rx.Observable
                .create(observer => {
                   observer.next(13);
                });

   // first subscriber (displayed first):
    obs.subscribe(n => console.log("subscriber 1: " + n));

    // second subscriber (displayed second):
    var subscription = obs.subscribe(

      x  => console.log('Called next:  %s', x),
      e  => console.log('Called error: %s', e),
      () => console.log('Called completed')
    );
      subscription.dispose();

    // delay and add a subscriber (displayed fourth)
    setTimeout(() =>
      obs.subscribe(v => console.log("delayed: " + v)), 2000)
    // subscribe again (displayed third):
    obs.subscribe(n => console.log("subscriber 2: " + n));
  </script>
 </body>
</html>
```

Listing 9.4 contains a cold `Observable` followed by two `Subscribers` to that `Observable`.

Launch the code in Listing 9.4 and you will see the following output:

```
subscriber 1: 13
Called next: 13
subscriber 2: 13
delayed: 13
```

Note All `Observables` are cold by default, whereas all `Promises` are hot by default.

One way to convert a cold `Observable` into a hot `Observable` is via the `publish()` operator, as shown here:

http://blog.thoughtram.io/angular/2016/06/16/cold-vs- hot-observables.html

Reactifying an HTML Web Page

Web pages that contain a button element for accessing external data are good candidates for the use of `Observables`.

Listing 9.5 displays the contents of `ObservableDivElement2.html` that illustrates how to use an `Observable` to reactify an **HTML** Web page.

Listing 9.5: ObservableDivElement2.html

```
<html>
 <head>
  <meta charset="utf-8">
  <title>Working with Observables and div Elements</title>
 </head>

 <body>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                           rxjs/4.1.0/rx.js">
  </script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                     rxjs/4.1.0/rx.async.js">
  </script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                      rxjs/4.1.0/rx.time.js">
  </script>

  <div id="div1">This is a DIV element</div>
  <div id="div2">This is a DIV element</div>

  <script>
    let div1 = document.querySelector('#div1')
    let div2 = document.querySelector('#div2')

    var stream = Rx.Observable
                   .interval(500)
                   .take(10)
                   .map(x => x*x)
                   .subscribe(x => {
                      div1.innerHTML += x;
                      div2.innerHTML += x;
                    })
  </script>
 </body>
</html>
```

Listing 9.5 contains two `<div>` elements, followed by a `<script>` element that references both of them. The `<script>` element also defines an `Observable` that emits the first `10` integers, with a `500`-millisecond delay between data items.

The next part of the `Observable` invokes the `map()` operator to compute the square of each data item. The `subscribe()` method appends the newly computed number to the current contents of both `<div>` elements.

Launch the code in Listing 9.5 and you will see the following output in your browser session upon completion of the `Observable`:

```
This is a DIV element0149162536496481

This is a DIV element0149162536496481
```

RxJS and SVG Graphics/Animation

Listing 9.6 displays the contents of the Web page `SVG Observables 1 Anim1.html` that illustrates how to use `Observable` to generate and display SVG `<ellipse>` elements with an animation effect in an HTML Web page.

Listing 9.6: SVGObservables1Anim1.html

```
<html>
 <head>
  <meta charset="utf-8">
  <title>Observables and SVG</title>
 </head>

 <body>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                  rxjs/4.1.0/rx.js"></script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                            rxjs/4.1.0/rx.async.js"></script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/
                             rxjs/4.1.0/rx.time.js"></script>
  <script src="https://ajax.googleapis.com/ajax/
                          libs/jquery/1.11.1/jquery.min.js">
  </script>

  <div id="outer1">
    <svg height="300" id="svg1" />
  </div>

  <script>
    var XPoints =  [], factor=30;
    var minorAxis = 20,  majorAxis = 40, strokeWidth = 1;
    var colors = ["#FF0000", "#FFFF00", "#0000FF"];
    var color  = colors[0], colorIndex = 0;
    var svgNS  = "http://www.w3.org/2000/svg";
    var svg    = document.querySelector("#svg1");
    var svgDocument;

    window.onload = function(evt){
      svgDocument = document.getElementById("svg1").
                                            ownerDocument;

      // generate some positions for the ellipses
      for(var i=2; i<10; i++) {
         XPoints.push(i*factor);
      }

      var delay = Rx.Observable.empty().delay(1000);
      var items = Rx.Observable.fromArray(XPoints)
                    .map(function (x) {
                      return Rx.Observable.return(x).
                                            concat(delay);
                    })
                    .concatAll();
      items.subscribe(x => {
             // create an SVG graphics element:
             var elem = svgDocument.createElementNS(svgNS,
                                                  "ellipse");
             elem.setAttribute("fill",
                               colors[(colorIndex++)%colors.
                                                    length]);
             elem.setAttribute("stroke",
                               colors[(1+colorIndex)%colors.
                                                    length]);
             elem.setAttribute("stroke-width", strokeWidth);
             elem.setAttribute("cx", x);
             elem.setAttribute("cy", x);
             elem.setAttribute("rx", majorAxis);
             elem.setAttribute("ry", minorAxis);
             // append the SVG element to the <svg> element:
             $("#svg1").append(elem);
           });
    }
  </script>
 </body>
</html>
```

Listing 9.6 contains a `<script>` element that initializes an assortment of JavaScript variables. The next portion of Listing 9.6 obtains a reference to a `<div>` element whose `id` attribute has the value `svg1`, followed by a simple loop that initializes the JavaScript array `XPoints`. After initializing `XPoints`, there is an interesting block of code that creates `Observable items` based on the `XPoints` array, and also a delay property that creates an animation effect (i.e., a `1`-second delay between rendering adjacent SVG `<ellipse>` elements).

The third portion of Listing 9.6 invokes the `subscribe()` method, which retrieves the data values in the populated array to calculate the position of a set of ellipses. An SVG ellipse is created via the method `createElementNS()`, and then its attributes are assigned via the `setAttribute()` method. The ellipse is fully populated and rendered, with a 1-second delay between consecutive ellipses, until a total of 10 ellipses have been rendered.

RxJS and Mouse Events in an HTML Web Page

Listing 9.7 displays the contents of `SVGObservables1MouseMove1.html` that illustrates how to handle mouse-related events. This code sample creates a follow-the-mouse effect that renders a different colored ellipse whenever users move their mouse.

Listing 9.7: SVGObservables1MouseMove1.html

```
<html>
 <head>
  <meta charset="utf-8">
  <title>Observables, MouseEvents, and SVG</title>
 </head>

 <body>
 <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                            rxjs/4.1.0/rx.js">
 </script>
 <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                      rxjs/4.1.0/rx.async.js">
 </script>
 <script src="http://cdnjs.cloudflare.com/ajax/libs/
                               rxjs/4.1.0/rx.coincidence.js">
 </script>
 <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                    rxjs/4.1.0/rx.binding.js">
 </script>
 <script src="http://cdnjs.cloudflare.com/ajax/libs/
                                       rxjs/4.1.0/rx.time.js">
 </script>
 <script src="http://cdnjs.cloudflare.com/ajax/libs/rxjs-
                                          dom/2.0.7/rx.dom.js">
 </script>
 <script src="https://ajax.googleapis.com/ajax/ libs/
                                jquery/1.11.1/jquery.min.js">
 </script>

 <div id="outer1">
   <svg height="300" id="svg1" />
 </div>

 <script>
   var minorAxis = 30,  majorAxis = 10, strokeWidth = 1;
   var colors = ["#FF0000","#FFFF00","#0000FF","#FFFFFF"];
   var color  = colors[0],  colorIndex = 0;
   var svgNS  = "http://www.w3.org/2000/svg";
   var svg    = document.querySelector("#svg1");
   var svgDocument;

   window.onload = function(evt){
     svgDocument = document.getElementById("svg1").
                                               ownerDocument;
     // set up RxJS-related stuff...
     var mouseDownEvt =
            Rx.Observable.fromEvent(svg,"mousedown");
     var mouseUpEvt   =
               Rx.Observable.fromEvent(svg,"mouseup");
        var mouseMoveEvt =
               Rx.Observable.fromEvent(document,"mousemove");

        mouseDownEvt.map(function () {
           return mouseMoveEvt.takeUntil(mouseUpEvt);
        })
        .concatAll()
        .subscribe(function (e) {
           // create an SVG graphics element:
           var elem = svgDocument.createElementNS(svgNS,
                                                     "ellipse");

           elem.setAttribute("fill",
                             colors[(colorIndex++)%colors.
                                                      length]);
            elem.setAttribute("stroke",
                              colors[(1+colorIndex)%colors.
                                                          length]);
            elem.setAttribute("stroke-width", strokeWidth);

            elem.setAttribute("cx", e.x);
            elem.setAttribute("cy", e.y);
            elem.setAttribute("rx", majorAxis);
            elem.setAttribute("ry", minorAxis);

            // append the SVG element to the <svg> element:
            $("#svg1").append(elem);
        });

       // set initial click position
       mouseDownEvt.subscribe(function (e) {
        //offsetX = e.x - parseInt(svg.offsetLeft);
        //offsetY = e.y - parseInt(svg.offsetTop);
       });
     }
   </script>
  </body>
</html>
```

Listing 9.7 contains a `<script>` element that initializes some JavaScript variables. The next portion of Listing 9.7 defines the `Observables` to handle mouse down, mouse up, and mouse move events. The third portion of Listing 9.7 invokes the `subscribe()` method, which retrieves the data values in the populated array to calculate the position of a set of ellipses.

An SVG ellipse is created via the method `createElementNS()`, and then its attributes are assigned via the `setAttribute()` method. The code uses the value of the JavaScript variable `colorIndex` as an index into the `colors` array, as shown here:

```
elem.setAttribute("fill",
                  colors[(colorIndex++)%colors.length]);
elem.setAttribute("stroke",
                  colors[(1+colorIndex)%colors.length]);
```

After the other mandatory attributes are initialized, the newly created SVG `<ellipse>` is appended to the Document Object Model (DOM) via the jQuery "$" function.

Additional examples of handling mouse events with `RxJS` are available here:

- http://jsfiddle.net/dinkleburg/ay8afp5f
- http://reactivex.io/learnrx/
- http://rxmarble.com
- http://cycle.js.org/basic-examples.html

Other code samples involving `RxJS` and SVG graphics/animation are located here:

https://github.com/ocampesato/rxjs-svg-graphics

The following website illustrates how to create a toggle button with `RxJS`:

https://www.themarketingtechnologist.co/create-a-simple- toggle-button-with-rxjs-using-scan-and-startwith

An Observable Form

The code sample in this section shows you how to combine the intermediate operator `interval()` with the `async` operator in an Angular application to emit integers at regular intervals.

DVD Copy the directory `ObservableForm` from the companion disc into a convenient location. Listing 9.9 displays the contents of `app.component.ts` that illustrates how to watch for changes in an `<input>` element and then dynamically update another `<input>` element with a randomly selected last name from an array of names.

Listing 9.9: app.component.ts

```
import { Component }   from '@angular/core';
import { FormBuilder } from '@angular/forms';
import { FormGroup }   from '@angular/forms';
import 'rxjs/Rx'

@Component({
  selector: 'app-root',
  template: '
    <div>
       <h2>An Observable Form</h2>

       <form [formGroup]="myForm"
             (ngSubmit)="onSubmit(myForm.value)">

         <div class="field">
           <label for="fname">fname</label>
           <input type="text" id="lname"
                  [formControl]="myForm.controls['fname']">
        </div>

        <div class="field">
          <label for="lname">lname</label>
          <input type="text" id="lname"
                 [formControl]="myForm.controls['lname']">
        </div>

        <button type="submit">Submit</button>
       </form>

       <div class="field">
         <label for="randomName">Random:</label>
         <input id="randomName" [(ngModel)]="randomName">
       </div>
     </div>
    '

})
export class AppComponent {
  myForm: FormGroup;
  randomName = "";
  lNames = ["Anderson", "Smith", "Jones", "Edwards"];

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'fname': [''],
      'lname': ['']
    });

    this.myForm.controls['fname'].valueChanges
        .debounceTime(750)
        .subscribe(data => this.getUserName(data));
  }
  getUserName(fname) {
      var index = Math.floor(20*Math.random())+1;
      var lname = this.lNames[index%this.lNames.length];
      this.randomName = lname;
  }

  onSubmit(value: string): void {
    console.log('you submitted value:', value);
  }
}
```

Listing 9.9 is a copy of the directory `FormBuilder` from Chapter 5, with the new code sections shown in bold. The new functionality is straightforward: When users enter a text string in the first `<input>` element and then pause for 750 milliseconds, the `subscribe()` method in the constructor invokes the `getUserName()` method with the inputted string (which is not actually used in this example). Next, the `getUserName()` method randomly selects a last name from the `lNames` array, and displays the selected name in the third `<input>` element.

As you can see, this type of functionality is useful when you want to perform dynamic validation before users click the `<submit>` button.

Unsubscribing in Angular Applications

The Angular documentation indicates that Angular will invoke the unsubscribe method on your behalf for the Observables that you have defined in an Angular application. However, there is a bug (discovered as this book goes to print) in this functionality, which is described via a code sample in this blog post:

https://netbasal.com/when-to-unsubscribe-in-angular- d61c6b21bad3

An RxJS and Timer Example

The code sample in this section shows you how to combine the intermediate operator `interval()` with the `async` operator in an Angular application to emit integers at regular intervals.

DVD Copy the directory `NGObservableTimer` from the companion disc into a convenient location. Listing 9.8 displays the contents of `app.component.ts` that illustrates how to use a simple timer in an Angular application.

Listing 9.8: app.component.ts

```
import { Component } from '@angular/core';
import {Observable}  from 'rxjs/Observable';
import 'rxjs/add/observable/interval';

@Component({
  selector: 'app-root',
  template: '<h2>{{mytimer |async}}</h2>'
})
export class AppComponent {
   pause:number = 1500;
   mytimer = Observable.interval(this.pause);
}
```

Listing 9.8 contains a standard `import` statement, followed by two more `import` statements for an `Observable` and the `interval()` method. Next, a `template` property contains the `Observable` called `mytimer` whose output is "piped" to the `async` operator.

The `AppComponent` class contains the definition of `mytimer` that emits an integer every `1500` milliseconds (which is the value of `pause` in this example).

At this point you can easily modify this code sample by incorporating the `Observables` in this chapter.

RxJS Version 5

`RxJS` version 5 provides better debugging and better modularity. Version 5 also follows the ES7 specification, whereas `RxJS` version 4 follows the ES6 specification.

`RxJS` version 5 provides some notable performance improvement: `RxJS` version 5 is also 5 times faster (on average) than `RxJS` v4 in Google Chrome V8.

A code snippet with `RxJS` v5 in ES6:

```
import {Observable} from "rxjs/Observable";
import { map } from "rxjs/operator/map";
map.call(Observable.of(1,2,3), x => x*x)
                   .subscribe(console.log.bind(console));
```

A code snippet with `RxJS` v5 with Babel in ES6:

```
import {Observable} from "rxjs/Observable";
import { map } from "rxjs/operator/map";

Observable.of(1,2,3)::map(x => x*x)
          .subscribe(::console.log);
```

The following code snippet uses the syntax for `RxJS` version 5 in TypeScript:

```
import {Observable} from "rxjs/Observable";
import "rxjs/add/operator/map";

Observable.of(1,2,3).map(x => x*x)
          .subscribe(console.log.bind(console));
```

Creating Observables in Version 5 of RxJS

The syntax for creating an `Observable` in version 5 is slightly different from the syntax in version 4: the "on" prefix has been dropped in version 5.

An example of the syntax for version 5 of `RxJS` is shown here:

```
let obs = new Observable(observer => {
    myAsyncMethod((err,value) => {
       if(err) {
          observer.error(err);
       } else {
          observer.next(value);  // older v4: onNext
          observer.complete();   // older v4: onComplete
         }
     });
});
```

Compare the preceding syntax with earlier examples of `Observables` in this chapter.

Caching Results in RxJS

Version 5 of `RxJS` supports the `cache()` operator, which enables you to cache results of an `Observable`. If you are using `RxJS` version 4, and you would like use caching in your code, the following link contains information about caching results:

http://www.syntaxsuccess.com/viewarticle/caching-with- rxjs-observables-in-angular-2.0

d3.express: The Integrated Discovery Environment

In Chapter 3 you saw some Angular applications that create D3-based graphics. As this book goes to print, Mike Bostock (creator of D3.js) is currently developing `d3.express`, which he describes here:

https://medium.com/@mbostock/a-better-way-to-code-2b1d2876a3a0

Based on the contents of this article, it's possible that `d3.express` will "play well" with RxJS (and hence its inclusion in this chapter), which bodes well for `d3.express`. This looks like yet another very interesting project from Mike Bostock that could provide more sophisticated data visualization functionality in Angular applications.

As this book goes to print, an alpha release of `d3.express` may be available, and you can sign up for early access here:

https://d3.express

Summary

This chapter introduced you to `FRP`, focusing primarily on `RxJS` for Web applications. You then learned about various operators in `FRP`, such as `filter()`, `map()`, and `reduce()`. You saw the similarities and differences between a `Promise` and an `Observable`. Then you learned about the difference between cold `Observables` and hot `Observables`, and how to convert a cold `Observable` into a hot `Observable`.

In addition, you learned how to reactify HTML elements in HTML Web pages. Finally, you saw how to generate SVG graphics and animation effects using `Observables`, and a follow-the-mouse example that displays SVG-based ellipses during mouse move events.

Finally, you saw how to combine the intermediate operator `interval()` with the `async` operator in an Angular application to emit integers at regular intervals.