Reactive Extensions / part 3

In the previous part, you learned the very basics of functional reactive programming, using the `bacon.js` library. We discussed what is an observable and saw two different flavors of it in the `bacon.js` library (EventStream and Property). We also looked at how to create observables from common event sources (interval, array, or user input) and even from any other possible source (using the `fromBinder()` method).

After an introduction to observables, we looked at how to subscribe to it. You can react to events, errors, or even detect the end of an observable when you subscribe to it. So, you learned how to create an observer for your observables.

One of the really important lessons from the previous part was the usage of diagrams to explain how observables and operators occur, along with text explanations and example code. Understanding diagrams is the key to understanding the behavior of operators and observables; you will see a lot of diagrams throughout the guide. Read this section with special attention. Read it again in case you're not comfortable with it yet, but don't be afraid if it seems complicated (it seemed that way to me the first time I saw it). You will keep seeing it with text explanations and will eventually learn how it works.

The previous part added a lot of new concepts to your toolbox, and one of these was operators. The usage of operators to transform your events into something completely new gives you the taste of functional reactive programming and lets you see how powerful and composable it is. If you liked the initial operators you used, then be happy; we will see a lot more operators in the upcoming parts. Operators are the backbone of functional reactive programming and they help you write cleaner, concise, and testable code with ease.

Finally, we saw the laziness of observables in the `bacon.js` library. We will talk about the behavior of observables in RxJS in this part, so don't worry.

This part will introduce you to the world of Reactive Extensions. In this guide, we will base our implementation on RxJS. However, Reactive Extensions is an agnostic framework (this means it has implementations for several languages), so a lot of the concepts described here can be used in other platforms (such as RxJava, RxSwift, and so on). This makes learning Reactive Extensions (and functional reactive programming) really useful, as you can use it to improve your code in different platforms.

The focus in this part is to teach you the different types of observables possible in the RxJS world. You will see that they differ a little bit from the observables of bacon.js but are easy to use. You will also learn how to react to events in our observables using observers and how to finish your observers.

This part will cover the following points:

- The different types of observables in the context of Reactive Extensions and the difference between them
- Using the RxJS API to create observables
- Reacting to events in our observables
- Reacting to errors in our observables

RxJS observables

RxJS lets you have even more control over the source of your data. In this section, we will learn the differences between RxJS Observables and bacon.js EventStreams and Properties. We will also learn some different flavors of Observables and how we can better control their life cycle.

Difference between bacon.js and RxJS observables

In part 2, you learned that an observable is basically an abstraction over possible asynchronous data. The observable gives you the power to transform data using different operators and take an action when a piece of new data becomes available, using a subscriber. The `bacon.js` library uses the term **subscriber** to the object listening to incoming data, but on Reactive Extensions, we will use a different term; we will call it Observer.

Conceptually, there is no difference between the two. Basically, it was just names chosen by the developers of both libraries; however, it is important that you're aware of both the names as you will see both being used in the documentation of the libraries.

In bacon.js, we saw two different flavors of observables:

- EventStreams
- Properties

In RxJS, we will always talk about:

- Observables
- Observers

Hot and cold observables

In bacon.js, an observable only emits a value when a subscriber subscribes to it; therefore, we will call it a **cold observables**. In Reactive Extensions, things can become a little trickier: we have both hot and cold observables.

A hot observable is an observable that propagates the data independently, irrespective of whether we have some Observer attached to it or not. An example of a hot observable is an observable created from mouse movements.

A cold observable, on the other hand, is an observable that fires the same sequence for all the subscribers. An example of a cold observable is an observable created from an array.

This behavior is important to understand, and RxJS has special methods to replay events in a hot observable or turn a cold observable into a hot observable.

Installing RxJS

RxJS is divided into modules. This way, you can create your own bundle with only the modules you're interested in. In this guide, we will always use the official bundle with all the contents from RxJS; by doing so, we'll not have to worry about whether a certain module exists in our bundle or not. So, let's follow the steps described here to install RxJS.

To install it on your server, just run the following command inside a node project:

```
npm i rx@4.1.0 -save
```

To add it to a HTML page, just paste the following code snippet inside your HTML:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/4.1.0/rx.all.js">
</script>
```

If you're running inside a node program, you need to have the `RxJS` library in each JavaScript file that you want to use. To do this, add the following line to the beginning of your JavaScript file:

```
var Rx = require('rx');
```

The preceding line will be omitted in all examples, as we expect you to have added it before testing the sample code.

Creating an observable

Here we will see a list of methods to create an observable from common event sources. This is not an exhaustive list, but it contains the most important ones.

Creating an observable from iterable objects

We can create an observable from iterable objects using the `from()` method. An iterable in JavaScript can be an array (or an array-like object) or other iterates added in ES6 (such as `Set()` and `map()`). The `from()` method has the following signature:

```
Rx.Observable.from(iterable,[mapFunction],[context],[scheduler]);
```

Usually, you will pass only the first argument. Others arguments are optional; you can see them here:

- `iterable`: This is the `iterable` object to be converted into an observable (can be an array, set, map, and so on)
- `mapFunction`: This is a function to be called for every element in the array to map it to a different value
- `context`: This object is to be used when `mapFunction` is provided
- `scheduler`: This is used to iterate the input sequence

Now let's see some examples on how we can create observables from iterables.

To create an observable from an array, you can use the following code:

```
Rx.Observable
    .from([0,1,2])
    .subscribe((a)=>console.log(a));
```

This code prints the following output:

```
0
1
2
```

Now let's introduce a minor change in our code, to add the `mapFunction` argument to it, instead of creating an observable to propagate the elements of this array. Let's use `mapFunction` to propagate the double of each element of the following array:

```
Rx.Observable
    .from([0,1,2], (a) => a*2)
    .subscribe((a)=>console.log(a));
```

This prints the following output:

```
0
2
4
```

We can also use this method to create an observable from an `arguments` object. To do this, we need to run `from()` in a function. This way, we can access the `arguments` object of the function. We can implement it with the following code:

```
var observableFromArgumentsFactory = function(){
    return Rx.Observable.from(arguments);
};
observableFromArgumentsFactory(0,1,2)
    .subscribe((a)=>console.log(a));
```

If we run this code, we will see the following output:

```
0
1
2
```

One last usage of this method is to create an observable from either `Set()` or `Map()`. These data structures were added to ES6. We can implement it for a `set` as follows:

```
var set = new Set([0,1,2]);
Rx.Observable
    .from(set)
    .subscribe((a)=>console.log(a));
```

This code prints the following output:

```
0
1
2
```

We can also use a `map` as an argument for the `from()` method, as follows:

```
var map = new Map([['key0',0],['key1',1],['key2',2]]);
Rx.Observable
    .from(map)
    .subscribe((a)=>console.log(a));
```

This prints all the key-value tuples on this `map`:

```
    [ 'key0', 0 ]
    [ 'key1', 1 ]
    [ 'key2', 2 ]
```

All observables created from this method are cold observables. As discussed before, this means it fires the same sequence for all the observers. To test this behavior, create an `observable` and add an Observer to it; add another observer to it after a second:

```
var observable = Rx.Observable.from([0,1,2]);

observable.subscribe((a)=>console.log('first subscriber receives => '+a));

setTimeout(()=>{
    observable.subscribe((a)=>console.log('second subscriber receives =>
'+a));
},1000);
```

If you run this code, you will see the following output in your console, showing both the subscribers receiving the same data as expected:

```
    first subscriber receives => 0
    first subscriber receives => 1
    first subscriber receives => 2
    second subscriber receives => 0
    second subscriber receives => 1
    second subscriber receives => 2
```

Creating an observable from a sequence factory

Now that we have discussed how to create an observable from a sequence, let's see how we can create an observable from a sequence factory. RxJS has a built-in method called `generate()` that lets you create an observable from an iteration (such as a `for()` loop). This method has the following signature:

```
Rx.Observable.generate(initialState, conditionFunction, iterationFunction,
resultFactory, [scheduler]);
```

In this method, the only optional parameter is the last one. A brief description of all the parameters is as follows:

- `initialState`: This can be any object, it is the first object used in the iteration
- `conditionFunction`: This is a function with the condition to stop the iteration
- `iterationFunction`: This is a function to be used on each element to iterate
- `resultFactory`: This is a function whose return is passed to the sequence
- `scheduler`: This is an optional scheduler

Before checking out an example code for this method, let's see some code that implements one of the most basic constructs in a program: a `for()` loop. This is used to generate an array from an initial value to a final value. We can produce this array with the following code:

```
var resultArray=[];
for(var i=0;i < 3;i++){
    resultArray.push(i)
}
console.log(resultArray);
```

This code prints the following output:

```
[0,1,2]
```

When you create a `for()` loop, you basically give to it the following: an initial state (the first argument), the condition to stop the iteration (the second argument), how to iterate over the value (the third argument), and what to do with the value (block). Its usage is very similar to the `generate()` method. Let's do the same thing, but using the `generate()` method and creating an observable instead of an array:

```
Rx.Observable.generate(
    0,
    (i) => i<3,
    (i) => i+1,
    (i) => i
).subscribe((i) => console.log(i));
```

This code will print the following output:

```
0
1
2
```

Creating an observable using range ()

Another common source of data for observables are ranges. With the `range()` method, we can easily create an observable for a sequence of values in a range. The `range()` method has the following signature:

```
Rx.Observable.range(first, count, [scheduler]);
```

The last parameter in the following list is the only optional parameter in this method:

- `first`: This is the initial integer value in the sequence
- `count`: This is the number of sequential integers to be iterated from the beginning of the sequence
- `scheduler`: This is used to generate the values

We can create an observable using a range with the following code:

```
Rx.Observable
    .range(0, 4)
    .subscribe((i)=>console.log(i));
```

This prints the following output:

```
0
1
2
3
```

Creating an observable using period of time

In the previous part, we discussed how to create timed sequences in bacon.js. In RxJS, we have two different methods to implement observables emitting values with a given interval. The first method is `interval()`. This method emits an infinite sequence of integers starting from one every *x* milliseconds; it has the following signature:

```
Rx.Observable.interval(interval, [scheduler]);
```

The interval parameter is mandatory, and the second argument is optional:

- `interval`: This is an integer number to be used as the interval between the values of this sequence
- `scheduler`: This is used to generate the values

Run the following code:

```
Rx.Observable
    .interval(1000)
    .subscribe((i)=> console.log(i));
```

You will see an output as follows; you will have to stop your program (hitting Ctrl+C) or it will keep sending events:

```
0
1
2
```

The `interval()` method sends the first value of the sequence after the given period of interval and keeps sending values after each interval.

RxJS also has a method called `timer()`. This method lets you specify a due time to start the sequence or even generate an observable of only one value emitted after the due time has elapsed. It has the following signature:

```
Rx.Observable.timer(dueTime, [interval], [scheduler]);
```

Here are the parameters:

- `dueTime`: This can be a date object or an integer. If it is a date object, then it means it is the absolute time to start the sequence; if it is an integer, then it specifies the number of milliseconds to wait for before you could send the first element of the sequence.
- `interval`: This is an integer denoting the time between the elements. If it is not specified, it generates only one event.
- `scheduler`: This is used to produce the values.

We can create an observable from the `timer()` method with the following code:

```
Rx.Observable
    .timer(1000,500)
    .subscribe((i)=> console.log(i));
```

You will see an output that will be similar to the following; you will have to stop your program or it will keep sending events:

```
0
1
2
```

We can also use this method to generate only one value and finish the sequence. We can do this omitting the `interval` parameter, as shown in the following code:

```
Rx.Observable
    .timer(1000)
    .subscribe((i)=> console.log(i));
```

If you run this code, it will only print `0` in your console and finish.

Creating an observable from callbacks

In JavaScript, we have a lot of APIs that use callbacks to let you control the flow of the application. We can use functional reactive programming to have better control of it. We can use observables to wrap callbacks (following the Node.js callback pattern). This way, we can reuse and compose these callbacks.

To do so, we can use the `fromCallback()` method. It has the following signature:

```
 Rx.Observable.fromCallback(func,[context],[selector]);
```

The first argument is mandatory and the next two are optional. They are as follows:

- `func`: This is the function that usually receives a callback when it finishes
- `context`: This is the context to be used in the callback
- `selector`: This is a function that takes the arguments from the callback to produce a single item to be propagated by this observable

Before showing examples of the usage of this method, let's create our own function accepting a `callback` function, as follows:

```
var myAsyncComputation = function(name,callback){
    setTimeout(()=>{
        callback(null,'Finished computation for '+name);
},100);
};
```

The `myAsyncComputation` variable holds a function that accepts a `callback` function. It calls this `callback` function after 100 milliseconds with a success message: `Finished computation for SOME_NAME`. To test our function, we can just call it in the following way:

```
myAsyncComputation('John Doe',(err,result)=>console.log(result));
```

This will print the `Finished computation for John Doe` message.

Remember, callbacks can receive two parameters: the first is `error` (if any error has occurred) and the second is `result`, representing the result of the asynchronous computation made by the function.

Now let's create an observable for our asynchronous function, as follows:

```
var observableFromCallback =
Rx.Observable.fromCallback(myAsyncComputation);

observableFromCallback('John Doe')
        .subscribe((result)=> console.log(result));
```

If you run this code, you will see the following message printed on the console:

```
    [ null, 'Finished computation for John Doe' ]
```

This happens because as we discussed, a callback receives two arguments: the first is the error (if any) and the second is the successful result. If we don't want to propagate both the values, we can use the selector parameter from the `fromCallback()` method to map the response in a different way:

```
var observableFromCallback = Rx.Observable.
        fromCallback(myAsyncComputation, null, (error,result)=>result);
observableFromCallback('John Doe')
        .subscribe((result)=> console.log(result));
```

Now we added an argument to omit the error, so when we run this code, we will see the following output:

```
    Finished computation for John Doe
```

Creating an observable from a promise

The promises are objects that hold the result of an asynchronous computation. They are becoming more common now. One of the cool things about promises is that you can compose them with other promises or synchronous values, giving you an extra layer of abstraction over asynchronous computations. We can wrap a promise into an observable. This provides us a lot more power to handle the result of our computation using the operators from RxJS.

To do this, we can use the `fromPromise()` built-in method. This method can turn any A+ Promise into an observable.

Promises/A+ is a specification of promises. All the famous promises libraries in JavaScript follow this pattern; it is the same pattern followed by the standard ES6 promise.

This method has the following signature:

```
Rx.Observable.fromPromise(promise);
```

It receives only one parameter:

- `promise`: This is a Promises/A+ object or a factory function that returns an A+ Promise object.

To create an observable from a promise, use the following code:

```
Rx.Observable
    .fromPromise(Promise.resolve('Hello World'))
    .subscribe((result)=> console.log(result));
```

This prints the following output:

```
        Hello World
```

We can also use this method to create an observable from a factory function returning a promise, as you can see in the following code:

```
var promiseFactory = () => Promise.resolve('Hello World') Rx.Observable
    .fromPromise(promiseFactory)
    .subscribe((result)=> console.log(result));
```

When you run this code, you will see the following output:

```
        Hello World
```

Creating empty observables

Sometimes, when composing multiple observables, you might need to create an empty observable. You can do this in RxJS using two different methods. The first one is the `empty()` method; this method will only fire an `onComplete` event.

The following code creates an observable that finishes and doesn't propagate any value:

```
 Rx.Observable.empty();
```

Sometimes you might need an observable that wouldn't emit any value and would never terminate. To create this kind of observable, you can use the `never()` method:

```
Rx.Observable.never();
```

These two methods are used more for composition or as a fallback to prevent the absence of an observable. The only difference between the two is that only the first one terminates.

Creating an observable from a single value

When composing multiple observables or mocking observables for testing purposes, you might need to create an observable that would emit only one value and then terminate itself. RxJS has two methods to implement this behavior: the `return()` method and the `just()` method. They work exactly in the same way and have the following signature:

```
Rx.Observable.return(value, [scheduler]);
Rx.Observable.just(value, [scheduler]);
```

The first argument is mandatory and the second is optional:

- `value`: This can be any object; it is the value to be emitted on the sequence
- `scheduler`: This is used to emit the value

The following code illustrates an example of this method:

```
Rx.Observable
    .just('Hello World')
    .subscribe((i)=> console.log(i));
```

This code uses the `just()` method, but we can also use the `return()` method with a simple change in it:

```
Rx.Observable
    .return('Hello World')
    .subscribe((i)=> console.log(i));
```

If you run any of these code lines, you will see the `Hello World` message printed in your console.

Creating an observable from a factory function

There is a method in RxJS where you can create an observable from a function that returns an observable or an observable factory function; it is called `defer()`. The signature of this method is as follows:

```
Rx.Observable.defer(factoryFunction);
```

It receives only one parameter, and this parameter is mandatory:

- `factoryFunction`: This is a function that returns an observable

The following code shows an implementation of this method:

```
Rx.Observable
    .defer(function(){
        return Rx.Observable.just('Hello World');
    })
    .subscribe((data)=>console.log(data));
```

If you run this code, you will see the following output:

```
Hello World
```

As the name implies, the `defer()` method only calls the factory function when an Observer subscribes to it.

Creating an observable from arbitrary arguments

You have already learned how to create an observable from a sequence using the `from()` method, but there is a way to create an observable from an arbitrary sequence of arguments. We can do this using the `of()` method.

This method creates an observable that emits each argument passed to it.

The `of()` method has the following signature:

```
Rx.Observable.of(...args);
```

The three dots before `args` illustrate an arbitrary number of arguments.

- `...args`: This refers to any number of objects

The following code shows an implementation of the `of()` method:

```
Rx.Observable
    .of(0,1,2)
    .subscribe((i)=>console.log(i));
```

If you run this code, you will see the following output:

```
0
1
2
```

Now let's see what happens if we use the `of()` method with only one parameter on it.

Say you run the following code:

```
Rx.Observable
    .of('Hello World')
    .subscribe((i)=>console.log(i));
```

When you do this, you will see the following output:

```
Hello World
```

As you can see, if you run the `of()` method with only one argument, it works the same way as the `just()` method.

Now you might be wondering what happens if you run the `of()` method without any argument. To check this, let's run the following code:

```
Rx.Observable
    .of()
    .subscribe((i)=>console.log(i));
```

If you run the preceding code, your program will finish without any output in the console. So if you run the `of()` method without any argument, it will have the same behavior as the `empty()` method.

Remember that the `empty()` method creates an observable without any element and ends it, while the `never()` method creates an empty observable that never ends. So creating an observable using the `of()` method without any argument is the same as creating an observable using the `empty()` method.

Creating an observable from an error

We can use an observable to wrap an error; this way, we will have all the operators and compositions of the concerned observable.

RxJS gives us two methods to wrap an error. Both work in exactly the same way and are basically aliases of each other.

The methods to wrap an error to an observable are `throw()` and `throwError()`. We also have a method called `throwException()`, but this method is deprecated and should not be used anymore.

Both methods have the same signature, as follows:

```
Rx.Observable.throw(err,[scheduler]);
Rx.Observable.throwError(err,[scheduler]);
```

The first method is the error you want to wrap around the observable and it is mandatory; the second is optional:

- `err`: This is any object representing an error
- `scheduler`: This is used to generate the sequence of the observable

Lets see a code snippet that uses the `throw()` method:

```
Rx.Observable
    .throw(new Error('AN ERROR HAPPENED'))
    .subscribe((data)=>console.log(data));
```

If you run the preceding code, you will see an error stack trace on your console and your program will finish.

Now let's introduce a minor change in our code.

The `subscribe()` method from RxJS can receive a second function to react to errors, so let's add another function to print the message of the error:

```
Rx.Observable
    .throw(new Error('AN ERROR HAPPENED'))
    .subscribe(
    (data)=>console.log(data),
    (err)=>{
        console.log('Running the subscription function for error');
        console.log(err.message)
    }
);
```

If you run the code, you will see the following error message in your console:

```
Running the subscription function for error
AN ERROR HAPPENED
```

In our example, we passed an `Error` object for the `throw()` method, but what happens when we pass another type of object, such as a number? To answer this question, let's introduce a minor change in our code:

```
Rx.Observable
    .throw(1)
    .subscribe(
    (data)=>console.log(data),
    (err)=>{
        console.log('Running the subscription function for error');
        console.log(err.message)
    }
);
```

If you run the preceding code, you will see the following output:

```
Running the subscription function for error
undefined
```

So as you can see, it doesn't matter what is the object. You can use the `throw()` method to wrap any object as an error. Using another object to represent an error is an anti-pattern, so be careful and always wrap your errors inside an `Error` object.

Never throw a string (or any object) without wrapping it in an `Error` object.

Creating observables from DOM events (or EventEmitter)

One of the most important usages of RxJS in frontend applications is to create observables from user input such as mouse clicks, mouse moves, or keystrokes.

RxJS can bind to DOM elements, jQuery (or Zepto.js) elements, Ember elements, or Angular elements, if any of these libraries are present. RxJS will attempt to detect the libraries automatically.

This method can also be used to bind to a Node.js EventEmitter.

The method `fromEvent()` is used to create observables from DOM elements. This method has the following signature:

```
Rx.Observable.fromEvent(element,eventName,[mapFunction]);
```

The first two parameters are mandatory, and the last one is optional:

- `element`: This represents either the DOM element, jQuery element, Angular element, Ember element, NodeList, or EventEmitter to attach the event
- `eventName`: This is a string representing the event
- `mapFunction`: This is a function that takes arguments from EventEmitter and maps it to a single value

To use the `fromEvent()` method, create an HTML page with a single element, as follows:

```
<html>
    <head></head>
    <body>
        <div id="myDiv">Foo</div>
        <script
src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/4.1.0/rx.all.js"></script>
    </body>
</html>
```

Now let's show an alert every time the user clicks on the word **Foo**. In the first example, let's create an observable from a DOM element.

Add a script tag to your `HTML` file to listen to the click on `<div>`:

```
Rx.Observable
   .fromEvent(document.getElementById('myDiv'),'click')
   .subscribe(function(e){
         alert('Clicked');
   });
```

Open this file in your browser and click on the word **Foo**; once you click on it, an alert will pop up on your screen.

Remember, before adding this code to your `HTML` file, you have to add RxJS as a script to your page.

Now let's use a jQuery element instead of a DOM element. Change your `<script>` tag to use the following code:

```
Rx.Observable
   .fromEvent($('#myDiv'),'click')
   .subscribe(function(e){
         alert('Clicked');
   });
```

Open this `HTML` file in your browser and click on the word **Foo**; once you click on it, an alert will pop up on your screen.

Remember, before adding this code to your `HTML` file, you have to add jQuery as a script to your page.

As you can see, you don't need any changes to make RxJS support the jQuery element; it just works out of the box.

We can also use the `fromEvent()` method on Node.js EventEmitters. To illustrate this usage, let's read a file and print its content to the console using RxJS:

```
var fs = require('fs');
var Rx = require('rx');
var readStream = fs.createReadStream(__filename,'utf8');
Rx.Observable
    .fromEvent(readStream,'data')
    .subscribe((i)=> console.log(i));
```

In this example, we first imported the `fs` module to read files and the RxJS module. Then, we created `readStream` from our own file (the `__filename` variable holds the location of the current file) and we added the string `utf8` to read our file as a string (if we omit this, it will read the file as bytes). And finally, we used the `fromEvent()` method to create an observable from this stream, listening to all the events of the type `data`. We need to subscribe to the observable in order to print the content in the console.

If you run the preceding code, you will see your own program printed in the console.

Now lets make a minor change in our code. If we remove the `utf8` string in the `createReadStream()` method and re-execute the program, we will see an output as follows:

```
<Buffer 76 61 72 20 66 73 20 3d 20 72 65 71 75 69 72 65 28 27 66 73
27 29 3b 0a 76 61 72 20 52 78 20 3d 20 72 65 71 75 69 72 65 28 27
72 78 27 29 3b 0a 0a 76 ... >
```

We already know we can add the `utf8` string as a parameter to make the `createReadStream()` method convert it into a string.

If we haven't, we could use RxJS.

In JavaScript, the buffer object has a method called `toString()` to convert it into a string using the given character encoding. So if I want to convert a buffer into an `utf8` string, all I have to do is write it as follows:

```
var myString = myBuffer.toString('utf8');
```

The `fromEvent()` method can receive a third parameter to map the input from the EventEmitter to another value before propagating it to the listeners.

We can use this method to avoid marking `readStream` as an `utf8` string, and we can do this only in the observable. To do this, use the following code:

```
var fs = require('fs');
var Rx = require('rx');

var readStream = fs.createReadStream(__filename);
Rx.Observable
    .fromEvent(readStream,'data',(chunk)=>chunk.toString('utf8'))
    .subscribe((i)=> console.log(i));
```

Creating an observable from an arbitrary source

In the previous part, we discussed how to implement an observable from an arbitrary source in bacon.js, using the `fromBinder` event. On RxJS, we can also create an observable from any source that we want using the `create()` method. There is also an alias for this method, called `createWithDisposable`.

The `create()` method has the following signature:

```
Rx.Observable.create(sourceFunction);
```

This method accepts only one parameter and it is mandatory:

- `sourceFunction`: This is a function that receives an object capable of pushing data into an observable

We can create an observable with the `create()` method using the following code:

```
Rx.Observable.create(function(source){
    source.onNext(0);
    source.onNext(1);
    source.onNext(2);
    source.onCompleted();
}).subscribe((i)=> console.log(i));
```

If you run this code, you will see the following output:

```
0
1
2
```

We can also use it to send errors to the observable, as you can see in this code:

```
Rx.Observable.create(function(source){
    source.onNext(0);
    source.onNext(1);
    source.onError(new Error('ops'));
    source.onNext(2);
    source.onCompleted();
}).subscribe(
    (i)=> console.log(i),
    (err)=> console.log('An error happened: '+err.message ) );
```

If you run this code, you will see the following output:

```
0
1
An error happened: ops
```

Sometimes when creating an observable, you might need to release some allocated resources (such as database connections or file handlers). To do this, RxJS has a special type of object called Disposable.

The `create()` method lets you define a function (or a Disposable object) to be fired when you no longer want to hold the resources you just allocated.

To do this, all you have to do is return a function (or a Disposable object) and call the

`dispose()`method of the subscription, as seen in the following code:

```
var observable = Rx.Observable.create(function(source){
    source.onNext(0);
    source.onNext(1);
    source.onNext(2);
    source.onCompleted();
    return function(){
        console.log('dispose called: releasing connections or other
resources');
    };
});
var subscription = observable.subscribe(
        (i)=> console.log(i)
);
subscription.dispose();
```

If you run this code, you will see the following output:

```
0
1
2
dispose called: releasing connections or other resources
```

As can be seen, the function returned by the parameter passed to the `create()` method is called when you call the `dispose()` method on the subscription of the observable. We could also return a Disposable object instead of a function to be executed when the `dispose()` method of the subscription is called.

Subscribing to changes (Observer)

To listen to data on an observable, we must call the `subscribe()` method. This method returns a subscription, which we can use later to stop reacting to the incoming data if we are no longer interested in it.

The `subscribe()` method of observables has the following signature:

```
observable.subscribe(onNext,onError,onCompleted);
```

All parameters are optional and can be omitted if we are not interested in this type of event:

- `onNext`: This is a function to be called every time new data is propagated through the observable
- `onError`: This is a function to be called every time an error occurs in the observable
- `onCompleted`: This is a function to be called when the observable is completed

The easiest way to subscribe to an observable is to just pass the `onNext` parameter (as we have been doing in most of our code snippets in this part):

```
Rx.Observable
    .just('Hello World!!!')
    .subscribe((message)=>console.log(message));
```

If you run the preceding code, it will print the `Hello World!!!` message in your console.

We can also add an `onError` function to be called when an error happens:

```
Rx.Observable
    .throw(new Error('ops'))
    .subscribe(
        (message)=>console.log(message),
        (err)=>console.log('An error happened: '+err.message) );
```

If you run this code, you will see the following output in your console:

```
An error happened: ops
```

Lastly, we can also add the `onCompleted()` function to take an action when the observable is completed:

```
Rx.Observable
    .just('Hello World!!!')
    .subscribe(
        (message)=>console.log(message),
        (err)=>console.log('An error happened: '+err.message),
        ()=>console.log('END')
    );
```

If you run this code, you will see the following output:

```
Hello World!!!
END
```

All subscriptions have a method called `dispose()`. This method can be used to stop listening to incoming data in the observable. We can test it in the following code:

```
var subscription = Rx.Observable
        .interval(100)
        .subscribe(
            (message)=>console.log(message),
            (err)=>console.log('An error happened: '+err.message),
            ()=>console.log('END')
```

```
    );

setTimeout(()=>subscription.dispose(),290);
```

As illustrated in the code, we first create an observable that would emit data every 100 milliseconds and we subscribe to it. Then we use the `setTimeout()` function to call the `dispose()` method of this subscription and stop listening to data from the original observable.

If you run this code, you will see the following output in your console:

```
0
1
```

Remember, the `interval()` method will wait for 100 milliseconds before sending the number `0`. Then, it will wait for 100 milliseconds more to send the number `1`.

In RxJS, we also have a special class of objects called Observer. Basically, this object wraps the functions `onNext()`, `onError()`, and `onCompleted()` that you use to subscribe to an observable, giving you an extra layer of abstraction.

To create an `observer`, we can use the `create()` method and then use this `observer` to subscribe to an observable, as you can see in following code:

```
var observer = Rx.Observer.create(
            (message)=> console.log(message),
            (err)=>console.log('An error happened: '+err.message),
            ()=>console.log('END') );

Rx.Observable
    .just('Hello World!!!')
    .subscribe(observer);
```

If you run this code, you will see the following output:

```
Hello World!!!
END
```

Observer is the object responsible for reacting to data sent by an Observable.

RxJS Subjects

Subjects could be both an Observable and an Observer. They can be seen as a pushable Observable; they let you add more data to be propagated through them.

Subjects expose three important methods: `onNext()`, `onError()`, and `onCompleted()`. These methods can be used to send events through the observable sequence. You can see their usage in the following example:

```
var subject = new Rx.Subject();

subject.subscribe(
    (message)=> console.log(message),
    (err)=>console.log('An error happened: '+err.message),
    ()=>console.log('END')
);

subject.onNext('Hello World!!!');
subject.onCompleted();
```

In this example, we created a new `subject` and subscribed to it, using the `subscribe()` method. Then, we pushed data on this `subject` using the `onNext()` method, and we finally finished it calling the `onCompleted()` method.

If you run this code, you will see the following output:

```
Hello World!!!
END
```

We can also propagate an error through a subject using its `onError()` method:

```
var subject = new Rx.Subject();

subject.subscribe(
    (message)=> console.log(message),
    (err)=>console.log('An error happened: '+err.message),
    ()=>console.log('END')
);

subject.onNext('Hello World!!!');
subject.onError(new Error('ops'));
```

First, we propagate the value `Hello World`, which will be printed in the console. Then, we propagate an error. If you run the preceding code, you will see the following output in your console:

```
Hello World!!!
An error happened: ops
```

There is another type of subject called `AsyncSubject()`. It can be used to represent the result of an asynchronous operation. An `AsyncSubject()` can receive only one value, and this value is then cached for all future subscriptions of this subject.

An example of `AsyncSubject()` creation can be seen in the following code:

```
var subject = new Rx.AsyncSubject();

subject.subscribe(
    (message)=> console.log(message),
    (err)=>console.log('An error happened: '+err.message),
    ()=>console.log('END')
);

subject.onNext('Hello World!!!');
subject.onCompleted();
```

As you can see, the only difference to create it–when compared to a Subject–appears on the first line of code, where we use `new Rx.AsyncSubject()` instead of `new Rx.Subject()`. If you run this code, you will see the following output in your console:

```
Hello World!!!
END
```

As discussed earlier, an `AsyncSubject()` must be used to propagate only one piece of data. So let's compare it with a regular Subject.

First, let's see what the output of a Subject would be when we call the `onNext()` method multiple times:

```
var subject = new Rx.Subject();

subject.subscribe(
    (message)=> console.log(message),
    (err)=>console.log('An error happened: '+err.message),
    ()=>console.log('END')
);

subject.onNext(0);
subject.onNext(1);
subject.onNext(2);
subject.onCompleted();
```

If you run this code, you will have the following content printed in the console:

```
0
1
2
END
```

This shows us that the Observer was fired three times (for elements `0`, `1`, and `2`). Now let's change our code to use an `AsyncSubject()`:

```
var subject = new Rx.AsyncSubject();

subject.subscribe(
    (message)=> console.log(message),
    (err)=>console.log('An error happened: '+err.message),
    ()=>console.log('END')
);

subject.onNext(0);
subject.onNext(1);
subject.onNext(2);
subject.onCompleted();
```

If we look at the console after running this code, we will see the following printed:

```
2
END
```

As expected, an `AsyncSubject()` propagates only one element, and for this reason, it fires the Observer once (with two elements).

Lastly, let's see what happens if we don't call the `onCompleted()` method in our Subjects.

First, let's use a regular subject:

```
var subject = new Rx.Subject();

subject.subscribe(
    (message)=> console.log(message),
    (err)=>console.log('An error happened: '+err.message),
    ()=>console.log('END')
);
```

```
    subject.onNext(0);
    subject.onNext(1);
    subject.onNext(2);
```

If we run this code, we will see the following output:

```
0
1
2
```

We can see that a Subject doesn't wait for the `onCompleted()` method to be called to propagate the elements.

Now let's check out the output of `AsyncSubject()`:

```
    var subject = new Rx.AsyncSubject();

    subject.subscribe(
        (message)=> console.log(message),
        (err)=>console.log('An error happened: '+err.message),
        ()=>console.log('END')
    );

    subject.onNext(0);
    subject.onNext(1);
    subject.onNext(2);
```

If we run this code, nothing is printed in the console, so an `AsyncSubject()` will wait for the `onCompleted()` method before it can start propagating the element. So it's easy to see that when dealing with `AsyncSubject()`, we must always call the `onNext()` and `onCompleted()` methods together.

The easiest way to understand a Subject is to see it as a pushable observable.

RxJS Disposable

The `Disposable` class gives us a method to release allocated resources (database connections, file handlers, and so on). We can do this by calling the `dispose()` method of this object.

In this section, we will use the `dispose()` method to unsubscribe from an observable.

To create a `Disposable`, we can use the `create()` function from `Rx.Disposable`:

```
var disposable = Rx.Disposable
        .create(()=>console.log('Releasing allocated resources'));

disposable.dispose();
```

If you run this code, it will print the following message in your console:

```
    Releasing allocated resources
```

As discussed earlier in this part, we can call the `dispose()` method to unsubscribe from an observable. In some cases, we can even define a `Disposable` to be used when someone calls the `dispose()` method from this observable (remember the `create()` and `createWithDisposable()` functions to create an observable).

The `Disposable` class also gives us functions to control groups of `Disposable` objects. The two most important types of Disposable are **CompositeDisposable** and **RefCountDisposable**.

A `CompositeDisposable()` object wraps other Disposables. So when you call the `dispose()` method of the parent object, it will call dispose for all children. You can see this behavior in this example:

```
var firstDisposable = Rx.Disposable
        .create(()=>console.log('disposing first')); var secondDisposable =
Rx.Disposable
        .create(()=>console.log('disposing second'));

var composite = new
Rx.CompositeDisposable(firstDisposable,secondDisposable);

composite.dispose();
```

In this example, we create two Disposable objects (`firstDisposable` and `secondDisposable`) and wrap them under `CompositeDisposable()` (composite). So when we call the `dispose()` method of the `CompositeDisposable()` object (composite), it will call the `dispose()` method of the two children as well (`firstDisposable` and `secondDisposable`).

If you run this code, you will see the following output:

```
disposing first
disposing second
```

The `RefCountDisposable` object does the opposite. It takes a `Disposable` object and lets you distribute references to this Disposable (using the `getDisposable()` method) but protects it from being disposed unless all the references already have been disposed. This is really useful to manage resources being used by multiple Observables/Observers.

This code shows a `RefCountDisposable` usage:

```
var disposable = Rx.Disposable.create(function () {
    console.log('releasing connection');
});

var refCountDisposable = new Rx.RefCountDisposable(disposable);

var firstDisposable = refCountDisposable.getDisposable();
var secondDisposable = refCountDisposable.getDisposable();

firstDisposable.dispose();
secondDisposable.dispose();

console.log('first and second disposed. Disposing refCount');

refCountDisposable.dispose();
```

If you run this code, you will see the following output:

```
first and second disposed. Disposing refCount
releasing connection
```

The output shows that we need to dispose everything to dispose the `RefCountDisposable` children.

Disposables are an important feature to enhance the control of the life cycle of your resources.

RxJS Schedulers

Schedulers are used to determine where a task is going to be executed (current thread, thread pool, and so on).

You can use them to run any type of task that you want, but RxJS Observables use schedulers to propagate data. For this reason, you can optionally define a Scheduler when creating a new Observable.

As you might expect, having control of which execution context you task should run in (or your Observable should process elements in) is specially useful when you are running in a multithread environment.

As RxJS runs in a single-thread environment, other than the default settings, you should be really cautious when using a scheduler to propagate data from an Observable. Incorrect use can block your thread.

In this part, you learned the basics of functional reactive programming using RxJS, and it became clear that it is a more extensive framework.

We started to use different objects such as Observables, Observers, Subjects, Disposables, and Schedulers.

Some of these concepts don't even exist in the bacon.js world, and they give us more power over our code.

With Subjects, you learned how to create an Observable using a push style.

With Disposable, you gained more control over the life cycle of your code, as it lets you tear down your resources gracefully.

With Schedulers, you learned that if you want, you can control in which context your code will be executed, giving you more power over how Observables would propagate data. You also learned the importance of avoiding schedulers, other than the default ones, when using RxJS.