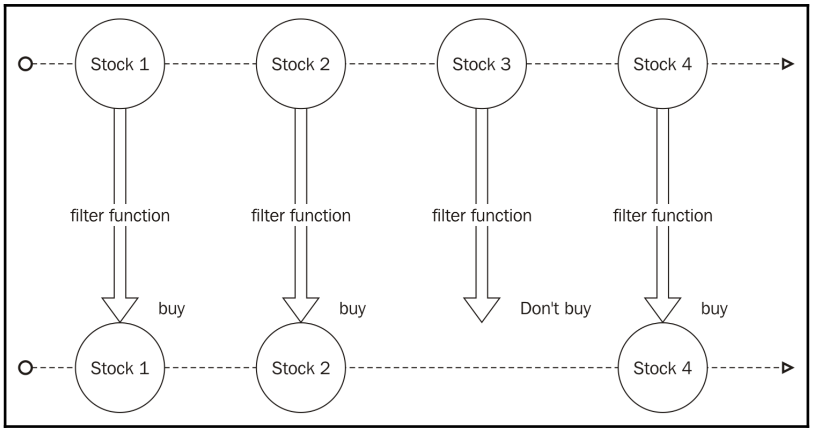
FRP - next level \ part 2

In the previous part, you started understanding the motivations behind using functional reactive programming in your systems; you also saw how a program using this paradigm fared against a program without it. You learned how reactive programming can improve code readability and testability by decoupling your event sources from the action you take when the action occurs.

We started with some basic examples using `bacon.js` as the reactive programming library for JavaScript. In the examples, we began with creating our first EventStream from an interval. Then we started using some operators (`map()` and `take()`). Finally, we subscribed to this event source to take actions in the case of an event occurrence. This was just a kind introduction to functional reactive programming.

When reading most of the functional reactive programming libraries (for any language), you will see a lot of diagrams explaining how the operator works. In the previous part, I presented the following diagram to illustrate a **filter function**:



We will cover what this illustration means in detail in this part. Learning how to read this will make reading documentation on libraries a lot easier.

Functional reactive programming is a paradigm which is hard to master, so after the quick introduction, you'll now move on to learning how to implement more complex programs using the `bacon.js` library.

This part will cover the following points:

- Understanding bacon.js observable objects (EventStream and Property) and their differences
- Modeling a functional reactive program
- Subscribing to an observable
- Unsubscribing from an observable
- Reading a reactive programming operator diagram
- Using operators to transform an observable

The bacon.js observables

In functional reactive programming, an observable is an object where you can listen for events. This way, you can, for instance, create an observable for a button and then listen and act when a click happens.

The bacon.js gives you two flavors of an observable: the first one is EventStream and the other is Property. We will see the difference between the two objects later. To listen to events in an observable (or subscribe to an observable), you can use the `onValue()` method with a callback. So if you want to log every event in an EventStream, you can use the following code:

```
myEventStream.onValue(function(event){
    console.log(event);
});
```

As we saw in the example in the last part, we can transform our observable using bacon.js operators. These operators let us filter, combine, map, buffer, and do a lot of other interesting things with our EventStream.

An observable can either finish or stay open to propagate events forever; it can also contain (and propagate) errors. We will see this in more detail later.

Creating our first observable

The bacon.js gives us a multitude of methods to create observables. We can create them from DOM events, promises, interval, and so on. Sometimes we might need to create an observable from our own source, so let's learn how to create our own event streams.

Observables from DOM events (asEventStream)

To create an EventStream from DOM events (a mouse click for instance), we will need an HTML page with jQuery (or **Zepto.js**); bacon.js adds the `asEventStream()` method for all jQuery objects. So if we want to create an EventStream from a button click, we can use the following code:

```
var clickEventStream = $('#myButton').asEventStream('click');
```

This line creates an EventStream from button clicks on a DOM object with the ID `myButton`. If we want to execute an action every time this button is clicked, we will need to use the `onValue()` method from this event stream. The following code shows an alert on the screen every time the user clicks on the button:

```
clickEventStream.onValue(function(){
    alert('Button clicked');
});
```

This code adds a function to be called every time an event happens in this EventStream object, as it emits an event every time `myButton` is clicked. The following code will show an alert every time this button is clicked.

As this is your first HTML code, I will paste the full HTML here so you can create a file with it and test it in your own browser:

```
<html>
  <head></head>
  <body>
    <button id="myButton">CLICK</button>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js"></s
cript>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/bacon.js/0.7.88/Bacon.min.js"><
/script>
    <script>
      var clickEventStream = $('#myButton').asEventStream('click');
      clickEventStream.onValue(function(){
        alert('Button clicked');
      });
    </script>
  </body>
</html>
```

Observables from promises (fromPromise)

You can also create EventStreams from ES6 promises or jQuery AJAX. These streams will contain only a single value (or an error in case of failure), and finish. We will need a promise to test this method:

```
var promiseObject = Promise.resolve(10);
```

Here, we are creating a successful promise that will return the value `10`. To transform this promise into an EventStream, we use the following code:

```
Bacon
   .fromPromise(promiseObject);
```

Observable node event emitter (fromEvent)

Lots of Node.js modules implement an event emitter, which is basically an object with the `on()` method that is called every time an event happens. We can transform these objects into EventStreams of bacon.js to take advantage of all the transformations. The method `fromEvent()` has the following signature:

```
Bacon
   .fromEvent(eventEmitter,eventName);
```

The parameters are `eventEmitter`; the node `eventEmitter` object; and `eventName`, the name of the event. We can use the `fs` module to show the usage of this method.

To read a file using the `fs` node module that uses an event emitter, we need to create a `readStream` for the file using the following code:

```
var fs = require('fs');
var FILE_PATH = 'SOME FILE';
var readStream = fs.createReadStream(FILE_PATH,'utf8');
```

Then, subscribe to the events on this emitter using the `on()` method, as follows:

```
readStream.on('data',(content)=>console.log(content));
```

To transform this code into a code using a bacon.js EventStream, we need to change the last line of the following code:

```
var eventStream = Bacon.fromEvent(readStream,'data');
eventStream
    .onValue((content)=> console.log(content));
```

So comparing with the method signature, our `eventEmitter` parameter is the `readStream` object and `eventName` is `data`.

The `fromEvent()` method can also be used to listen to DOM events (similar to the `asEventStream()` method), as you can see in the following code:

```
var clickEventStream =
Bacon.fromEvent(document.getElementById('#myButton'), 'click');
```

Observables from an array (fromArray)

To create an observable from an array, you can use the `fromArray()` method, as shown in the following code:

```
var myArray = [1,2,3];
Bacon
   .fromArray(myArray);
```

Observables from an array (sequentially)

To create an observable from an array with a given interval to deliver each item from the array, we can use the `sequentially()` method, as follows:

```
var myArray = [1,2,3];
var intervalBetweenItens = 100;
Bacon
   .sequentially(intervalBetweenItens ,myArray);
```

This code emits the itens of the array with `100` milliseconds of interval between each emission.

Observables from an interval (interval)

We can create an observable from an interval; this is especially useful if you wish to execute repetitive tasks, and it can be used as a substitution for the `setInterval()` method. An EventStream created by the `interval()` method will never end; this method has the following signature:

```
Bacon.interval(intervalInMilliseconds);
```

It receives only one parameter, that is the interval between the events in milliseconds, and it will emit an empty object as the event. Let's say you have run the following code:

```
Bacon
    .interval(100)
    .onValue((event)=>{
        console.log(event);
    });
```

If so, you will see the following output:

```
{}
{}
{}
{}
```

It will keep printing an empty object until you kill the program.

Observables from other sources

The bacon.js has other built-in methods to create observables from common EventStream sources. The objective of this part is not to explain all the functions of the bacon.js API. You can check out other possible EventStream sources by referring to bacon.js documentation. The last important method to create an EventStream is the method that lets you create an EventStream from any arbitrary source. This is the `fromBinder()` method.

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

```
Bacon.fromBinder(publisher);
```

It receives a `publisher` function as a parameter; this is a function with only one parameter that lets you push events for your EventStream, as illustrated in the following example:

```
var myCustomEventStream = Bacon.fromBinder(function(push){
   push('some value');
   push(new Bacon.End()); });
```

The following code creates an EventStream that emits a string and then finishes the EventStream. As you can see, `Bacon.End()` is a special object that tells bacon.js when to close the EventStream.

The following example sends more events to the EventStream and prints them to the console:

```
var myCustomEventStream = Bacon.fromBinder(function(push){
    push('some value');
    push('other value');
    push('Now the stream will finish');
    push(new Bacon.End()); });
myCustomEventStream
   .onValue((value)=>
       console.log(value)
   );
```

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

```
some value
other value
Now the stream will finish
```

When we want to propagate an error on our observable, we need to use the `Bacon.Error()` type to wrap the error. The bacon.js is not capable of automatically encapsulating a thrown exception; you always need to encapsulate it manually. It is also important to notice that when an error occurs, bacon.js doesn't end your observable (but you can configure it). Let's change the previous code to add an error as follows:

```
var myCustomEventStream = Bacon.fromBinder(function(push){
    push('some value');
    push('other value');
    push(new Bacon.Error('NOW AN ERROR HAPPENED'));
    push('Now the stream will finish');
    push(new Bacon.End());
});
myCustomEventStream
    .onValue((value)=>
        console.log(value)
    );
```

It will give you the same output and ignore your error, as you haven't added a handler for errors in this observable (we will see later how to add a handler using the `onError()` method). But if we want to finish the observable as soon as an error happens, we can call the `endOnError()` method to our observable, as you can see in this code:

```
var myCustomEventStream = Bacon.fromBinder(function(push){
    push('some value');
    push('other value');
    push(new Bacon.Error('NOW AN ERROR HAPPENED'));
    push('Now the stream will finish');
    push(new Bacon.End());
}).endOnError();
myCustomEventStream
   .onValue((value)=>
       console.log(value)
   );
```

It will give you the following output:

```
some value
other value
```

As you can see, it never prints `Now the stream will finish` because the error happened before it.

Properties

Now you know what an EventStream is and how to create it; as discussed, it is an observable. However, bacon.js has another special form of observable called Property. A Property is an EventStream with the concept of current value. You can create a Property from any EventStream using the `toProperty()` or `scan()` method; it is especially useful to represent DOM data binding.

To create a Property from an EventStream, you can use the `toProperty()` method that has the following signature:

```
eventStream.toProperty(initialValue);
```

The `initialValue` parameter is optional. If you decide to omit it, you will have a Property without an initial value. If you pass it, it will be used as the current value of this Property until the first value is emitted from the stream.

The other way to create a Property is using the `scan()` method. This method is similar to the `reduce()` method from the JavaScript array object. Given a seed object and accumulator function, the `scan()` method will iterate over your EventStream to create a Property from the result:

```
Bacon
    .sequentially(100,['a','b','c','d'])
    .scan('=> ',(acc,b)=> acc+b)
    .onValue((value)=>console.log(value));
```

In this example, we first create an EventStream from an array of strings, then we call the `scan()` method to create a Property with a seed value of `=>` , and using a function to concatenate these value, we print the result to the console for each scan iteration. So running this code will give the following output:

```
=>
=> a
=> ab
=> abc
=> abcd
```

Changes in an observable

In the last section, we saw how to use the `scan()` method to create a Property from an EventStream. When we use this method (actually, when we use any operator), we do not change the original observable; we always create a new observable. So, it never interferes with the other subscriptions (and transformations) of the observable. We can change the previous code to listen to events on both the EventStream and the Property; now we will see that making changes in one of these two doesn't affect the other:

```
var eventStream = Bacon
    .sequentially(100,['a','b','c','d']);

eventStream.onValue((value)=>{
    console.log('From the eventStream :'+value);
});

var property = eventStream.scan('=> ',(acc,b)=> acc+b);

property.onValue((value)=>{
    console.log('From the property :'+value);
});
```

It gives the following output:

```
From the property :=>
From the eventStream :a
From the property :=> a
From the eventStream :b
From the property :=> ab
From the eventStream :c
From the property :=> abc
From the eventStream :d
From the property :=> abcd
```

This shows that the usage of the `scan()` method to concatenate the array in the Property does not change the behavior of the original `eventStream`.

Reacting to changes

As discussed earlier, an observable is an object where you can listen to events. The act of listening to events in this object is called **subscription**. Here, we will see the different ways in which we can subscribe to an observable and also how we can stop listening to events in this observable (unsubscribe).

Subscribing

We call subscribing to an observable an act of adding a function to be called when an event happens. Using bacon.js, we can be notified when a value is emitted (the `onValue()`, `log()`, and `assign()` methods), when an error has occurred (the `onError()` method), or when our observable is closed (at the end).

Subscribing using the onValue() method

The most common way of subscribing to an observable is using the `onValue()` method. This method has the following signature:

```
observable.onValue(functionToBeCalledWhenAnEventOccurs);
```

So let's subscribe to `eventStream` to log every event on this stream, as follows:

```
Bacon
    .fromArray([1,2,3,4,5])
    .onValue((number)=>console.log(number));
```

This code gives you the following output to the console:

```
1
2
3
4
5
```

This function can be used for any type of observable (EventStream and Property). The only difference is that in Properties, if the initial value of the Property exists, then it triggers the `onValue()` function. Check out the following code:

```
var initialValue =0;
Bacon
    .fromArray([1,2,3,4,5])
    .toProperty(initialValue)
    .onValue((number)=>console.log(number));
```

This will give you the following output:

```
0
1
2
3
4
5
```

When subscribing to an observable, it's important you know a way to unsubscribe it as well. The `onValue()` method returns a function; this function when called will unsubscribe your function from this observable. So we can create an observable from an interval and print a message every time an event is propagated, as follows:

```
Bacon
    .interval(1000)
    .onValue(()=>(console.log("event happened")));
```

It will print the message `event happened` every second until we kill the process. But if we want, we can use the return of `onValue()` to unsubscribe from our observable after a certain amount of time, as follows:

```
var unsubscribe = Bacon
        .interval(1000)
        .onValue(()=>(console.log("event happened")));

setTimeout(function(){
    console.log("unsubscribing")
    unsubscribe();
},4000);
```

With this code, the program will unsubscribe from your observable. This way, it exits normally and prints the following output:

```
event happened
event happened
event happened
unsubscribing
```

Subscribing using the log method

So far, every time we created an observable, we used the `onValue()` method to listen to events in the observable. In our examples, we usually just printed the value to the console,as this is common usage when testing observables and operators bacon.js has a special method through which you can print all the events to the console. All observables have the `log()` method. This method prints every event to the console and prints the `<end>` string when the event stream finishes. We can use it with EventStreams:

```
Bacon
    .fromArray([1,2,3,4,5])
    .log();
```

This code gives you the following output to the console:

```
1
2
3
4
5
<end>
```

As you can see, after all the events, it will print the `<end>` string to indicate the end of the EventStream. If we decide to use it with an infinite stream (a stream created with the `interval()` method), it will never print the `<end>` string (as you should expect). Refer to the following code:

```
Bacon
    .interval(100)
    .log()
```

This will print the following output:

```
{}
{}
{}
{}
```

It will keep on printing until you close the program.

We can also use the `log()` method with a Property, as you can see in the following example:

```
var stringProperty = Bacon
        .fromArray(['a','b','c','d'])
        .scan('=> ',(acc,b)=> acc+b);

stringProperty.log();
```

This will print the following output:

```
=>
=> a
=> ab
=> abc
=> abcd
<end>
```

As you can see, this prints the `<end>` string for a Property as well.

The `log()` method is especially useful for debug purposes and to test and see how an operator works. It is the fastest way to see an operator in action.

Subscribing using the assign method

The `assign()` method lets you call a function on an object every time an event occurs. It is especially useful to set the content of DOM elements. You can use it to set the value of a DOM object class using jQuery:

```
observable
   .assign($("#myElement"), "class");
```

The `assign()` method is just a synonym for the `onValue()` method.

Acting when an error occurs

In the previous sections, we saw how to encapsulate an error on bacon.js. Now, we will use the `OnError()` method to take an action whenever this happens. This method has the same signature as that of the `onValue()` method and works similarly, but only for bacon errors. Now with this method, we can change the code we used in the previous section to print an error when it occurs. This is shown as follows:

```
var myCustomEventStream = Bacon.fromBinder(function(push){
   push('some value');
   push('other value');
   push(new Bacon.Error('NOW AN ERROR HAPPENED'));
   push('Now the stream will finish');
   push(new Bacon.End()); }); myCustomEventStream
   .onError((value)=>
       console.log(value)
   );
```

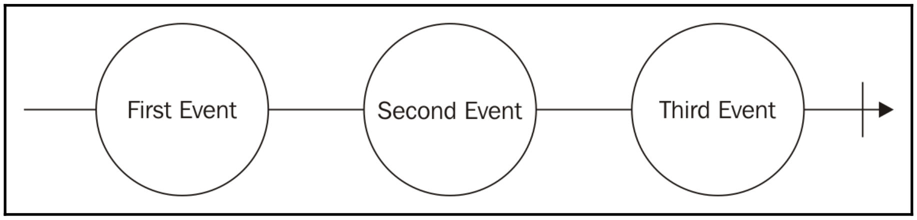
It will print the following:

```
NOW AN ERROR HAPPENED
```

As you might expect, the `OnError()` method also returns a function to unsubscribe.

Reading how an operator works

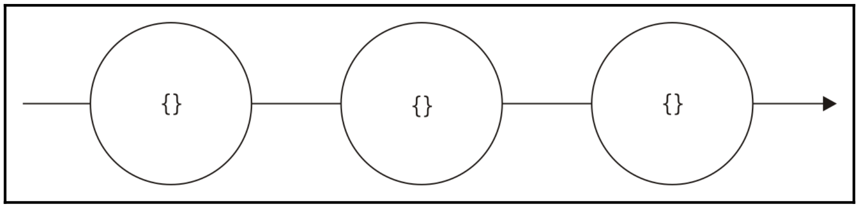
Throughout this part, you will see a lot of diagrams explaining how an operator works. These diagrams are a graphical representation of observables and operations in those observables. Usually, the diagrams consist of three parts. On the top they shows a line with balls representing the initial state of an observable. The line itself represents an observable. The circles are events that happened in this observable and they are pushed from left to right; therefore, the leftmost ball is the first event, the second one is the second, and so on. This is illustrated in the following diagram:



So, the preceding line represents an observable with three emitted events. The string at the center of each circle is the value emitted by that event. The short vertical line on the right-hand side of the diagram represents the end of this observable. This diagram is a graphical representation of the following observable:

```
Bacon
   .fromArray([
         'First Event',
         'Second Event',
         'Third Event'
   ]);
```

As we know, an observable can exist forever (the ones created using the `Bacon.interval()` method). To represent this kind of observable, we remove the short line on the right-hand side. This way, we show that other events might happen in this observable in the future, but we are not interested in them, , as shown in the following diagram:

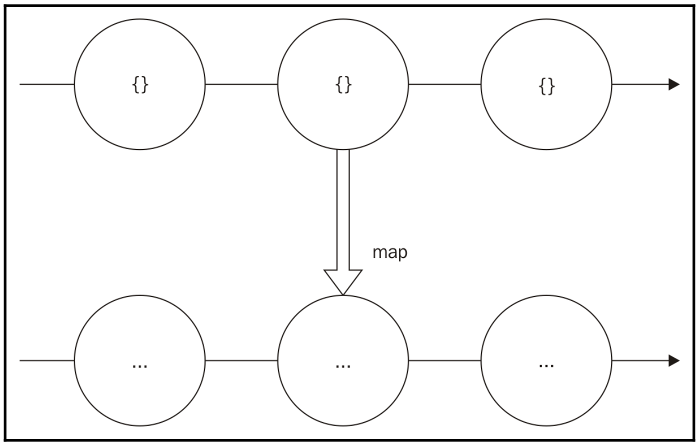


This diagram is a possible representation of the following observable:

```
Bacon
   .interval(1000);
```

Keep in mind, the time between events is irrelevant in this case, so we decided to omit it from the graphical representation (we usually omit this information, as this is an implementation detail). The `Bacon.interval()` method always emits an empty object, and for this reason, we draw brackets at the center of each circle.

The main reason for using this graphical representation is to show how an operator transforms an observable. So to show which operator is being used, we add an arrow pointing downward with the name of the operator and the content of the operator if needed. So if we want to represent the use of the `map()` operator, we can use the following diagram:



As discussed before, when an operator is applied to an observable, it creates a new observable (it does not change the original observable). For this reason, we add a new line that represents the new observable created after the operation. On this new line, I decided to omit the value at the center of the circle (adding only three dots), because the way the `map()` operator works is out of the scope of this discussion.

Sometimes we want to represent multiple operators. In such cases, we just keep chaining more lines. Some operators let you work with multiple observables, which makes the diagram more complex. But don't worry, we will explain the changes the first time they happen.

Transforming events using bacon.js

One important thing when learning functional reactive programming is how you can transform the events emitted by an observable. This way, you can use successive function calls to create new objects from the original input. This also improves the reuse of your code; every transformation of an observable creates a new observable, and each observable can have several listeners subscribed to it.

Before applying any transformations to our observables, let's implement an observable to generate and print the current date. To do this, let's use the `Bacon.interval()` method. So, the following code will emit an empty object every second:

```
var eventSource = Bacon
    .interval(1000);
```

Remember, `Bacon.interval()` emits an empty object every *x* milliseconds, where *x* is the argument passed to the function–in our example, every `1000` milliseconds, which is the same as every second.

Now we can just subscribe a function to print the current date to the console. We can do this using the `onValue()` function by adding the following line:

```
eventSource
    .onValue(()=>{
        console.log(new Date());
    });
```

This will keep printing an output as follows:

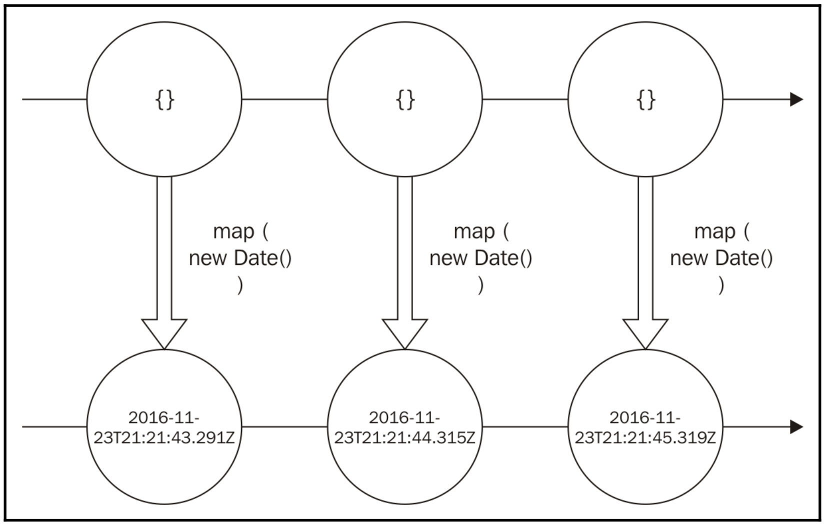
```
2016-11-23T21:21:43.291Z
2016-11-23T21:21:44.315Z
2016-11-23T21:21:45.319Z
```

It works fine and solves our problem, but remember the problem–We wanted an observable to generate and print the current date to the console. Unfortunately, bacon.js doesn't have any observable that could generate the current date every given second, but we can use an operator to change the empty object to the current date. The return of this operator will be a new observable that will emit the current date every second, which is exactly what we wanted from the beginning. Then, we can use the `onValue()` method to subscribe to this operator in order to print the current date.

Luckily, such an operator exists and it is one of the most commonly used operators as well. It's called the `map()` operator. It lets you add a function to map an event emitted in another object. It has the following signature:

```
eventStream.map(handler);
```

The `handler` in the signature is a function that receives the emitted event and returns a new object to substitute the original event (so basically, we **map** an input to an output). You can see this illustrated in the following diagram:



This diagram describes exactly how we want our observable to work. We use the `Bacon.inverval()` method to generate a new observable that will emit the current date using the `map()` function. This function is executed every time an object is emitted.

Looking at this diagram makes it a lot easier to understand how we can use the `map()` function to implement the desired behavior. We will need to do some minor changes in our original code to create a new EventStream using the `map()` operator, and change our subscription to only log the emitted event. So, the final implementation of the described diagram using the `map()` operator is as follows:

```
Bacon
    .interval(1000)
    .map(
    (i)=> new Date()
)
.onValue((date)=>console.log(date));
```

This gives you the same kind of output as from the previous code:

```
    2016-11-23T21:21:43.291Z
    2016-11-23T21:21:44.315Z
    2016-11-23T21:21:45.319Z
```

In this section, I want to show you an example of reusing an `eventStream`. To do this, we will slightly change our original problem. Now, we want to print only those dates where the seconds are even. To do this, we will need a new operator: the `filter()` operator. This operator lets you create a new observable by omitting some events that you are not interested in. It has the following signature:

```
eventStream.filter(handler);
```

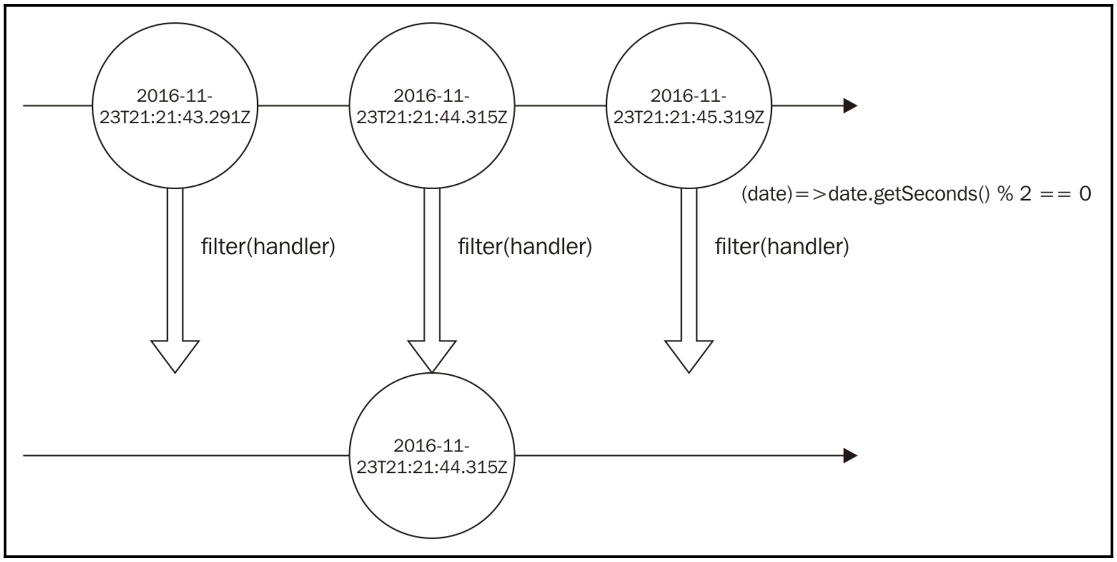
Here, `handler` is a function that receives the emitted event as an input and returns `true` or `false` (actually, any truthy or falsy value). So, because we want to print only the dates where the seconds are even, we will pass the following handler as an argument to the `filter` operator:

```
(date)=>date.getSeconds() % 2 == 0
```

As you can see, this function receives the current date as an argument and gets the seconds part of this date. Also, it uses the `mod` operator to decide whether it is an even number or not.

Notice that one of the advantages of this approach is the creation of small functions to implement each part of our code. This makes our code more testable, as we can easily create unit tests for each of these functions.

Now, to implement this new observable that emits only even dates, all we need to do is chain the `filter()` function to our previous observable in order to create a new observable that we can subscribe to finally print what we want. You can see the use of this `filter()` operator in the following diagram:



As you can see in this diagram, on the created observable, we don't emit the dates where the seconds are odd numbers. For this reason, we don't have their circles. Now, let's change our code using the `map()` function to use the `filter()` operator as well. This can be easily done with the following code:

```
Bacon
    .interval(1000)
    .map(
    (i)=> new Date()
)
.filter(
    (date)=>date.getSeconds() % 2 == 0
)
.onValue((date)=>console.log(date));
```

If you run this code in a node program, you will see a result like this:

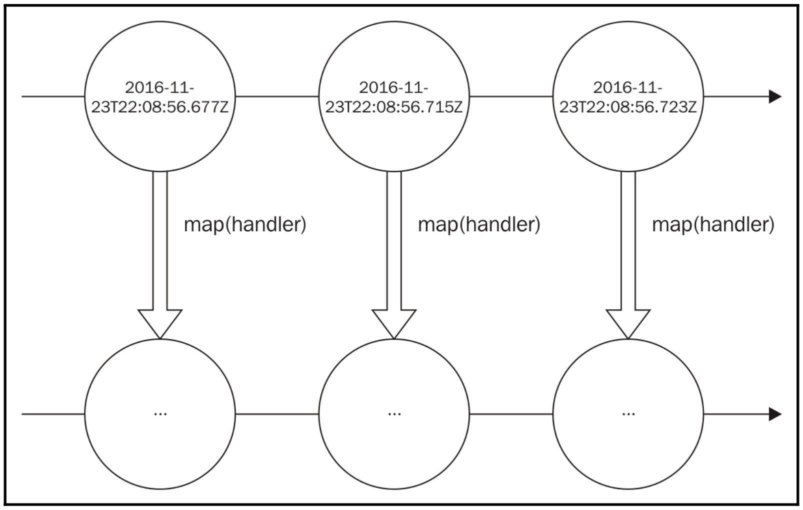
```
        2016-11-23T22:08:56.677Z
        2016-11-23T22:08:58.715Z
        2016-11-23T22:09:00.723Z
```

Notice that our new output contains only dates with even seconds.

This output is a little confusing. It might be hard to see that it prints only dates with even seconds (because the last part is milliseconds and not seconds). So, let's make a last change in our code. As you might expect, we can use the same operator we used before. To show this, let's use the `map()` operator again to generate a string from the date. Now we want to change our output to something like this:

```
        The number in the second part of the date XXX is YYY which is as
      even number
```

To do this, let's chain our filtered observable with a `map()` operator to generate this string for each event in the observable. The new observable can be represented by the following diagram:



The `handler` function will be used to generate the final string from the date. As the result string is big, I decided to omit it in the result circle, but it will follow the pattern I've described. So, first our `handler` function on the `map()` function should be as follows:

```
(date)=> 'The number in the second part of the date ' +
    date.toISOString() + ' is ' + date.getSeconds() +' which is as even
number'
```

Now, all we have to do is change our last observable with the `map()` operator. We can do this as follows:

```
Bacon
    .interval(1000)
    .map(
    (i)=> new Date())
.filter(
    (date)=>date.getSeconds() % 2 == 0)
.map(
    (date)=> 'The number in the second part of the date ' +
    date.toISOString() + ' is ' + date.getSeconds() +' which is as even
number'
)
.onValue((date)=>console.log(date));
```

Running this code will display the following output, which accentuates the seconds part, making it easier for us to see that it really is an even number:

```
The number in the second part of the date 2016-11-23T22:29:42.694Z
      is 42 which is as even number
The number in the second part of the date 2016-11-23T22:29:44.749Z
      is 44 which is as even number
The number in the second part of the date 2016-11-23T22:29:46.757Z
      is 46 which is as even number
```

The bacon.js has a lot of built-in operators. You can see their descriptions in their APIs. In this section, we only wanted to show how you can use operators to create new observables. We will see a few more operators in depth in the parts which follows, using RxJS.

Reusing observables

In the previous section, we saw how we can use operators to create new observables, but until this moment, we have never tried reusing an observable. To do this, I propose a change in the program we created in the previous section. We still want to print the string for every date where the seconds part is an even number. But now, let's also print the string `a second has passed` every second. To do this, we can use another `Bacon.interval` call to generate a new observable that emits an empty object every second. Once this is done, we can map each object in the observable to the string we want and finally subscribe to print, as you can see in the following code:

```
Bacon
    .interval(1000)
    .map(
    ()=> 'a second has passed'
)
.onValue((str)=> console.log(str));
```

If you add the preceding code to the last program, you will see the following output:

```
    a second has passed
    The number in the second part of the date 2016-11-23T22:42:18.683Z
         is 18 which is as even number
    a second has passed
    a second has passed
    The number in the second part of the date 2016-11-23T22:42:20.696Z
         is 20 which is as even number
    a second has passed
    a second has passed
    The number in the second part of the date 2016-11-23T22:42:22.702Z
         is 22 which is as even number
```

This solution is okay, but we can reuse the first `Bacon.interval()`, storing it in a variable. This way, we don't have to keep repeating this code every time we want a source of events emitting every second, using a variable as follows:

```
var emitsEverySecondStream = Bacon.interval(1000);
```

We can create better code by reusing this variable as follows:

```
var emitsEverySecondStream = Bacon.interval(1000);
emitsEverySecondStream.map(
    (i)=> new Date()
)
.filter(
    (date)=>date.getSeconds() % 2 == 0
)
```

```
.map(
    (date)=> 'The number in the second part of the date ' +
    date.toISOString() + ' is ' + date.getSeconds() +' which is as even
number'
)
.onValue((date)=>console.log(date)); emitsEverySecondStream.map(
    ()=> 'a second has passed'
)
.onValue((str)=> console.log(str));
```

This code gives us the same kind of output as from the previous example.

Another important feature when reusing observables is the ability to add multiple subscribers to the same observable. You can see this happening in the following code:

```
var eventStream = Bacon
        .interval(1000)
        .map(
        ()=> 'a second has passed'
   ); eventStream
    .onValue((str)=>console.log('Subscriber 1 prints => ' + str));
eventStream
    .onValue((str)=>console.log('Subscriber 2 prints => ' + str));
eventStream
    .onValue((str)=>console.log('Subscriber 3 prints => ' + str));
```

In this code, we added three subscriptions to the same EventStream. This will give us the following output:

```
Subscriber 1 prints => a second has passed
Subscriber 2 prints => a second has passed
Subscriber 3 prints => a second has passed
Subscriber 1 prints => a second has passed
Subscriber 2 prints => a second has passed
Subscriber 3 prints => a second has passed
```

Remember that the `onValue()` function returns a function to unsubscribe that function from the observable. So if you want, you can unsubscribe from each subscription individually without affecting other subscriptions.

Observables' lazy evaluation

In bacon.js, an observable doesn't emit any event unless someone is subscribed to listen to it. We can easily check this behavior using `doAction()`. This operator lets us run an arbitrary function every time an event is emitted from an observable.

If you create an observable from an array and add a `doAction()` operator to log your events then, without a subscription, it will print nothing to the console. So if you run the following code, you will see your program doesn't give you any output:

```
Bacon
    .fromArray([1,2,3])
    .doAction((value)=>
        console.log('running doAction')
    );
```

As you can see in this code, we have no subscriber for the EventStream. This is why no event is emitted. However, if we add a subscriber to this EventStream, we will see the program print `running doAction` for each value:

```
Bacon
    .fromArray([1,2,3])
    .doAction((value)=>
        console.log('running doAction')
    )
    .log();
```

The preceding code gives you the following output:

```
running doAction
1
running doAction
2
running doAction
3
<end>
```

It is really important to keep this behavior in mind because ignoring this can lead to bugs that are really hard to identify.

In this part, you learned the basics of functional reactive programming using the `bacon.js` library. We wanted to give you the taste of functional reactive programming and also get your hands dirty with a lot of different examples.

You learned about observables and the different types of observables that bacon.js has implemented. You also learned about some of the built-in methods that help you create observables in bacon.js and also how you can create your own observable using the `fromBinder()` method.

Apart from this, you also learned how to subscribe and unsubscribe from an observable, how to use operators to transform an observable, and the importance of reuse in this context.

In the next part, we will start using RxJS as it gives us more tools for functional reactive programming. We will see how it compares with bacon.js and understand the basics of this new tool.