1)RxJS – What and Why?

In official webpage, it is written-

RxJS is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code. Path is-

<http://reactivex.io/rxjs/>

it is not clear, so lets use it. On above link , click on install it. For now I wnt use npm. Later we will use it. Scroll to very bottom to pick up a CDN. Pick last one bcoz its later. Now from CDN remove @version and paste link in browser URL bar. Then hit enter. You will get CDN for latest version.

We write some code in jsbin(similar example can also be found on official page, click on learn it link in home page)-

Html-

<body>

<button>Click Me</button>

<script src= "https://unpkg.com/@reactivex/rxjs/dist/global/Rx.js"></script>

<script src= "app.js"></script>

</body>

Js code-

**var button = document.querySelector('button');**

**button.addEventListener('click', (event) => console.log(event));**

when I click on button, event object is printed on console.

now lets rebuild it using rxjs-

var button = document.querySelector('button');

Rx.Observable.fromEvent(button,'click')

.subscribe((event) => {

console.log(event);

});

Here we use Rx package which is avalaible due to import. Then I use observable object, then a helper method- fromEvent. This method creates a new observable based on an event. So here I specify event source, button, where I want to listen to event. then type of event- click. With that I got my observable. But nothing will happen now. To react to event, I need to subscribe to an observable. To subscribe we can pass 3 functions , here we will pass only 1.

Now when we click on button, same result as we saw with previous code.

In first look this second setup looks complicated. Why should we use it? Because with observables, you can already kind of see here, we have funnel like setup, our event data travels from top to bottom, so from data source which is our click event on button, to code we execute in subscribe function. here funnel is very short we only have subscribe function but powerful thing about observables and why they are so useful and used that much in angular are operators ,which allow you to transform the way this data is handled, used or looks like immensely.

Lets say we want to make sure that we only react to click events once a second, so if I spam this click button, we omit or we drop all event pieces, all events which have more than once a second.

In vanilla js we have to use this code-

var count = 0;

var rate = 1000;

var lastClick = Date.now() - rate;

var button = document.querySelector('button');

button.addEventListener('click', () => {

if (Date.now() - lastClick >= rate) {

console.log(`Clicked ${++count} times`);

lastClick = Date.now();

}

});

now with observable we can use built in operator – **throttleTime** for this. It takes number as argument, which is milli second that we want to have as a time window. Code-

var button = document.querySelector('button');

Rx.Observable.fromEvent(button,'click')

.throttleTime(1000)

.subscribe((event) => {

console.log(event);

});

So that’s the thing with observable, it is very easy to add this functionality with one of the many,many,many operators.

So that’s the advantage of observables-

This funnel like approach which makes it very easy to write structured statements handling asynchronous code and operators we can use in funnel to transform the data.

Lets consider a new case-

Lets say I want to print X coordinate of click event. We can do this by-

var button = document.querySelector('button');

Rx.Observable.fromEvent(button,'click')

.throttleTime(1000)

.subscribe((event) => {

console.log(event.clientX);

});

Now whenever we click on button, we get 23,45,78 printed. They are X coordinate.

Now lets say this is very long code and we are using this x coordinate and suddenly we decided to use Y coordinate instead of X. it is very difficult to change in many places.

Another approach is we can add new operator to party – **map,** operator.

We add it after throttleTime, ordering is very important here, it will also receive new event once in a second.

Map works in a way that it takes a function as a input. this function then returns a new observable. This function gets Observable data(here event data) as input, then it has to return something which will then automatically be wrapped into new observable(since we can only subscribe to an observable, and subscribe function works on result of map).

var button = document.querySelector('button');

Rx.Observable.fromEvent(button,'click')

.throttleTime(1000)

.map((data) => data.clientY)

.subscribe((event) => {

console.log(event);

});

From this function passed to map we return data. clientY for example, in subscribe I know I will get a coordinate, which one it’s not important to me here, previous operator sort it out. We just get our coornidate in subscribe function.

If we run our code it works fine. So we got throttling in place , we got map. This is power of operators, this is the power of writing it in structured way and looking at it like a funnel.this is how you can think of RXJS and observable, we have this funnel like approach for handling asynchronous code, which is very powerful and which might give you more features and power than normal vanilla js or promise using ways of handling asynchronous code gives you.

# 2)OBSERVABLES, OBSERVERS & SUBSCRIPTIONS

**Observable is kind of wrapper around a data source, data source typically means stream of values**, because you might be aware observables are concept we mostly use for asynchronous data but it is not limited to it. You can also wrap it around any synchronous data source, we will see that in this video.

So we have stream of data possibly asynchronous or possible multiple values over time. Now we want to do something whenever new value occurs. This is job of observer. Observer is there to execute some code whenever we receive a new value or error or when observable reports that it is done. Therefore observer is part doing that and we need to connect it with observable. we do this through subscription. Subscription basically means, with one method(subscribe method ) we tell observable(wrapper around stream of values) that someone is caring about these values, someone is listening to them – the Observer.The observer on other hand implements upto 3 methods. we can implement some of these or none of these.

**Next** method will be called whenever a new value is emitted. Whenever we receive new value

**Error** method is executed whenever observable throws an error

**Complete** is executed when observable is done, whenever we know that no more value will be emitted in future. As a side note, some observables never finish. Like we wrap it around ,click listen on button, user can anytime click on button.

But how does the observer know that it should call next, error and complete? Well it is kind of contract the two Observable and Observer sign through subscription. the observable knows that it could fire a next, complete or error method on an observer and observer on the other hand knows that observable will only fire one of these 3 methods. so you can easily implement them on observer and react whenever they are fired. So it is this invocation of these methods which allows us to communicate and handle our data. but depicting in this way (figure 1)is not best way.

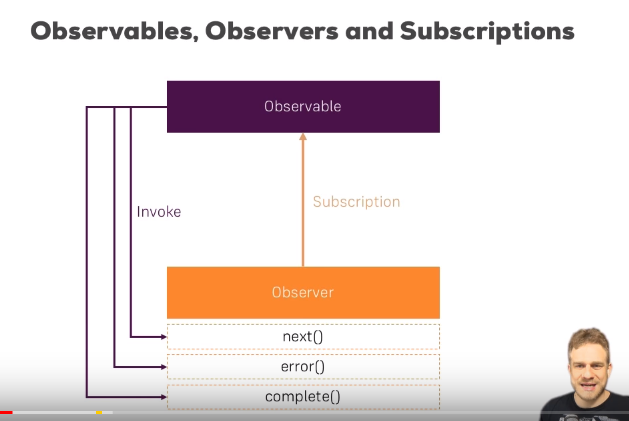


Figure 1

Instead Depiction that you commonly see is depiction as a stream(figure 2) . as I mentioned observable is just wrapper around a stream of value and we can have one value which instantly occurs to have synchronous data stream( then it is not stream) or we have multiple values. Whatever the case may be , we have observer with 3 methods where we can handle any values. We can handle one or multiple value. In the end, some observable may finish, in that case complete, method is executed. See figure 2

We can have stream which not only emit values but also throws errors at some time. Think of stream (or observable I should say) wrapping an HTTP request, we know that we will eventually get back a response, but that response may be error either, timeout or server side error. In this case observable will throw the error and we will handle it in error function of observer.

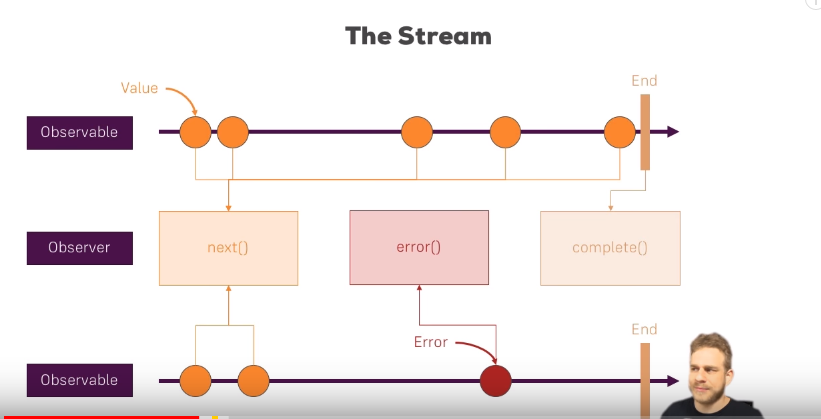


Figure 2

Lets see this in code. Open js fiddle. In html import rxjs through cdn

Code-

var button = document.querySelector('button');

Rx.Observable.fromEvent(button,'click')

.subscribe((event) => {

console.log(event);

});

Here we listen to button clicks with observable. Here we wrap the button click and create a new observable with fromEvent helper method. In official docs you will find a lot of helper methods to create observables(on home page click on observables) and lot of these methods are actually nice methods to conveniently create observable which do something specific, like here wrapping a event, others will emits a value every x seconds. Later in this video we will see how we can create observable from scratch.

Here code used in first lecture is explained. What happens there behind the scenes is that this observable has infinite stream of values. New value is emitted when we click on button, this is how this observable is configured. then in subscribe method we pass observer. Now this may not look like this because all I pass is a method but subscribe method takes 2 possible arguments- list of functions(next,error and complete) or single object which implements these functions. So we can have our code like this-

var button = document.querySelector('button');

var observer ={

next: function(value){

console.log(value);

},

error: function(error){

console.log(error);

},

complete: function(){

console.log("complete");

}

};

Rx.Observable.fromEvent(button,'click').

subscribe(observer);

so that is observable, an infinite observable, a asynchronous one bcoz we can emit multiple values over the time.

now lets built an observable from scratch. We will build same behavior that we saw in observable above ,in observable that we will create from scratch. Go to official documentation to see which helper method might me helpful. There are lot of methods, but create method is one I am looking for. With create method we can built our observable from scratch. Click on this function name in officials docs to see details. In docs we can see that, this function takes one argument, the observer kind of. you will see that how we built that in seconds.

Create actually takes a function as a argument, this function takes observer as a argument. So we pass anonymous function as argument to create. Now this function takes observer object as argument. Now important, we are not passing our observer variable(created by us in code) here, because ,keep in that is anonymous function. It was like if you were to declare new one. So we simply name the argument this function will eventually get. If we name it obsever we will get naming conflics with our variable. We don’t pass the data to the function here. We define a new one(function) which eventually receive one argument. Rxjs will pass that for us and this argument will be observer, we know it from documentation. So here we simply name it ops. Inside this function we can call ops.next() to emit new value.

Here in code we name this argument as ops, RXjs will pass this argument and it is observer. Inside this function we call next method of observer to emit a value. code-

var observer ={

next: function(value){

console.log(value);

},

error: function(error){

console.log(error);

},

complete: function(){

console.log("complete");

}

};

//Rx.Observable.fromEvent(button,'click').

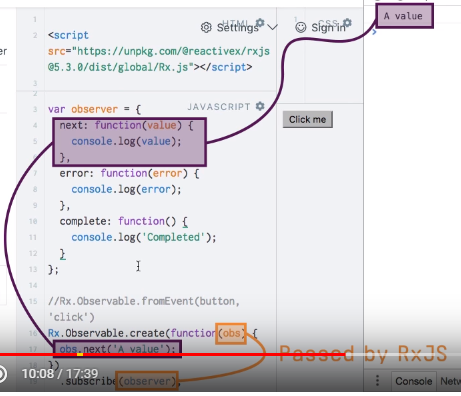
Rx.Observable.create(function(ops){

ops.next("a value");

}).

subscribe(observer);

if we run this code, then we see “a value” was already emitted because ofcourse we are not listening to any click right now , this code gets immediately gets executed. Why do we see a value? because we subscribe to an observable which takes a function as argument, where function takes the observer s argument, which we also pass to subscribe function and RXJS now passes our observer which we passed to subscribe, to that function and executes the function. that is how you create a observable.in that function we can call next method on observer because remember observer knows or Observable knows that observer can have next,error or complete, so these are 3 methods that we can call, therefore we see a value.



If we do this-

var button = document.querySelector("button");

var observer = {

  next: function (value) {

    console.log(value);

  },

  error: function (error) {

    console.log(error);

  },

  complete: function () {

    console.log("complete");

  },

};

Rx.Observable.create(function (ops) {

  ops.next("value emitted");

  ops.error("Error occured");

  ops.next("Second Value Emitted");

}).subscribe(observer);

here we dnt see second value on console, if error ocuurs the observable is finished. It wnt call complete . it just emits an error, then it emits no more value that’s why we dnt see value after error.

However if we call complete instead of error, then also we dnt see next value emitted. this is because observable has completed in this case also.

All this is ok, but here observable does’nt look asynchronous. Observable that wraps event looks asynchronous. because there we wrapped something that was asynchronous. There stream of data made more sense. Here we immediately print some value. So how is this observable?. remember I told you that a observable does’nt need to be asynchronous it can be synchronous. We have function which is executed when we subscribe to it and in that function we execute some synchronous code. So our whole observable is not asynchronous. we can make it asynchronous by-

var button = document.querySelector('button');

var observer ={

next: function(value){

console.log(value);

},

error: function(error){

console.log(error);

},

complete: function(){

console.log("complete");

}

};

//Rx.Observable.fromEvent(button,'click').

Rx.Observable.create(function(ops){

ops.next("value emitted");

setTimeout(function(){

ops.complete();

},4000);

ops.next("Second Value Emitted");

}).

subscribe(observer);

now we have asynchronous observable because now we have data stream where we have 2 synchronous value , emitted immediately and 1 event happening after 2 seconds.

Now let’s create our eventEmitter observable-

var button = document.querySelector("button");

var observer = {

  next: function (value) {

    console.log(value);

  },

  error: function (error) {

    console.log(error);

  },

  complete: function () {

    console.log("complete");

  },

};

Rx.Observable.create(function (ops) {

  button.onclick = function (event) {

    ops.next(event);

  };

}).subscribe(observer);

here basically recreated the behavior that we had from **fromEvent** Observable, by observable created from scratch.

So we saw how observable works and how we can built our observable from scratch.

There is a important point- this observable never completes as we can always click on button. If you have observable that never completes, that poses the danger of memory leak. So we should unsubscribe from any subscriptions that we dnt need any more. Imagine we have complex js app, for ex a single page application and we navigate to some other place in our app.all of sudden we might not need subscription any more,. If we come back we create a new one but old one still lives on in memory, that is memory leak. So clean any subscriptions that you do not need anymore

How to do it? Simple store your subscription in a variable then call unsubscribe method on it. Like this-

var subscription = Rx.Observable.create(function(ops){

button.onclick = function(event) {

obs.next(event);

}

}).

subscribe(observer);

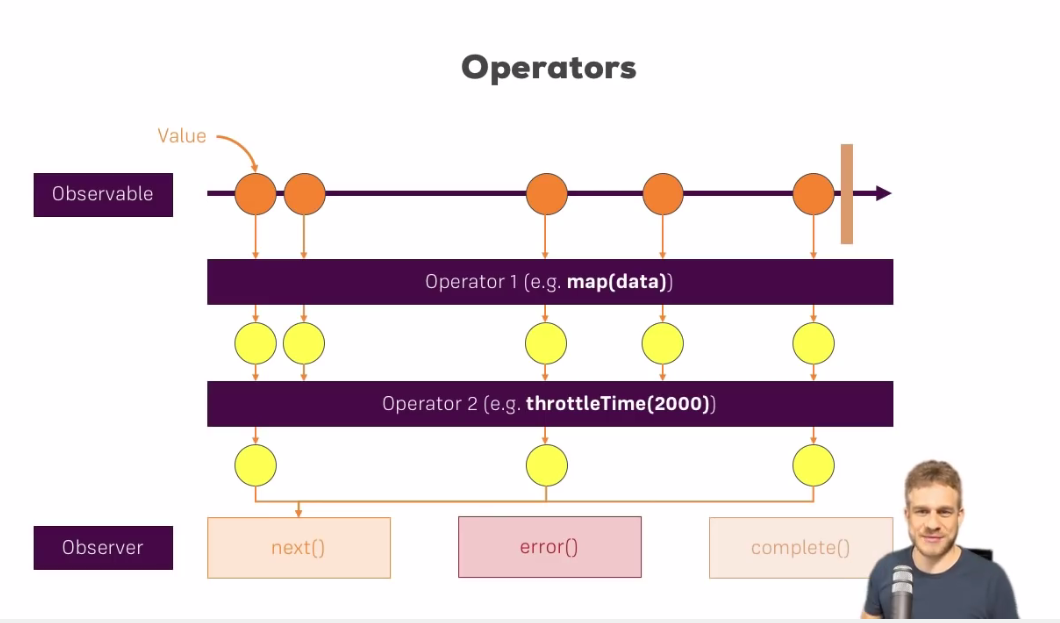
then-

subscription.unsubscribe();

3)RxJs Operators like map() OR throttleTime()

In stream diagram we saw how observable emits value and how we use those values in observer. even better thing about observable beside having nice structure is that they have this funnel like approach, where we can pass the value which are emitted through as many operators as we want before they finally reach observer.

First operaor is map, it transform our values. Logic of transformation lives in map function. may be want to convert number into string or anything like that. These transformed value then reach next function(next of observer). but may be we want to chain other operator using that funnel like approach. We can do that ,here we want to chain other operator using that funnel like approach. We can do that may be we can have **throttleTime** operator which make sure that no more than one value can reach the next function per window frame.



Here in first case only one of 2 value reach the observer because there is not enough time gap between them and so on. Now in next function of observer we will know what we are going to get.

Lets see this in code, here we use interval method to create a observable (all methods can be found in official docs). it emits value every n seconds, n being passed as input. It will emit ascending number of integars.

var observable = Rx.Observable.interval(1000);

var observer = {

next: function(value){

console.log(value);

}

};

observable.subscribe(observer);

we see increment number every 1 second. Now lets add operaors. We chain operators before calling subscribe. First we use **map**, it transforms the value that observable emits. Actually maps gives us new observable, but this observable holds the transformed data. Why do it gives us new observable? It has to, because we have subscribe in the end and we can only subscribe to observable. Code-

var observable = Rx.Observable.interval(1000);

var observer = {

next: function(value){

console.log(value);

}

};

observable.map(function(value){

return value\*10;

})

.subscribe(observer);

Map takes function as a argument. This function has logic for transforming the value. So we get the value emitted by observable in function that we pass to map, then in body of function we transform that value and return the transformed value. we can return any value that we want. Rxjs automatically creates a new observable which emits this transformed value, so that subscribe will work.

Now I want to chain another operator – **throttleTime**. As a value we pass the time to which we want to wait till observer receives next value.

var observable = Rx.Observable.interval(1000);

var observer = {

next: function(value){

console.log(value);

}

};

observable.map(function(value){

return value\*10;

})

.throttleTime(2000)

.subscribe(observer);

Our output will be –

0

20

40

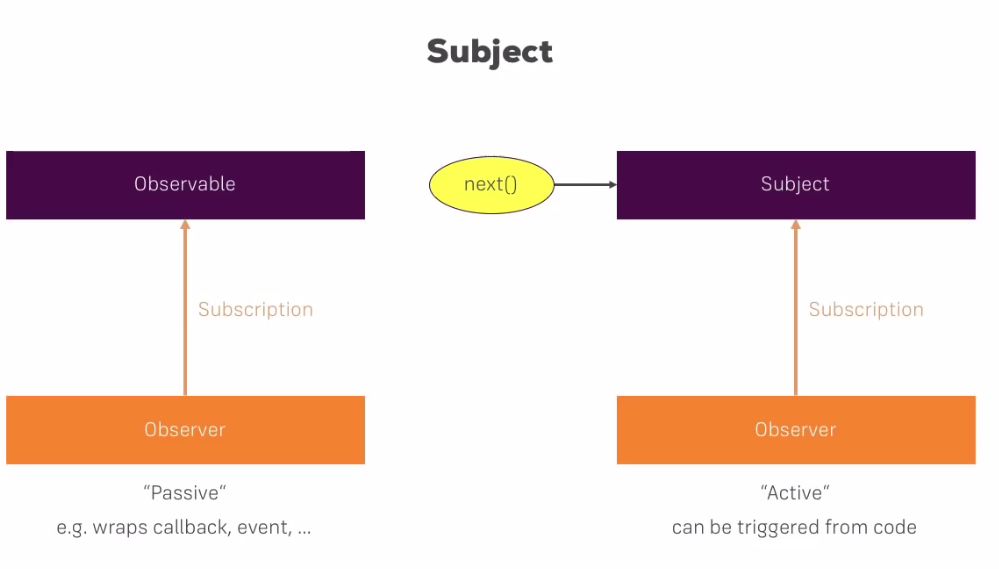
60

You can read about more operators in official documentation. Click on observable to see list of all operator.

4)RXJS SUBJECT (~ Event Emitter)

You might have noticed that observables are kind of passive. You wrap a event, you wrap a http request. You cnt trigger the emission of new value manually. So may be you want to emit a new value manually.So we might want to use observer as event emitter. Where you can emit value when you want.

We can do this by something called Subject. It inherits from Observerable. but there we can call next method manually to force it to emit some value. therefore we can have more active approach of using observable with subject. It does’nt mean you should always use subject. Use it if you need something like event emitter.



In code we create subject by calling using constructor. Unlike observable, we dnt need helper method. Now I can set up my observer but I can save some time , by calling subscribe on subject, then to this subscribe function we can pass a object that have next,error or complete method Or we can pass list of these functions. Here we pass js object.

Now to demonstrate how this works, I will add more than one subscription. Because great thing about event emitters is that we can trigger them one place in code and react to them in multiple places. So we typically have more than one subscriber. Then we call next on our subject to emit new data.

When we run our code , we see 2 times new value on console. This is because we have 2 subscriptions. We can do same for error and complete. You can call both these functions on subject. However we are not handling error case in second subscription, so we will get error on console, if our subjects emits error. We can emit error by-

subject.error(‘error’);

This is how we use subject. We can call this at any other place , we can call it when user clicked button, we can call it when timeOut finishes. You decide when to emit a value.

As a side note, once we use complete method on subject, emit any new value wnt happen.

So subject works like observable but you decide when something happens.

Code-

var subject = new Rx.Subject();

subject.subscribe({

next: function(value){

console.log(value);

},

error: function(error){

console.log(error);

},

complete: function() {

console.log('Completed');

}

});

subject.subscribe({

next: function(value){

console.log(value);

}

});

subject.next("new Data");

5)The filter Operator

This operator allows us to filter out some of value emitted by observable.

To filter operator we pass a function as a argument. Then we can pass second argument. This second argument defines what this keyword will refer to inside the function that we passed as first argument. Here we dnt need second argument. Inside that function we receive the values emitted by observable, we just need to return true or false. True means value should pass, false means it should not pass. We dnt receive error for value that are not allowed to pass. We just drop these values.

Code-

var observable = Rx.Observable.interval(1000);

observable

.filter(function(value){

return value %2 ===0;

})

.subscribe({

next: function(value){

console.log(value);

},

error: function(value){

consol.log(value);

}

});

Note- even if we change the value of argument in function passed to filter operator, subscribe will get original value

# 6)debounceTime & distinctUntilChanged

# observables and operators are great, but it can hard to find practical use cases for operator because you need some practice to be aware of where you can can use them. Here we will see specific pattern that uses 2 operators that rxJs provides. This is awesome pattern and it might come in handy

Code-

Html-

<body>

<button>Click Me</button>

<input type="text">

<script src= "https://unpkg.com/@reactivex/rxjs/dist/global/Rx.js"></script>

<script src= "app.js"></script>

</body>

js-

var input = document.querySelector('input');

var observable = Rx.Observable.fromEvent(input,'input');

observable.subscribe({

next: function(event){

console.log(event.target.value);

}

});

Here we are wrapping our input event with observable and are reacting to every event.

With each keystroke we are printing the value in input field on console.

Now lets say when user type something we want to connect to database and see if it is valid name or not. But if you send request at each keystroke then we will be sending a lot of requests. For same reason we can use rxjs operators to control how often we do something.

First operator that can help us in that is **deBounceTime**. It takes a argument, this argument defines the time in milliseconds , we need to wait before we emit a new value. code-

var input = document.querySelector('input');

var observable = Rx.Observable.fromEvent(input,'input');

observable

.debounceTime(2000)

.subscribe({

next: function(event){

console.log(event.target.value);

}

});

Now we dnt get new value emitted , if it’s within this timeframe. Lets say we type Sumeet, now event will be emitted only after 2 seconds. Rest events will be dropped.so we see Sumeet on console. We dnt see s, then su , then sum and so on.

Here event is emitted after every 2 seconds even if we continue to type.

Now you might say that debounceTime is like throttleTime, where we pass time and then it would only emit new value when that time span has passed. But they works differently. Here deBounceTime value will only be emitted if nothing happens(data stream is not sent) for 2 seconds.

Here event will be emitted after 2 seconds we stop typing. If keep to typing then nothing will happen. So after 2 seconds of no value being emitted, so of me not typing, we get the current value. this is what debounceTime does, it simply checks that whether there has been pause of value emissions for the timespan that you specify then it will simply give you the latest value and others will be dropped.

This can be really useful, if we set it to 5 seconds we can allow user to type, correct any mistake then check it’s typed value with that in database.

This is fine. But we have an issue here, what if user decides, oh I miss typed, and hen oh I am correct, in this case we will be emitting same value again.

It would be nice, if I can filter the value that are equal to last one. For this use this another operator- **distinctUntillChanged().**  We chain it after debounceTime. So we use it like this-

var input = document.querySelector('input');

var observable = Rx.Observable.fromEvent(input,'input');

observable

.debounceTime(2000)

.distinctUntilChanged()

.subscribe({

next: function(event){

console.log(event.target.value);

}

});

Use distinctUntilChanged() after debounceTime().

But our issue is not solved. Again we can see 2 values that are exactly same are emitted. here issue is- data emitted is event object, it is not event.value.target. event object is not same in both cases. We need to use map operator.

Code-

var input = document.querySelector('input');

var observable = Rx.Observable.fromEvent(input,'input');

observable

.map((event) => event.target.value)

.debounceTime(2000)

.distinctUntilChanged()

.subscribe({

next: function(value){

console.log(value);

}

});

so this is ow we can combine one or more operators. Now we can send one http request to check whether user name is valid or not, instead of sending many requests.

7)scan() vs reduce()

These 2 operator can be mistaken for each other because they do same kind of thing but they have one distinct feature and we will see that.

These 2 operators are useful, if we have a ongoing operation, a couple of values being emitted which you often have, while using observables and you want to reduce these values to a final result.

Lets see reduce first. We know how reduce method works on arrays in js. Reduce operator works in same way.

We create Observable using **of** mehod. It takes list of elements and it emits this list synchronously immediately. Code-

var observable = Rx.Observable.of(1,2,3,4,5);

observable

.subscribe({

next: function(data){

console.log(data);

}

}

);

Output-

1

2

3

4

5

Now we want to reduce these value into single value. so we use reduce operator. To this operator we pass a function. now this function has 2 arguments- total and currentValue. It will do this for each emission, so for each value to say so and then it has to return us a new value, a new total which will be used for next element as a input total. Beside this function we can also pass second argument which is our starting total, because for our first element we have no total.

Code-

var observable = Rx.Observable.of(1,2,3,4,5);

observable

.reduce((total,currentValue) => {

return total+currentValue;

},0)

.subscribe({

next: function(data){

console.log(data);

}

});

So we get the sum of all values. Now this operator is useful if you have a observable and you know that it will be eventually completed and you want to reduce all these individual values to one value.

Example-

var subject = new Rx.Subject();

subject

.reduce((total, currentValue) =>{

return total + currentValue;

})

.subscribe({

next: function(value){

console.log(value);

}

});

console.log("sumeet");

subject.next(1);

subject.next(2);

subject.next(3);

subject.next(4);

subject.next(5);

subject.complete();

important thing to note is reduce function will give us final result only when our observable completes. If we remove last line from code then we won’t get some of all numbers. Same is true for this code also-

var subject = new Rx.Subject();

subject

.reduce((total, currentValue) =>{

return total + currentValue;

})

.subscribe({

next: function(value){

console.log(value);

}

});

var i = 0;

setInterval(() => {

i++;

subject.next(i);

if (i == 5) {

subject.complete();

}

}, 2000);

So reduce makes sense only when our observable completes.

Now lets see scan-

**var observable = Rx.Observable.of(1,2,3,4,5);**

**observable**

**.scan((total,currentValue)=>{**

**return total+currentValue;**

**},0)**

**.subscribe({**

**next: function(data){**

**console.log(data);**

**}**

**});**

Output-

**1**

**3**

**6**

**10**

**15**

Here instead of final total we also get the induvial total of a loop. So that’s the difference .

Scan is useful if you want to reduce your elements , but you are also interested in in between totals and reduce is interested in final total only. Reduce only makes sense if your observable completes.

8)Pluck

This operator makes it easy for us to pluck property or properties of object emitted as value of our observables.

Consider this code-

var input = document.querySelector('input');

var observable = Rx.Observable.fromEvent(input, 'input');

observable

.subscribe({

    next: function(event) {

  console.log(event.target.value);

}

});

Here we are accessing the property of object that is emitted by observable . we can argue that digging into event.target.value property in subscribe method is not optimal. Why? Because If you chain more and more operators into this pipe of operators, then you might hit the point where you want to extract value right at the start and other operators should be build on top of that so that they dnt have to extract value in every operator.

In lecture 6, we saw that how used map operator to solve our problem. Here pluck makes it easier to do that step(getting value property from event.target object). So we replace map with pluck operator.pluck now is special case. You may only use it if you are working with objects.We use pluck when value emitted is object. It allows to pluck property or properties out of the object. We pass list of strings to pluck operator, each string is property of object, the property that you want to pluck out. First we pluck we pluck target from event. now target again is an object because target is simply html element which event occurred. Therefore we added second argument to pluck. We can add as many arguments as needed to extracted the final property we are interested in. we are interested in value of target , so we pass target as second argument. These string that we pass to pluck has to be properties that exist on object.

Now this code wroks in same way it was working before, but now with shorter syntax. So whenever you want to get property from object, use pluck instead of map.

Code-

var input = document.querySelector('input');

var observable = Rx.Observable.fromEvent(input, 'input');

observable

.pluck('target','value')

.debounceTime(500)

.distinctUntilChanged()

.subscribe({

    next: function(value) {

  console.log(value);

}

});

Here subscribe object gets the value plucked by plucked operator.

9)mergeMap()

Here we will see how we can merge the result value of 2 observables into into one using mergeMap. Code-

Htlml-

<body>

<input type="text" id="input1">

<input type="text" id="input2">

<p>Combined value: <span></span></p>

<script src= "https://unpkg.com/@reactivex/rxjs/dist/global/Rx.js"></script>

<script src= "app.js"></script>

</body>

Js-

var input1 = document.querySelector('#input1');

var input2 = document.querySelector('#input2');

var span = document.querySelector('span');

var obs1 = Rx.Observable.fromEvent(input1, 'input')

.subscribe((event) => span.textContent = event.target.value);

var obs2 = Rx.Observable.fromEvent(input2, 'input')

.subscribe((event) => span.textContent = event.target.value);

Here when I type in one input, it overwrites the paragraph i.e value of other input in paragraph is overwritten. This is because now we have 2 data sources,2 inputs with 2 observables listening to input event which will basically change the text content of span.

We want to combine the value from 2 inputs and show it in paragraph. Right now we overwrite their values.

We want to merge these 2 observables to get combined value. We can do it by mergeMap operator. mergeMap takes a outer observable and then merge inner Observable into it and whenever inner observable then emits a value, it will also take value of outer Observable into account to give us a combined value. sounds complicated lets see it in action.

Here we want to merge observable 2 into observable 1. So we call mergeMap on obs1. This operator takes function as a argument. this function gets the value of observable, on which we are using mergeMap operator.in our case it is obs1. Since we know it will event object , so we can name this parameter as event1. Now in function body we have to return another observable. So let’s return obs2. Now that alone won’t do the trick bcoz we want to combine the value of 2 observable. So I call map on obs2. Here i know that I will get event object of second observable. In function passed to map operator, we then combine the value of 2 obsrvables. Why can I do that. How can I return this-

return obs2.map(event2 => event1.target.value+ ' ' + event2.target.value);

remember we are still in function that we passed to mergeMap, here we have to return an observable. We can just return obs2, but that won’t do the trick. Because what we want to return in the end is combine observable, which combines the values of 2 observables. Therefore we call map operator on observable 2 which makes sure we transform the value that we get back. So since we have access to values of both observables, it is easy for us to combine them and in the end we return combined observable. Then we call subscribe after merge map operator. Inside subscribe we get combined and value and we set the value of paragraph.

Now if I type something in textbox1, nothing happens. But if type in textbpx 2, then combined value of 1 and 2 appears in paragraph.

Now why does not something happens if I type in textbox1? That is how merge map works. The inner observable, (observable connected with input box2 in our case) has to emit value , for whole observable to update and emit value. This is what happens in our case. Moment we type something in input2, combined observable emits value.

mergeMap might come in handy if you have multiple data sources and you want to merge them into one , where one of the data sources is the one, telling you when event emission happens like here inner observable. Code-

var input1 = document.querySelector('#input1');

var input2 = document.querySelector('#input2');

var span = document.querySelector('span');

var obs1 = Rx.Observable.fromEvent(input1, 'input');

var obs2 = Rx.Observable.fromEvent(input2, 'input');

obs1.mergeMap(

    event1 => {

        console.log('Insidse merge');

        return obs2.map(event2 => event1.target.value+ ' ' + event2.target.value);

    }

).subscribe(combinedValue => {

    console.log(combinedValue);

    span.textContent = combinedValue;

});

We call mergeMap on observable 1, it takes a function as argument. This function gets the value of the observable, on which we use mergeMap on

10)SwitchMap

It allows us to trigger some value emissions whenever some other observable emits a value. Our code is-

Html-

<body>

<button>Click Me</button>

<p>Combined value: <span></span></p>

<script src= "https://unpkg.com/@reactivex/rxjs/dist/global/Rx.js"></script>

<script src= "app.js"></script>

</body>

Js-

var input1 = document.querySelector('button');

var obs1 = Rx.Observable.fromEvent(button, 'click');

Now I want to start an Interval, so an observable which emits a new value every x seconds(lets say every second), when ever I click the button but I want that emitting observable, that interval to start over when I click the button again. So the old emission ,the old interval should be cancelled automatically. Now why is it useful?

It might not be useful if we talk about intervals but it is useful when we talk about http requests, if you have lets say some auto complete functionality and you reach out to your server whenever the user types something, you do not want these old request to continue whenever user changes its opinion. You are sending a new request and you want to cancel old ones, you want to cancel old observables, so you don’t have to handle the data which will eventually come back.Rxjs will handle that for you and switchMap specifically is what helps us with that.

So we create obs2 which will do interval emission thing. Now easiest way of connecting 2 observables is to subscribe to first observable and in subscribe function of obs1 we also subscribe to obs2. Code-

var button = document.querySelector('button');

var obs1 = Rx.Observable.fromEvent(button, 'click');

var obs2 = Rx.Observable.interval(1000);

obs1.subscribe(event => obs2.subscribe(value => console.log(value)));

now whenever we click on button countdown starts. When I click on button again a new observable starts emitting value along with old one. If click again then we have 3 obsevables.this is bcoz old ones are not cancelled. switchMap can help us with that.

We remove the subscription from obs1, we use switchMap, which help us to connect 2 observables in a way I want them to connect here. So to switchMap I basically have to return a function where i still pass the value of outer observable, so of obs1 here(which is event) but inside of that body of that function now , I will now have to return second observable(obs 2) which I want to connect with firstone . that’s all. What switchMap will now do is, it will react to values being emitted on that outer observables(obs1) and it will then trigger the inner observable and basically switch the values you could say. We wnt receive click events, we will receive the values of inner observable .hence the name it switches values. The keything is that , it cancels all old subscriptions automatically if we click button again, so if outer observable is triggered again. Then we subscribe to that switch observable. Code-

var button = document.querySelector('button');

var obs1 = Rx.Observable.fromEvent(button, 'click');

var obs2 = Rx.Observable.interval(1000);

obs1

.switchMap((event) =>{

    return obs2;

})

.subscribe((value) => console.log(value));

Now when we click on button, all old observables are cancelled and we got only one interval running at a time.

So that is switchMap, we switch 2 values or 2 observables kind of and we cancel any old ones.

11)Behaviour Subject

We already saw subjects. This code uses subjects-

Html-

<body>

<script src="https://unpkg.com/@reactivex/rxjs@5.3.0/dist/global/Rx.js"></script>

<button>Click me</button>

<div></div>

<script src="app.js"></script>

</body>

Js-

var clickEmitted = new Rx.Subject();

var button = document.querySelector('button');

var div = document.querySelector('div');

button.addEventListener('click', () => clickEmitted.next('Clicked!'));

clickEmitted.subscribe(

    (value) => div.textContent = value

);

Normal subjects have a limitation. There is one limitation that normal subjects has though it does’nt have an initial value or starting value. So if we load our project gain in starting div does not have any text. Workaround will be to add this line the end-

clickEmitted.next('Not clicked');

RXjs has special type of subject that we can use, the behavior subject. To BehaviourSibjects we can pass a argument to constructor that was not possible in normal subject. Here we can assign default value for the subject.code-

var clickEmitted = new Rx.BehaviorSubject('Sumit');

var button = document.querySelector('button');

var div = document.querySelector('div');

button.addEventListener('click', () => clickEmitted.next('Clicked!'));

clickEmitted.subscribe(

    (value) => div.textContent = value

);

Now when we run the code, we can see that div has text that we passed as default value to our constructor. What happens is , this starting value is automatically emitted by our subject. So our subscription code is triggered and we get the text of div.

So when will we use behavior subject? You typically use it whenever you do use your subject like a normal variable which should hold the value but which you want to change eventually and which should then inform other parts of your application about this change that is aperfect useCase for behavior subject.

Code whisperer-

Code-

var sub = new Rx.BehaviorSubject(42);

var observer = n => console.log('The Meaning of Life is' + n);

sub.subscribe(observer);

sub.next(108);

var observer2 = n => console.log('Combination to Safe is'+ n);

sub.subscribe(observer2);

sub.next(666);

output-

The Meaning of Life is42

The Meaning of Life is108

Combination to Safe is108

The Meaning of Life is666

Combination to Safe is666

Behaviour subjects have all features of subject but it remembers the last value that subject has pushed and whenever a new subscriber appears , it automatically pushes the most recent thing. The idea here is that new subscribers that subscribe on you know in middle of process or late in process still get one value to configure their initial state. If you don’t use bahaviour subject then either you have to cache last value manually from subject or wait until the next data comes. Either way it’s not pretty and behavior solves this nicely.

Here we created a behavior subject. Now the value passed to constructor is initial value that will be returned in case something subscribes to it before anything has been sent out.

Now when observer subscribes to our behaviour subject, no value was sent out by our subject. So its gets the default value that we passed in constructor. Then our behavior subject dispatches a new value. Now observer prints gets this new value and executes the function. Then we define a new observable. The this observable2 subscribes to our function. Now the subject has not emitted any value. But still this observable gets last emitted value. As subject has emitted a value, in this case default value(that we passed to constructor is ignored). Finally when we push a new value both subscribers get same value.

We will see Behaviour subjects gain and again as they made ideal format for representing state of an app.