18)Delay-DelayWhen

Delay- the way delay works is to basically take any emission from source and it add some kind of element of time between when it got value and when it moves on. Typically when you use delay, you use delayWhen. So you specify a number and then after that time passes each emission is released

delayWhen- in delayWhen on other hand you use a number stick with deayWhen you specify method instead of number. So value returned can vary based on what input was.

Code-

import Rx from 'rxjs';

Rx.Observable

.of(42)

.do(() => console.log('What is meaning of Life'))

.delay(1200)

.do(()=> console.log('Comeone you know this'))

.delay(2000)

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

Output-

What is meaning of Life

Comeone you know this

42

First it does the first log then it waits, then it do second log and waits and finally third log. You can imagine how difficult it would be to get this functionality if you are just using es5 and how to use setTimeout and call backs. You can actually create a stack of as many things as you want to create a really kind of curated experience could take a minute or longer.

Next question, what if instead of just one thing we have got many things going on code-

Rx.Observable.range(1,9)

.delay(1000)

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

Here all the numbers come at same time, i.e after delay of 1 seconds.

Lets see this Code-

Rx.Observable.interval(100)

.delay(10000)

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

Here only first value is delayed. Rest values are emitted every 1 second. but in this case-

var clicks = Rx.Observable.fromEvent(document, 'click');

var delayedClicks = clicks.delay(5000); // each click emitted after 1 second

delayedClicks.subscribe(x => console.log(x));

we see log after 5 seconds of delay and this is for each click. So there is delay in each value to be emitted.

In code-

Rx.Observable.range(1,9)

.delay(1000)

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

We dnt want values to be emitted in one go. We can use dleayWhen for that. Code-

Rx.Observable.range(1,9)

.delayWhen((n) => Rx.Observable.timer(n\* 1000))

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

See documentation for details.

Another example-

var clicks = Rx.Observable.fromEvent(document, 'click');

var delayedClicks = clicks.delayWhen(event =>

Rx.Observable.interval(Math.random() \* 5000)

);

delayedClicks.subscribe(x => console.log(x));

here we print event object on screen with random delays.

Delay object is useful lets say we are making simple calculation and we want to fake that calculation is very complex, we want to show things like analysis, consulting experts etc.

19)Take- TakeWhile – TakeUntil

Take works like this it takes fist x emission from source observable and emits those, after it has exhausted that number of emisisons it actually completes(observable on which it is applied is also complted) letting any following observables know, nothing else is coming. It is different form its cousins (Takewhile,TakeUntill), because here we specify number like 5, 10etc, which indicates how many number of values we want to take.

So its good, say if you have an observable that returns the first thousand users in your database but you need only lets say 10 to run simple check.

Then comes takeWhile. It works same as take. Instead of giving it a number we give it a predicate and until that predicate returns false it emits everything. Once that predicate emits false, it stops emitting and completes. Difference between filter and takeUntil is that even if predicate fails filter will still emits value for which predicate is valid but takeUntil completes as soon predicate becomes false.

Then we have takeUntil. Unlike other 2 takeUntil is provided an observable, observable can be anything but most probably we see it as timer and it is going to take values until that observable resolves. Code-

Rx.Observable.range(1,9)

.take(5)

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

Output-

1

2

3

4

5

This code also has output-

Rx.Observable.interval(1000)

.take(5)

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

But values appear one after another with delay.

Lets use takeWhile-

Rx.Observable.interval(1000)

// .take(5)

.takeWhile((n) => n <7)

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

Lets use takeUntil-

Rx.Observable.interval(100)

// .take(5)

// .takeWhile((n) => n <7)

.takeUntil(Rx.Observable.timer(1000))

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

Output-

0

1

2

3

4

5

6

7

8

Here we see values till observables passed to takeUntill does not emit value, when it emit value then our main observable completes.

20)Throw

It is slightly more modified and useful version of empty. So it creates a observable which immediately enters an error state and then completes as opposed to empty which immediately enters a complete state. None of them emits any values. You are basically only going to use throw for testing or error handling. So if you have function , which is supposed to handle an error and you want to see if it works , then this operator can be used.

Code-

import Rx from 'rxjs';

Rx.Observable

.throw()

.subscribe({

next : n => console.log(n),

error: e => console.log('Entered Error State', e)

});

Output-

Entered Error State undefined

If we dnt use error method then we get error and our application will stop. Throw is static operator there is no way to chain it on to something else like a range of numbers. Its just operator is just born, right out of method and it throws an error. Ofcourse it does pair well with error handling operators catch and retry. We will see them later.

21)Interlude - Creating an Observable Wrapper for Standard Keyboard Input

Here we will make observable from what we create everyday, in this case entering stuff with keyboard. We will create a function which creates an observable out of standard keyboard input. We create a new file utility.js. there we have a function named fromStdIn, which is going to return an observable which emits every time we press something on keyboard. In node we use process.stdin to get input from keyboard. Here also we do same. Then set RawMode of stdin to true. Then we set encoding to utf-8, which is typical text-encoding format. Then we create a observable using fromEvent. To fromEvent we pass stdin which is our event emitter and stdin emits a event data when something is clicked. Code-

import Rx from 'rxjs';

export const fromStdIn = () => {

const stdin = process.stdin;

stdin.setRawMode(true);

stdin.setEncoding('utf-8');

const observable = Rx.Observable.fromEvent(stdin,'data');

return observable;

};

fromStdIn().subscribe(a => console.log(a));

save this file as utility.js in project root folder. Then run this command-

**npm run babel-node ./utility**

now whenever we press any key, it is logged into console. But we are not able to close it using ctrl+c. so we need to refine it a bit, before we can actually use it. We need to listen for ctrl+c input and end our standard input. We do it like this-

import Rx from 'rxjs';

export const fromStdIn = () => {

const stdin = process.stdin;

stdin.setRawMode(true);

stdin.setEncoding('utf-8');

const observable = Rx.Observable.fromEvent(stdin,'data');

observable.subscribe( key => {

if (key === '\u0003') {

process.exit();

}

});

return observable;

};

// fromStdIn().subscribe(a => console.log(a));

So now we import fromStdIn to any other program, we have cool asynchronously emitting observable that we really have a lot a control over. In upcoming videos we will use it.

22)Skip-skipWhile-skipUntil

Skip is case of inverted take. It ognotres first few elemnts of source observable.

To skipWhile takes a predicate , we wnt get any value until that predicate returns true and once it returns false , all subsequest value are emitted.

skipsUntil takes a observable, when that observable resolves, then skipping stops and emissions begin(from main observable).

Unlike there is not implicit completion here. So if you have long running observable and you call skipUntill, its not actually going to complete when skip until starts to kick in. lets see in code-

import Rx from 'rxjs';

import {fromStdIn} from '../utility';

fromStdIn()

.skip(2)

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

Here we are going to use fromStdIn that we made in last video. Here our first 2 keypress does not appear on console, because we used skip and 2 as argument. Lets say we also use take after skip like this-

import Rx from 'rxjs';

import {fromStdIn} from '../utility';

fromStdIn()

.skip(2)

.take(2)

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

In this case only 3rd and 4th keypress will appear on console. All others are igored and our observable is completed. If we had a comleted handler, it would have been executed but application isn’t actually exited, since we have this standard in opened it will remain open until we close it. It has no way of knowing it that we don’t need it any more.

Lets see skipWhile-

import Rx from 'rxjs';

import {fromStdIn} from '../utility';

fromStdIn()

// .skip(2)

// .take(2)

.skipWhile(v => v !== 'g')

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

Here we wnt see any input until we press g(i.e our predicate becomes false), after that all values will be printed. Now lest see skipUntil. Skip until best lends irself to examples involving timers. Code-

import Rx from 'rxjs';

import {fromStdIn} from '../utility';

console.log('Input Will be accepted in 2 Seconds');

fromStdIn()

// .skip(2)

// .take(2)

// .skipWhile(v => v !== 'g')

.skipUntil(Rx.Observable.timer(2000))

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

Here input will only be accepted after 2 seconds, i.e after inner observable resolves.

23)last

It emits the last element of source observable which passes a predicate and it only does so after the observable completes. So unlike first source must complete if source never completes, last will never emit and you got yourself a useless operator. Code-

import Rx from 'rxjs';

Rx.Observable

.range(1,9)

.last()

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

Output-9