2. Setting Up The Project!

Go to your project, run

**npm init –yes**

it will create create pakage.json file.

now we need to get .gitignore file. it is used to tell which files to ignore while using git commands. Now we are not going to be accidentally indexing our node modues.

The create a new file index.js, now run this file using node.it runs fine. But what If I want to use little bit of ES6. To test whether ES6 is working we write this coe-

import fs from 'fs';

this is file system module which is avalaible by default. So we can test this import statement whther it works. We get this error-

SyntaxError: Unexpected token import

So to use es6 features we need to use bable. So lets configure bable. Create file named **.bablerc**. it is going to be a json file.

**{**

**"presets": ["env"]**

**}**

So this is going to tell bable that when we call it , look for bable preset -env. But we have none of them installed yet, so we will have to install them right now. Run this command-

**npm install --save-dev babel@6.23.0 babel-cli@6.26.0 babel-loader@7.1.2 babel-preset-env@1.6.1** [**babel-register@6.26.0**](mailto:babel-register@6.26.0)

in package.json-

"devDependencies": {

"babel": "^6.23.0",

"babel-cli": "^6.26.0",

"babel-loader": "^7.1.2",

"babel-preset-env": "^1.6.1",

"babel-register": "^6.26.0"

}

If you dnt want to install these packages manually then download package.json from git and run npm install.

Now we have babel cli so we can run this command-

**babel-node “filename”**

now file will be processed by babel first using babel preset env and we wnt see error for import keyword.

3)Range

Observable emits these value and completes. Lets see it in code. For this we need rxjs package, so run this command-

**npm install --save** [**rxjs@5.5.6**](mailto:rxjs@5.5.6)

create a new folder called range. In it create a index.js file.

import Rx from 'rxjs';

const rangeObservable = Rx.Observable.range(1,9);

rangeObservable.subscribe(e => console.log(e));

now run this command-

**npm run babel-node range**

you should be in project folder. Now here observable starts emitting asa sson as we subscribe to it.

Main thing to remember is that range operator is synchronous.

4)of

Slide 10

Of is very simple static operator, and all it does is you pass it any number of argument, that can be in any format, array,object,number and they don’t all have to be the same format. And it just turns it into observable. It’s a simple utility you can use to turn literally any collection into an observable

It is utility knife for making observable out of any object. Because of its broadness it is recommended you only use it if you cnt find more better and specific operator. That’s kind of rule.

Code-

import Rx from 'rxjs';

Rx.Observable

.of(1,2,3,'Four',[1,2,3],{name: 'sumeet'},5)

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

Output-

**1**

**2**

**3**

**Four**

**[ 1, 2, 3 ]**

**{ name: 'sumeet' }**

**5**

5)from

Whatever you pass to from will get turned into observable. So where as of is something that we used to take bunch of things and turned them into observable, with from we are only taking one thing that represents a lot of things like array and turning it into observable where each elment of array will be emitted induivally in contrast if we provide an array as first argument of **of**, it will just emit that whole array all at once. Code-

import Rx from 'rxjs';

const fibonacciArray = [1,1,2,3,5,8,13];

Rx.Observable.from(fibonacciArray)

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

Output-

1

1

2

3

5

8

13

That was simple. lets try it with promise. We will make a promise that resolves instantly to Fibonacci array. Pomise can return only one value, but if that returned value is array, like here, lets see how our from operator handles it, how it emits values.

Code-

import Rx from 'rxjs';

const fibonacciArray = [1,1,2,3,5,8,13];

const fibonacciPromise = new Promise(r => r(fibonacciArray));

Rx.Observable.from(fibonacciPromise)

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

Output-

[ 1, 1, 2, 3, 5, 8, 13 ]

So Fibonacci promise is not equivalent because all numbers are logged in single array. So we are takig what ever is returned from that promise even if it’s an array and popping it all out just like that.

All this is fine what if we want to make observable from generator function? if you are not familiar generator functions lokks like this so the way it works with observable is, generally like this, it has while sort of loop in it. Actualy I will write it now. Sso this Fibonacci generator will produce Fibonacci numbers at infinity.

function \* fibonacciGenerator () {

let a =1, b =1;

while (true) {

let c = a+b;

yield c;

a =b;

b = c;

}

};

Lets pass our iterator to **from** operator.

Code-

import Rx from 'rxjs';

const fibonacciArray = [1,1,2,3,5,8,13];

function \* fibonacciGenerator () {

let a =1, b =1;

while (true) {

let c = a+b;

yield c;

a =b;

b = c;

}

};

const fibonacciPromise = new Promise(r => r(fibonacciArray));

Rx.Observable

// .from(fibonacciPromise)

.from(fibonacciGenerator)

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

We dnt see any output. So lets try invoking iterator with pair of brackets. Like this-

.from(fibonacciGenerator())

Now we see long list of numbers and then we see infinity on screen.

We can use another observable here. We want certain numners of value from a observable we can use **take**  operator. So here we will take only 10 values. Code-

import Rx from 'rxjs';

const fibonacciArray = [1,1,2,3,5,8,13];

function \* fibonacciGenerator () {

let a =1, b =1;

while (true) {

let c = a+b;

yield c;

a =b;

b = c;

}

};

const fibonacciPromise = new Promise(r => r(fibonacciArray));

Rx.Observable

// .from(fibonacciPromise)

.from(fibonacciGenerator())

.take(20)

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

Output-

2

3

5

8

13

21

34

55

89

144

233

377

610

987

1597

2584

4181

6765

10946

17711

Both generators and observables returns lot of values over time.

6)Interval – Timer

Interval creates a observable that emits value every time duration passes. You dnt specify what values it emits. Instead by default it emits 0,1,2,3 and so on. You can then map these values. We will learn about mapping techniques.

Timer emits once when duration passes, it is like setTimeOut. Howver if you pass it second value then it becomes interval.

Code-

import Rx from 'rxjs';

const ticker = Rx.Observable.interval(2000);

ticker.subscribe(a => console.log("A", a));

setTimeout(()=> ticker.subscribe( a=> console.log("B", a)),1000);

output-

A 0

B 0

A 1

B 1

A 2

B 2

A 3

here each subscription is getting its own sequence of values. Every time you subscribe it with something else , it starts a new interval from 0.

Now lets see timer.

import Rx from 'rxjs';

const stopWatch = Rx.Observable.timer(2000);

stopWatch.subscribe(a => console.log("A", a));

output –

A 0

So it runs once with number 0 and then exists. Now lets add second argument.

import Rx from 'rxjs';

const stopWatch = Rx.Observable.timer(2000, 1000);

stopWatch.subscribe(a => console.log("A", a));

output

A 0

A 1

A 2

A 3

A 4

A 5

So it takes 2 seconds to get initial value then it swiches to 1 second timer and behaves just like interval.

Note- these should be only used for demonstration purposes.

7)empty

Slide 13

Code-

import Rx from 'rxjs';

Rx.Observable.empty().subscribe({

next: a => console.log('Emit', a),

complete: a=> console.log('Complete',a)

});

Output-

Complete undefined

If you ever need a obsvable that completes immediately, emitting no values, then use empty.

8)Map-Mapto

Basically every element that;s emitted from source observable , the map function is going to turn that, is going to mutate it into something else. And whatever that something else is we pass it long the next thing in obsever chain, as the first thing never existed. Its very similar to array.prototype.map

Now surprisingly useful map to works like map but it does’nt care what provided value is. It always maps to a constant value, I mean you cold use globals to mutate value over time which in anti pattern but generally mapto is what you use to make things into single contsnat value which I add gain is surprisingly useful for day to day cases.

Lets see map-

import Rx from 'rxjs';

const range = Rx.Observable.range(1,9);

range

.map((n)=> n\*n)

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

Now lets see mapTo. Lets take kind of more real world approach for this. Lets write a method that kind of pretends to be notification socket api was called notification api. Code-

const notificationApI = Rx.Observable.interval(1000);

notificationApI

.mapTo({type: "NOTIFICATION\_ARRIVED"})

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

Output-

{ type: 'NOTIFICATION\_ARRIVED' }

{ type: 'NOTIFICATION\_ARRIVED' }

{ type: 'NOTIFICATION\_ARRIVED' }

{ type: 'NOTIFICATION\_ARRIVED' }

{ type: 'NOTIFICATION\_ARRIVED' }

{ type: 'NOTIFICATION\_ARRIVED' }

Here we put a redux style action in mapTo. So all values are replaced by consistent value. If this was redux application, this action could prompt a different part of application to refresh or pull ore data or something like that. So this mapTo is a subtle and refined operator which we will be revisiting again.

9)Filter

import Rx from 'rxjs';

const range = Rx.Observable

.range(1,9)

.filter(n => n%2 === 0)

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

We can also use filter for asynchronous observables.

import Rx from 'rxjs';

const range = Rx.Observable

.interval(400)

.filter(n => n%2 === 0)

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

10)Do

Slide 16

Do operator discretly executes a side effect, most common example of which is console.log. you can do basically anything with do including calling an api, you know directly using request but that’s but that’s an anti pattern. You should’nt use do for it. For console its fine because that’s just for developing but like say you want to call an api, be best if you wrapped that api call in another observable like ath observable and subscribe to it instead of using do.

When do is called it gets last emitted value as argument. It does’nt matter what is returned , it does’nt return anything because that same value id passed to next step. The do cannot do anything to change the value, it cnt alter it like map does. Do can console log it. If it’s an object that can be mutable then maybe it can be edited. But that’s a huge anti-pattern and you should not do that.

Only useful case for do I can think of is, it can kind of log the interstate process for very long complicated observable observable. Code-

import Rx from 'rxjs';

const interval = Rx.Observable.interval(1000);

interval

.map((n) => n\*n)

.do(n => console.log("N after Map#1::", n))

.map((n) => n / 3)

.do(n => console.log("N after Map#2::", n))

.map((n) => n\*n\*n)

.do(n => console.log("N after Map#3::", n))

.map((n) => n%42)

.do(n => console.log("N after Map#4::", n))

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

Output-

N after Map#1:: 0

N after Map#2:: 0

N after Map#3:: 0

N after Map#4:: 0

0

N after Map#1:: 1

N after Map#2:: 0.3333333333333333

N after Map#3:: 0.037037037037037035

N after Map#4:: 0.037037037037037035

0.037037037037037035

N after Map#1:: 4

N after Map#2:: 1.3333333333333333

N after Map#3:: 2.3703703703703702

N after Map#4:: 2.3703703703703702

2.3703703703703702

N after Map#1:: 9

N after Map#2:: 3

N after Map#3:: 27

N after Map#4:: 27

27

Here we are doing many complex operations on observables. Do is used to see the interstial state without modifying and passing it to next operator which is either map or subscribe.

We can use do to make api calls also. but it is advised not to use so. Dnt use do other than console.log.

12)First

Slide 18

It means from observable we are going to get first value that matches predicate.

Code-

import Rx from 'rxjs';

Rx.Observable.interval(1000)

.first((n) => (n!== 0 && n %2 ===0))

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

Output-

2

Here we get first value that fulfils the criteria, after that no value is emitted.

13)startWith

It basically creates an observable, you chain it to source observable and provide it a value. Then new observable first emits the value provided and after that emits everything that came from source.

So it’s useful for dealing with observable, lets say we have had observable like timeout which does’nt actually in running application does’nt actually return value for seven seconds or 10 seconds. You could have other components that could be displaying some kind of intermediate value like zero if only knew what to display. Ofcoure you can could program all your components individually to just deal with that or you could use startWith. Code-

import Rx from 'rxjs';

console.log("Made API Request");

Rx.Observable.timer(2000,500)

.map(n => n+1)

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

Here first value is emitted after 2 seconds, the after every 0.5 seconds we get a new value. Lets now use the startWith.

import Rx from 'rxjs';

console.log("Made API Request");

Rx.Observable.timer(2000,500)

.map(n => n+1)

.startWith(0)

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

Output-

Made API Request

0

1

2

3

4

5

Here we get the value that we pass to starWith immediately, after that we get value from regular observable.

14)Create

Create in many ways is one of the simplest operators but I waited a long time to reveal it bcoz it is great power tends to tempt developers to the dark side of anti patterns. So basically create can be used to create a new observable with almost any structure that completes, emits and errors under all circumstances, that are 100% in your control. Which is very powerful but you get into a situation where excuting a massive amount of code inside you create call would raise the question if you are running so much of your code outside of chained observable context why are we using observable. In other words the more code you put inside your create call , the less code could have gone into more testable, readable and maintainable observer chains.

Create is hammer that can hammer any nail, but we almost never want to use it. You can only use it when you can’t think of any other creation operator that could be applied or if you are writing some kind of custom operator, like we will be doing much later in this course.

Code-

import Rx from 'rxjs';

const customObservable = Rx.Observable.create((observer)=> {

observer.next(42);

observer.complete(100);

});

customObservable.subscribe({

next: a => console.log("Next", a),

complete: a => console.log("Complete",a)

});

Output-

Next 42

Complete undefined

So create is passed method and argument argument to this function is observer. So it means we can call observer.next inside function. in output we see 42. It is because 42 gets emitted right away, this is what is function does. function that we pass to create prescribes the behaviour of observervable. Bot ofcourse since our behaviour here is functional and not declarative it’s anyones guess what’s going on, when you are trying to maintain this code. Another thing to be noted is value emitted by complete is not passed to observable.

So there is no limit to custom functionality , you can literally put your whole application inside the create call, that is why it is so dangerous.

But lets also imagine here, will just set an interval. And this is actually another great example. So here I am in create and I am suing setInterval , so when I am using RXJS the whole point is that I can use operator creators like interval and timeout to get better versions of these things tat are easier to test. But since I am writing everything inside my create statement, I am just tempted to fall back on a plane old javascript and wrire setInterval. Code-

import Rx from 'rxjs';

const customObservable = Rx.Observable.create((observer)=> {

// observer.next(42);

// observer.complete(100);

setInterval(() => {

observer.next('New Value');

}, 500);

setTimeout(()=> {

observer.complete(100);

}, 2500);

});

customObservable.subscribe({

next: a => console.log("Next", a),

complete: a => console.log("Complete",a)

});

Output-

Next New Value

Next New Value

Next New Value

Next New Value

Complete undefined

Here our code is competed but our application cannot exit because this interval that we have created, you cant get rid of itself, you have to get rid of it manually. Hence this is why rxjs intervals are much better.

So you can see, if you have really specific function that for some reason had to do all these different things and really just could’nt kind of spread it out amoung then create would be great. But you can also see , now I am putting tons of stuffs in create and it’s not very observable, you know.

15)Every

It creates an observable which eits true if every elemnt that was passed to it by sorce function observable, everyone passed a predicate function. the important details about every is, it only emits, once the source completes. So it belongs to special category of observables which have no use if they are chinned to another observable which runs forever like a long running api cal. If you chain every to anything that does’nt complete , it will literally never resolve ,otherwise as soon as thing completes it emits true or false if every element passed the predicate.

Code-

import Rx from 'rxjs';

const range = Rx.Observable.range(1,9);

range

.every(n => n % 2 ==0 )

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

Output-

False

Lets use do to log value.

Code-

import Rx from 'rxjs';

const range = Rx.Observable.range(1,9);

range

.do(v => console.log("Value: ",v))

.every(n => n % 2 ==0 )

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

Output-

Value: 1

False

Actualy it stop running the range after first value. Its actually cancelled because as soon as it gets one false value, it knows it does’nt need to do anything else. Lets run this code-

import Rx from 'rxjs';

const range = Rx.Observable.range(1,9);

range

.map(n => n\*2)

.do(v => console.log("Value: ",v))

.every(n => n % 2 ==0 )

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

Output-

Value: 2

Value: 4

Value: 6

Value: 8

Value: 10

Value: 12

Value: 14

Value: 16

Value: 18

True

So here observable ends, as all values passed predicate we get true. So we will get true or false either when observable completes or when one of value fail test.

Code-

import Rx from 'rxjs';

const range = Rx.Observable.interval(1000);

range

.map(n => n\*2)

.do(v => console.log("Value: ",v))

.every(n => n % 2 ==0 )

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

Output-

Value: 0

Value: 2

Value: 4

Value: 6

Value: 8

Value: 10

This code keeps running bcoz observable does not complete and all values are passing the test. To complete this observable we use take operator, i.e we will just take 10 values from this observable. Code-

import Rx from 'rxjs';

const range = Rx.Observable

.interval(1000)

.take(10);

range

.map(n => n\*2)

.do(v => console.log("Value: ",v))

.every(n => n % 2 ==0 )

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

Output-

Value: 0

Value: 2

Value: 4

Value: 6

Value: 8

Value: 10

Value: 12

Value: 14

Value: 16

Value: 18

True

We get true because our observable has completed and all values passed the predicate.

16)DistinctUntilChanged

Code-

import Rx from 'rxjs';

Rx.Observable

.interval(100)

.map(n => ~~ (n/10))

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

Output-

0

0

0

0

0

0

0

0

0

0

1

1

1

1

1

1

1

1

1

1

2

2

2

2

2

2

2

2

2

2

3

3

3

3

3

3

3

3

3

3

4

4

4

4

4

4

4

4

4

4

5

5

5

Lets use our operator now-

Code-

import Rx from 'rxjs';

Rx.Observable

.interval(100)

.map(n => ~~ (n/10))

.distinctUntilChanged()

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

Output-

0

1

2

3

17)DefaultIfEmpty

It ceates a new observable from source observable that, if source observable copletes without emitting anything then the value provided is emitted. If source observable ever emitted any value, then this operator has no effect. It also has no effect of source observable is never complted. Code-

import Rx from 'rxjs';

Rx.Observable

.empty()

.defaultIfEmpty(42)

.subscribe(a => console.log('Where is my value ??'));

Here observable completes without any value so our operator kicks in.