Lecture 8th starting- how all observables are created, listen about create method

Lecture 11 paragraph 3, data is not available for modification in component

Observable Creation Methods-

**1)interval**

const interval$ = interval(1000);

interval$.subscribe(val => console.log(`Stream 1 ${val}`));

**2)timer**

const interval$ = timer(3000, 2000);

interval$.subscribe(val => console.log(`Stream 1 ${val}`));

interval$.subscribe(val => console.log(`Stream 2 ${val}`));

this observable will wait for 3 seconds, then it will start emitting values at interval of 2 seconds.

**3)fromEvent**

const click$ = fromEvent(document, 'click');

click$.subscribe(evt => console.log(evt));

first argument is source of click and second argument is event that we are subscribing to .it returns us only defination of stream, tis is not an instance of stream. We are not specifiying what we to do with stream values, we have not yet instantiated a stream. When we subscribe to it , then we are going to get values emitted by these observables.

**4)Observable.create**

We will use **Observable.create** method. This will allow you to create an observable from scratch and this how all observable that we have seen before , such as interval, timer are created by calling **Observable.create** . this function returns a observable, that we store in http$. In order to create observable we pass a function as argument to create method. This function is going to implement the behaviour of our observable. In this case , it’s the function that is going to do network fetch. This function takes one parameter which is known as observer. The observer is what is going to allow us to emit new values , error out the observable or complete the observable. So the observable here(Http$), does not allow us to emit values on its behalf, instead we can only subscribe to it and get values from stream of values. The observer on the other hand should be kept private here, to the implementation of the observable and this is what will allow us to either emit a new value for our stream i.e calling next, , or error it ,by calling error, or even complete it by calling complete. The observer is what we use internally to implement observable. The function that we pass as argument to create will only be called when we subscribe to our observable

const http$ = Observable.create(observer => {

fetch('/api/courses')

.then(response => {

return response.json();

})

.then(body => {

observer.next(body);

observer.complete();

})

.catch(err => {

observer.error(err);

});

});

5))**of**

ngOnInit() {

const source1$ = interval(2000);

const source2$ = of(4, 5, 6);

const source3$ = of(7, 8, 9);

const result$ = concat(source1$, source2$, source3$);

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

}

our observable will emit these values that we pass it of and after emitting these it will complete.

6)**fromPromise**

It generates observable from promise.

const saveCourse$ = fromPromise(fetch(`/api/courses/${this.course.id}`, {

method: 'PUT',

body: JSON.stringify(changes),

headers: {

'content-type': 'application/json'

}

}));

saveCourse$.subscribe();

In order to trigger http request we have to subscribe to this observable

Operators-

1)**Map**

2)**shareReplay**

So we want to somehow share the execution of this particular stream across multiple subscribers, so we somehow want to avoid the default observable behaviour which is to create a complete new stream by subscription. Instead we want this stream(courses$) to be executed only once and then result of that stream we want it to be replayed to each new subscriber

ngOnInit() {

const http$ = createHttpObservable("/api/courses");

const courses$: Observable<Course[]> = http$

.pipe(

tap(() => console.log('HTTP request executed')),

map(res => Object.values(res['payload']) ),

shareReplay()

);

this.beginnerCourses$ = courses$

.pipe(

map(courses => courses

.filter(course => course.category === 'BEGINNER')));

this.advancedCourses$ = courses$

.pipe(

map(courses => courses

.filter(course => course.category === 'ADVANCED')));

}

3)**tap**

Tap is operator that is meant to be used to produce side effects in our observable chain. So whenever we want to update something outside of observable chain such as for ex updating here a variable at level of the component or in this case to issue a logging statement

Tap is operator that is meant to be used to produce side effects in our observable chain. So whenever we want to update something outside of observable chain such as for ex updating here a variable at level of the component or in this case to issue a logging statement

4)**filter**

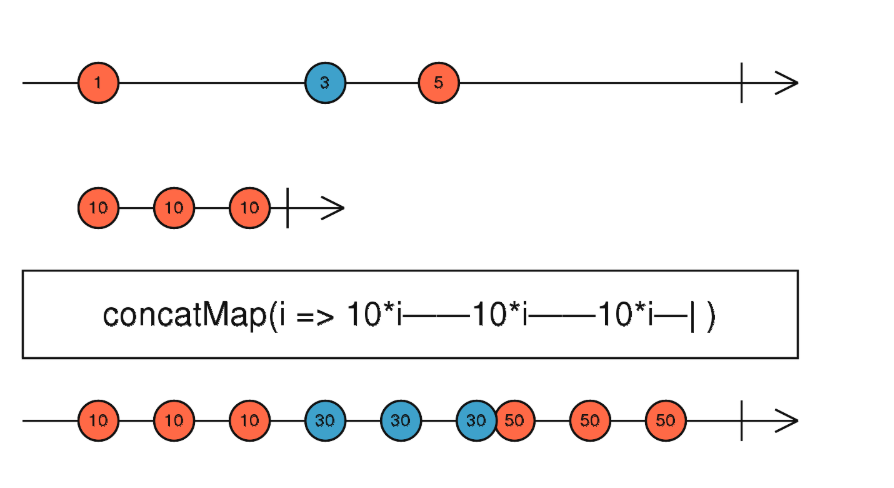
This operator takes a function as argument, this function receives value of emitted in stream as argument. this function returns true or false. For each value in stream that fuction is executed. Only if this function returns true then that value in stream makes it to output.

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

var clicksOnDivs = clicks.filter(ev => ev.target.tagName === 'DIV');

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

5)**concatmap –**



We have source observable that is emitting here series of values- 1,3,5. To concatMap we pass mapping function. what mapping function does is it takes input value and then it produces observables. This observable will emit 3 values for each source observable , each value is multiplied by 10 and then it will complete. So it’s a function that transform value into observable. so here is how concatMap works. We are going to take values of first observable until it completes and for each value for source observable we are going to create a second derived observable. so in our case values of the first observable are going to be valid form values that are getting emitted over time. our derived observable is our save operation. Here derived observable is emitting multiple values but in our case we are going to do http put call to backend and we are going to complete it immediately. But idea is same we take 1 value from source observable and we convert it into an observable. now as long as derived observable is emitting values, those values will be emitted here in output of concatMap. Only when derived observable completes, only in that case we are going to be creating second derived observable from second values of source observable. so once derived observable completes, we take next value from source observable and we are going to create second derived observable. and this process goes on until source observable and last derived observable(which is derived from last value of source observable) completes. Then our resultant observable also completes.

**Code-**

ngOnInit() {

this.form.valueChanges

.pipe(

filter(() => this.form.valid),

concatMap(changes => this.saveCourse(changes))

)

.subscribe();

}

saveCourse(changes) {

return fromPromise(fetch(`/api/courses/${this.course.id}`, {

method: 'PUT',

body: JSON.stringify(changes),

headers: {

'content-type': 'application/json'

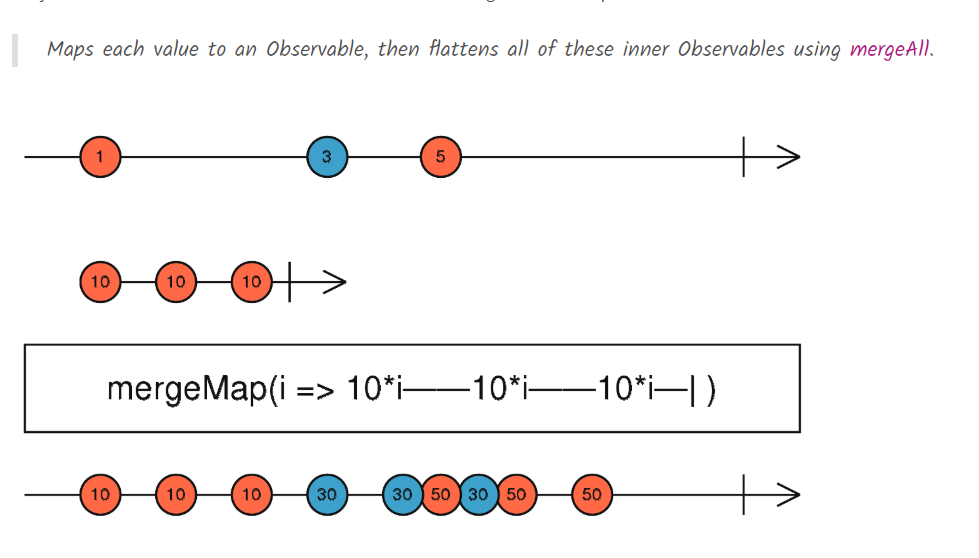
}

}));

}

**6)mergeMap**

but if you find yourself in situation where you would like to perform multiple calls to your backend in parallel and fetch results of each call as they arrive over time. then in that case you can use mergeMap operator



mergeMap is very similar to concat map that we have already covered. Principle is same , we are going to take values of source observable and we are going to apply a mapping function, that is going to take values and produce a new observable. we took value1 and by running mapping function we produce an observable that emitted 3 times 10. These3 values appear in final result. Now take a look at what happens if several other values are emitted without a large gap between them. Second value 3 emits 2 values 30 and 30 but before it emits third value., 5 is emitted from source observable. so we are going to create a new observable using this mapping function and we are going to subscribe to it and start emitting values without having to wait that the previous observable all have completed. We are deriving from each value(of source observable) a new observable, subscribing to everything in parallel and outputting the values as they come from each source observable into the output. We will only complete the output observable when source observable has completed. In video we replace concatMap with mergeMap. Then in network tab we saw that all save request are being made in parallel.

Code-

var letters = Rx.Observable.of('a', 'b', 'c');

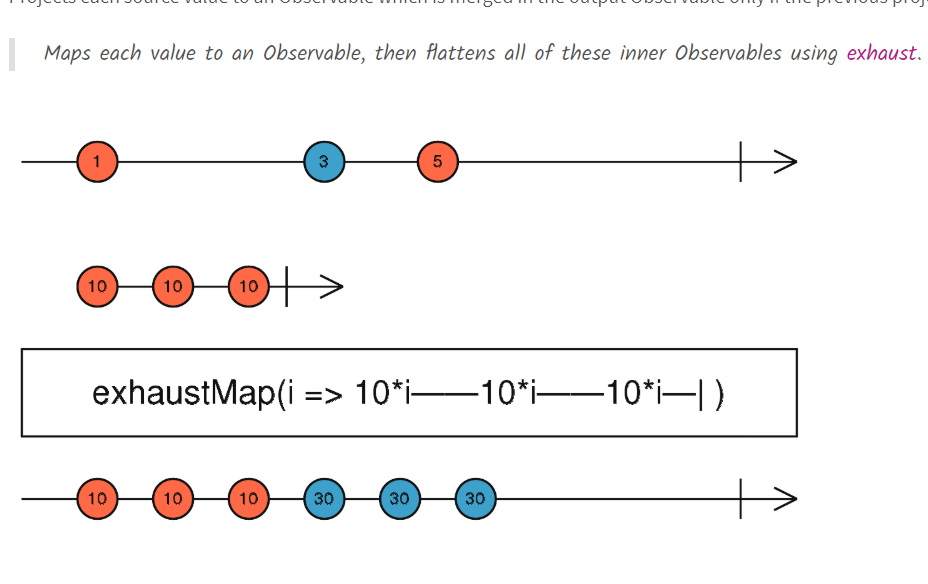
var result = letters.mergeMap(x =>

Rx.Observable.interval(1000).map(i => x + i)

);

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

7)**exhaustMap**



Source observable emits multiple values 1,3,5. As usual each value is going to be transformed into separate observable by using mapping function that we pass to exhaustMap. It takes a value and returns a observable. this derived observable emits multiple values and then completes. values emitted by derived observable(by first value of source observable) are passed to output of exhaut map. This derived observable will complete. Critical thing is if our derived observable(form first value) is still emitting values(not completed) and our source observable emits another value, then that value will be ignored. That is why we do not see values corresponding to 5 in output observable. Because 5 was emitted from source observable at a time when derived observable from value 3 was not completed. So as a result 5 was ignored. so derived observable was not created from 5.

So critical notion here is ignoring values while current observable is till ongoing. This is what we want in our code. We want to ignore clicks if a save request is still ongoing.

Code-

ngAfterViewInit() {

fromEvent(this.saveButton.nativeElement, 'click')

.pipe(

exhaustMap(() => this.saveCourse(this.form.value))

)

.subscribe();

}

saveCourse(changes) {

return fromPromise(fetch(`/api/courses/${this.course.id}`, {

method: 'PUT',

body: JSON.stringify(changes),

headers: {

'content-type': 'application/json'

}

}));

}

8)**switchMap**

Function-

1)**concat**-

here we want to combine these 2 observable togather, to first emit source 1 values and then only after this observable completes only then we would like to show the values of second observable. when all observables complete(here we have 2), then our resultant concatenation observable will also complete

ngOnInit() {

const source1$ = of(1, 2, 3);

const source2$ = of(4, 5, 6);

const source3$ = of(7, 8, 9);

const result$ = concat(source1$, source2$, source3$);

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

}

Output-

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

So the key notion when it comes to concatenation is the notion of completion. If source1$ observable never completes, then values of other observables will never appear on screen. Code-

ngOnInit() {

const source1$ = interval(2000);

const source2$ = of(4, 5, 6);

const source3$ = of(7, 8, 9);

const result$ = concat(source1$, source2$, source3$);

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

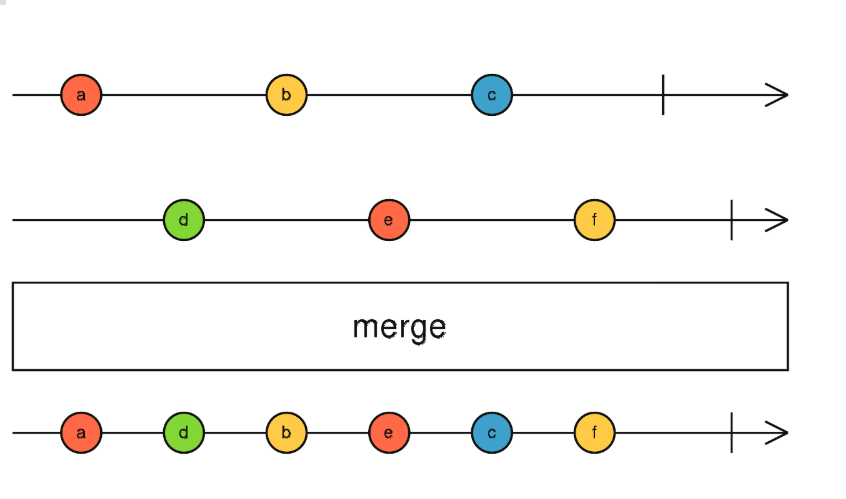
}

Here we will never see values from source2$ and source3$ because source1$ never completes.

Doubts-

Lecture 15, how to make http requests. As fetch method again returns the promise.

2)**merge**



Merge should be used, when we want to take multiple observables, subscribe to all of them and notice here at the same time, and take the values of each of these observable. mergeMap is ideal for performing asynchronous operation in parallel. When one of these observable emits a value we immediately add it to the output. Our combined merge observable will complete when all observables are completed. Also if any of these observable throws an error then resultant observable is going to throw an error immediately.

ngOnInit() {

const interval1$ = interval(1000);

const interval2$ = interval1$.pipe(map(val => val \* 10));

const result$ = merge(interval1$, interval2$);

result$.subscribe(val => console.log(val));

}

Merge straregy is ideal for performing long running operations in parallel and getting the results of each of these operations combined. lets see practical example of this.