*RxJS Operators Explained with Example (2022)*

RxJS Tutorial - Table of Contents

1. [RxJS Observable](https://www.techgeeknext.com/angular/angular-rxjs-observables)
2. [RxJS Subject](https://www.techgeeknext.com/angular/angular-rxjs-subject)
3. [**RxJS Operators**](https://www.techgeeknext.com/angular/angular-rxjs-operators)
4. [RxJS - zip, combineLatest, withLatestFrom](https://www.techgeeknext.com/angular/angular-rxjs-combine-latest-zip)
5. [RxJS Higher-Order Observable Mapping](https://www.techgeeknext.com/angular/angular-rxjs-higher-order-mapping-operator)
6. [RxJS concatMap](https://www.techgeeknext.com/angular/angular-rxjs-concatmap-operator)
7. [RxJS mergeMap](https://www.techgeeknext.com/angular/angular-rxjs-mergemap-operator)
8. [RxJS switchMap](https://www.techgeeknext.com/angular/angular-rxjs-switchmap-operator)

Lets understand first what operators really are.

Observable produces a sequence of data and we can subscribe to fetch it but what if we need to do some modifications to initial data, before we feed them to some component HTML template.

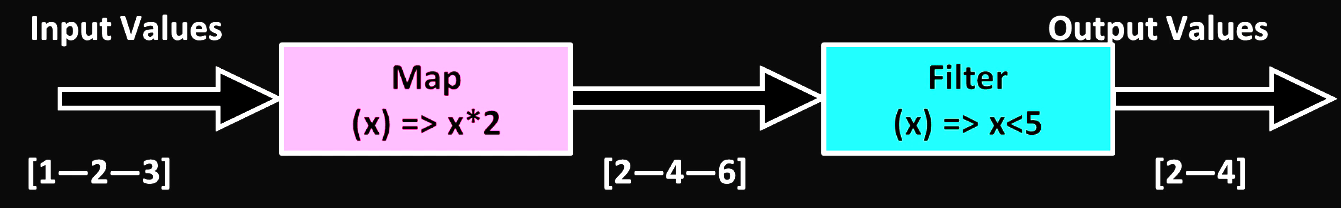
As RxJS implements declarative programming parity, it means that each value can be transformed with predefined function or as we name it in RxJS, operators.

RxJS Operators:

An operator is simply a method that acts on an Observable and changes the stream in some way. Operators are by nature immutable. This immutability makes the code easier to test and reason about.

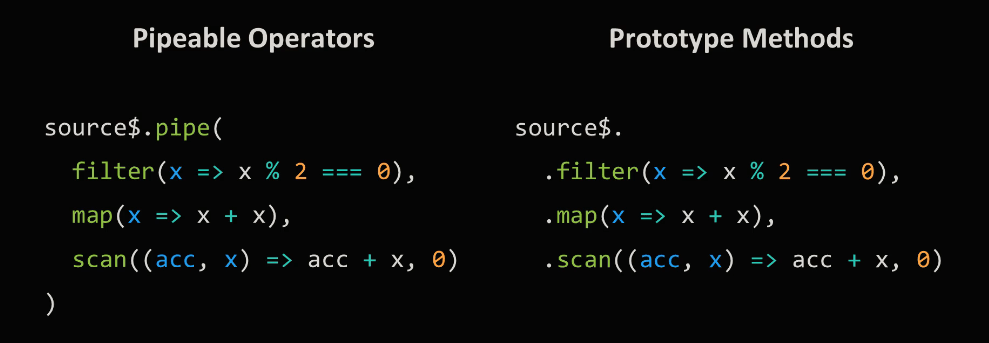
Let's review definitions of operators from pre ActiveX project:

*An operator is a function which creates a new observable based on the input observed. The purpose of operator, to modify or filter originally emitted values in the way we need for the project tasks.*

[](https://www.techgeeknext.com/img/rxjs/rxjs-operators.png)As can be seen in the diagram, operators transfer values to an observable new output and change the values simultaneously.

There are two types of operators :

* Pipeable operators.
* Observable prototype methods.

So what is the difference and which one should we use.[](https://www.techgeeknext.com/img/rxjs/rxjs-operators-type.png)So one uses pipeable operators and another prototype method both of which do the same thing.

Pipeable operator is just a function so it becomes easier to combine existing operators in custom operators with specific logic. Starting from RxJS version 5.5, they are announced as preferable way of using operators.

On the other side, operators in prototype do not use Tree shaking.. if you did not know that Tree Shaking is in the process of deleting the code imported but not used in the program to reduce bundled size.

RxJS from Operator

*The from() operator allows us to create an Observable from some other asynchronous/synchronous concept. It's really powerful when almost anything (array, promise, or iterable) can be made into an Observable, as this allows for rich composition.*

Let's see typical example of from Operator

// RxJS v6+

import { from } from 'rxjs';

//emit array as a sequence of values

let stream$ = from([1,2,3,4]);

//output: 1,2,3,4

stream$.subscribe( data => console.log(data));

RxJS Map Operator

*Map operator applies a given function to each value emitted by source observable and any resulting values as an observable.*

Let's see simple example of Map Operator

**EXAMPLE 1:** Add 10 to each number (RxJS v6+)

import { from } from 'rxjs';

import { map } from 'rxjs/operators';

//emit (1,2,3,4)

const source = from([1, 2, 3, 4]);

//add 10 to each value

const example = source.pipe(map(val => val + 10));

//output: 11,12,13,14

const subscribe = example.subscribe(val => console.log(val));

**EXAMPLE 2:** MAP Object property (RxJS v6+)

import { from } from 'rxjs';

import { map } from 'rxjs/operators';

//emit ({name: 'Joe', age: 30}, {name: 'Frank', age: 20},{name: 'Ryan', age: 50})

const source = from([

{ name: 'John', age: 20 },

{ name: 'Alek', age: 25 },

{ name: 'Rosy', age: 18 }

]);

//grab each persons name, could also use pluck for this scenario

const example = source.pipe(map(({ name }) => name));

//output: "John","Alek","Rosy"

const subscribe = example.subscribe(val => console.log(val));

RxJS Filter Operator

*Filter operator, filters source observable items by only omitting those that satisfy a specified predicate.*

Filter operator takes items emitted by the source observable and emit only those value that satisfy a specified predicate. As you know, predicate is a function that returns true or false.

Let's see simple example of filter Operator

import { from } from 'rxjs';

import { filter } from 'rxjs/operators';

let stream$ = from([1, 2]).map( x => x +1 ) .filter( x > 2 );

// output : data 3

stream$.subscribe( data => console.log('data', data))

Here, we can see that we are using the .map() operator and .filter() to change our stream's data. map() operates on each value in the stream by incrementing each value by one. .filter() operates on the changed stream; a change brought about by calling .map(). It also operates on each value in the stream but conditionally decides what should be emitted. The end result is only one value being emitted, 3.

RxJS distinct Operators

*The main purpose of distinct operators is preventing subscribers on next handler to run, if observable emits same value again. Sometimes it makes sense since why to do view update or recalculate values if nothing is actually changed.*

At the moment, RxJS has three distinct operators :

* **distinctUntilChanged** - Distinct until changed operator emits all items that are distinct by comparison from only one previous item, complicating compared logic is possible here.
* **distinctUntilKeyChanged** - Distinct until key changed works similar but allows to specify which key value you want to compare.
* **distinct**

- Distinct operator emits all items that are distinct by comparison from all previous items in internal buffer, you can empty its internal buffer with the second flush argument which should be an observable.

distinctUntilChanged

distinct until changed operator emits all items that are distinct by comparison (===) from only one previous item, complicating compared logic is possible here.

Distinct until changed returns an observable that emits all items emitted by the source observable that are distinct by comparison from the previous item. Pay attention that operator only compares next value with one previous value.

Let's see an example distinctUntilChanged with basic value (RxJS v6+)

import { from } from 'rxjs';

import { distinctUntilChanged } from 'rxjs/operators';

// only output distinct values, based on the last emitted value

const source$ = from([1, 1, 2, 2, 3, 3]);

source$

.pipe(distinctUntilChanged())

// output: 1,2,3

.subscribe(console.log);

To compare previous and next values, it uses comparison function as a first param. If a comparator function is not provided, the Equality check is used by default. If a comparator function is provided, then it will be called for each item to test for whether or not this value should be emitted.

If values are objects, you can use second pair on key selector to specify which key of the value you want to compare. But actually, the same result can be easily achieved with the justa comparison function.

distinctUntilKeyChanged

Now let's review distinct until key changed. Actually, it is very similar to distinct until changed but it has some minor differences. They have different arguments order and key name argument here is a string, so we can use only with flat objects without deep nesting. Key selector argument of distant until changed operator is a function, so we can reach object property value at any depth.

Both statements do the same thing, which one you want to use it's up to you to decide. Definitely if value is a flat object without deep nesting, distinct until key changed demands much less to type.

Example with object value (RxJS v6+)

from([{v:1}, {v:2}, {v:2}, {v:3}])

.pipe(

// returns all objects as, quality comparison will not work since each object has different reference.

distinctUntilChanged()

// output : {v:1}, {v:2}, {v:2}, {v:3}

// with comparison callback function as the first argument for distinct until changed by using second parameter key selector

distinctUntilChanged((prev, next) => prev.v === next.v)

// output : {v:1}, {v:2}, {v:3}

// used null as the first argument so the default strict JavaScript equality will be used.

distinctUntilChanged(null, (item) => item.v)

// output : {v:1}, {v:2}, {v:3}

// very similar to distinct until changed but it has some minor differences.

// They have different arguments order and key name argument here is a string

distinctUntilKeyChanged('v')

// output : {v:1}, {v:2}, {v:3}

)

distinct

let's try distinct. As you can see, it compares with all previous values.

Let's see an example of distinct

import { of } from 'rxjs';

import { distinct } from 'rxjs/operators';

of(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)

.pipe(distinct())

// OUTPUT: 1,2,3,4,5

.subscribe(console.log);

What is same and what is different between distinct until changed. Distinct doesn't have comparison callback so some complicated compare logic is not possible here.

It has key selector so we can specify what property value should be compared, but the main difference is next. Distinct operator compares next value, not only with one but with all previous values, so it should store values somewhere in a buffer and if number of values is significant, it can cause a lack of memory issue.

To prevent this, we have second argument here named flush. Flush should be observable, each time flush emits, all buffered values will be erased so the process starts from the beginning. For example, to empty operator internal buffer each ten seconds, we can use interval function.

As you can see, distinct compares next value with all previous values from a sequence, that's why ending value here is filtered out despite of previous item is different

PART-11

1. [RxJS Observable](https://www.techgeeknext.com/angular/angular-rxjs-observables)
2. [RxJS Subject](https://www.techgeeknext.com/angular/angular-rxjs-subject)
3. [**RxJS Operators**](https://www.techgeeknext.com/angular/angular-rxjs-operators)
4. [RxJS Higher-Order Observable Mapping](https://www.techgeeknext.com/angular/angular-rxjs-higher-order-mapping-operator)
5. [RxJS concatMap](https://www.techgeeknext.com/angular/angular-rxjs-concatmap-operator)
6. [RxJS mergeMap](https://www.techgeeknext.com/angular/angular-rxjs-mergemap-operator)
7. [RxJS switchMap](https://www.techgeeknext.com/angular/angular-rxjs-switchmap-operator)

[RxJS Interview Questions](https://www.techgeeknext.com/angular/angular-interview-questions)

Overview

In this PART 2

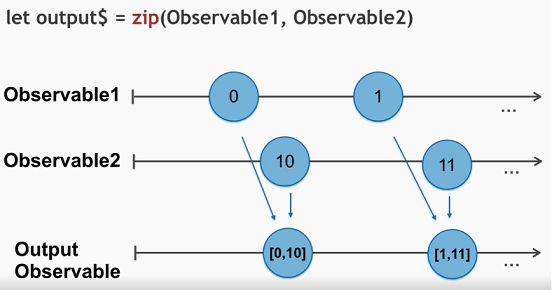
we will learn new functions and operator that allows to combine a few observable sequences in a different ways. We will review Zip and combineLatest functions and withLatestFrom operator.

Composing functions and operators usually accept observables as their params and also they return observable that emits array with values produced by argument observables.

This result observable emission logic is different depending on which operator or function we use, let's start reviewing them one by one.

RxJS Zip function:

*Zip Combines mulitple Observables emitted values with same index. Return array of respective values.*

[](https://www.techgeeknext.com/img/rxjs/rxjs-zip-function.png)

As can be seen in the diagram, So we've had two observable sequences to zip function and want to receive array of values, only after both observables produce values. So the output Observable value would be : [[[0, 10], [1, 11]]

**zip (observable1, [ observable2, observable3,..., resultSelector]);**

* Zip waits until all the params observables emit values with same index number and only then emits array of these values.
* Zip accepts one or more observables as a params and you can also provide last param as function it is treated as a resultSelector
* resultSelector : You can use result selector for additional emitted data manipulation.

Here is a code example of using zip function :

import { zip, of } from 'rxjs';

import { map } from 'rxjs/operators';

let age$ = of<number>(27, 25, 29);

let name$ = of<string>('Foo', 'Bar', 'Beer');

let isDev$ = of<boolean>(true, true, false);

zip(age$, name$, isDev$).pipe(

map(([age, name, isDev]) => ({ age, name, isDev })),

)

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

// outputs

// { age: 27, name: 'Foo', isDev: true }

// { age: 25, name: 'Bar', isDev: true }

// { age: 29, name: 'Beer', isDev: false }

We have three observable sequences here, ages name and isdev. Each of them will emit three values one by one, we feed them to zip function and subscribe to result observable.

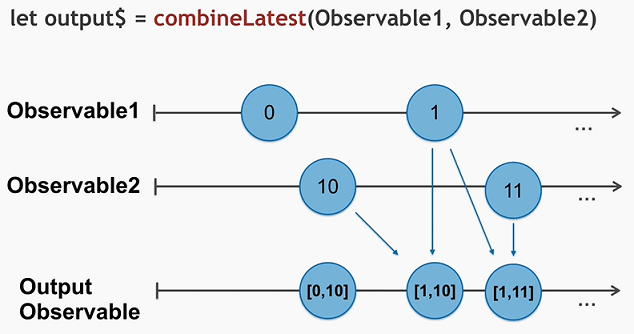
As you can see, each time of the argument observables emit values with same index result observable produces array of these values.

We can use zip function in our web application at time of bootstrap of application or component, where we are waiting for more than one http data to render the component. So zip function can help in better state management.

RxJS combineLatest function

*CombineLatest combines multiple observables to create an observable, those values are calculated from the latest values of each of its input observables, once any of them emit irrespective of their index.*

Please note that combineLatest result observable only starts producing values only if all arguments observable has at least one value already emitted, and it doesn't wait for all argument observables to have same index value emitted

Let's take a look at this diagram :[](https://www.techgeeknext.com/img/rxjs/rxjs-combinelatest-function.png)

As you can see, if all argument observables has at least one value emitted, combined latest result observable produces array of emitted values ([0,10]). And each time any of param observable sends new data result observable will meet new array of values as well. ([1,10],[1,11])

Let's see simple example

import { combineLatest, of } from 'rxjs';

import { map } from 'rxjs/operators';

const amount = of(70, 72, 76, 79, 75);

const conversionRate = of(0.06, 0.07, 0.08);

const fees = combineLatest(amount, fees).pipe(

map(([a, r]) => (a \* r)),

);

fees.subscribe(x => console.log('commission is ' + x));

RxJS withLatestFrom Operator

*withlatestFrom combine the source observable with other observables to create a result observable. Those values are calculated from the latest values of each, only when source emits so source emission is the trigger.*