



RXjs

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

RxJS is a javascript library that uses observables to work with reactive programming that deals with asynchronous data calls, callbacks and event-based programs. RxJS can be used with other javascript libraries and frameworks. It is supported by Javascript and also with typescript.

Audience

This tutorial is designed for software programmers who want to learn the basics of Reactive extension for Javascript (RxJS) and its programming concepts in simple and easy way. This tutorial will give you enough understanding on various functionalities of RxJS with suitable examples.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of Javascript.

Copyright & Disclaimer

© Copyright 2019 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial	ii
Audience.....	ii
Prerequisites.....	ii
Copyright & Disclaimer	ii
Table of Contents	iii
1. RxJS — Overview	1
What is RxJS?	1
Features of RxJS.....	1
When to use RxJS?.....	2
Advantages of using RxJS.....	2
Disadvantages of using RxJS	2
2. RxJS — Environment Setup	3
NODEJS and NPM Installation	3
RxJS Package Installation	4
Testing RxJS in Browser	6
3. RxJS6 — Latest Updates	12
Imports for operators	12
Import of Methods to create Observables	12
Import of Observables.....	13
Import of Subject.....	13
How to use operators in RxJS 6?	13
4. RxJS — Observables	15
Create Observable	15
Subscribe Observable	16
Execute Observable.....	16
5. RxJS — Operators	18

Working with Operators	18
Creation Operators	19
Mathematical Operators	20
Join Operators	21
Transformation Operators	21
Filtering Operators	22
Utility Operators	23
Conditional Operators	24
Multicasting Operators	25
Error Handling Operators	26
ajax	26
from	27
fromEvent	28
fromEventPattern	29
interval	30
of	30
range	31
throwError	32
timer	33
iif	34
count	35
max	36
min	37
Reduce	38
concat	39
forkJoin	40
merge	41
race	41

buffer	42
bufferCount	43
bufferTime	45
bufferToggle	46
bufferWhen	47
expand	48
groupBy	49
map.....	50
mapTo.....	51
mergeMap	52
switchMap	53
window	53
debounce.....	54
debounceTime.....	55
distinct.....	56
elementAt.....	57
filter	58
first	59
last	59
ignoreElements.....	60
sample	61
skip	62
throttle	62
tap	63
delay	64
delayWhen	65
subscribeOn.....	67
timeInterval.....	68

timestamp	69
timeout	70
toArray.....	71
defaultIfEmpty	72
every	72
find.....	73
findIndex.....	74
isEmpty	75
multicast	75
publish	77
publishBehavior	78
publishLast.....	79
publishReplay	80
share	81
catchError	82
retry.....	83
6. RxJS — Working with Subscription.....	85
count() operator	85
7.RxJS — Working with Subjects	86
Create a subject.....	86
What is the Difference between Observable and Subject?.....	89
Behaviour Subject.....	91
Replay Subject	92
AsyncSubject	93
8. RxJS — Working with Scheduler.....	95
9. RxJS — Working with RxJS and Angular	97
10. RxJS — Working with RxJS and ReactJS.....	99

1. RxJS — Overview

This chapter deals with information about features, advantages and disadvantages of RxJS. Here, we will also learn when to use RxJS.

The full form of RxJS is **Reactive Extension for Javascript**. It is a javascript library that uses observables to work with reactive programming that deals with asynchronous data calls, callbacks and event-based programs. RxJS can be used with other Javascript libraries and frameworks. It is supported by javascript and also with typescript.

What is RxJS?

As per the official website of [RxJS](https://rxjs.dev), it is defined as a library for composing asynchronous and event-based programs by using observable sequences. It provides one core type, the Observable, satellite types (Observer, Schedulers, Subjects) and operators inspired by Array#extras (map, filter, reduce, every, etc.) to allow handling asynchronous events as collections.

Features of RxJS

In RxJS, the following concepts takes care of handling the async task:

Observable

An observable is a function that creates an observer and attaches it to the source where values are expected, for example, clicks, mouse events from a dom element or an Http request, etc.

Observer

It is an object with next(), error() and complete() methods, that will get called when there is interaction to the with the observable i.e. the source interacts for an example button click, Http request, etc.

Subscription

When the observable is created, to execute the observable we need to subscribe to it. It can also be used to cancel the execution.

Operators

An operator is a pure function that takes in observable as input and the output is also an observable.

Subject

A subject is an observable that can multicast i.e. talk to many observers. Consider a button with an event listener, the function attached to the event using addlistener is called every time the user clicks on the button similar functionality goes for subject too.

Schedulers

A scheduler controls the execution of when the subscription has to start and notified.

When to use RxJS?

If your project consists of lots of async task handling than RxJS is a good choice. It is loaded by default with the Angular project.

Advantages of using RxJS

The following are the advantages of using RxJS:

- RxJS can be used with other Javascript libraries and frameworks. It is supported by javascript and also with typescript. Few examples are Angular, ReactJS, Vuejs, nodejs etc.
- RxJS is an awesome library when it comes to the handling of async tasks. RxJS uses observables to work with reactive programming that deals with asynchronous data calls, callbacks and event-based programs.
- RxJS offers a huge collection of operators in mathematical, transformation, filtering, utility, conditional, error handling, join categories that makes life easy when used with reactive programming.

Disadvantages of using RxJS

The following are the disadvantages of using RxJS:

- Debugging the code with observables is little difficult.
- As you start to use Observables, you can end up your full code wrapped under the observables.

2. RxJS — Environment Setup

In this chapter, we are going to install RxJS. To work with RxJS, we need the following setup:

- NodeJS
- Npm
- RxJS package installation

NODEJS and NPM Installation

It is very easy to install RxJS using npm. You need to have nodejs and npm installed on your system. To verify if NodeJS and npm is installed on your system, try to execute the following command in your command prompt.

```
E:\>node -v && npm -v  
v10.15.1  
6.4.1
```

In case you are getting the version, it means nodejs and npm is installed on your system and the version is 10 and 6 right now on the system.

If it does not print anything, install nodejs on your system. To install nodejs, go to the homepage <https://nodejs.org/en/download/> of nodejs and install the package based on your OS.

The download page of nodejs will look like the following:

nodejs.org/en/download/

node

HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | NEWS | FOUNDATION

Downloads

Latest LTS Version: **10.16.3** (includes npm 6.9.0)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users	Current Latest Features
<p>Windows Installer</p> <p>node-v10.16.3-x84.msi</p>	<p>macOS Installer</p> <p>node-v10.16.3.pkg</p>
	<p>Source Code</p> <p>node-v10.16.3.tar.gz</p>

Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)	64-bit	
macOS Binary (.tar.gz)	64-bit	

Based on your OS, install the required package. Once nodejs is installed, npm will also be installed along with it. To check if npm is installed or not, type `npm -v` in the terminal. It should display the version of the npm.

RxJS Package Installation

To start with RxJS installation, first create a folder called **rxjsproj/** where we will practice all RxJS examples.

Once the folder **rxjsproj/** is created, run command **npm init**, for project setup as shown below:

```
E:\>mkdir rxjsproj

E:\>cd rxjsproj

E:\rxjsproj>npm init
```

Npm init command will ask few questions during execution, just press enter and proceed. Once the execution of npm init is done, it will create **package.json** inside **rxjsproj/** as shown below:

```
rxjsproj/
  package.json
```

Now you can install rxjs using below command:

```
npm install ---save-dev rxjs
```

```
E:\rxjsproj>npm install --save-dev rxjs
npm notice created a lockfile as package-lock.json. You should commit this file.

npm WARN rxjsproj@1.0.0 No description
npm WARN rxjsproj@1.0.0 No repository field.

+ rxjs@6.5.3
added 2 packages from 7 contributors and audited 2 packages in 21.89s
found 0 vulnerabilities
```

We are done with RxJS installation. Let us now try to use RxJS, for that create a folder **src/** inside **rxjsproj/**

So, now, we will have the folder structure as shown below:

```
rxjsproj/
  node_modules/
  src/
  package.json
```

Inside **src/** create a file **testrx.js**, and write the following code:

testrx.js

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

map(x => x * x)(of(1, 2, 3)).subscribe((v) => console.log(`Output is: ${v}`));
```

When we go to execute the above code in command prompt, using command: **node testrx.js**, it will display error for import, as nodejs does not know what to do with import.

To make import work with nodejs, we need to install ES6 modules package using npm as shown below:

```
E:\rxjsproj\src>npm install --save-dev esm
npm WARN rxjsproj@1.0.0 No description
npm WARN rxjsproj@1.0.0 No repository field.
```

```
+ esm@3.2.25
added 1 package from 1 contributor and audited 3 packages in 9.32s
found 0 vulnerabilities
```

Once the package is installed, we can now execute **testrx.js** file as shown below:

```
E:\rxjsproj\src>node -r esm testrx.js
Output is: 1
Output is: 4
Output is: 9
```

We can see the output now, that shows RxJS is installed and ready to use. The above method will help us test RxJS in the command line. In case, you want to test RxJS in the browser, we would need some additional packages.

Testing RxJS in Browser

Install following packages inside rxjsproj/ folder:

```
npm install --save-dev babel-loader @babel/core @babel/preset-env webpack
webpack-cli webpack-dev-server
```

```
E:\rxjsproj>npm install --save-dev babel-loader @babel/core @babel/preset-env
webpack webpack-cli webpack-dev-server
```

```
npm WARN rxjsproj@1.0.0 No description
```

```
npm WARN rxjsproj@1.0.0 No repository field.
```

```
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9
```

```
(node_modules\fse
```

```
vents):
```

```
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for
fsevents@
```

```
1.2.9: wanted {"os":"darwin","arch":"any"} (current:
{"os":"win32","arch":"x64"}
)
```

```
+ webpack-dev-server@3.8.0
```

```
+ babel-loader@8.0.6
```

```
+ @babel/preset-env@7.6.0
```

```
+ @babel/core@7.6.0
+ webpack-cli@3.3.8
+ webpack@4.39.3
added 675 packages from 373 contributors and audited 10225 packages in 255.567s
found 0 vulnerabilities
```

To start the server to execute our Html file, we will use webpack-server. The command "publish" in package.json will help us start as well as pack all the js files using webpack. The packed js files which are our final js file to be used is saved at the path */dev* folder.

To use webpack, we need to run **npm run publish** command and the command is added in package.json as shown below:

Package.json

```
{
  "name": "rxjsproj",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "publish": "webpack && webpack-dev-server --output-public=/dev/",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.6.0",
    "@babel/preset-env": "^7.6.0",
    "babel-loader": "^8.0.6",
    "esm": "^3.2.25",
    "rxjs": "^6.5.3",
    "webpack": "^4.39.3",
    "webpack-cli": "^3.3.8",
```

```

    "webpack-dev-server": "^3.8.0"
  }
}

```

To work with webpack we must first create a file called webpack.config.js that has the configuration details for webpack to work.

The details in the file are as follows:

```

var path = require('path');

module.exports = {
  entry: {
    app: './src/testrx.js'
  },
  output: {
    path: path.resolve(__dirname, 'dev'),
    filename: 'main_bundle.js'
  },
  mode: 'development',

  module: {
    rules: [

      {

        test: /\.js$/,
        include: path.resolve(__dirname, 'src'),

        loader: 'babel-loader',

        query: {
          presets: ['@babel/preset-env']
        }
      }
    ]
  }
};

```

The structure of the file is as shown above. It starts with a path that gives the current path details.

```
var path = require('path'); //gives the current path
```

Next is module.exports object which has properties entry, output, and module. Entry is the start point. Here, we need to give the start js file we want to compile.

```
entry: {
  app: './src/testrx.js'
},
```

path.resolve(__dirname, 'src/testrx.js') -- will look for src folder in the directory and testrx.js in that folder.

Output

```
output: {
  path: path.resolve(__dirname, 'dev'),

  filename: 'main_bundle.js'
},
```

The output is an object with path and filename details. path will hold the folder in which the compiled file will be kept and the filename will tell the name of the final file to be used in your .html file.

Module

```
module: {
  rules: [
    {

      test: /\.js$/,

      include: path.resolve(__dirname, 'src'),
      loader: 'babel-loader',
      query: {
        presets: ['@babel/preset-env']
      }
    }
  ]
}
```

Module is object with rules details which has properties i.e. test, include, loader, query. The test will hold details of all the js file ending with .js and .jsx. It has the pattern which will look for .js at the end in the entry point given.

Include tells the folder to be used for looking at the files.

The loader uses babel-loader for compiling code.

The query has property presets which is an array with value '@babel/preset-env'. It will transpile the code as per the ES environment you need.

The final folder structure will be as follows:

```
rxjsproj/
  node_modules/
  src/
    testrx.js

  index.html
  package.json
  webpack.config.js
```

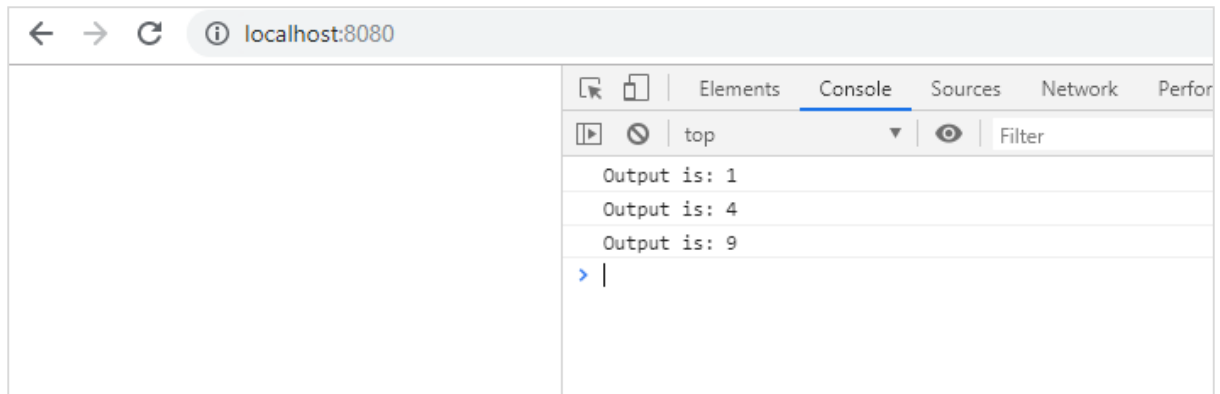
Run Command

npm run publish will create dev/ folder with main_bundle.js file in it. The server will be started and you can test your index.html in the browser as shown below.

```
E:\rxjsproj>npm run publish
> rxjsproj@1.0.0 publish E:\rxjsproj
> webpack && webpack-dev-server --output-public=/dev/

Hash: 28cb4821f7f87cb38e5e
Version: webpack 4.39.3
Time: 42464ms
Built at: 09/07/2019 5:25:49 PM
    Asset      Size  Chunks             Chunk Names
main_bundle.js  626 KiB          0  [emitted]  app
Entrypoint app = main_bundle.js
[./src/testrx.js] 191 bytes {app} [built]
  + 200 hidden modules
i ?wds?: Project is running at http://localhost:8080/
i ?wds?: webpack output is served from /
i ?wds?: Content not from webpack is served from E:\rxjsproj
i ?wdm?: Hash: ac26126f37561ea38bdd
Version: webpack 4.39.3
Time: 8913ms
Built at: 09/07/2019 5:26:10 PM
```

Open the browser and hit the url : <http://localhost:8080/>



The output is shown in the console.

3. RxJS6 — Latest Updates

We are using RxJS version 6 in this tutorial. RxJS is commonly used to deal with reactive programming and used more often with Angular, ReactJS. Angular 6 loads rxjs6 by default.

RxJS version 5 was handled differently in comparison to version 6. The code will break in case you update your RxJS 5 to 6. In this chapter, we are going to see the difference in ways of handling the version update.

In case you are updating RxJS to 6 and don't want to make the code changes, you can do that too, and will have to install the following package.

```
npm install --save-dev rxjs-compact
```

This package will take care of providing backward compatibility and old code will work fine with RxJS version 6. If you want to make the code changes that works fine with RxJS 6, here are the changes that needs to be done.

The packages for operators, observables, subject were restructured and hence, the major changes go in for imports and they are explained below.

Imports for operators

As per version 5, for operators the following import statements should be included:

```
import 'rxjs/add/operator/mapTo'  
import 'rxjs/add/operator/take'  
import 'rxjs/add/operator/tap'  
import 'rxjs/add/operator/map'
```

In RxJS version 6 the imports will be as follows:

```
import {mapTo, take, tap, map} from "rxjs/operators"
```

Import of Methods to create Observables

As per version 5, while working with Observables, the following import methods should be included:

```
import "rxjs/add/observable/from";  
import "rxjs/add/observable/of";  
import "rxjs/add/observable/fromEvent";  
import "rxjs/add/observable/interval";
```

In RxJS version 6 the imports will be as follows:

```
import {from, of, fromEvent, interval} from 'rxjs';
```

Import of Observables

In RxJS version 5, while working with Observables, the following import statements should be included:

```
import { Observable } from 'rxjs/Observable'
```

In RxJS version 6, the imports will be as follows:

```
import { Observable } from 'rxjs'
```

Import of Subject

In RxJS version 5, subject should be included as follows:

```
import { Subject } from 'rxjs/Subject'
```

In RxJS version 6, the imports will be as follows:

```
import { Subject } from 'rxjs'
```

How to use operators in RxJS 6?

pipe() method is available on the observable created. It is added to RxJS from version 5.5. Using pipe() now you can work on multiple operators together in sequential order. This is how the operators were used in RxJS version 5.

Example

```
import "rxjs/add/observable/from";

import 'rxjs/add/operator/max'

let list1 = [1, 6, 15, 10, 58, 2, 40];
from(list1).max((a,b)=>a-b).subscribe(x => console.log("The Max value is "+x));
```

From RxJS version 5.5 onwards, we have to use pipe() to execute the operator:

Example

```
import { from } from 'rxjs';
import { max } from 'rxjs/operators';
```

```
from(list1).pipe(max((a,b)=>a-b)).subscribe(x => console.log("The Max value is "+x));
```

Operators Renamed

During restructuring of the packages some of the operators were renamed as they were conflicting or matching with javascript keywords. The list is as shown below:

Operator	Renamed to
do()	tap()
catch()	catchError()
switch()	switchAll()
finally()	finalize()
throw()	throwError()

4. RxJS — Observables

An observable is a function that creates an observer and attaches it to the source where values are expected from, for example, clicks, mouse events from a dom element or an Http request, etc.

Observer is an object with callback functions, that will get called when there is interaction to the Observable, i.e., the source has interacted for an example button click, Http request, etc.

We are going to discuss following topics in this chapter:

- Create Observable
- Subscribe Observable
- Execute Observable

Create Observable

The observable can be created using observable constructor and also using observable create method and by passing subscribe function as an argument to it as shown below:

testrx.js

```
import { Observable } from 'rxjs';

var observable = new Observable(function subscribe(subscriber) {
    subscriber.next("My First Observable")
});
```

We have created an observable and added a message "My First Observable" using **subscriber.next** method available inside Observable.

We can also create Observable using, Observable.create() method as shown below: -

testrx.js

```
import { Observable } from 'rxjs';

var observer = Observable.create(function subscribe(subscriber) {
    subscriber.next("My First Observable")
});
```

Subscribe Observable

You can subscribe to an observable as follows:

testrx.js

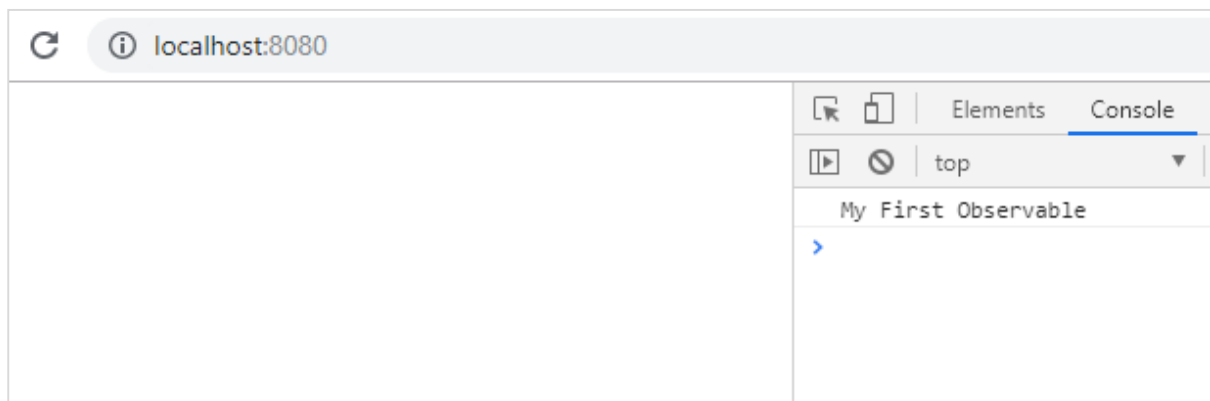
```
import { Observable } from 'rxjs';

var observer = new Observable(function subscribe(subscriber) {
  subscriber.next("My First Observable")
});

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

When the observer is subscribed, it will start the execution of the Observable.

This is what we see in the browser console:



Execute Observable

An observable gets executed when it is subscribed. An observer is an object with three methods that are notified,

next(): This method will send values like a number, string, object etc.

complete(): This method will not send any value and indicates the observable as completed.

error(): This method will send the error if any.

Let us create the observable with all three notifications and execute the same.

testrx.js

```
import { Observable } from 'rxjs';

var observer = new Observable(function subscribe(subscriber) {
  try{
```

```

        subscriber.next("My First Observable");
        subscriber.next("Testing Observable");
        subscriber.complete();
    } catch(e){
        subscriber.error(e);
    }
});
observer.subscribe(x => console.log(x), (e)=>console.log(e),
    ()=>console.log("Observable is complete"));

```

In the above code, we have added, next, complete and error method.

```

try{
    subscriber.next("My First Observable");
    subscriber.next("Testing Observable");
    subscriber.complete();
} catch(e){
    subscriber.error(e);
}

```

To execute next, complete and error, we have to call the subscribe method as shown below:

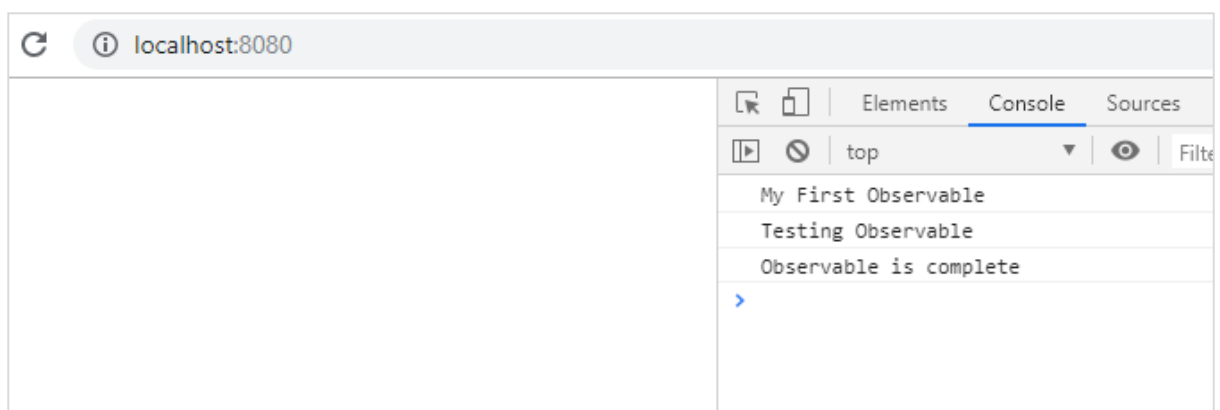
```

observer.subscribe(x => console.log(x), (e)=>console.log(e),
    ()=>console.log("Observable is complete"));

```

The error method will be invoked only if there is an error.

This is the output seen in the browser:



5. RxJS — Operators

Operators are an important part of RxJS. An operator is a pure function that takes in observable as input and the output is also an observable.

Working with Operators

An operator is a pure function which takes in observable as input and the output is also an observable.

To work with operators we need a `pipe()` method.

Example of using `pipe()`

```
let obs = of(1,2,3); // an observable
obs.pipe(
  operator1(),
  operator2(),
  operator3(),
  operator3(),
)
```

In above example we have created a observable using **`of()`** method that takes in values 1, 2 and 3. Now on this observable you can perform different operation using any numbers of operators using `pipe()` method as shown above. The execution of operators will go on sequentially on the observable given.

Below is a working example:

```
import { of } from 'rxjs';
import { map, reduce, filter } from 'rxjs/operators';

let test1 = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
let case1 = test1.pipe(
  filter(x => x % 2 === 0),
  reduce((acc, one) => acc + one, 0)
)

case1.subscribe(x => console.log(x));
```

Output

In above example, we have used filter operator that, filters the even numbers and, next we have used **reduce()** operator that will add the even values and give the result when subscribed.

Here is a list of Observables that we are going to discuss.

- Creation
- Mathematical
- Join
- Transformation
- Filtering
- Utility
- Conditional
- Multicasting
- Error handling

Creation Operators

Following are the operators we are going to discuss in Creation operator category:

Operator	Description
ajax	This operator will make an ajax request for the given URL.
from	This operator will create an observable from an array, an array-like object, a promise, an iterable object, or an observable-like object.
fromEvent	This operator will give output as an observable that is to be used on elements that emit an event for example buttons, clicks, etc.
fromEventPattern	This operator will create an observable from the input function that is used to register event handlers.
interval	This operator will create an Observable for every time for the time given.

of	This operator will take in the arguments passed and convert them to observable.
range	This operator will create an Observable that will give you a sequence of numbers based on the range provided.
throwError	This operator will create an observable that will notify an error.
timer	This operator will create an observable that will emit the value after the timeout and the value will keep increasing after each call.
iif	This operator will decide which Observable will be subscribed.

Mathematical Operators

The following are the operators we are going to discuss in the Mathematical operator category:

Operator	Description
Count	The count() operator takes in an Observable with values and converts it into an Observable that will give a single value
Max	Max method will take in an observable with all values and return an observable with the max value
Min	Min method will take in an observable with all values and return an observable with the min value.
Reduce	<p>In reduce operator, accumulator function is used on the input observable, and the accumulator function will return the accumulated value in the form of an observable, with an optional seed value passed to the accumulator function.</p> <p>The reduce() function will take in 2 arguments, one accumulator function, and second the seed value.</p>

Join Operators

The following are the operators we are going to discuss in the Join operator category.

Operator	Description
concat	This operator will sequentially emit the Observable given as input and proceed to the next one.
forkJoin	This operator will be taken in an array or dict object as an input and will wait for the observable to complete and return the last values emitted from the given observable.
merge	This operator will take in the input observable and will emit all the values from the observable and emit one single output observable.
race	It will give back an observable that will be a mirror copy of the first source observable.

Transformation Operators

The following are the operators we are going to discuss in the Transformation operator category.

Operator	Description
buffer	The buffer operates on an observable and takes in argument as an observable. It will start buffering the values emitted on its original observable in an array and will emit the same when the observable taken as argument, emits. Once the observable taken as arguments emits, the buffer is reset and starts buffering again on original till the input observable emits and the same scenario repeats.
bufferCount	In the case of buffercount() operator, it will collect the values from the observable on which it is called and emit the same when the buffer size given to buffercount matches.
bufferTime	This is similar to bufferCount, so here, it will collect the values from the observable on which it is called and emit the bufferTimeSpan is done. It takes in 1 argument <i>i.e. bufferTimeSpan</i> .

bufferToggle	In the case of bufferToggle() it takes 2 arguments, openings and closingSelector. The opening arguments are subscribable or a promise to start the buffer and the second argument closingSelector is again subscribable or promise an indicator to close the buffer and emit the values collected.
bufferWhen	This operator will give the values in the array form, it takes in one argument as a function that will decide when to close, emit and reset the buffer.
expand	The expand operator takes in a function as an argument which is applied on the source observable recursively and also on the output observable. The final value is an observable.
groupBy	In groupBy operator, the output is grouped based on a specific condition and these group items are emitted as GroupedObservable.
map	In the case of map operator, a project function is applied on each value on the source Observable and the same output is emitted as an Observable.
mapTo	A constant value is given as output along with the Observable every time the source Observable emits a value.
mergeMap	In the case of mergeMap operator, a project function is applied on each source value and the output of it is merged with the output Observable.
switchMap	In the case of switchMap operator, a project function is applied on each source value and the output of it is merged with the output Observable, and the value given is the most recent projected Observable.
window	It takes an argument windowboundaries which is an observable and gives back a nested observable whenever the given windowboundaries emits

Filtering Operators

The following are the operators we are going to discuss in the filtering operator category.

Operator	Description
debounce	A value emitted from the source Observable after a while and the emission is determined by another input given as Observable or promise.
debounceTime	It will emit value from the source observable only after the time is complete.
distinct	This operator will give all the values from the source observable that are distinct when compared with the previous value.
elementAt	This operator will give a single value from the source observable based upon the index given.
filter	This operator will filter the values from source Observable based on the predicate function given.
first	This operator will give the first value emitted by the source Observable.
last	This operator will give the last value emitted by the source Observable.
ignoreElements	This operator will ignore all the values from the source Observable and only execute calls to complete or error callback functions.
sample	This operator will give the most recent value from the source Observable, and the output will depend upon the argument passed to it emits.
skip	This operator will give back an observable that will skip the first occurrence of count items taken as input.
throttle	This operator will output as well as ignore values from the source observable for the time determined by the input function taken as an argument and the same process will be repeated.

Utility Operators

The following are the operators we are going to discuss in the utility operator category.

Operator	Description
tap	This operator will have the output, the same as the source observable, and can be

	used to log the values to the user from the observable. The main value, error if any or if the task is complete.
delay	This operator delays the values emitted from the source Observable based on the timeout given.
delayWhen	This operator delays the values emitted from the source Observable based on the timeout from another observable taken as input.
observeOn	This operator based on the input scheduler will reemit the notifications from the source Observable.
subscribeOn	This operator helps to asynchronous subscribes to the source Observable based on the scheduler taken as input.
timeInterval	This operator will return an object which contains current value and the time elapsed between the current and previous value that is calculated using scheduler input taken.
timestamp	Returns the timestamp along with the value emitted from source Observable which tells about the time when the value was emitted.
timeout	This operator will throw an error if the source Observable does not emit a value after the given timeout.
toArray	Accumulates all the source value from the Observable and outputs them as an array when the source completes.

Conditional Operators

The following are the operators we are going to discuss in the conditional operator category.

Operator	Description
defaultIfEmpty	This operator will return a default value if the source observable is empty.
every	It will return an Observable based on the input function satisfies the condition on each of the value on source Observable.

find	This will return the observable when the first value of the source Observable satisfies the condition for the predicate function taken as input.
findIndex	This operator will give you the index of the first value from source Observable which happens to satisfy the condition inside the predicate function.
isEmpty	This operator will give the output as true if the input observable goes for complete callback without emitting any values and false if the input observable emits any values.

Multicasting Operators

The following are the operators we are going to discuss in the multicasting operator category.

Operator	Description
multicast	A multicast operator shares the single subscription created with other subscribers. The params that multicast takes in, is a subject or a factory method that returns a ConnectableObservable that has connect() method. To subscribe, connect() method has to be called.
publish	This operator gives back ConnectableObservable and needs to use connect() method to subscribe to the observables.
publishBehavior	publishBehaviour make use of BehaviourSubject, and returns ConnectableObservable. The connect() method has to be used to subscribe to the observable created.
publishLast	publishBehaviour make use of AsyncSubject, and returns back ConnectableObservable. The connect() method has to be used to subscribe to the observable created.
publishReplay	publishReplay make use of behaviour subject wherein it can buffer the values and replay the same to the new subscribers and returns ConnectableObservable. The connect() method has to be used to subscribe to the observable created.

share	It is an alias for multicast() operator with the only difference is that you don't have to call connect () method manually to start the subscription.
-------	-------------------------------------------------------------------------------------------------------------------------------------------------------

Error Handling Operators

The following are the operators we are going to discuss in error handling operator category.

Operators	Description
catchError	This operator takes care of catching errors on the source Observable by returning a new Observable or an error.
retry	This operator will take care of retrying back on the source Observable if there is error and the retry will be done based on the input count given.

ajax

This operator will make an ajax request for the given url. To work with ajax we need to import it first as follows:

```
import { ajax } from 'rxjs/ajax';
```

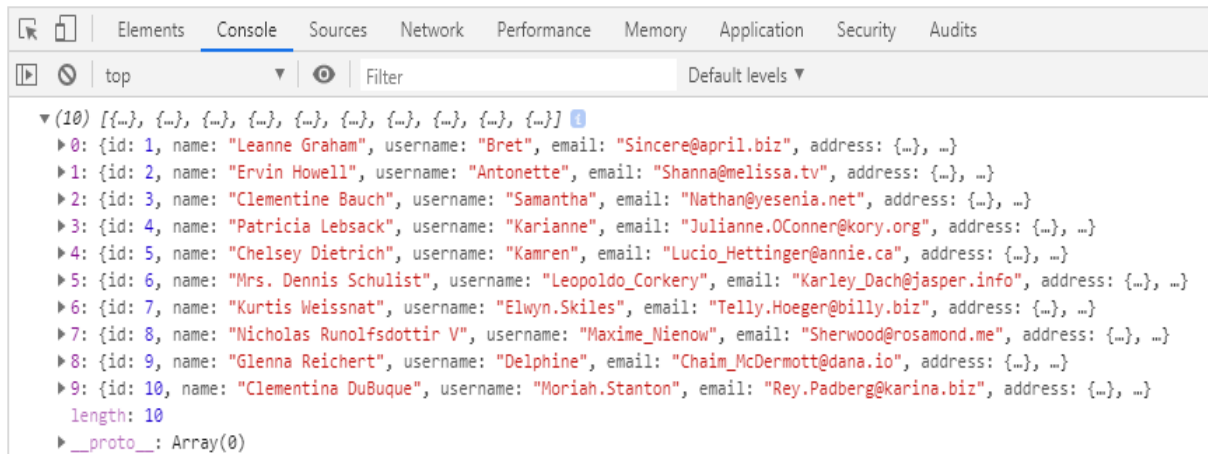
Let us see an example to see the working of ajax in RxJS:

Example

```
import { map, retry } from 'rxjs/operators';
import { ajax } from 'rxjs/ajax';

let final_val = ajax('https://jsonplaceholder.typicode.com/users').pipe(map(e
=> e.response));
final_val.subscribe(x => console.log(x));
```

Output



from

This operator will create an observable from an array, an array-like object, a promise, an iterable object, or an observable-like object.

Syntax

```
from(input: ObservableInput): Observable
```

Parameters

input: The input given to this operator is an Observable.

Return value: It returns an observable.

Example

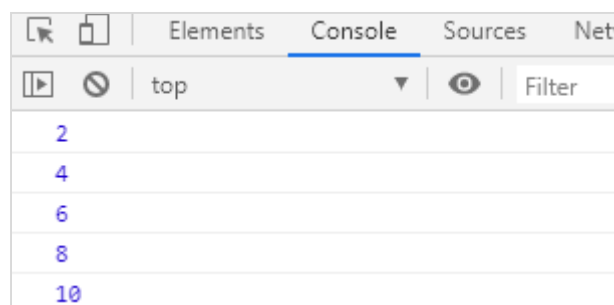
```

import { from } from 'rxjs';

let arr = [2, 4, 6, 8, 10];
let test = from(arr);
test.subscribe(x => console.log(x));

```

Output



fromEvent

This operator will give output as an observable that is to be used on elements that emit events for example buttons, clicks, etc.

Syntax

```
fromEvent(target: eventtarget, eventName: string): Observable
```

Parameters

target: The target is the dom element

eventName: eventName you want to capture for example click, mouseover, etc.

Return value

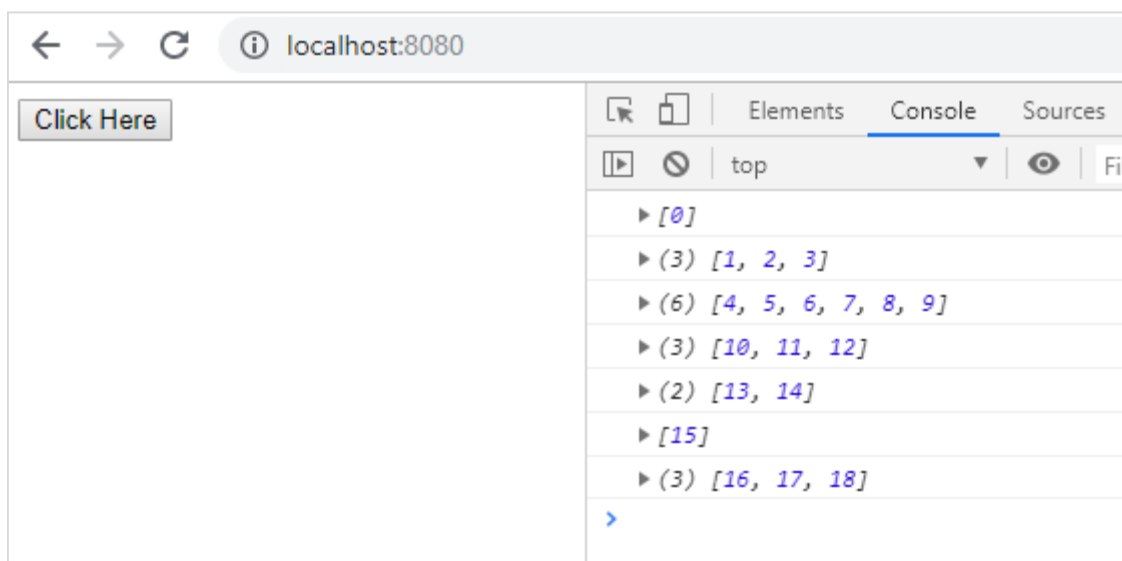
It returns an observable.

Example

```
import { fromEvent, interval } from 'rxjs';
import { buffer } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let interval_events = interval(1000);
let buffered_array = interval_events.pipe(buffer(btn_clicks));
buffered_array.subscribe(arr => console.log(arr));
```

Output



fromEventPattern

This operator will create an observable from the input function that is used to register event handlers.

Syntax

```
fromEventPattern(addHandler_func: Function): Observable
```

Parameters

addHandler_func: The argument given is addHandler_func, this will be attached to the actual event source.

Return value

Returns an observable when the event happens, for example, click, mouseover, etc.

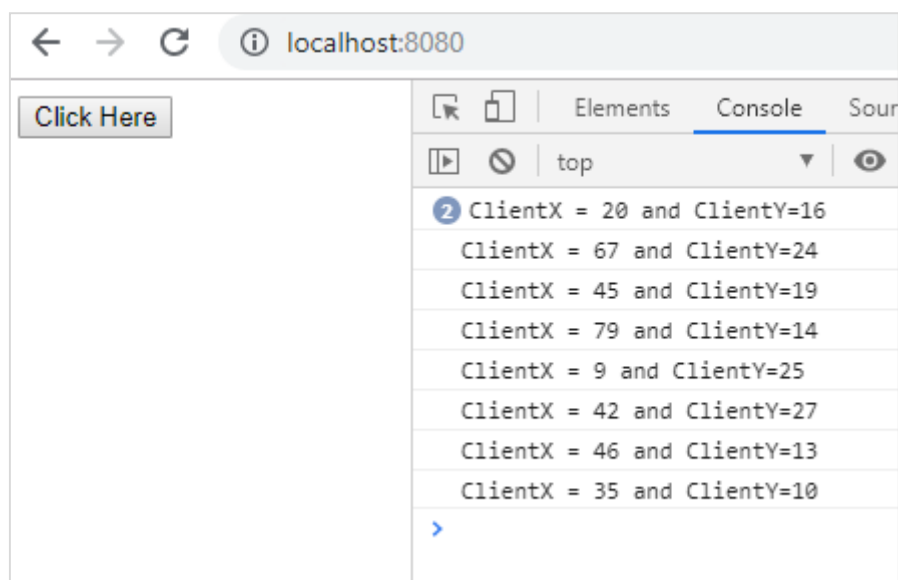
Example

```
import { fromEventPattern } from 'rxjs';

function addBtnClickHandler(handler) {
  document.getElementById("btnclick").addEventListener('click', handler);
}

const button_click = fromEventPattern(addBtnClickHandler);
button_click.subscribe(x => console.log("ClientX = " + x.clientX + " and ClientY=" + x.clientY));
```

Output



interval

This operator will create an Observable every time for the time given.

Syntax

```
interval(time): Observable
```

Parameters

time: The time given is in milliseconds.

Return value

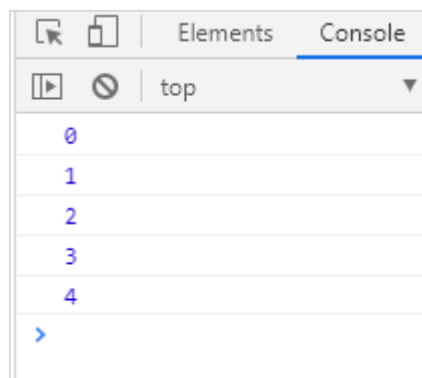
It returns an observable, that will give a sequential number for the timeinterval given.

Example

```
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';

let test = interval(2000);
let case1 = test.pipe(take(5));
case1.subscribe(x => console.log(x));
```

Output



of

This operator will take in the arguments passed and convert them to observable.

Syntax

```
of(input: array[]):Observable
```

Parameters

input: The input given is an array form.

Return value

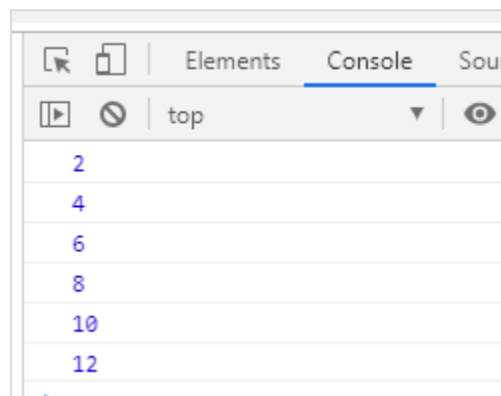
It returns an observable with values from the source observable.

Example

```
import { of } from 'rxjs';

let ints = of(2,4,6,8,10,12);
ints.subscribe(x => console.log(x));
```

Output



range

This operator will create an Observable that will give you a sequence of numbers based on the range provided.

Syntax

```
range(start: number, count: number): Observable
```

Parameters

start: The first value will be from start and will emit sequentially until the count given.

count: the count of numbers to be emitted.

Return value

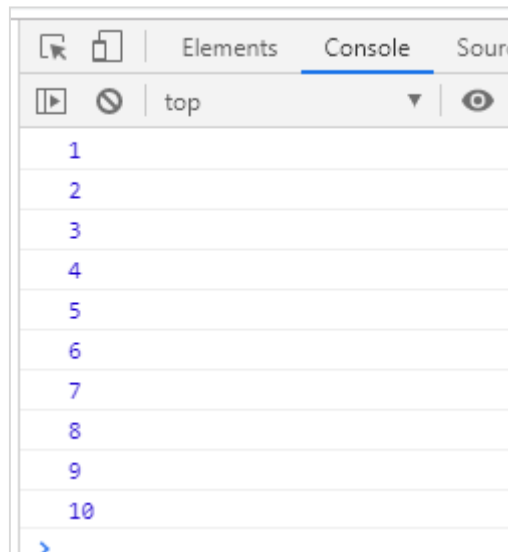
It returns an Observable that will give you a sequence of numbers based on the range provided.

Example

```
import { range } from 'rxjs';

let ints = range(1, 10);
ints.subscribe(x => console.log(x));
```

Output



throwError

This operator will create an observable that will notify an error.

Syntax

```
throwError(error: any): Observable
```

Parameters

error: The argument the operator takes in is the error that you need to notify.

Return value

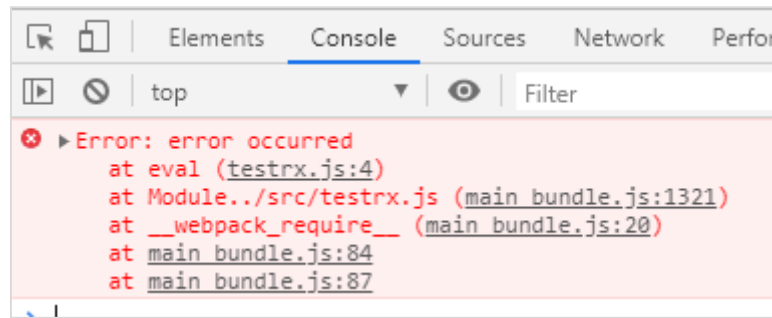
It returns an observable that will notify an error.

Example

```
import { throwError, concat, of } from 'rxjs';

const result = throwError(new Error('error occurred'));
result.subscribe(x => console.log(x), e => console.error(e));
```

Output



timer

This operator will create an observable that will emit the value after the timeout and the value will keep increasing after each call.

Syntax

```
timer(dueTime: number | Date): Observable
```

Parameters

dueTime: The dueTime can be in milliseconds or date.

Return value

An observable that will emit the value after the timeout and the value will keep increasing after each call.

Example

```
import { timer, range } from 'rxjs';

let all_numbers = timer(10, 10);
all_numbers.subscribe(x => console.log(x));
```

Output

Elements	Console	Sources	Network
top	▼	Filter	
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

iif

This operator will decide which Observable will be subscribed.

Syntax

```
iif(condition: Function):Observable
```

Parameters

condition: The condition is a function if its return true the observable will be subscribed.

Return value

An observable will be returned based on the condition.

Example

```
import { iif, of } from 'rxjs';
import { mergeMap, first, last } from 'rxjs/operators';
let task1 = iif(
  () => (Math.random() + 1) % 2 === 0,

  of("Even Case"),
  of("Odd Case")
)
```



```
);
task1.subscribe(value => console.log(value));
```

iff() operator acts like a ternary operator and mostly used for if-else condition cases.

Output

```
Odd Case
```

count

count() takes in an Observable with values and converts it into an Observable that will give a single value. The count function takes in predicate function as an optional argument. The function is of type boolean and will add value to the output only if the value is truthy.

Syntax

Here is the syntax for Count:

```
count(predicate_func? : boolean): Observable
```

Parameters

predicate_func : (optional) Function that will filter the values to be counted from the source observable and return a boolean value.

Return value

The return value is an observable that has the count of the given numbers.

Let us see some examples of count without predicate and with function.

Example 1

The following example is without predicate function:

```
import { of } from 'rxjs';
import { count } from 'rxjs/operators';

let all_nums = of(1, 7, 5, 10, 10, 20);
let final_val = all_nums.pipe(count());
final_val.subscribe(x => console.log("The count is "+x));
```

Output

```
The count is 6
```

Example 2

The following example is with predicate function:

```
import { of } from 'rxjs';
import { count } from 'rxjs/operators';

let all_nums = of(1, 6, 5, 10, 9, 20, 40);
let final_val = all_nums.pipe(count(a => a % 2 === 0));
final_val.subscribe(x => console.log("The count is "+x));
```

The function we have used in the count is to give the count only of even numbers.

Output

```
The count is 4
```

max

max() method will take in an observable with all values and return an observable with the max value. It takes in a compare function as an argument, which is optional.

Syntax

```
max(comparer_func?: number): Observable
```

Parameters

comparer_func: (optional). A function that will filter the values to be considered for max value from the source observable. If not provided the default function is considered.

Return value

The return value is an observable that will have the max value.

Example 1

The following example is with the max value:

```
import { of } from 'rxjs';
import { max } from 'rxjs/operators';

let all_nums = of(1, 6, 15, 10, 58, 20, 40);
let final_val = all_nums.pipe(max());
final_val.subscribe(x => console.log("The Max value is "+x));
```

The output is:

```
The Max value is 58
```

Example 2

The following example is the max value with comparer function:

```
import { from } from 'rxjs';
import { max } from 'rxjs/operators';

let list1 = [1, 6, 15, 10, 58, 2, 40];
let final_val = from(list1).pipe(max((a,b)=>a-b));
final_val.subscribe(x => console.log("The Max value is "+x));
```

We are using arrays and the values inside the array are compared using the function given in max function, the max value from the array is returned.

Output

```
The Max value is 58
```

min

min() method will take in an observable with all values and return an observable with the min value. It takes in a compare function as an argument, which is optional.

Syntax

```
min(comparer_func?: number): Observable
```

Parameters

comparer_func: (optional). A function that will filter the values to be considered for min value from the source observable. If not provided the default function is considered.

Return value

The return value is an observable that will have the min value.

Example 1

```
import { of } from 'rxjs';
import { min } from 'rxjs/operators';

let list1 = [1, 6, 15, 10, 58, 2, 40];

let final_val = of(1, 6, 15, 10, 58, 2, 40).pipe(min());
final_val.subscribe(x => console.log("The Min value is "+x));
```

Output

```
The Min value is 1
```

Example 2

```
import { of ,from} from 'rxjs';
import { min } from 'rxjs/operators';

let list1 = [1, 6, 15, 10, 58, 2, 40];
let final_val = from(list1).pipe(min((a,b) => a - b));
final_val.subscribe(x => console.log("The Min value is "+x));
```

Output

```
The Min value is 1
```

Reduce

In reduce operator, accumulator function is used on the input observable, and the accumulator function will return the accumulated value in the form of an observable, with an optional seed value passed to the accumulator function.

The reduce() function will take in 2 arguments, one accumulator function, and second is the seed value.

Syntax

```
reduce(accumulator_func, seeder?) : Observable
```

Parameters

accumulator_func: a function that is called on the source values from the observables.

seeder: (optional) By default it is undefined. The initial value to considered for accumulation.

Return Value

It will return an observable that will have a single accumulated value.

We will see some examples to see how the reduce operator works.

Example 1

```
import { from } from 'rxjs';
import { reduce } from 'rxjs/operators';

let items = [
  {item1: "A", price: 1000.00},
```

```

    {item2: "B", price: 850.00},
    {item2: "C", price: 200.00},
    {item2: "D", price: 150.00}
  ];
  let final_val = from(items).pipe(reduce((acc, itemsdet) => acc+itemsdet.price,
  0));
  final_val.subscribe(x => console.log("Total Price is: "+x));

```

Output

```
Total Price is: 2200
```

concat

This operator will sequentially emit the Observable given as input and proceed to the next one.

Syntax

```
concat(observables: Array): Observable
```

Parameters

observables: The input given is an array of Observables.

Return value

An observable is returned with a single value merged from the values of the source observable.

Example

```

import { of } from 'rxjs';

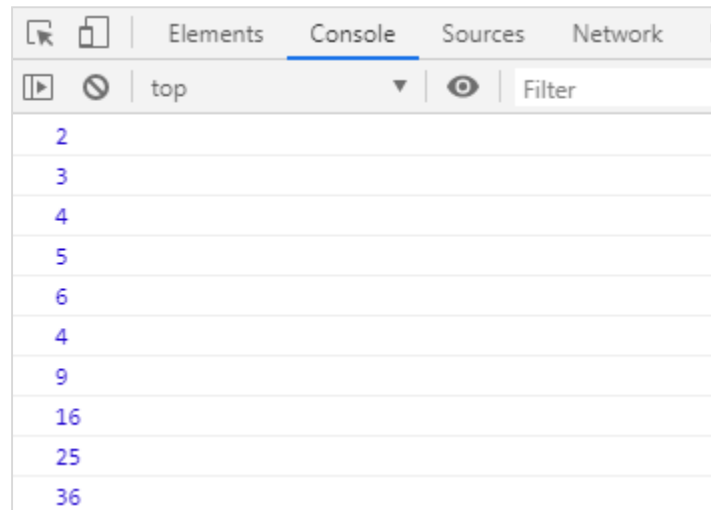
import { concat } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let list2 = of(4, 9, 16, 25, 36)
let final_val = list1.pipe(concat(list2));
final_val.subscribe(x => console.log(x));

```

We have concat two observables into one. Below is the output.

Output



forkJoin

This operator will take in an array or dict object as an input and will wait for the observable to complete and return the last values emitted from the given observable.

Syntax

```
forkJoin(value: array or dict object): Observable
```

Parameters

value: The value is the input which can be an array or dict object.

Return value

An observable is returned with last values emitted from the given observable.

Example

```
import { of, forkJoin } from 'rxjs';
import { concat } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let list2 = of(4, 9, 16, 25, 36);
let final_val = forkJoin([list1, list2]);
final_val.subscribe(x => console.log(x));
```

Output

```
[6, 36]
```

merge

This operator will take in the input observable and will emit all the values from the observable and emit one single output observable.

Syntax

```
merge(observable:array[]): Observable
```

Parameters

observable: The input will be an array of Observable.

Return value

It will return an observable with a single value as output.

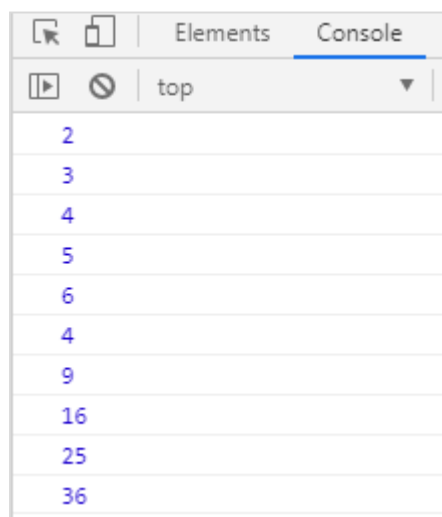
Example

```
import { of, merge } from 'rxjs';
import { concat } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let list2 = of(4, 9, 16, 25, 36)

let final_val = merge(list1, list2);
final_val.subscribe(x => console.log(x));
```

Output



race

It will give back an observable that will be a mirror copy of the first source observable.

Syntax

```
race(observables: array[]): Observable
```

Parameters

observables: The argument for this operator is an array of Observable.

Return value

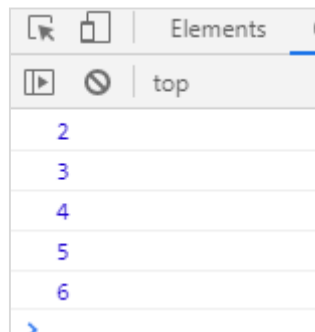
It will return an observable that will be a mirror copy of the first source observable.

Example

```
import { of, race } from 'rxjs';
import { concat } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let list2 = of(4, 9, 16, 25, 36);
let final_val = race(list1, list2);
final_val.subscribe(x => console.log(x));
```

Output



buffer

The buffer operates on an observable and takes in argument as an observable. It will start buffering the values emitted on its original observable in an array and will emit the same when the observable taken as argument emits. Once the observable taken as arguments emits the buffer is resets and starts buffering again on original till the input observable emits and the same scenario repeats.

Syntax

```
buffer(input_observable: Observable): Observable
```

Parameters

input_observable: an observable that will make the buffer emit values. For example, button click.

Return Value

An observable will be returned, that will have an array of buffered values. We will work on an example to understand the working of the buffer() operator.

In the example below, we are going to use a button click as an observable input to buffer. The interval of 1s will be as original observable on which buffer is called. The buffer will collect the clicks passed in the time interval given.

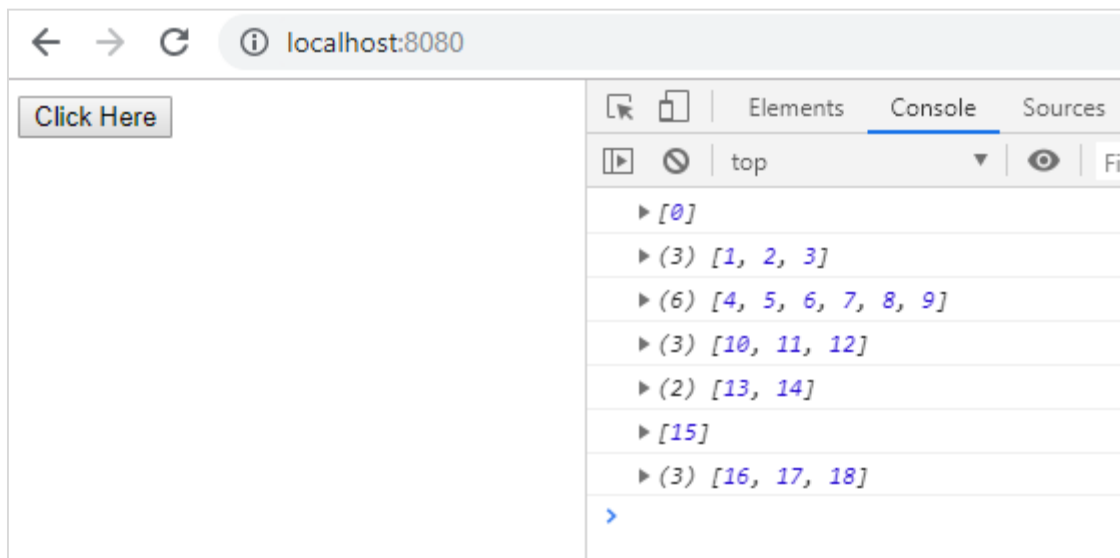
Example

```
import { fromEvent, interval } from 'rxjs';
import { buffer } from 'rxjs/operators';

let btn = document.getElementById("btnclick");

let btn_clicks = fromEvent(btn, 'click');
let interval_events = interval(1000);
let buffered_array = interval_events.pipe(buffer(btn_clicks));
buffered_array.subscribe(arr => console.log(arr));
```

Output



bufferCount

In the case of buffercount operator, it will collect the values from the observable on which it is called and emit the same when the buffer size given to buffercount matches. It takes 2 arguments **bufferSize** and the second one is **startBufferEvery** i.e. it will count the new values from startBufferEvery if given or else from the beginning of the source observable.

Syntax

```
bufferCount(bufferSize: number, startBufferEvery: number = null): Observable
```

Parameters

bufferSize: The size of the buffer to be emitted.

startBufferEvery: (optional) It is the time interval at which to start a new buffer.

Return value

An observable will be returned, that will have an array of buffered values.

We will see a working example of `bufferCount()`

Example 1

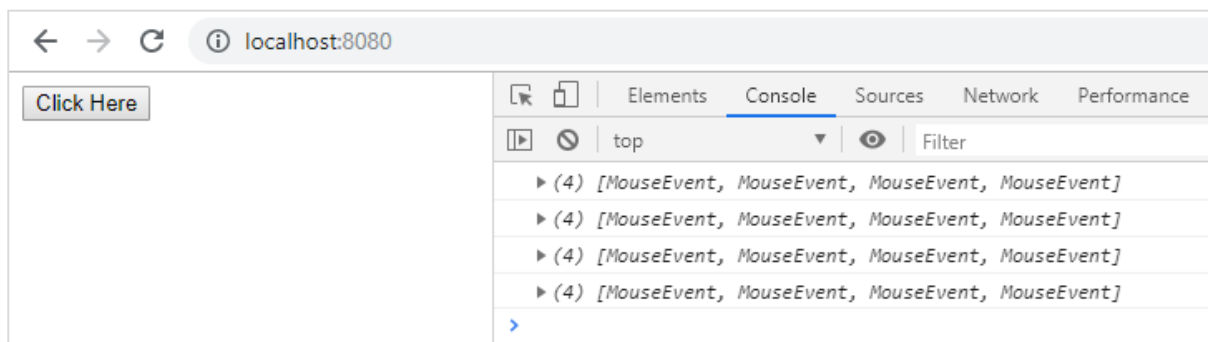
```
import { fromEvent } from 'rxjs';
import { bufferCount } from 'rxjs/operators';

let btn = document.getElementById("btnclick");

let btn_clicks = fromEvent(btn, 'click');
let buffered_array = btn_clicks.pipe(bufferCount(4));
buffered_array.subscribe(arr => console.log(arr));
```

In the above example, the `bufferSize` is 4. So, after a count of 4 clicks the array of click events is collected in an array and displayed. Since we have not given the `startBufferEvery` the values will be counted from the start.

Output



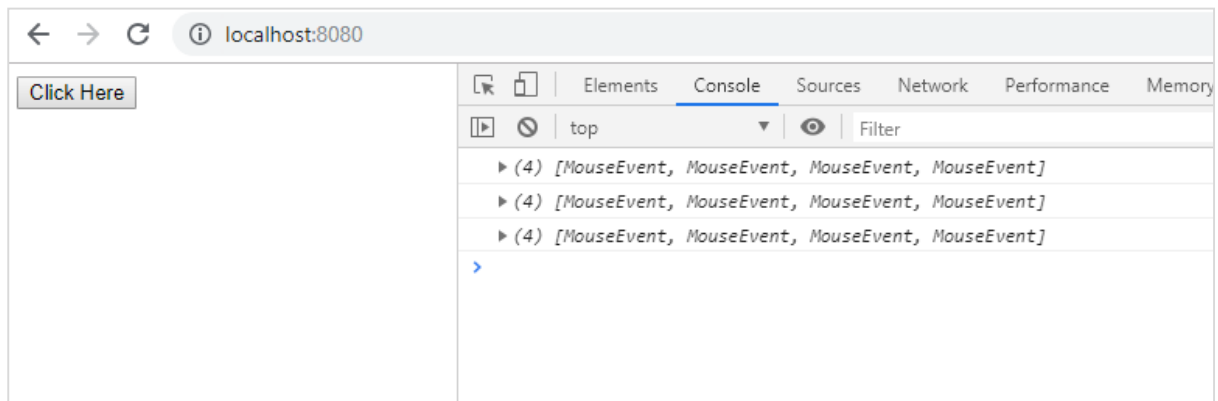
Example 2

```
import { fromEvent } from 'rxjs';
import { bufferCount } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let buffered_array = btn_clicks.pipe(bufferCount(4, 2));
buffered_array.subscribe(arr => console.log(arr));
```

In this example, we have added `startBufferEvery`, so after every 2 clicks, it will display a buffer count of 4 click events.

Output



bufferTime

This is similar to `bufferCount`, so here, it will collect the values from the observable on which it is called and emit the `bufferTimeSpan` is done. It takes in 1 argument, i.e., ***bufferTimeSpan***.

Syntax

```
bufferTime(bufferTimeSpan: number): Observable
```

Parameters

bufferTimeSpan: The time to fill the buffer array.

Return Value

An observable will be returned, that will have an array of buffered values.

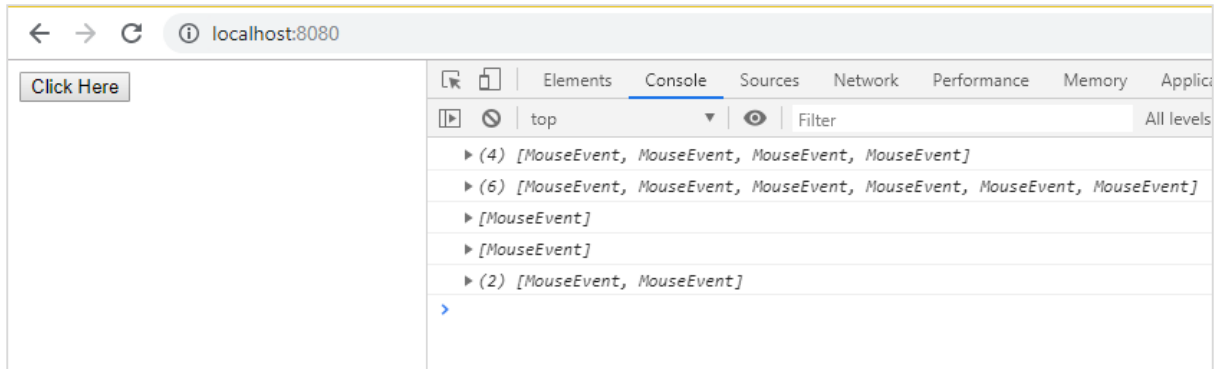
Example

```
import { fromEvent } from 'rxjs';
import { bufferTime } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let buffered_array = btn_clicks.pipe(bufferTime(4000));
buffered_array.subscribe(arr => console.log(arr));
```

In the example the time used is 4seconds, So, `bufferTime()` operator will accumulate the clicks and after every 4 seconds will display them as shown below.

Output



bufferToggle

In the case of `bufferToggle` it takes 2 arguments, `openings` and `closingSelector`. The opening arguments are a subscribable or a promise to start the buffer and the second argument `closingSelector` is again subscribable or promise an indicator to close the buffer and emit the values collected.

Syntax

```
bufferToggle(openings: SubscribableOrPromise, closingSelector:
SubscribableOrPromise): Observable
```

Parameters

openings: A promise or notification to start the new buffer.

closingSelector: A function that will take the values from *openings observable* and return Subscribable or promise.

Return Value

An observable will be returned, that will have an array of buffered values.

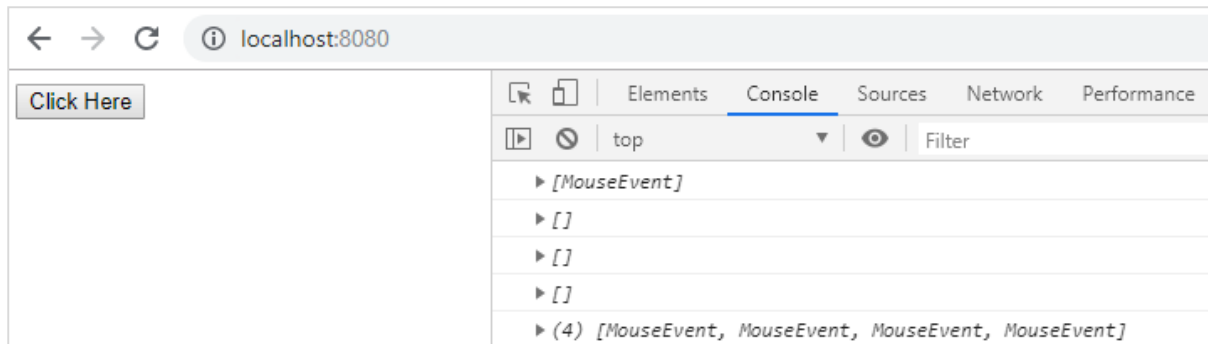
Example

```
import { fromEvent, interval, EMPTY } from 'rxjs';
import { bufferToggle } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let start = interval(2000);
let buffered_array = btn_clicks.pipe(bufferToggle(start, a => a%2 ==0 ?
interval(1000): EMPTY));
buffered_array.subscribe(arr => console.log(arr));
```

In the example above the buffer will start after 2s and end when we 1s interval if the value received is even otherwise it will empty the buffer values and emit empty values.

Output



bufferWhen

This operator will give the values in the array form, it takes in one argument as a function that will decide when to close, emit and reset the buffer.

Syntax

```
bufferWhen(closing_func: Observable): Observable
```

Parameters

closing_func: A function that returns an Observable indicating buffer closure.

Return Value

An observable will be returned, that will have an array of buffered values.

Example

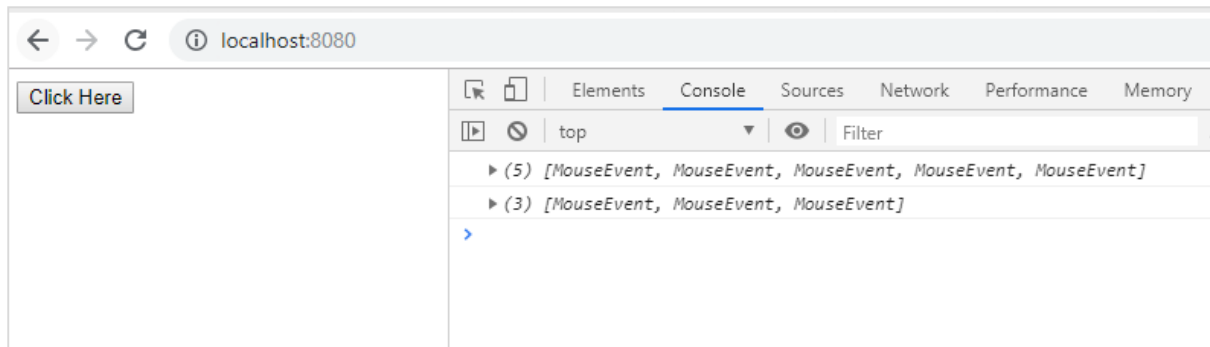
Here is a working example of bufferWhen:

```
import { fromEvent, interval } from 'rxjs';
import { bufferWhen } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let buffered_array = btn_clicks.pipe(bufferWhen(() => interval(5000)));
buffered_array.subscribe(arr => console.log(arr));
```

For **bufferWhen** we have given a function that executes at an interval of 5 seconds. So, after every 5 seconds, it will output all the clicks recorded and will get reset and start again.

Output



expand

The expand operator takes in a function as an argument which is applied on the source observable recursively and also on the output observable. The final value is an observable.

Syntax

```
expand(recursive_func:observable): Observable
```

Parameters

recursive_func: A function is applied to all the values coming from the source and returns an Observable.

Return Value

An observable, with values as per the result of the recursive_func.

Example

```
import { of } from 'rxjs';
import { expand } from 'rxjs/operators';

let buffered_array = of(2).pipe(expand(x => of(2 * x)));
buffered_array.subscribe(arr => console.log(arr));
```

Output

Elements

Console

Sources

top

Filter

2

4

8

16

32

64

128

256

512

1024

2048

4096

8192

16384

32768

65536

131072

262144

groupBy

In groupBy operator, the output is grouped based on a specific condition and these group items are emitted as GroupedObservable.

Syntax

```
groupBy(keySelector_func: (value: T) => K):GroupedObservables
```

Parameters

keySelector_func: A function that gives the key for each item from the source observable.

Return Value

The return value is an Observable that emits values as a GroupedObservables.

Example

```
import { of , from } from 'rxjs';
import { groupBy } from 'rxjs/operators';

const data = [
  {groupId: "QA", value: 1},
  {groupId: "Development", value: 3},
```

```

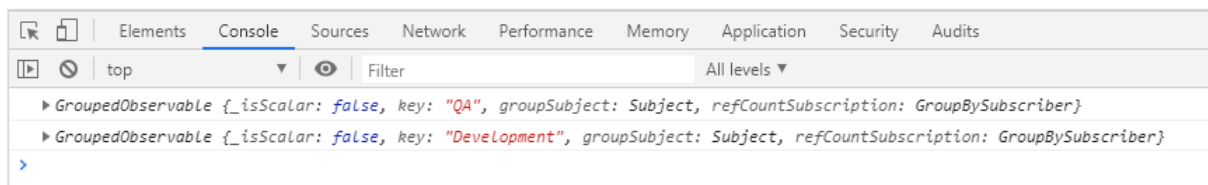
    {groupId: "QA", value: 5},
    {groupId: "Development", value: 6},
    {groupId: "QA", value: 2},
  ];

  from(data).pipe(
    groupBy(item => item.groupId)
  )
    .subscribe(x => console.log(x));

```

If you see the output, it is an observable wherein the items are grouped. The data we have given has 2 groups QA and Development. The output shows the grouping of the same as shown below:

Output



map

In the case of map operator, a project function is applied on each value on the source Observable and the same output is emitted as an Observable.

Syntax

```
map(project_func: function): Observable
```

Parameters

project_func: It takes in *project_func* as the argument which is applied to all the values of source observable.

Return Value

An observable, with values as per the result of the *project_func*.

Example

```

import { fromEvent } from 'rxjs';
import { map } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');

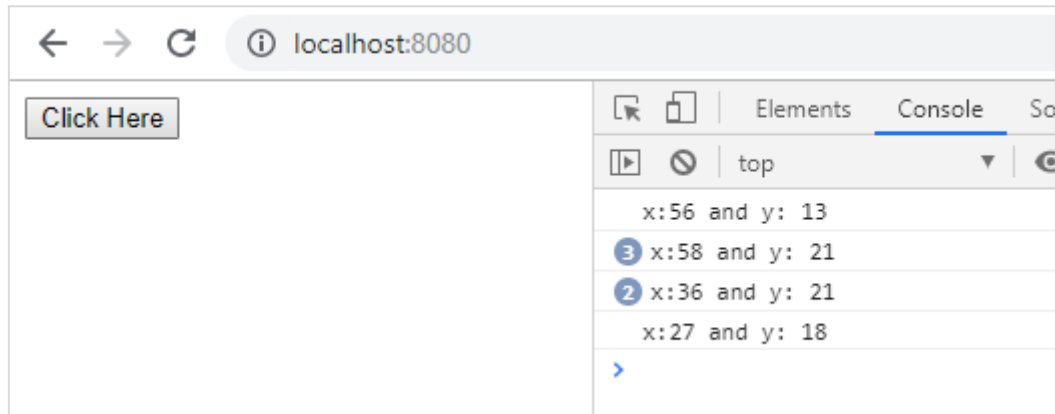
let positions = btn_clicks.pipe(map(ev => ev));

```



```
positions.subscribe(x => console.log("x:"+x.clientX + " and y: "+x.clientY));
```

Output



mapTo

A constant value is given as output along with the Observable every time the source Observable emits a value.

Syntax

```
mapTo(value: any): Observable
```

Parameters

value: It takes in the value as an argument and this value will be a map to the source value given.

Return Value

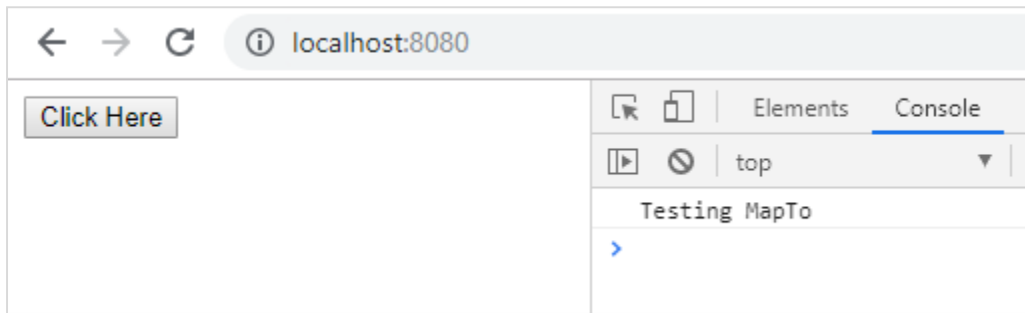
It returns an Observable with values emitted when source observable emits.

Example

```
import { fromEvent } from 'rxjs';
import { mapTo } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let positions = btn_clicks.pipe(mapTo ("Testing MapTo"));
positions.subscribe(x => console.log(x));
```

Output



mergeMap

In the case of mergeMap operator a project function is applied on each source value and the output of it is merged with the output Observable.

Syntax

```
mergeMap(project_func: function): Observable
```

Parameters

project_func: It takes in project_func as the argument which is applied to all the values of source observable.

Return value

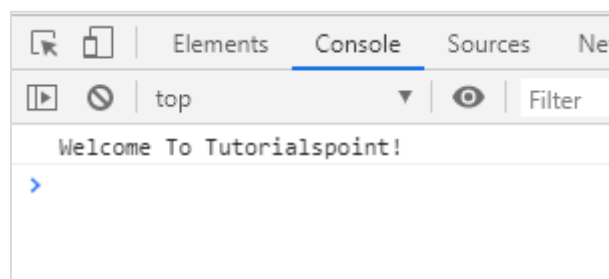
It returns an Observable that has values based on the project_func applied on each value of source observable.

Example

```
import { of } from 'rxjs';
import { mergeMap, map } from 'rxjs/operators';

let text = of('Welcome To');
let case1 = text.pipe(mergeMap((value) => of(value + ' Tutorialspoint!')));
case1.subscribe((value) => {console.log(value)});
```

Output



switchMap

In the case of switchMap operator, a project function is applied on each source value and the output of it is merged with the output Observable, and the value given is the most recent projected Observable.

Syntax

```
switchMap(project_func: function): Observable
```

Parameters

project_func: It takes in project_func as the argument which is applied to all the values emitted from source observable and returns an Observable.

Return Value

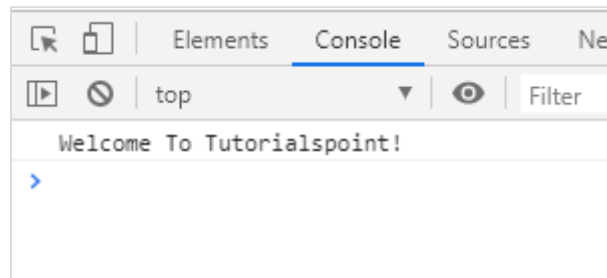
The return value is an Observable, that has values based on the project_func applied on each value of source observable.

Example

```
import { of } from 'rxjs';
import { switchMap } from 'rxjs/operators';

let text = of('Welcome To');
let case1 = text.pipe(switchMap((value) => of(value + ' Tutorialspoint!')));
case1.subscribe((value) => {console.log(value);});
```

Output



window

It takes an argument windowboundaries which is an observable and gives back a nested observable whenever the given windowboundaries emits

Syntax

```
window(windowBoundaries: Observable): Observable
```

Parameters

windowBoundaries: The argument windowboundaries is an observable.

Return value

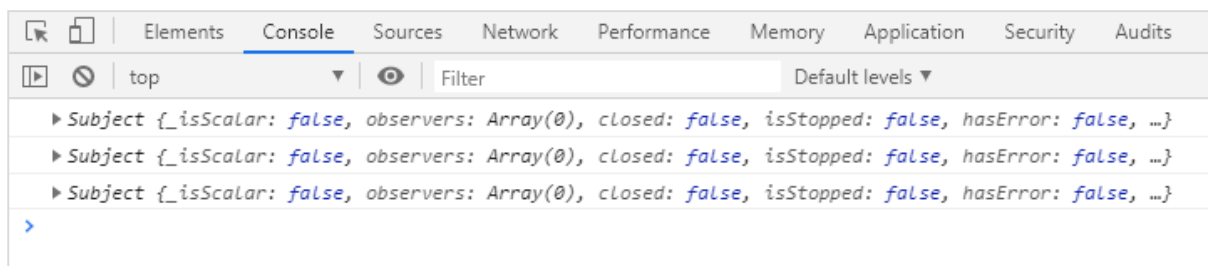
It returns an observable of windows.

Example

```
import { fromEvent, interval } from 'rxjs';
import { window } from 'rxjs/operators';

let btnclick = fromEvent(document, 'click');
let sec = interval(5000);
let result = btnclick.pipe(
  window(interval(5000))
);
result.subscribe(x => console.log(x));
```

Output



debounce

A value emitted from the source Observable after a while and the emission is determined by another input given as Observable or promise.

Syntax

```
debounce(durationSelector: Observable or promise): Observable
```

Parameters

durationSelector: It takes in an argument called *durationSelector* that returns an observable or a promise. This argument will get input from the source observable and decide the timeout for each source value.

Return value

It returns an observable wherein the emission of the source observable is delayed based on the *durationSelector*.

Example

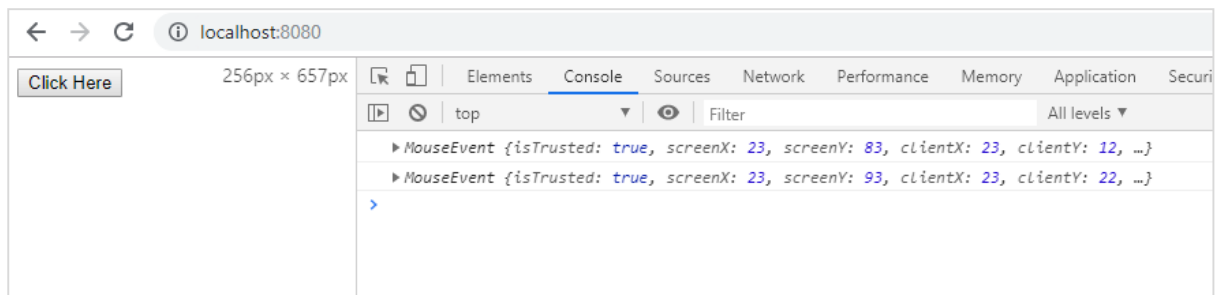
```
import { fromEvent, interval } from 'rxjs';
```

```
import { debounce } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(
  debounce(() => interval(2000))
);
case1.subscribe(x => console.log(x));
```

Here the click event is delayed using debounce() operator

Output



debounceTime

It will emit value from the source observable only after the time span is complete.

Syntax

```
debounceTime(dueTime: number): Observable
```

Parameters

debounceTime: The argument dueTime is the timeout in milliseconds.

Return value

It returns an observable wherein the emission of the source observable is delayed based on the dueTime.

Example

```
import { fromEvent } from 'rxjs';
import { debounceTime } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(
```

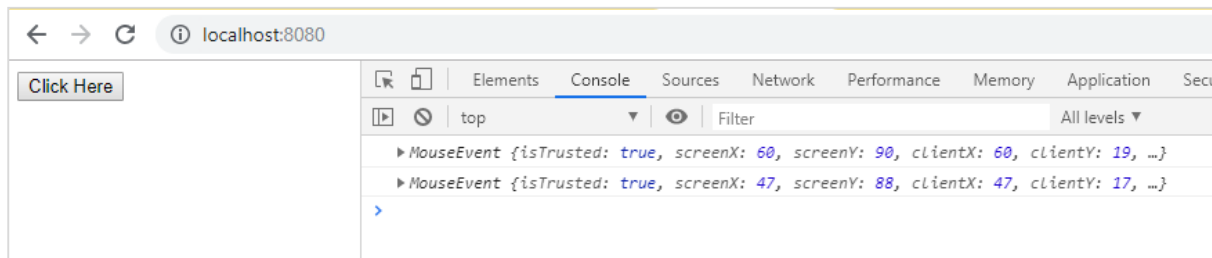
```

    debounceTime(2000)
  );
case1.subscribe(x => console.log(x));

```

Same as `debounce()` operator , with the only difference, is that you can pass the delay time to this operator directly.

Output



distinct

This operator will give all the values from the source observable that are distinct when compared with the previous value.

Syntax

```
distinct()
```

Return value

It returns an observable that has distinct values.

Example

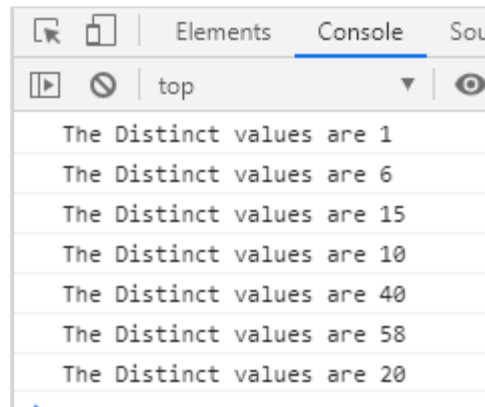
```

import { of } from 'rxjs';
import { distinct } from 'rxjs/operators';

let all_nums = of(1, 6, 15, 1, 10, 6, 40, 10, 58, 20, 40);
let final_val = all_nums.pipe(distinct());
final_val.subscribe(x => console.log("The Distinct values are "+x));

```

Output



elementAt

This operator will give a single value from the source observable based upon the index given.

Syntax

```
elementAt(index: number): Observable
```

Parameters

index: The argument passed is the index of type number, starting from 0. The value from the source observable for this index will be given back.

Return value

An observable will be returned with a value based on the index given.

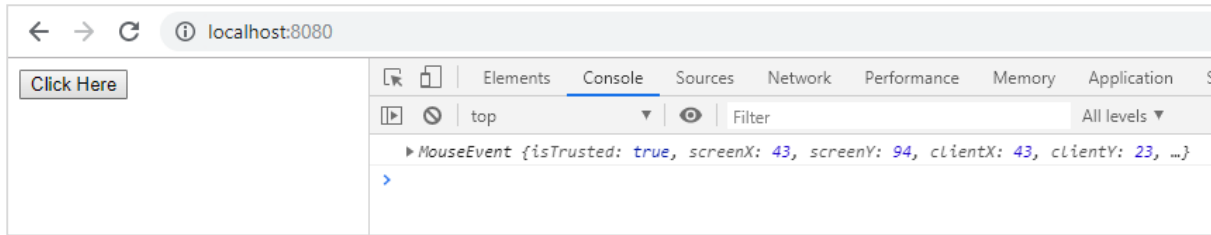
Example

```
import { fromEvent } from 'rxjs';
import { elementAt } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(
  elementAt(4)
);
case1.subscribe(x => console.log(x));
```

We have used `elementAt(4)`, so the 5th click will be emitted as the index starts from 0.

Output



filter

This operator will filter the values from source Observable based on the predicate function given.

Syntax

```
filter(predicate_func: function): Observable
```

Parameter

predicate_func: The predicate_func, will return a boolean value, and the output will get filtered if the function returns a truthy value.

Return value

It will return an observable with values that satisfies the predicate_func.

Example

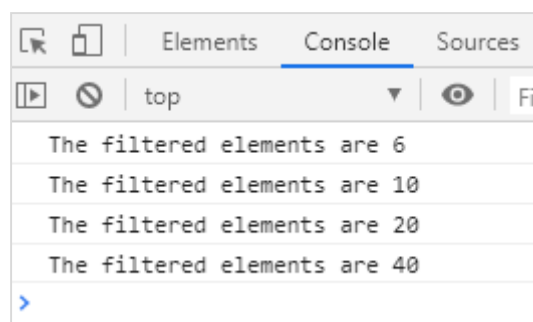
```
import { of } from 'rxjs';

import { filter } from 'rxjs/operators';

let all_nums = of(1, 6, 5, 10, 9, 20, 40);
let final_val = all_nums.pipe(filter(a => a % 2 === 0));
final_val.subscribe(x => console.log("The filtered elements are "+x));
```

We have filtered the even numbers using filter() operator.

Output



first

This operator will give the first value emitted by the source Observable

Syntax

```
first()
```

Return value

An observable will be returned with the first value.

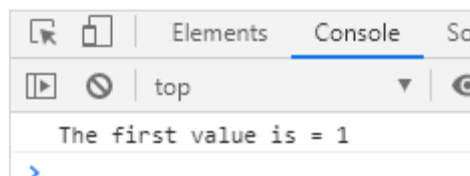
Example

```
import { of } from 'rxjs';
import { first } from 'rxjs/operators';

let all_nums = of(1, 6, 5, 10, 9, 20, 40);
let final_val = all_nums.pipe(first());
final_val.subscribe(x => console.log("The first value is = "+x));
```

The first() operator gives the first value from the list given.

Output



last

This operator will give the last value emitted by the source Observable.

Syntax

```
last()
```

Return value

It returns an observable with the last value.

Example

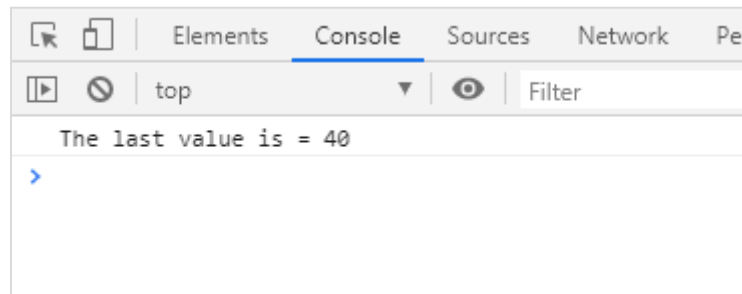
```
import { of } from 'rxjs';
import { last } from 'rxjs/operators';

let all_nums = of(1, 6, 5, 10, 9, 20, 40);
```

```
let final_val = all_nums.pipe(last());
final_val.subscribe(x => console.log("The last value is = "+x));
```

The last() gives the last value from the list provided.

Output



ignoreElements

This operator will ignore all the values from the source Observable and only execute calls to complete or error callback functions.

Syntax

```
ignoreElements()
```

Return value

It returns an observable that will call complete or error based on the source observable.

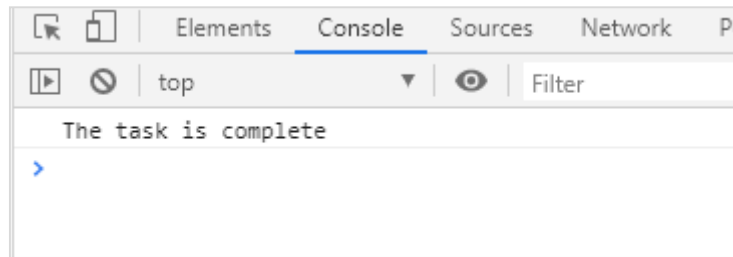
Example

```
import { of } from 'rxjs';
import { ignoreElements } from 'rxjs/operators';

let all_nums = of(1, 6, 5, 10, 9, 20, 40);
let final_val = all_nums.pipe(ignoreElements());
final_val.subscribe(
  x => console.log("The last value is = "+x),
  e => console.log('error:', e),
  () => console.log('The task is complete')
);
```

ignoreElements() operator will directly execute the complete method if success and error if failure and ignore everything else.

Output



sample

This operator will give the most recent value from the source Observable, and the output will depend upon the argument passed to it emits.

Syntax

```
sample(notifier: Observable): Observable
```

Parameters

notifier: The argument notifier is an Observable which will decide the output to be picked.

Return value

It returns an observable, based on values emitted by the source observable.

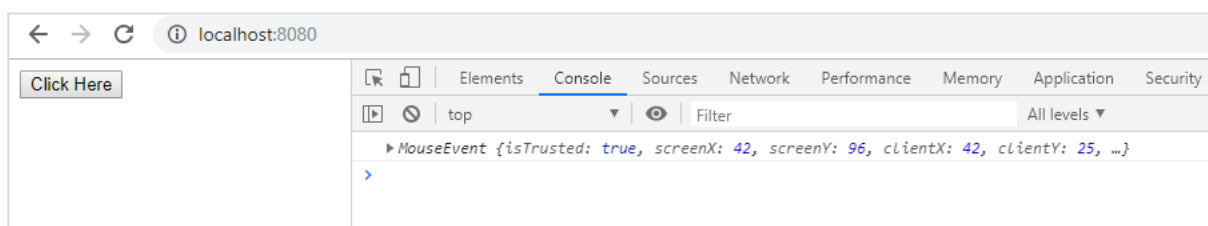
Example

```
import { fromEvent, interval } from 'rxjs';
import { sample } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(
  sample(interval(4000))
);
case1.subscribe(x => console.log(x));
```

The sample() operator is given interval(4000) so the click event will get emitted when the interval of 4seconds is done.

Output



skip

This operator will give back an observable that will skip the first occurrence of count items taken as input.

Syntax

```
skip(count: number): Observable
```

Parameters

count: The argument count is the number of times that the items will be skipped from the source observable.

Return value

It will return an observable that skips values based on the count given.

Example

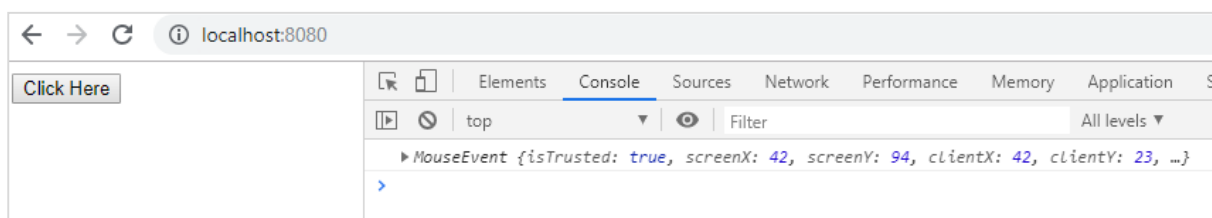
```
import { fromEvent, interval } from 'rxjs';

import { skip } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(
  skip(2)
);
case1.subscribe(x => console.log(x));
```

We have given count as 2 to skip() operator so the first two clicks are ignored and the third click event is emitted.

Output



throttle

This operator will output as well as ignore values from the source observable for the time period determined by the input function taken as an argument and the same process will be repeated.

Syntax

```
throttle(durationSelector: Observable or Promise): Observable
```

Parameters

durationSelector: The argument *durationSelector* is an Observable or Promise that will ignore values from the values emitted from the source Observable.

Return value

It will return an observable that will throttle the values emitted from the source observable.

Example

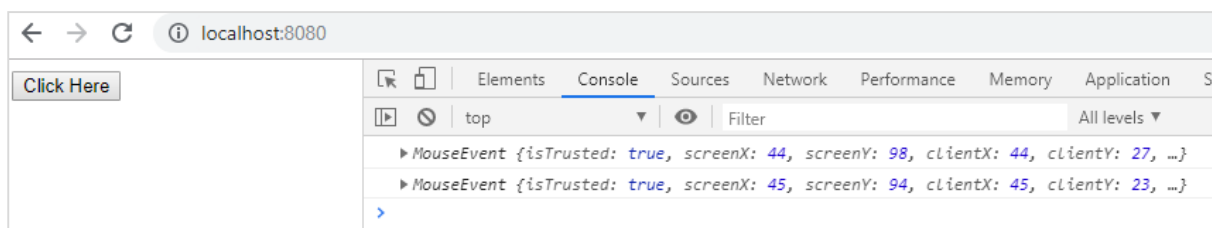
```
import { fromEvent, interval } from 'rxjs';

import { throttle } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(
  throttle(ev => interval(2000))
);
case1.subscribe(x => console.log(x));
```

When you click on the button the first click event will be emitted, the subsequent clicks will be delayed for the time given to throttle() operator.

Output



tap

This operator will have the output the same as the source observable and can be used to log the values to the user from the observable. The main value, error if any or is the task is complete.

Syntax

```
tap(observer, error, complete):Observable
```

Parameters

observer: (optional) this is the same as the source observable.

error: (optional) error method if any error occurs.

complete: (optional) complete() method will get called when the task is complete.

Return value

It returns an observable same like source observable with a callback function.

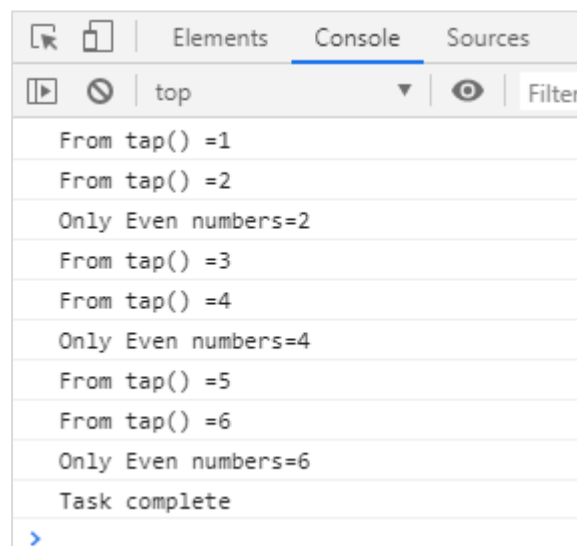
Example

```
import { of } from 'rxjs';

import { tap, filter } from 'rxjs/operators';

let list1 = of(1, 2, 3, 4, 5, 6);
let final_val = list1.pipe(
  tap(x => console.log("From tap() =" + x),
    e => console.log(e),
    () => console.log("Task complete")),
  filter(a => a % 2 === 0)
);
final_val.subscribe(x => console.log("Only Even numbers=" + x));
```

Output



delay

This operator delays the values emitted from the source Observable based on the timeout given.

Syntax

```
delay(timeout: number): Observable
```

Parameters

timeout: It will be in milliseconds or a Date which will delay the emission of the values from the source observable.

Return value

An observable will be returned that will use the timeout or date given to delay the source observable.

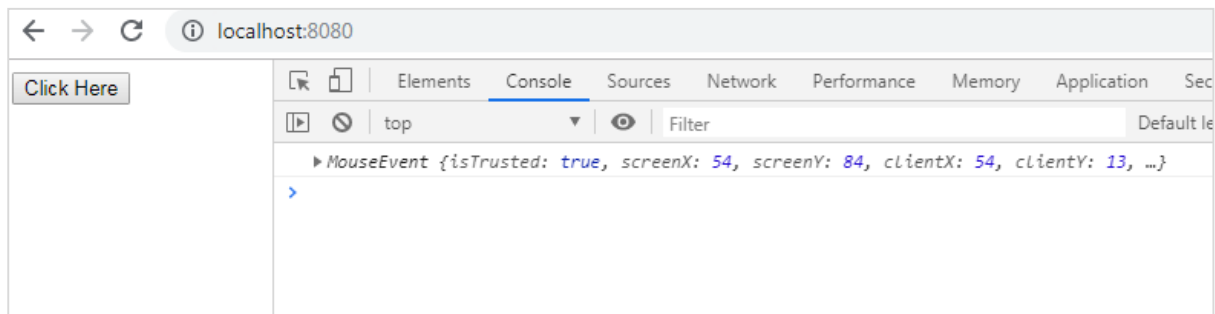
Example

```
import { fromEvent } from 'rxjs';
import { delay } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(
  delay(2000)
);
case1.subscribe(x => console.log(x));
```

The button click event will be delayed by 2seconds using delay() operator as shown below.

Output



delayWhen

This operator delays the values emitted from the source Observable based on the timeout from another observable taken as input.

Syntax

```
delayWhen(timeoutSelector_func: Observable<any>): Observable
```

Parameters

timeoutSelector_func: is an observable that decides about the timeout.

Return value

An observable will be returned that will use *timeoutSelector_func* output to delay the source observable.

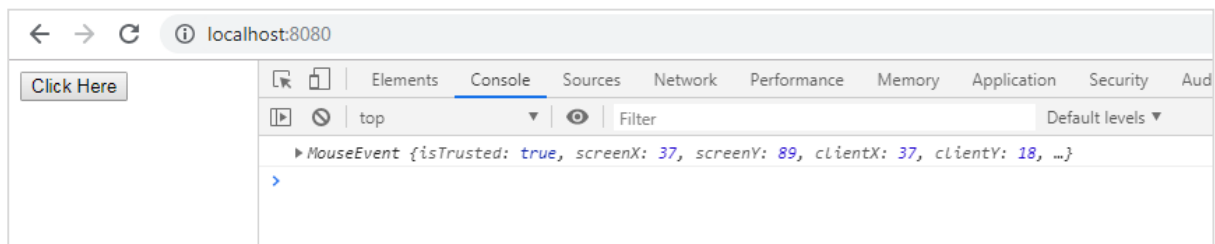
Example

```
import { fromEvent, timer } from 'rxjs';
import { delayWhen } from 'rxjs/operators';

let btn = document.getElementById("btnclick");
let btn_clicks = fromEvent(btn, 'click');
let case1 = btn_clicks.pipe(
  delayWhen(() => timer(1000)),
);
case1.subscribe(x => console.log(x));
```

We have used an observable for `delayWhen()`, and when that observable emits the click event is emitted.

Output



observeOn

This operator based on the input scheduler will reemit the notifications from the source Observable.

Syntax

```
observeOn(scheduler): Observable
```

Parameters

scheduler: The scheduler is used as an input that will help to re-emit the notifications from the source observable.

Return value

It will return an observable same as source observable, but with scheduler param.

Example

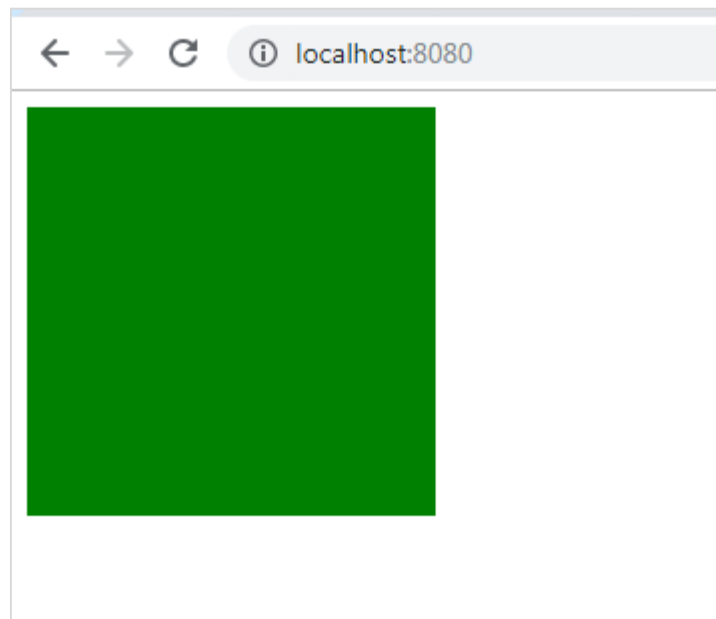
```
import { interval } from 'rxjs';
import { observeOn } from 'rxjs/operators';
```



```
import { animationFrameScheduler } from 'rxjs';

let testDiv = document.getElementById("test");
const intervals = interval(100);
let case1 = intervals.pipe(
  observeOn(animationFrameScheduler),
);
let sub1 = case1.subscribe(val => {
  console.log(val);
  testDiv.style.height = val + 'px';
  testDiv.style.width = val + 'px';
});
```

Output



subscribeOn

This operator helps to asynchronously subscribe to the source Observable based on the scheduler taken as input.

Syntax

```
subscribeOn(scheduler): Observable
```

Parameters

scheduler: The scheduler is used as an input that will help to re-emit the notifications from the source observable.

Return value

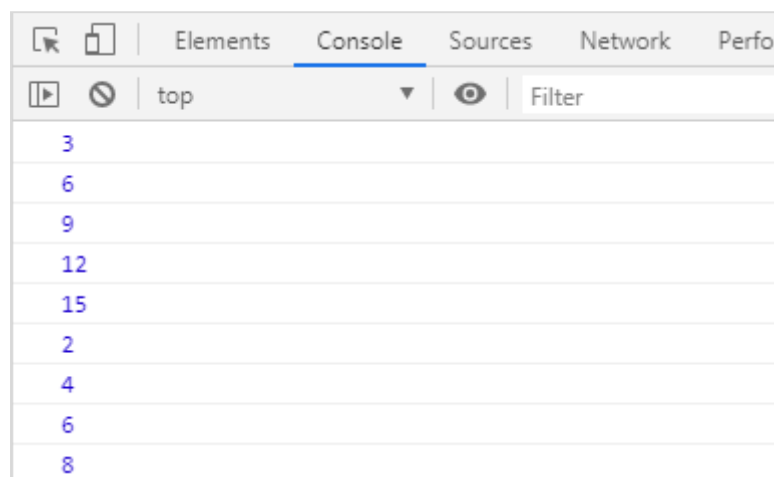
It will return an observable same as source observable, but with scheduler param.

Example

```
import { of, merge, asyncScheduler } from 'rxjs';
import { subscribeOn } from 'rxjs/operators';

let test1 = of(2, 4, 6, 8).pipe(subscribeOn(asyncScheduler));
let test2 = of(3, 6, 9, 12, 15);
let sub1 = merge(test1, test2).subscribe(console.log);
```

Output



timeInterval

This operator will return an object which contains current value and the time elapsed between the current and previous value that is calculated using scheduler input taken.

Syntax

```
timeInterval(scheduler): Observable
```

Parameters

scheduler: (optional) The scheduler input is used to calculate the time elapsed between the current and previous value from source Observable.

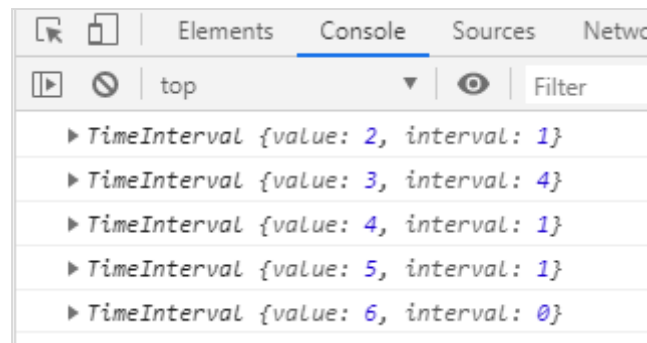
Return value

It will return an observable that will have source values and also the time interval.

Example

```
import { of } from 'rxjs';
import { filter, timeInterval } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let final_val = list1.pipe(
  timeInterval()
);
final_val.subscribe(x => console.log(x));
```

Output**timestamp**

Returns the timestamp along with the value emitted from source Observable which tells about the time when the value was emitted.

Syntax

```
timestamp(): Observable
```

Return value

Returns the timestamp along with the value emitted from source Observable which tells about the time when the value was emitted.

Example

```
import { of } from 'rxjs';
import { filter, timestamp } from 'rxjs/operators';

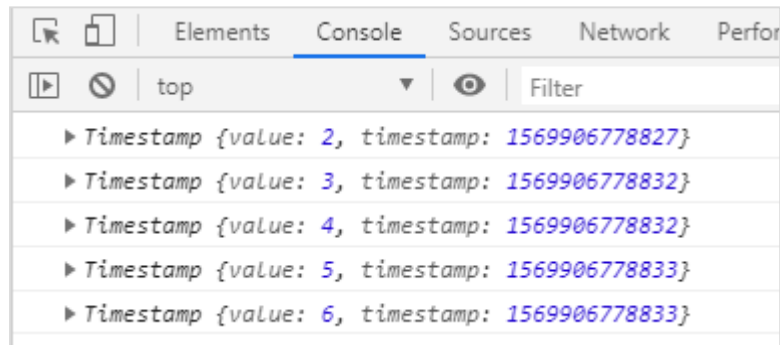
let list1 = of(2, 3, 4, 5, 6);
let final_val = list1.pipe(
```

```

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

```

Output



timeout

This operator will throw an error if the source Observable does not emit a value after the given timeout.

Syntax

```
timeout(timeout: number | Date): Observable
```

Parameters

timeout: The input to it is the timeout which can be of type number or Date within which the value from the source Observable must be emitted.

Return value

An observable is returned which will stop based on the timeout given.

Example

```

import { of, interval } from 'rxjs';
import { filter, timeout } from 'rxjs/operators';

let list1 = interval(1000);

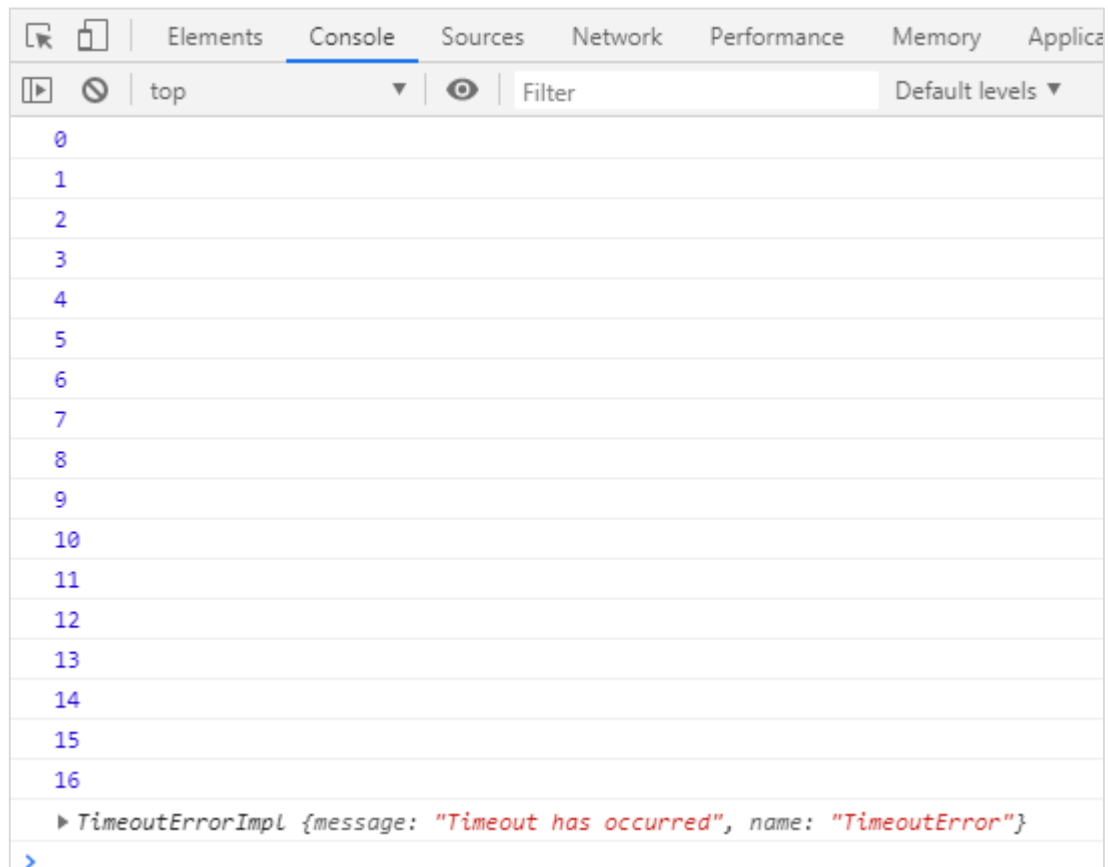
let final_val = list1.pipe(
  timeout(new Date("October 01, 2019 10:40:00"))
);
final_val.subscribe(
  x => console.log(x),

```

```
e => console.log(e),
() => console.log("Task complete")
);
```

The observable interval will go on and the timeout is given as new Date ("October 01, 2019 10:40:00"), so at that time the timeout will occur and it will throw an error as shown below.

Output



toArray

Accumulates all the source value from the Observable and outputs them as an array when the source completes.

Syntax

```
toArray():Observable
```

Return value

Returns an observable that outputs the values from the source observable as an array when the source completes.

Example

```
import { of } from 'rxjs';
```

```
import { toArray } from 'rxjs/operators';

let list1 = of(2, 3, 4, 5, 6);
let final_val = list1.pipe(toArray());
final_val.subscribe(x => console.log(x));
```

Output

```
[2, 3, 4, 5, 6]
```

defaultIfEmpty

This operator will return a default value if the source observable is empty.

Syntax

```
defaultIfEmpty(defaultValue = null): Observable
```

Parameters

defaultValue: The argument defaultValue can be given some value or if not given it is null by default.

Return value

It will return an observable with a default value if the source observable is empty.

Example

```
import { of } from 'rxjs';
import { defaultIfEmpty } from 'rxjs/operators';

let list1 = of();
let final_val = list1.pipe(
  defaultIfEmpty('Empty! No values')
);
final_val.subscribe(x => console.log(x));
```

Output

```
Empty! No values
```

every

It will return an Observable based on the input function satisfies the condition on each of the value on source Observable.

Syntax

```
every(predicate_func: function): Observable
```

Parameters

predicate_func: The input given to this operator is a *predicate_func* that will take in the source item and checks if it satisfies the condition given.

Return value

It returns an Observable based on the input function satisfies the condition on each of the value on source Observable.

Example

```
import { of } from 'rxjs';
import { every } from 'rxjs/operators';

let list1 = of(1, 3, 4, 9, 10, 15);
let final_val = list1.pipe(
  every(x => x % 2 === 0),
);
final_val.subscribe(x => console.log(x));
```

Output

```
false
```

find

This will return the observable when the first value of the source Observable satisfies the condition for the predicate function taken as input.

Syntax

```
find(predicate_func: function): Observable
```

Parameters

predicate_func: The input given to this operator is a *predicate_func* that will take in the source item and checks if it satisfies the condition given.

Return value

It will return the observable when the first value of the source Observable satisfies the condition for the predicate function taken as input.

Example

```
import { of } from 'rxjs';
```

```
import { find } from 'rxjs/operators';

let list1 = of(24, 3, 4, 9, 10, 15);
let final_val = list1.pipe(
  find(x => x % 2 === 0),
);
final_val.subscribe(x => console.log(x));
```

Output

```
24
```

findIndex

This operator will give you the index of the first value from source Observable which happens to satisfy the condition inside the predicate function.

Syntax

```
findIndex(predicate_func: function): Observable
```

Parameters

predicate_func: The predicate_function will be deciding the first index to be picked when the condition satisfies.

Return value

It will return an observable with the first value from source Observable which happens to satisfy the condition inside the predicate function

Example

```
import { of } from 'rxjs';
import { findIndex } from 'rxjs/operators';

let list1 = of(24, 3, 4, 9, 10, 15);
let final_val = list1.pipe(
  findIndex(x => x % 2 === 0),
);
final_val.subscribe(x => console.log(x));
```

Output

```
0
```


isEmpty

This operator will give the output as true if the input observable goes for complete callback without emitting any values and false if the input observable emits any values.

Syntax

```
isEmpty(): Observable
```

Return value

It will return an observable with a boolean value as true if the source observable is empty otherwise false.

Example

```
import { of } from 'rxjs';
import { isEmpty } from 'rxjs/operators';

let list1 = of();
let final_val = list1.pipe(
  isEmpty(),
);
final_val.subscribe(x => console.log(x));
```

Since the source observable is empty, the output given by observable is true.

Output

```
true
```

multicast

A multicast operator shares the single subscription created with other subscribers. The params that multicast takes in is a subject or a factory method that returns a ConnectableObservable that has to connect() method. To subscribe connect() method has to be called.

Syntax

```
multicast(subjectOrSubjectFactory: Subject): OperatorFunction
```

Params

subjectOrSubjectFactory: the parameter passed to multicast is a subject or factory method that returns a subject.

Before we get into the working of a multicast() operator, let us first understand how the multicast() operator is helpful.

Consider following example of a simple observable with subscription:

Example

```
import { Observable } from 'rxjs';

var observable = new Observable(function subscribe(subscriber) {
  try {
    subscriber.next(Math.random());

  } catch (e) {
    subscriber.error(e);
  }
});

const subscribe_one = observable.subscribe(val => console.log("Value from Sub1 = "+val));
const subscribe_two = observable.subscribe(val => console.log("Value from Sub2 = "+val));
```

Output

Elements	Console	Sources	Network
▶	top	👁	Filter
Value from Sub1 = 0.5830472945199348			
Value from Sub2 = 0.8939373128799453			

If you see the output the values for Sub1 and Sub2 are different. This is because when the subscriber gets called the observable restarts and gives the fresh value available. But we need the subscribers getting called to have the same value.

Here, we have multicast() operator to help us with it.

Example

```
import { Observable, Subject } from 'rxjs';
import { take, multicast, mapTo } from 'rxjs/operators';

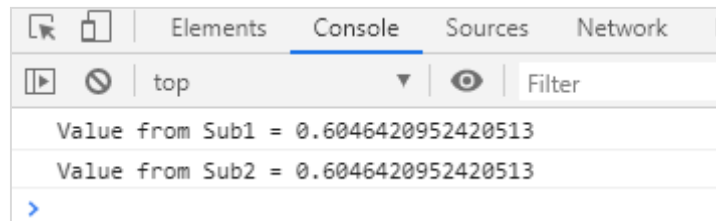
var observable = new Observable(function subscribe(subscriber) {
  try {
    subscriber.next(Math.random());

  } catch (e) {
    subscriber.error(e);
  }
});
```

```
const multi_op = observable.pipe(multicast(() => new Subject()));
const subscribe_one = multi_op.subscribe(x => console.log("Value from Sub1 = "+x));
const subscribe_two = multi_op.subscribe(x => console.log("Value from Sub2 = "+x));
multi_op.connect();
```

If you now see the same value is shared between the subscribers that are called.

Output



publish

publish() operator gives back ConnectableObservable and needs to use connect() method to subscribe to the observables.

Syntax

```
publish()
```

Example

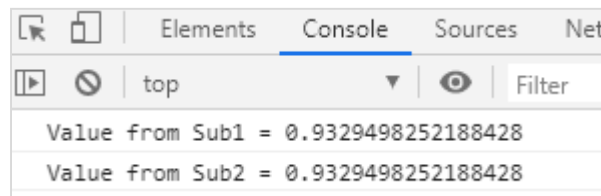
```
import { interval, Observable } from 'rxjs';
import { filter, publish } from 'rxjs/operators';

var observable = new Observable(function subscribe(subscriber) {
  try {
    subscriber.next(Math.random());
  } catch (e) {
    subscriber.error(e);
  }
});

const observable1 = publish()(observable);
const subscribe_one = observable1.subscribe(x => console.log("Value from Sub1 = "+x));
const subscribe_two = observable1.subscribe(x => console.log("Value from Sub2 = "+x));
```

```
observable1.connect();
```

Output



publishBehavior

publishBehaviour make use of BehaviourSubject, and returns ConnectableObservable. The connect() method has to be used to subscribe to the observable created.

Syntax

```
publishBehaviour(defaultvalue)
```

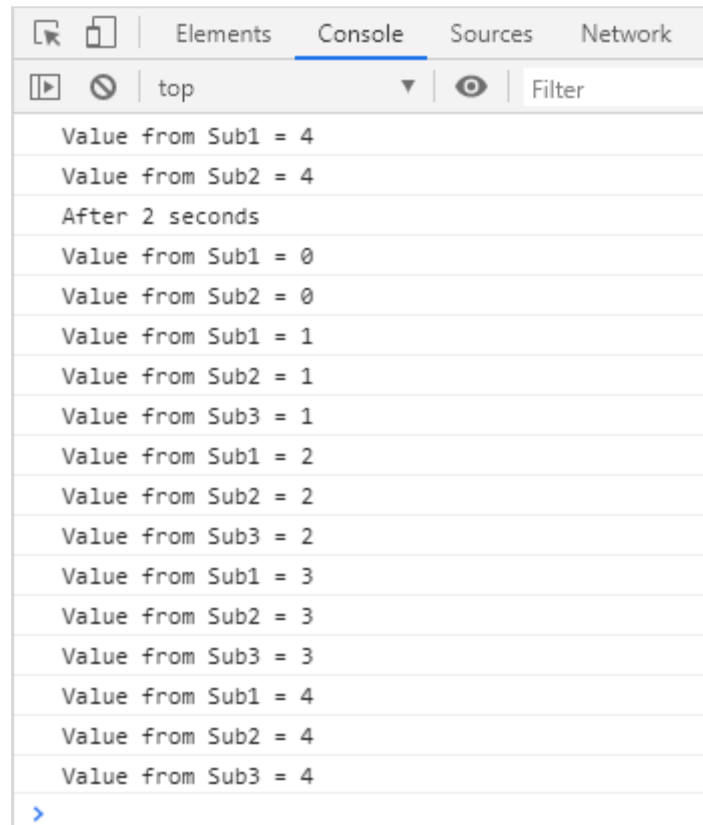
Example

```
import { interval } from 'rxjs';
import { take, publishBehavior } from 'rxjs/operators';

let observer = interval(1000).pipe(
  take(5),
  publishBehavior(4)
);

const subscribe_one = observer.subscribe(x => console.log("Value from Sub1 = "+x));
const subscribe_two = observer.subscribe(x => console.log("Value from Sub2 = "+x));
observer.connect();
console.log("After 2 seconds");
setTimeout(() => {
  const subscribe_three = observer.subscribe(x => console.log("Value from Sub3 = "+x));
}, 2000);
```

Output



The default value will be displayed first and later the value from the observable.

publishLast

`publishBehaviour` make use of `AsyncSubject`, and returns `ConnectableObservable`. The `connect()` method has to be used to subscribe to the observable created.

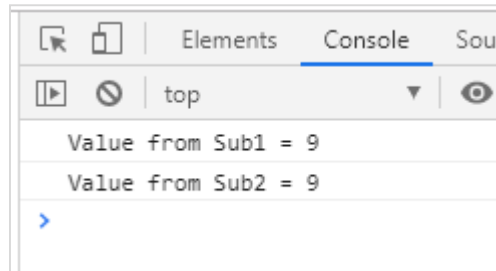
Example

```
import { interval } from 'rxjs';
import { take, publishLast } from 'rxjs/operators';

let observer = interval(1000).pipe(
  take(10),
  publishLast()
);

const subscribe_one = observer.subscribe(x => console.log("Value from Sub1 = " + x));
const subscribe_two = observer.subscribe(x => console.log("Value from Sub2 = " + x));
observer.connect();
```

Output



publishReplay

`publishReplay` make use of behaviour subject, wherein, it can buffer the values and replay the same to the new subscribers and returns `ConnectableObservable`. The `connect()` method has to be used to subscribe to the observable created.

Syntax

```
publishReplay(value); // here value is the number of times it has to replay.
```

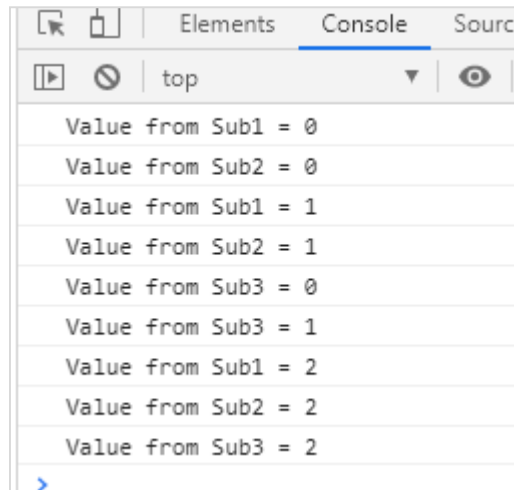
Example

```
import { interval } from 'rxjs';
import { take, publishReplay } from 'rxjs/operators';

let observer = interval(1000).pipe(
  take(3),
  publishReplay(2)
);

const subscribe_one = observer.subscribe(x => console.log("Value from Sub1 = "+x));
const subscribe_two = observer.subscribe(x => console.log("Value from Sub2 = "+x));
observer.connect();
setTimeout(() => {
  const subscribe_three = observer.subscribe(x => console.log("Value from Sub3 = "+x));
}, 2000);
```

Output



share

It is an alias for multicast() operator with the only difference is that you don't have to call connect () method manually to start the subscription.

Syntax

```
share()
```

Example

```
import { interval } from 'rxjs';
import { take, share } from 'rxjs/operators';

let observer = interval(1000).pipe(
  take(3),
  share()
);

const subscribe_one = observer.subscribe(x => console.log("Value from Sub1 = "+x));
const subscribe_two = observer.subscribe(x => console.log("Value from Sub2 = "+x));
setTimeout(() => {
  const subscribe_three = observer.subscribe(x => console.log("Value from Sub3 = "+x));
}, 2000);
```

Output

Elements	Console	Sources	Network	Perfor
top				
Value from Sub1 = 0				
Value from Sub2 = 0				
Value from Sub1 = 1				
Value from Sub2 = 1				
Value from Sub1 = 2				
Value from Sub2 = 2				
Value from Sub3 = 2				

catchError

This operator takes care of catching errors on the source Observable by returning a new Observable or an error.

Syntax

```
catchError(selector_func: (err_func: any, caught: Observable) => 0):Observable
```

Parameters

selector_func: The selector func takes in 2 arguments, error function and caught which is an Observable.

Return value

It returns an observable based on the value emitted by the selector_func.

Example

```
import { of } from 'rxjs';
import { map, filter, catchError } from 'rxjs/operators';

let all_nums = of(1, 6, 5, 10, 9, 20, 40);
let final_val = all_nums.pipe(
  map(el => {
    if (el === 10) {

      throw new Error("Testing catchError.");
    }

    return el;
  }),
  catchError(err => {
    console.error(err.message);
  })
);
```

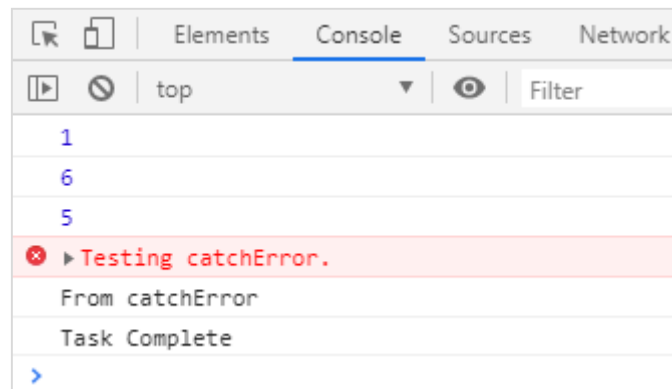


```

        return of("From catchError");
    })
);
final_val.subscribe(
    x => console.log(x),
    err => console.error(err),
    () => console.log("Task Complete")
);

```

Output



retry

This operator will take care of retrying back on the source Observable if there is error and the retry will be done based on the input count given.

Syntax

```
retry(retry_count: number): Observable
```

Parameters

retry_count: The argument `retry_count`, is the number of times you want to retry.

Return value

It will return back source observable with retry count logic.

Example

```

import { of } from 'rxjs';
import { map, retry } from 'rxjs/operators';
import { ajax } from 'rxjs/ajax';

let all_nums = of(1, 6, 5, 10, 9, 20, 10);
let final_val = ajax('http://localhost:8081/getData').pipe(retry(4));

```

```
final_val.subscribe(
  x => console.log(x),
  err => console.error(err),
  () => console.log("Task Complete")
);
```

In the example, we are making a call to a url using ajax. The url: <http://localhost:8081/getData> is giving a 404 so the `retry()` operator tries to make a call to url again for 4 times. The output is shown below:

Output

```
✖ ▶ GET http://localhost:8081/getData net::ERR_CONNECTION_REFUSED AjaxObservable.js:187
✖ ▶ GET http://localhost:8081/getData net::ERR_CONNECTION_REFUSED AjaxObservable.js:187
✖ ▶ GET http://localhost:8081/getData net::ERR_CONNECTION_REFUSED AjaxObservable.js:187
✖ ▶ GET http://localhost:8081/getData net::ERR_CONNECTION_REFUSED AjaxObservable.js:187
✖ ▶
  ▶ AjaxErrorImpl {message: "ajax error", name: "AjaxError", xhr: XMLHttpRequest, request: {...}, status: 0, ...} testrx.js:13
✖ ▶ GET http://localhost:8081/getData net::ERR_CONNECTION_REFUSED AjaxObservable.js:187
> |
```

6. RxJS — Working with Subscription

When the observable is created, to execute the observable we need to subscribe to it.

count() operator

Here, is a simple example of how to subscribe to an observable.

Example 1

```
import { of } from 'rxjs';
import { count } from 'rxjs/operators';

let all_nums = of(1, 7, 5, 10, 10, 20);
let final_val = all_nums.pipe(count());
final_val.subscribe(x => console.log("The count is "+x));
```

Output

```
The count is 6
```

The subscription has one method called `unsubscribe()`. A call to `unsubscribe()` method will remove all the resources used for that observable i.e. the observable will get canceled. Here, is a working example of using `unsubscribe()` method.

Example 2

```
import { of } from 'rxjs';
import { count } from 'rxjs/operators';

let all_nums = of(1, 7, 5, 10, 10, 20);
let final_val = all_nums.pipe(count());
let test = final_val.subscribe(x => console.log("The count is "+x));
test.unsubscribe();
```

The subscription is stored in the variable `test`. We have used `test.unsubscribe()` the observable.

Output

```
The count is 6
```

7.RxJS — Working with Subjects

A subject is an observable that can multicast i.e. talk to many observers. Consider a button with an event listener, the function attached to the event using add listener is called every time the user clicks on the button similar functionality goes for subject too.

We are going to discuss the following topics in this chapter:

- Create a subject
- What is the Difference between Observable and Subject?
- Behaviour Subject
- Replay Subject
- AsyncSubject

Create a subject

To work with subject, we need to import Subject as shown below:

```
import { Subject } from 'rxjs';
```

You can create a subject object as follows:

```
const subject_test = new Subject();
```

The object is an observer that has three methods:

- next(v)
- error(e)
- complete()

Subscribe to a Subject

You can create multiple subscription on the subject as shown below:

```
subject_test.subscribe({
  next: (v) => console.log(`From Subject : ${v}`)
});

subject_test.subscribe({

  next: (v) => console.log(`From Subject: ${v}`)
});
```

The subscription is registered to the subject object just like addlistener we discussed earlier.

Passing Data to Subject

You can pass data to the subject created using the next() method.

```
subject_test.next("A");
```

The data will be passed to all the subscription added on the subject.

Example

Here, is a working example of the subject:

```
import { Subject } from 'rxjs';

const subject_test = new Subject();

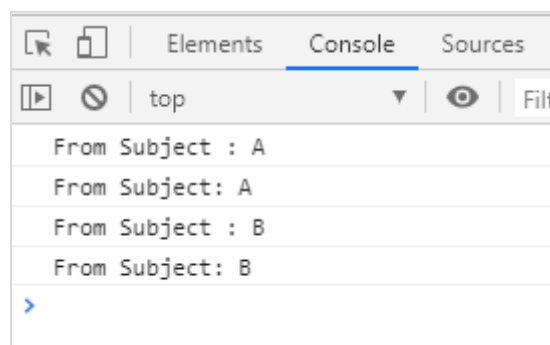
subject_test.subscribe({
  next: (v) => console.log(`From Subject : ${v}`)
});

subject_test.subscribe({
  next: (v) => console.log(`From Subject: ${v}`)
});

subject_test.next("A");
subject_test.next("B");
```

The subject_test object is created by calling a new Subject(). The subject_test object has reference to next(), error() and complete() methods. The output of the above example is shown below:

Output



We can use complete() method to stop the subject execution as shown below.

Example

```
import { Subject } from 'rxjs';

const subject_test = new Subject();

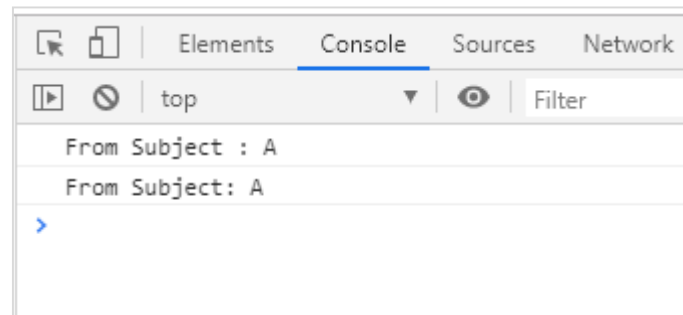
subject_test.subscribe({
  next: (v) => console.log(`From Subject : ${v}`)
});

subject_test.subscribe({
  next: (v) => console.log(`From Subject: ${v}`)
});

subject_test.next("A");
subject_test.complete();
subject_test.next("B");
```

Once we call complete the next method called later is not invoked.

Output



Let us now see how to call error () method.

Example

Below is a working example:

```
import { Subject } from 'rxjs';

const subject_test = new Subject();

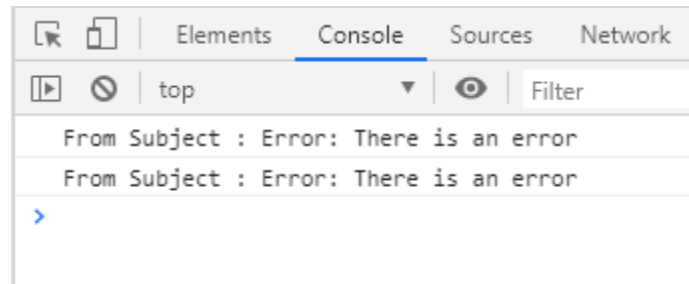
subject_test.subscribe({
  error: (e) => console.log(`From Subject : ${e}`)
});
```

```

subject_test.subscribe({
  error: (e) => console.log(`From Subject : ${e}`)
});
subject_test.error(new Error("There is an error"));

```

Output



What is the Difference between Observable and Subject?

An observable will talk one to one, to the subscriber. Anytime you subscribe to the observable the execution will start from scratch. Take an Http call made using ajax, and 2 subscribers calling the observable. You will see 2 Http requests in the browser network tab.

Example

Here is a working example of same:

```

import { ajax } from 'rxjs/ajax';

import { map } from 'rxjs/operators';

let final_val = ajax('https://jsonplaceholder.typicode.com/users').pipe(map(e
=> e.response));

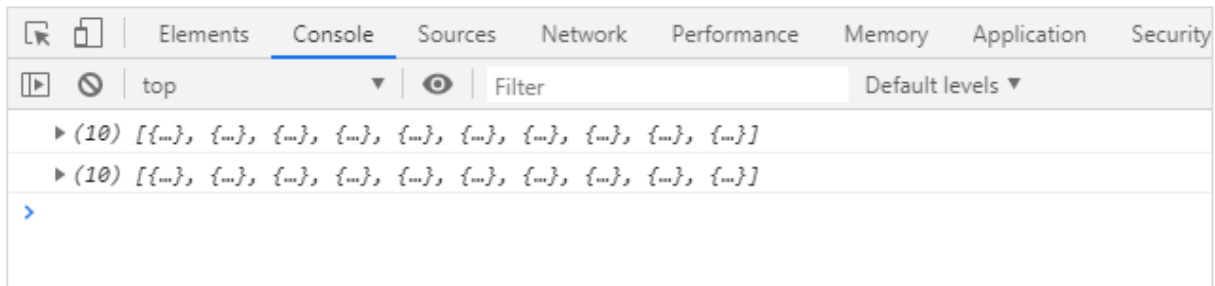
let subscriber1 = final_val.subscribe(a => console.log(a));
let subscriber2 = final_val.subscribe(a => console.log(a));

```

Output

The screenshot shows the Chrome DevTools Network tab. It displays a list of network requests. Two requests to 'users' are highlighted with an orange box. The first request is a 2.2 KB JSON response (550 ms), and the second is a 44 B response (715 ms). The waterfall chart on the right shows the timing of these requests.

Name	Status	Type	Initiator	Size	Time	Waterfall
localhost	200	document	Other	729 B	337 ms	
main_bundle.js	200	script	(index)	570 KB	15 ms	
users	200	xhr	AjaxObservable.js:187	2.2 KB	550 ms	
users	200	xhr	AjaxObservable.js:187	44 B	715 ms	



Now, here the problem is, we want the same data to be shared, but not, at the cost of 2 Http calls. We want to make one Http call and share the data between subscribers.

This will be possible using Subjects. It is an observable that can multicast i.e. talk to many observers. It can share the value between subscribers.

Example

Here is a working example using Subjects:

```
import { Subject } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { map } from 'rxjs/operators';

const subject_test = new Subject();

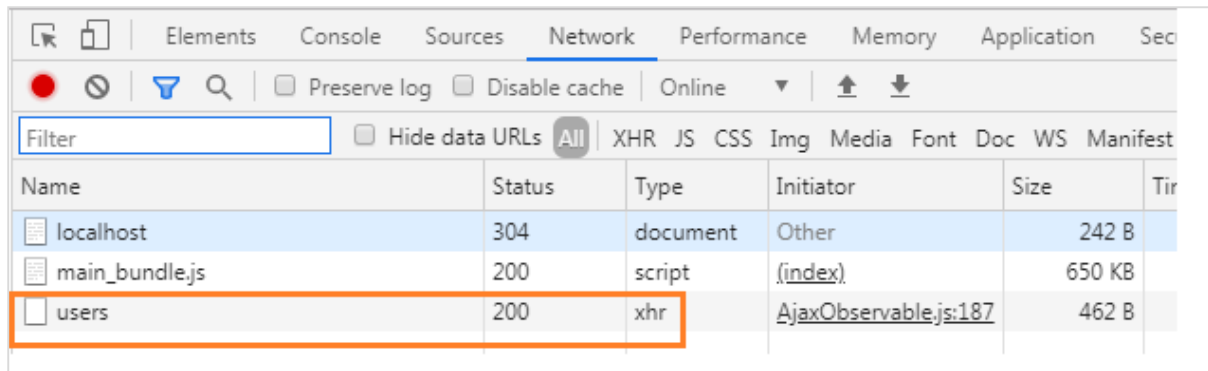
subject_test.subscribe({
  next: (v) => console.log(v)
});

subject_test.subscribe({

  next: (v) => console.log(v)
});

let final_val = ajax('https://jsonplaceholder.typicode.com/users').pipe(map(e
=> e.response));
let subscriber = final_val.subscribe(subject_test);
```

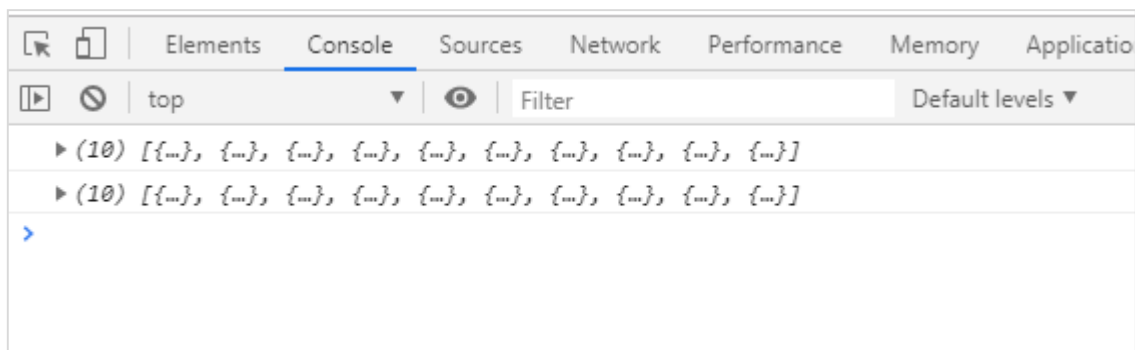
Output



The screenshot shows the Chrome DevTools Network tab. The 'Filter' field is empty, and the 'All' radio button is selected. The table below lists the network requests:

Name	Status	Type	Initiator	Size	Time
localhost	304	document	Other	242 B	
main_bundle.js	200	script	(index)	650 KB	
users	200	xhr	AjaxObservable.js:187	462 B	

Now you can see only one Http call and the same data is shared between the subscribers called.



The screenshot shows the Chrome DevTools Console. The 'top' dropdown is selected, and the 'Filter' field is empty. The console output shows two identical array logs:

```

▶ (10) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}]
▶ (10) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}]

```

Behaviour Subject

Behaviour subject will give you the latest value when called.

You can create behaviour subject as shown below:

```

import { BehaviorSubject } from 'rxjs';
const subject = new BehaviorSubject("Testing Behaviour Subject"); //
initialized the behaviour subject with value:Testing Behaviour Subject

```

Example

Here is a working example to use Behaviour Subject:

```

import { BehaviorSubject } from 'rxjs';
const behavior_subject = new BehaviorSubject("Testing Behaviour Subject"); //
0 is the initial value

behavior_subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});

behavior_subject.next("Hello");

```

```
behavior_subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});
behavior_subject.next("Last call to Behaviour Subject");
```

Output

Elements	Console	Sources	Network	Performance
top				
observerA: Testing Behaviour Subject				
observerA: Hello				
observerB: Hello				
observerA: Last call to Behaviour Subject				
observerB: Last call to Behaviour Subject				

Replay Subject

A `ReplaySubject` is similar to `behaviour subject`, wherein, it can buffer the values and replay the same to the new subscribers.

Example

Here is a working example of `replay subject`:

```
import { ReplaySubject } from 'rxjs';

const replay_subject = new ReplaySubject(2); // buffer 2 values but new subscribers

replay_subject.subscribe({
  next: (v) => console.log(`Testing Replay Subject A: ${v}`)
});

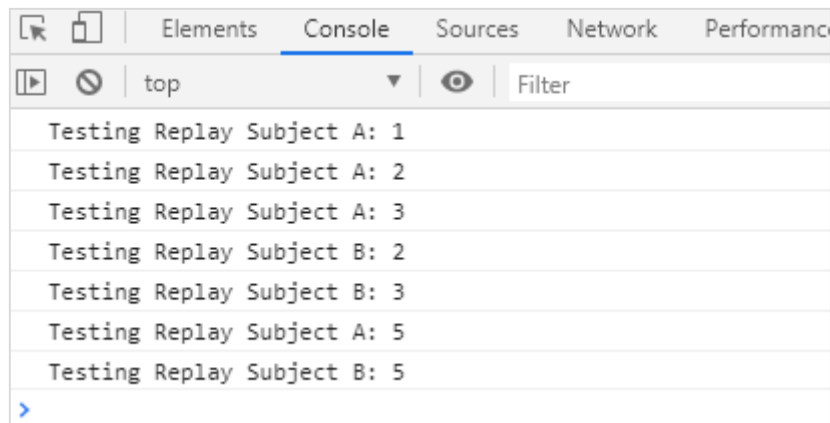
replay_subject.next(1);
replay_subject.next(2);
replay_subject.next(3);

replay_subject.subscribe({
  next: (v) => console.log(`Testing Replay Subject B: ${v}`)
});
```

```
replay_subject.next(5);
```

The buffer value used is 2 on the replay subject. So the last two values will be buffered and used for the new subscribers called.

Output



AsyncSubject

In the case of AsyncSubject the last value called is passed to the subscriber and it will be done only after complete() method is called.

Example

Here is a working example of the same:

```
import { AsyncSubject } from 'rxjs';

const async_subject = new AsyncSubject();

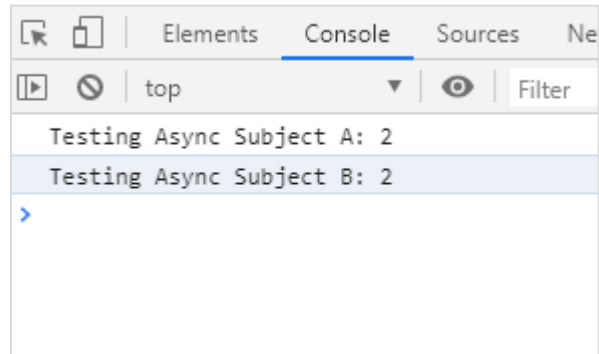
async_subject.subscribe({
  next: (v) => console.log(`Testing Async Subject A: ${v}`)
});

async_subject.next(1);
async_subject.next(2);
async_subject.complete();

async_subject.subscribe({
  next: (v) => console.log(`Testing Async Subject B: ${v}`)
});
```

Here, before complete is called the last value passed to the subject is 2 and the same it given to the subscribers.

Output



8. RxJS — Working with Scheduler

A scheduler controls the execution of when the subscription has to start and notified.

To make use of scheduler we need the following:

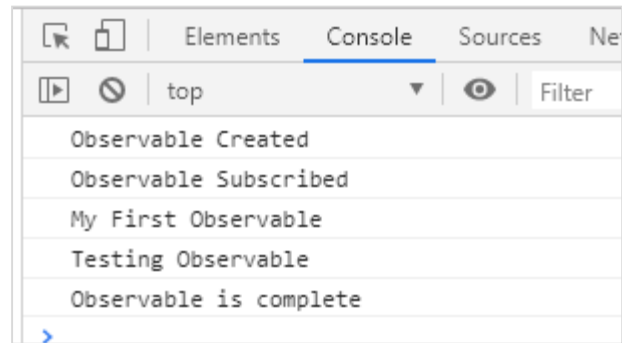
```
import { Observable, asyncScheduler } from 'rxjs';  
import { observeOn } from 'rxjs/operators';
```

Here is a working example, wherein, we will use the scheduler that will decide the execution.

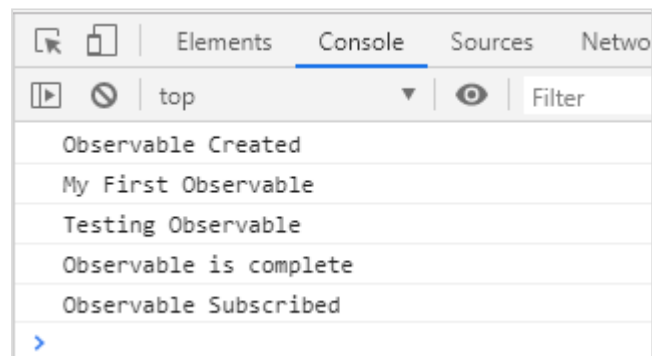
Example

```
import { Observable, asyncScheduler } from 'rxjs';  
import { observeOn } from 'rxjs/operators';  
  
var observable = new Observable(function subscribe(subscriber) {  
    subscriber.next("My First Observable");  
    subscriber.next("Testing Observable");  
    subscriber.complete();  
}).pipe(  
    observeOn(asyncScheduler)  
);  
  
console.log("Observable Created");  
  
observable.subscribe(  
    x => console.log(x),  
    (e)=>console.log(e),  
    ()=>console.log("Observable is complete")  
);  
  
console.log('Observable Subscribed');
```

Output



Without scheduler the output would have been as shown below:



9. RxJS — Working with RxJS and Angular

In this chapter, we will see how to use RxJs with Angular. We will not get into the installation process for Angular here, to know about Angular Installation refer this link: https://www.tutorialspoint.com/angular7/angular7_environment_setup.htm

We will directly work on an example, where will use Ajax from RxJS to load data.

Example

app.component.ts

```
import { Component } from '@angular/core';
import { environment } from '../environments/environment';
import { ajax } from 'rxjs/ajax';
import { map } from 'rxjs/operators'

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = '';
  data;
  constructor() {
    this.data = "";
    this.title = "Using RxJs with Angular";
    let a = this.getData();
  }

  getData() {
    const response =
    ajax('https://jsonplaceholder.typicode.com/users').pipe(map(e => e.response));
    response.subscribe(res => {
      console.log(res);

      this.data = res;
    });
  }
}
```

```

    });
  }
}

```

app.component.html

```

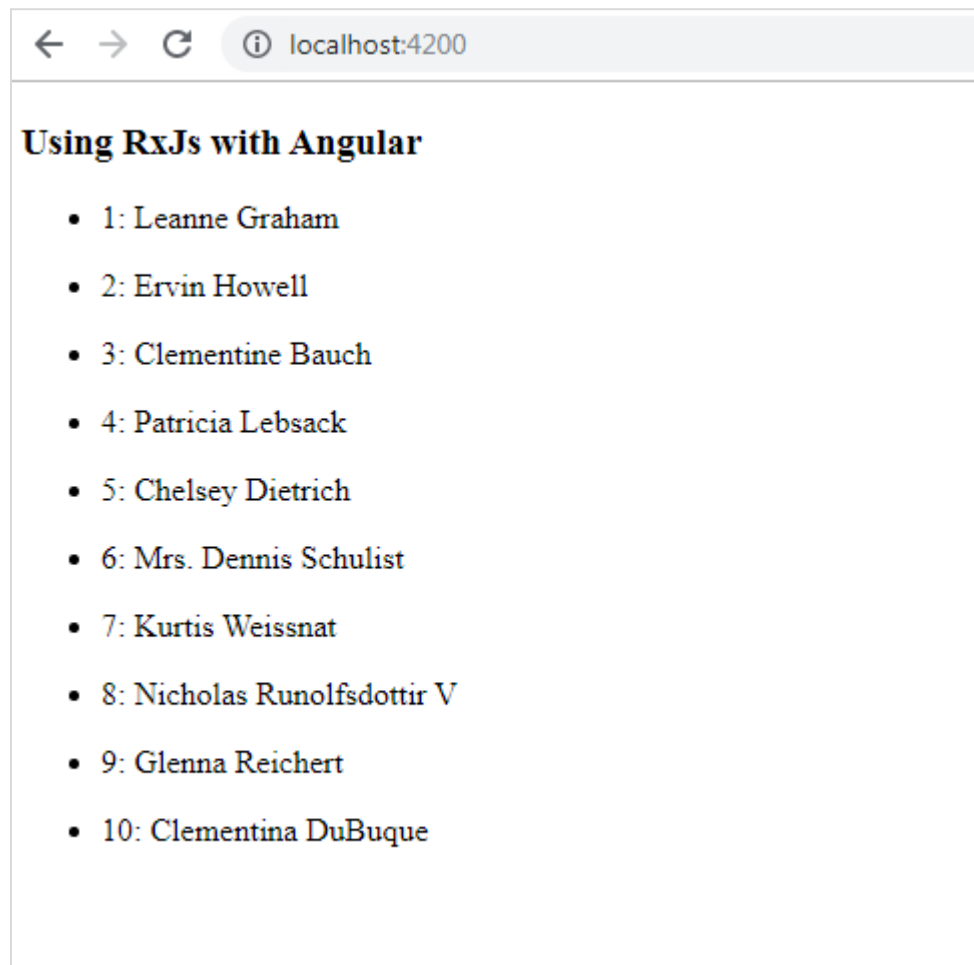
<div>
  <h3>{{title}}</h3>
  <ul *ngFor="let i of data">
    <li>{{i.id}}: {{i.name}}</li>
  </ul>
</div>

<router-outlet></router-outlet>

```

We have used ajax from RxJS that will load data from this url: <https://jsonplaceholder.typicode.com/users>.

When you compile the display is as shown below:



10. RxJS — Working with RxJS and ReactJS

In this chapter, we will see how to use RxJS with ReactJS. We will not get into the installation process for ReactJS here, to know about ReactJS Installation refer this link: https://www.tutorialspoint.com/reactjs/reactjs_environment_setup.html

Example

We will directly work on an example below, where will use Ajax from RxJS to load data.

index.js

```
import React, { Component } from "react";
import ReactDOM from "react-dom";
import { ajax } from 'rxjs/ajax';
import { map } from 'rxjs/operators';
class App extends Component {
  constructor() {
    super();
    this.state = { data: [] };
  }
  componentDidMount() {
    const response =
      ajax('https://jsonplaceholder.typicode.com/users').pipe(map(e => e.response));
    response.subscribe(res => {
      this.setState({ data: res });
    });
  }
  render() {
    return (
      <div>
        <h3>Using RxJS with ReactJS</h3>
        <ul>
          {this.state.data.map(el => (
            <li>
              {el.id}: {el.name}
            </li>
          ))}
        </ul>
      </div>
    );
  }
}
```

```

        </ul>
      </div>
    );
  }
}

ReactDOM.render(<App />, document.getElementById("root"));

```

index.html

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>ReactJS Demo</title>
  </head>
  <body>
    <div id = "root"></div>
  </body>
</html>

```

We have used ajax from RxJS that will load data from this Url : <https://jsonplaceholder.typicode.com/users>.

When you compile, the display is as shown below:

