
Table of Contents

Learn RxJS

Introduction	1.1
Operators	1.2
Combination	1.2.1
combineAll	1.2.1.1
combineLatest	1.2.1.2
concat	1.2.1.3
concatAll	1.2.1.4
forkJoin	1.2.1.5
merge	1.2.1.6
mergeAll	1.2.1.7
pairwise	1.2.1.8
race	1.2.1.9
startWith	1.2.1.10
withLatestFrom	1.2.1.11
zip	1.2.1.12
Conditional	1.2.2
defaultIfEmpty	1.2.2.1
every	1.2.2.2
iif	1.2.2.3
Creation	1.2.3
create	1.2.3.1
empty	1.2.3.2
from	1.2.3.3
fromEvent	1.2.3.4
interval	1.2.3.5

of	1.2.3.6
range	1.2.3.7
throw	1.2.3.8
timer	1.2.3.9
Error Handling	1.2.4
catch / catchError	1.2.4.1
retry	1.2.4.2
retryWhen	1.2.4.3
Multicasting	1.2.5
publish	1.2.5.1
multicast	1.2.5.2
share	1.2.5.3
shareReplay	1.2.5.4
Filtering	1.2.6
audit	1.2.6.1
auditTime	1.2.6.2
debounce	1.2.6.3
debounceTime	1.2.6.4
distinctUntilChanged	1.2.6.5
filter	1.2.6.6
first	1.2.6.7
ignoreElements	1.2.6.8
last	1.2.6.9
sample	1.2.6.10
single	1.2.6.11
skip	1.2.6.12
skipUntil	1.2.6.13
skipWhile	1.2.6.14
take	1.2.6.15
takeUntil	1.2.6.16

takeWhile	1.2.6.17
throttle	1.2.6.18
throttleTime	1.2.6.19
Transformation	1.2.7
buffer	1.2.7.1
bufferCount	1.2.7.2
bufferTime	1.2.7.3
bufferToggle	1.2.7.4
bufferWhen	1.2.7.5
concatMap	1.2.7.6
concatMapTo	1.2.7.7
exhaustMap	1.2.7.8
expand	1.2.7.9
groupBy	1.2.7.10
map	1.2.7.11
mapTo	1.2.7.12
mergeMap / flatMap	1.2.7.13
partition	1.2.7.14
pluck	1.2.7.15
reduce	1.2.7.16
scan	1.2.7.17
switchMap	1.2.7.18
window	1.2.7.19
windowCount	1.2.7.20
windowTime	1.2.7.21
windowToggle	1.2.7.22
windowWhen	1.2.7.23
Utility	1.2.8
do / tap	1.2.8.1
delay	1.2.8.2

delayWhen	1.2.8.3
dematerialize	1.2.8.4
finalize / finally	1.2.8.5
let	1.2.8.6
repeat	1.2.8.7
timeout	1.2.8.8
toPromise	1.2.8.9
Full Listing	1.2.9
Recipes	1.3
Http Polling	1.3.1
Game Loop	1.3.2
Progress Bar	1.3.3
Smart Counter	1.3.4
Concepts	1.4
RxJS v5 -> v6 Upgrade	1.4.1
Understanding Operator Imports	1.4.2

Learn RxJS

Clear examples, explanations, and resources for RxJS.

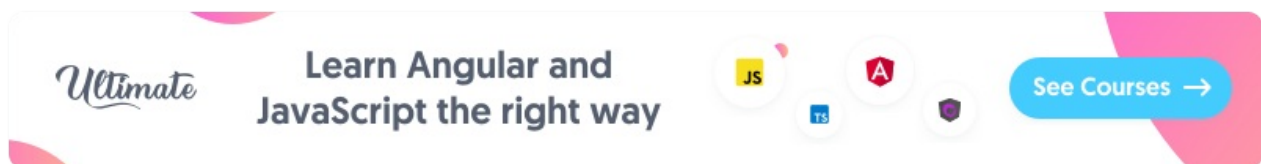
By [@btroncone](#)

Introduction

[RxJS](#) is one of the hottest libraries in web development today. Offering a powerful, functional approach for dealing with events and with integration points into a growing number of frameworks, libraries, and utilities, the case for learning Rx has never been more appealing. Couple this with the ability to utilize your knowledge across [nearly any language](#), having a solid grasp on reactive programming and what it can offer seems like a no-brainer.

But...

Learning RxJS and reactive programming is [hard](#). There's the multitude of concepts, large API surface, and fundamental shift in mindset from an [imperative to declarative style](#). This site focuses on making these concepts approachable, the examples clear and easy to explore, and features references throughout to the best RxJS related material on the web. The goal is to supplement the [official docs](#) and pre-existing learning material while offering a new, fresh perspective to clear any hurdles and tackle the pain points. Learning Rx may be difficult but it is certainly worth the effort!



Content

Operators

Operators are the horse-power behind observables, providing an elegant, declarative solution to complex asynchronous tasks. This section contains all [RxJS operators](#), included with clear, executable examples in both [JSBin](#) and [JSFiddle](#). Links to additional resources and recipes for each operator are also provided, when applicable.

Categories

- [Combination](#)
- [Conditional](#)
- [Creation](#)
- [Error Handling](#)
- [Multicasting](#)
- [Filtering](#)
- [Transformation](#)
- [Utility](#)

OR...

[Complete listing in alphabetical order](#)

Concepts

Without a solid base knowledge of how Observables work behind the scenes, it's easy for much of RxJS to feel like 'magic'. This section helps solidify the major concepts needed to feel comfortable with reactive programming and Observables.

[Complete Concept Listing](#)

Recipes

Recipes for common use-cases and interesting solutions with RxJS.

[Complete Recipe Listing](#)

Introductory Resources

New to RxJS and reactive programming? In addition to the content found on this site, these excellent articles and videos will help jump start your learning experience!

Reading

- [RxJS Introduction](#) - Official Docs
- [The Introduction to Reactive Programming You've Been Missing](#) - André Staltz

Videos

- [Asynchronous Programming: The End of The Loop](#) - Jafar Husain
- [What is RxJS?](#) - Ben Lesh
- [Creating Observable from Scratch](#) - Ben Lesh
- [Introduction to RxJS Marble Testing](#) - Brian Troncone
- [Introduction to Reactive Programming](#) 📺 - André Staltz
- [Reactive Programming using Observables](#) - Jeremy Lund

Exercises

- [Functional Programming in JavaScript](#) - Jafar Husain

Tools

- [Rx Marbles](#) - Interactive diagrams of Rx Observables - André Staltz
- [Rx Visualizer](#) - Animated playground for Rx Observables - Misha Moroshko
- [Reactive.how](#) - Animated cards to learn Reactive Programming - Cédric Soulas
- [Rx Visualization](#) - Visualizes programming with RxJS - Mojtaba Zarei

Interested in RxJS 4? Check out [Denis Stoyanov's](#) excellent [eBook](#)!

Translations

- [简体中文](#)

A Note On References

All references included in this GitBook are resources, both free and paid, that helped me tremendously while learning RxJS. If you come across an article or video that you think should be included, please use the *edit this page* link in the top menu and submit a pull request. Your feedback is appreciated!

RxJS Operators By Example

A complete list of RxJS operators with clear explanations, relevant resources, and executable examples.

[Prefer a complete list in alphabetical order?](#)

Contents (By Operator Type)

- **Combination**
 - [combineAll](#)
 - [combineLatest](#) ★
 - [concat](#) ★
 - [concatAll](#)
 - [forkJoin](#)
 - [merge](#) ★
 - [mergeAll](#)
 - [race](#)
 - [startWith](#) ★
 - [withLatestFrom](#) ★
 - [zip](#)
- **Conditional**
 - [defaultIfEmpty](#)
 - [every](#)
 - [iif](#)
- **Creation**
 - [create](#)
 - [empty](#)
 - [from](#) ★
 - [fromEvent](#)
 - [interval](#)
 - [of](#) ★
 - [range](#)
 - [throw](#)
 - [timer](#)

- Error Handling
 - catch / catchError ★
 - retry
 - retryWhen
- Filtering
 - audit
 - auditTime
 - debounce
 - debounceTime ★
 - distinctUntilChanged ★
 - filter ★
 - first
 - ignoreElements
 - last
 - sample
 - single
 - skip
 - skipUntil
 - skipWhile
 - take ★
 - takeUntil ★
 - takeWhile
 - throttle
 - throttleTime
- Multicasting
 - multicast
 - publish
 - share ★
 - shareReplay ★
- Transformation
 - buffer
 - bufferCount
 - bufferTime ★
 - bufferToggle
 - bufferWhen
 - concatMap ★

- `concatMapTo`
- `expand`
- `exhaustMap`
- `groupBy`
- `map` ★
- `mapTo`
- `mergeMap` / `flatMap` ★
- `partition`
- `pluck`
- `reduce`
- `scan` ★
- `switchMap` ★
- `window`
- `windowCount`
- `windowTime`
- `windowToggle`
- `windowWhen`
- **Utility**
 - `do` / `tap` ★
 - `delay`
 - `delayWhen`
 - `finalize` / `finally`
 - `let`
 - `repeat`
 - `toPromise`
 - `timeout`

★ - *commonly used*

Additional Resources

- [What Are Operators?](#) 📄 - Official Docs
- [What Operators Are](#) 📺 📄 - André Staltz

Combination Operators

The combination operators allow the joining of information from multiple observables. Order, time, and structure of emitted values is the primary variation among these operators.

Contents

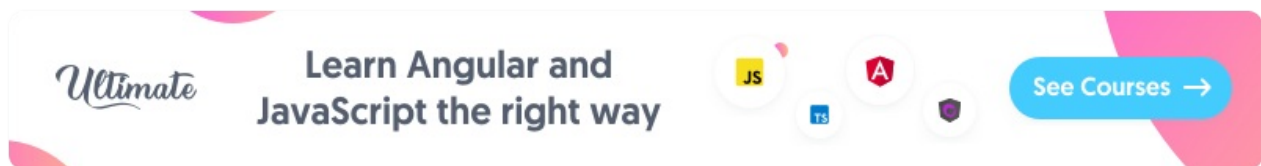
- [combineAll](#)
- [combineLatest](#) ★
- [concat](#) ★
- [concatAll](#)
- [forkJoin](#)
- [merge](#) ★
- [mergeAll](#)
- [pairwise](#)
- [race](#)
- [startWith](#) ★
- [withLatestFrom](#) ★
- [zip](#)

★ - *commonly used*

combineAll

signature: `combineAll(project: function): Observable`

When source observable completes use `combineLatest` with collected observables.



Examples

([example tests](#))

Example 1: Mapping to inner interval observable

([StackBlitz](#))

```
// RxJS v6+
import { take, map, combineAll } from 'rxjs/operators';
import { interval } from 'rxjs';

//emit every 1s, take 2
const source = interval(1000).pipe(take(2));
//map each emitted value from source to interval observable that
  takes 5 values
const example = source.pipe(
  map(val => interval(1000).pipe(map(i => `Result (${val}): ${i}`
), take(5)))
);
/*
  2 values from source will map to 2 (inner) interval observable
s that emit every 1s
  combineAll uses combineLatest strategy, emitting the last valu
e from each
  whenever either observable emits a value
*/
const combined = example.pipe(combineAll());
/*
  output:
  ["Result (0): 0", "Result (1): 0"]
  ["Result (0): 1", "Result (1): 0"]
  ["Result (0): 1", "Result (1): 1"]
  ["Result (0): 2", "Result (1): 1"]
  ["Result (0): 2", "Result (1): 2"]
  ["Result (0): 3", "Result (1): 2"]
  ["Result (0): 3", "Result (1): 3"]
  ["Result (0): 4", "Result (1): 3"]
  ["Result (0): 4", "Result (1): 4"]
*/
const subscribe = combined.subscribe(val => console.log(val));
```

Additional Resources

- [combineAll](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/combineAll.ts>

combineLatest

signature: `combineLatest(observables: ...Observable, project: function): Observable`

When any observable emits a value, emit the latest value from each.

💡 This operator can be used as either a static or instance method!

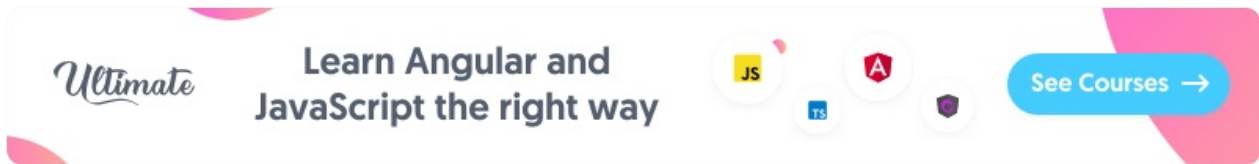
💡 `combineAll` can be used to apply `combineLatest` to emitted observables when a source completes!

Why use `combineLatest` ?

This operator is best used when you have multiple, long-lived observables that rely on each other for some calculation or determination. Basic examples of this can be seen in [example three](#), where events from multiple buttons are being combined to produce a count of each and an overall total, or a [calculation of BMI](#) from the RxJS documentation.

Be aware that `combineLatest` **will not emit an initial value until each observable emits at least one value**. This is the same behavior as `withLatestFrom` and can be a *gotcha* as there will be no output and no error but one (or more) of your inner observables is likely not functioning as intended, or a subscription is late.

Lastly, if you are working with observables that only emit one value, or you only require the last value of each before completion, `forkJoin` is likely a better option.



Examples

([example tests](#))

Example 1: Combining observables emitting at 3 intervals

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { timer, combineLatest } from 'rxjs';

//timerOne emits first value at 1s, then once every 4s
const timerOne = timer(1000, 4000);
//timerTwo emits first value at 2s, then once every 4s
const timerTwo = timer(2000, 4000);
//timerThree emits first value at 3s, then once every 4s
const timerThree = timer(3000, 4000);

//when one timer emits, emit the latest values from each timer as an array
const combined = combineLatest(timerOne, timerTwo, timerThree);

const subscribe = combined.subscribe(
  ([timerValOne, timerValTwo, timerValThree]) => {
    /*
      Example:
      timerOne first tick: 'Timer One Latest: 1, Timer Two Latest: 0, Timer Three Latest: 0
      timerTwo first tick: 'Timer One Latest: 1, Timer Two Latest: 1, Timer Three Latest: 0
      timerThree first tick: 'Timer One Latest: 1, Timer Two Latest: 1, Timer Three Latest: 1
    */
    console.log(
      `Timer One Latest: ${timerValOne},
      Timer Two Latest: ${timerValTwo},
      Timer Three Latest: ${timerValThree}`
    );
  }
);
```

Example 2: combineLatest with projection function

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { timer, combineLatest } from 'rxjs';

//timerOne emits first value at 1s, then once every 4s
const timerOne = timer(1000, 4000);
//timerTwo emits first value at 2s, then once every 4s
const timerTwo = timer(2000, 4000);
//timerThree emits first value at 3s, then once every 4s
const timerThree = timer(3000, 4000);

//combineLatest also takes an optional projection function
const combinedProject = combineLatest(
  timerOne,
  timerTwo,
  timerThree,
  (one, two, three) => {
    return `Timer One (Proj) Latest: ${one},
           Timer Two (Proj) Latest: ${two},
           Timer Three (Proj) Latest: ${three}`;
  }
);
//log values
const subscribe = combinedProject.subscribe(latestValuesProject
=>
  console.log(latestValuesProject)
);
```

Example 3: Combining events from 2 buttons

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { fromEvent, combineLatest } from 'rxjs';
import { mapTo, startWith, scan, tap, map } from 'rxjs/operators';

// helper function to set HTML
const setHtml = id => val => (document.getElementById(id).innerHTML = val);

const addOneClick$ = id =>
  fromEvent(document.getElementById(id), 'click').pipe(
    // map every click to 1
    mapTo(1),
    startWith(0),
    // keep a running total
    scan((acc, curr) => acc + curr),
    // set HTML for appropriate element
    tap(setHtml(`${id}Total`))
  );

const combineTotal$ = combineLatest(addOneClick$('red'), addOneClick$('black'))
  .pipe(map(([val1, val2]) => val1 + val2))
  .subscribe(setHtml('total'));
```

HTML

```
<div>
  <button id='red'>Red</button>
  <button id='black'>Black</button>
</div>
<div>Red: <span id="redTotal"></span> </div>
<div>Black: <span id="blackTotal"></span> </div>
<div>Total: <span id="total"></span> </div>
```

Additional Resources

- [combineLatest](#) 📖 - Official docs

- [Combining streams with combineLatest](#) 🖨️ 💻 - John Linquist
 - [Combination operator: combineLatest](#) 🖨️ 💻 - André Staltz
-

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/combineLatest.ts>

concat

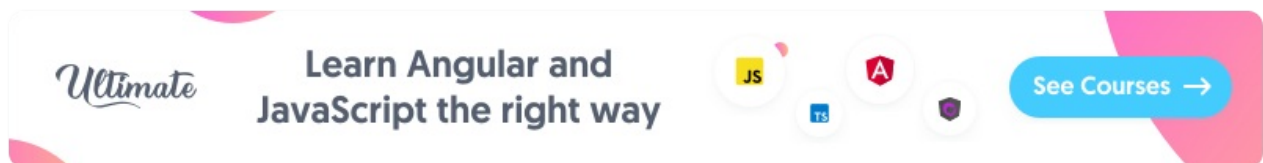
signature: `concat(observables: ...*): Observable`

Subscribe to observables in order as previous completes, emit values.

💡 You can think of concat like a line at a ATM, the next transaction (subscription) cannot start until the previous completes!

💡 This operator can be used as either a static or instance method!

💡 If throughput, not order, is a primary concern, try [merge](#) instead!



Examples

([example tests](#))

Example 1: concat 2 basic observables

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { concat } from 'rxjs/operators';
import { of } from 'rxjs';

//emits 1,2,3
const sourceOne = of(1, 2, 3);
//emits 4,5,6
const sourceTwo = of(4, 5, 6);
//emit values from sourceOne, when complete, subscribe to source
Two
const example = sourceOne.pipe(concat(sourceTwo));
//output: 1,2,3,4,5,6
const subscribe = example.subscribe(val =>
  console.log('Example: Basic concat:', val)
);
```

Example 2: concat as static method

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of, concat } from 'rxjs';

//emits 1,2,3
const sourceOne = of(1, 2, 3);
//emits 4,5,6
const sourceTwo = of(4, 5, 6);

//used as static
const example = concat(sourceOne, sourceTwo);
//output: 1,2,3,4,5,6
const subscribe = example.subscribe(val => console.log(val));
```

Example 3: concat with delayed source

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { delay, concat } from 'rxjs/operators';
import { of } from 'rxjs';

//emits 1,2,3
const sourceOne = of(1, 2, 3);
//emits 4,5,6
const sourceTwo = of(4, 5, 6);

//delay 3 seconds then emit
const sourceThree = sourceOne.pipe(delay(3000));
//sourceTwo waits on sourceOne to complete before subscribing
const example = sourceThree.pipe(concat(sourceTwo));
//output: 1,2,3,4,5,6
const subscribe = example.subscribe(val =>
  console.log('Example: Delayed source one:', val)
);
```


Example 4: concat with source that does not complete

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, of, concat } from 'rxjs';

//when source never completes, the subsequent observables never
runs
const source = concat(interval(1000), of('This', 'Never', 'Runs'))
);
//outputs: 0,1,2,3,4....
const subscribe = source.subscribe(val =>
  console.log(
    'Example: Source never completes, second observable never ru
ns:',
    val
  )
);
```


Additional Resources

- [concat](#)  - Official docs
 - [Combination operator: concat, startWith](#)   - André Staltz
-

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/concat.ts>

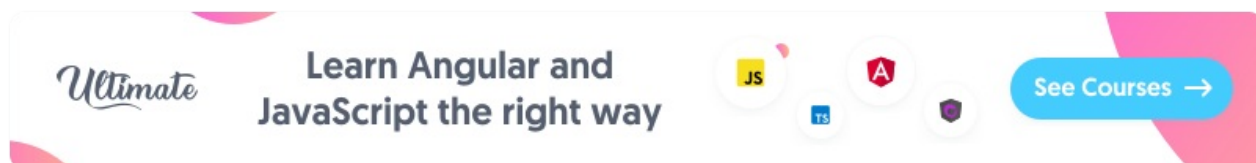
concatAll

signature: `concatAll(): Observable`

Collect observables and subscribe to next when previous completes.

⚠ Be wary of [backpressure](#) when the source emits at a faster pace than inner observables complete!

💡 In many cases you can use [concatMap](#) as a single operator instead!



Examples

([example tests](#))

Example 1: concatAll with observable

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { map, concatAll } from 'rxjs/operators';
import { of, interval } from 'rxjs';

//emit a value every 2 seconds
const source = interval(2000);
const example = source.pipe(
  //for demonstration, add 10 to and return as observable
  map(val => of(val + 10)),
  //merge values from inner observable
  concatAll()
);
//output: 'Example with Basic Observable 10', 'Example with Basic Observable 11'...
const subscribe = example.subscribe(val =>
  console.log('Example with Basic Observable:', val)
);
```

Example 2: concatAll with promise

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { map, concatAll } from 'rxjs/operators';
import { interval } from 'rxjs';

//create and resolve basic promise
const samplePromise = val => new Promise(resolve => resolve(val))
;
//emit a value every 2 seconds
const source = interval(2000);

const example = source.pipe(
  map(val => samplePromise(val)),
  //merge values from resolved promise
  concatAll()
);
//output: 'Example with Promise 0', 'Example with Promise 1'...
const subscribe = example.subscribe(val =>
  console.log('Example with Promise:', val)
);
```

Example 3: Delay while inner observables complete

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { take, concatAll } from 'rxjs/operators';
import { interval, of } from 'rxjs/observable/interval';

const obs1 = interval(1000).pipe(take(5));
const obs2 = interval(500).pipe(take(2));
const obs3 = interval(2000).pipe(take(1));
//emit three observables
const source = of(obs1, obs2, obs3);
//subscribe to each inner observable in order when previous completes
const example = source.pipe(concatAll());
/*
  output: 0,1,2,3,4,0,1,0
  How it works...
  Subscribes to each inner observable and emit values, when complete subscribe to next
  obs1: 0,1,2,3,4 (complete)
  obs2: 0,1 (complete)
  obs3: 0 (complete)
*/

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

Related Recipes

- [Progress Bar](#)

Additional Resources

- [concatAll](#)  - Official docs
- [Flatten a higher order observable with concatAll in RxJS](#)   - André Staltz

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/concatAll.ts>

forkJoin

signature: `forkJoin(...args, selector : function): Observable`

When all observables complete, emit the last emitted value from each.

💡 If you want corresponding emissions from multiple observables as they occur, try [zip](#)!

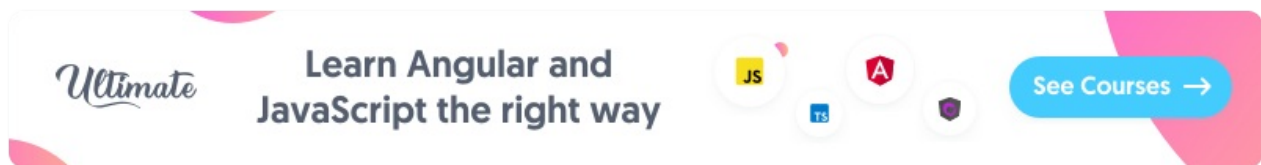
⚠️ If an inner observable does not complete `forkJoin` will never emit a value!

Why use `forkJoin` ?

This operator is best used when you have a group of observables and only care about the final emitted value of each. One common use case for this is if you wish to issue multiple requests on page load (or some other event) and only want to take action when a response has been received for all. In this way it is similar to how you might use `Promise.all`.

Be aware that if any of the inner observables supplied to `forkJoin` error you will lose the value of any other observables that would or have already completed if you do not `catch` the [error correctly on the inner observable](#). If you are only concerned with all inner observables completing successfully you can [catch the error on the outside](#).

It's also worth noting that if you have an observable that emits more than one item, and you are concerned with the previous emissions `forkJoin` is not the correct choice. In these cases you may be better off with an operator like [combineLatest](#) or [zip](#).



Examples

Example 1: Observables completing after different durations

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { delay, take } from 'rxjs/operators';
import { forkJoin, of, interval } from 'rxjs';

const myPromise = val =>
  new Promise(resolve =>
    setTimeout(() => resolve(`Promise Resolved: ${val}`), 5000)
  );

/*
  when all observables complete, give the last
  emitted value from each as an array
*/
const example = forkJoin(
  //emit 'Hello' immediately
  of('Hello'),
  //emit 'World' after 1 second
  of('World').pipe(delay(1000)),
  //emit 0 after 1 second
  interval(1000).pipe(take(1)),
  //emit 0...1 in 1 second interval
  interval(1000).pipe(take(2)),
  //promise that resolves to 'Promise Resolved' after 5 seconds
  myPromise('RESULT')
);
//output: ["Hello", "World", 0, 1, "Promise Resolved: RESULT"]
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Making a variable number of requests

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { mergeMap } from 'rxjs/operators';
import { forkJoin, of } from 'rxjs';

const myPromise = val =>
  new Promise(resolve =>
    setTimeout(() => resolve(`Promise Resolved: ${val}`), 5000)
  );

const source = of([1, 2, 3, 4, 5]);
//emit array of all 5 results
const example = source.pipe(mergeMap(q => forkJoin(...q.map(myPromise))));
/*
  output:
  [
    "Promise Resolved: 1",
    "Promise Resolved: 2",
    "Promise Resolved: 3",
    "Promise Resolved: 4",
    "Promise Resolved: 5"
  ]
*/
const subscribe = example.subscribe(val => console.log(val));
```

Example 3: Handling errors on outside

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { delay, catchError } from 'rxjs/operators';
import { forkJoin, of, throwError } from 'rxjs';

/*
  when all observables complete, give the last
  emitted value from each as an array
*/
const example = forkJoin(
  //emit 'Hello' immediately
  of('Hello'),
  //emit 'World' after 1 second
  of('World').pipe(delay(1000)),
  // throw error
  _throw('This will error')
).pipe(catchError(error => of(error)));
//output: 'This will Error'
const subscribe = example.subscribe(val => console.log(val));
```

Example 4: Getting successful results when one inner observable errors

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { delay, catchError } from 'rxjs/operators';
import { forkJoin, of, throwError } from 'rxjs';

/*
  when all observables complete, give the last
  emitted value from each as an array
*/
const example = forkJoin(
  //emit 'Hello' immediately
  of('Hello'),
  //emit 'World' after 1 second
  of('World').pipe(delay(1000)),
  // throw error
  _throw('This will error').pipe(catchError(error => of(error)))
);
//output: ["Hello", "World", "This will error"]
const subscribe = example.subscribe(val => console.log(val));
```

Example 5: forkJoin in Angular

([plunker](#))

```
@Injectable()
export class MyService {
  makeRequest(value: string, delayDuration: number) {
    // simulate http request
    return of(`Complete: ${value}`).pipe(
      delay(delayDuration)
    );
  }
}

@Component({
  selector: 'my-app',
  template: `
    <div>
      <h2>forkJoin Example</h2>
      <ul>
        <li> {{propOne}} </li>
```

```
        <li> {{propTwo}} </li>
        <li> {{propThree}} </li>
    </ul>
</div>
    ,
})
export class App {
    public propOne: string;
    public propTwo: string;
    public propThree: string;
    constructor(private _myService: MyService) {}

    ngOnInit() {
        // simulate 3 requests with different delays
        forkJoin(
            this._myService.makeRequest('Request One', 2000),
            this._myService.makeRequest('Request Two', 1000),
            this._myService.makeRequest('Request Three', 3000)
        )
        .subscribe(([res1, res2, res3]) => {
            this.propOne = res1;
            this.propTwo = res2;
            this.propThree = res3;
        });
    }
}
```

Additional Resources

- [forkJoin](#)  - Official docs

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/observable/ForkJoinObservable.ts>

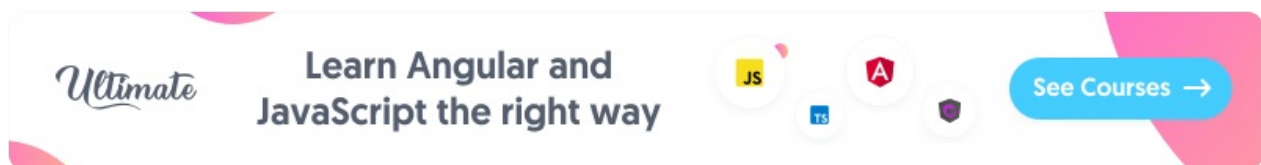
merge

signature: `merge(input: Observable): Observable`

Turn multiple observables into a single observable.

💡 This operator can be used as either a static or instance method!

💡 If order not throughput is a primary concern, try [concat](#) instead!



Examples

Example 1: merging multiple observables, static method

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { mapTo } from 'rxjs/operators';
import { interval, merge } from 'rxjs';

//emit every 2.5 seconds
const first = interval(2500);
//emit every 2 seconds
const second = interval(2000);
//emit every 1.5 seconds
const third = interval(1500);
//emit every 1 second
const fourth = interval(1000);

//emit outputs from one observable
const example = merge(
  first.pipe(mapTo('FIRST!')),
  second.pipe(mapTo('SECOND!')),
  third.pipe(mapTo('THIRD')),
  fourth.pipe(mapTo('FOURTH'))
);
//output: "FOURTH", "THIRD", "SECOND!", "FOURTH", "FIRST!", "THIRD", "FOURTH"
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: merge 2 observables, instance method

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))








```
// RxJS v6+
import { merge } from 'rxjs/operators';
import { interval } from 'rxjs';

//emit every 2.5 seconds
const first = interval(2500);
//emit every 1 second
const second = interval(1000);
//used as instance method
const example = first.pipe(merge(second));
//output: 0,1,0,2,...
const subscribe = example.subscribe(val => console.log(val));
```

Related Recipes

- [HTTP Polling](#)

Additional Resources

- [merge](#)  - Official docs
- [Handling multiple streams with merge](#)   - John Linquist
- [Sharing network requests with merge](#)   - André Staltz
- [Combination operator: merge](#)   - André Staltz



Source Code:

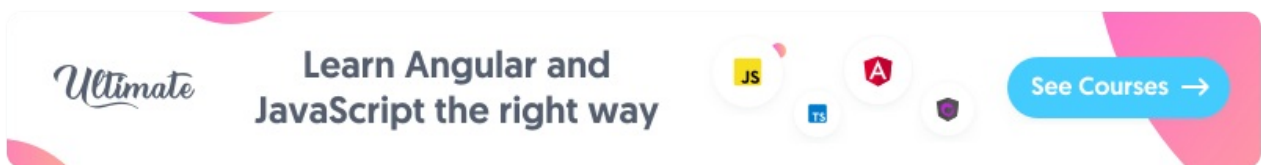
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/merge.ts>

mergeAll

signature: `mergeAll(concurrent: number): Observable`

Collect and subscribe to all observables.

💡 In many cases you can use [mergeMap](#) as a single operator instead!



Examples

([example tests](#))

Example 1: mergeAll with promises

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))


```
// RxJS v6+
import { map, mergeAll } from 'rxjs/operators';
import { of } from 'rxjs';

const myPromise = val =>
  new Promise(resolve => setTimeout(() => resolve(`Result: ${val}`), 2000));
//emit 1,2,3
const source = of(1, 2, 3);

const example = source.pipe(
  //map each value to promise
  map(val => myPromise(val)),
  //emit result from source
  mergeAll()
);

/*
  output:
  "Result: 1"
  "Result: 2"
  "Result: 3"
*/
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: mergeAll with *concurrent* parameter

([StackBlitz](#) | [jsFiddle](#))




```
// RxJS v6+
import { take, map, delay, mergeAll } from 'rxjs/operators';
import { interval } from 'rxjs';

const source = interval(500).pipe(take(5));

/*
  interval is emitting a value every 0.5s. This value is then being mapped to interval that
  is delayed for 1.0s. The mergeAll operator takes an optional argument that determines how
  many inner observables to subscribe to at a time. The rest of the observables are stored
  in a backlog waiting to be subscribe.
*/
const example = source
  .pipe(
    map(val =>
      source.pipe(
        delay(1000),
        take(3)
      )
    ),
    mergeAll(2)
  )
  .subscribe(val => console.log(val));

/*
  The subscription is completed once the operator emits all values.
*/
```

Additional Resources

- [mergeAll](#)  - Official docs
- [Flatten a higher order observable with mergeAll in RxJS](#)   - André Staltz



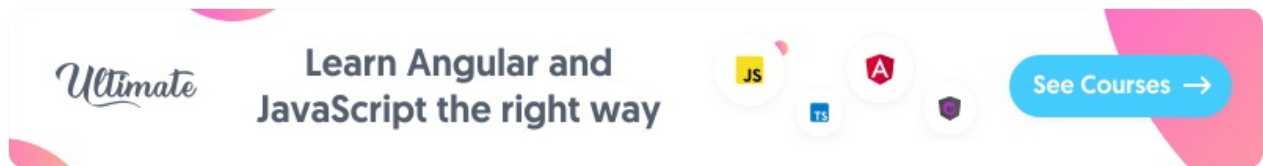
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/mergeAll.ts>

pairwise

signature: `pairwise(): Observable<Array>`

Emit the previous and current values as an array.



Examples

Example 1:

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { pairwise, take } from 'rxjs/operators';
import { interval } from 'rxjs';

//Returns: [0,1], [1,2], [2,3], [3,4], [4,5]
interval(1000)
  .pipe(
    pairwise(),
    take(5)
  )
  .subscribe(console.log);
```

Additional Resources

- [pairwise](#)  - Official docs



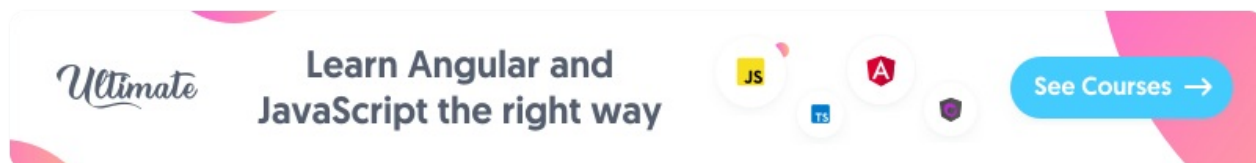
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/pairwise.ts>

race

signature: `race(): Observable`

The observable to emit first is used.



Examples

Example 1: race with 4 observables

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { mapTo } from 'rxjs/operators';
import { interval } from 'rxjs/observable/interval';
import { race } from 'rxjs/observable/race';

//take the first observable to emit
const example = race(
  //emit every 1.5s
  interval(1500),
  //emit every 1s
  interval(1000).pipe(mapTo('1s won!')),
  //emit every 2s
  interval(2000),
  //emit every 2.5s
  interval(2500)
);
//output: "1s won!"..."1s won!"...etc
const subscribe = example.subscribe(val => console.log(val));
```


Example 2: race with an error

([StackBlitz](#) | [jsFiddle](#))

```
// RxJS v6+
import { delay, map } from 'rxjs/operators';
import { of, race } from 'rxjs';

//Throws an error and ignores the other observables.
const first = of('first').pipe(
  delay(100),
  map(_ => {
    throw 'error';
  })
);
const second = of('second').pipe(delay(200));
const third = of('third').pipe(delay(300));
// nothing logged
race(first, second, third).subscribe(val => console.log(val));
```

Additional Resources

- [race](#)  - Official docs

 Source Code:

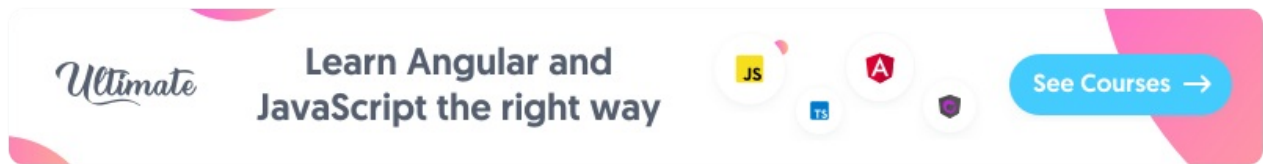
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/race.ts>

startWith

signature: `startWith(an: Values): Observable`

Emit given value first.

💡 A [BehaviorSubject](#) can also start with an initial value!



Examples

([example tests](#))

Example 1: startWith on number sequence

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { startWith } from 'rxjs/operators';
import { of } from 'rxjs';

//emit (1,2,3)
const source = of(1, 2, 3);
//start with 0
const example = source.pipe(startWith(0));
//output: 0,1,2,3
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: startWith for initial scan value

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { startWith, scan } from 'rxjs/operators';
import { of } from 'rxjs';

//emit ('World!', 'Goodbye', 'World!')
const source = of('World!', 'Goodbye', 'World!');
//start with 'Hello', concat current string to previous
const example = source.pipe(
  startWith('Hello'),
  scan((acc, curr) => `${acc} ${curr}`)
);
/*
  output:
  "Hello"
  "Hello World!"
  "Hello World! Goodbye"
  "Hello World! Goodbye World!"
*/
const subscribe = example.subscribe(val => console.log(val));
```

Example 3: startWith multiple values

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))








```
// RxJS v6+
import { startWith } from 'rxjs/operators';
import { interval } from 'rxjs';

//emit values in sequence every 1s
const source = interval(1000);
//start with -3, -2, -1
const example = source.pipe(startWith(-3, -2, -1));
//output: -3, -2, -1, 0, 1, 2....
const subscribe = example.subscribe(val => console.log(val));
```

Related Recipes

- [Smart Counter](#)

Additional Resources

- [startWith](#)  - Official docs
- [Displaying initial data with startWith](#)   - John Linquist
- [Clear data while loading with startWith](#)   - André Staltz
- [Combination operator: concat, startWith](#)   - André Staltz

 Source Code:

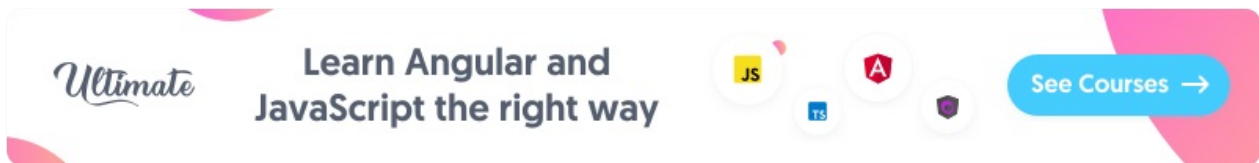
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/startWith.ts>

withLatestFrom

signature: `withLatestFrom(other: Observable, project: Function): Observable`

Also provide the last value from another observable.

💡 If you want the last emission any time a variable number of observables emits, try [combineLatest!](#)



Examples

Example 1: Latest value from quicker second source

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { withLatestFrom, map } from 'rxjs/operators';
import { interval } from 'rxjs';

//emit every 5s
const source = interval(5000);
//emit every 1s
const secondSource = interval(1000);
const example = source.pipe(
  withLatestFrom(secondSource),
  map(([first, second]) => {
    return `First Source (5s): ${first} Second Source (1s): ${second}`;
  })
);
/*
  "First Source (5s): 0 Second Source (1s): 4"
  "First Source (5s): 1 Second Source (1s): 9"
  "First Source (5s): 2 Second Source (1s): 14"
  ...
*/
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Slower second source

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { withLatestFrom, map } from 'rxjs/operators';
import { interval } from 'rxjs';

//emit every 5s
const source = interval(5000);
//emit every 1s
const secondSource = interval(1000);
//withLatestFrom slower than source
const example = secondSource.pipe(
  //both sources must emit at least 1 value (5s) before emitting
  withLatestFrom(source),
  map(([first, second]) => {
    return `Source (1s): ${first} Latest From (5s): ${second}`;
  })
);
/*
  "Source (1s): 4 Latest From (5s): 0"
  "Source (1s): 5 Latest From (5s): 0"
  "Source (1s): 6 Latest From (5s): 0"
  ...
*/
const subscribe = example.subscribe(val => console.log(val));
```

Related Recipes

- [Progress Bar](#)
- [Game Loop](#)

Additional Resources

- [withLatestFrom](#) 📖 - Official docs
- [Combination operator: withLatestFrom](#) 🖥️💻 - André Staltz

📁 Source Code:

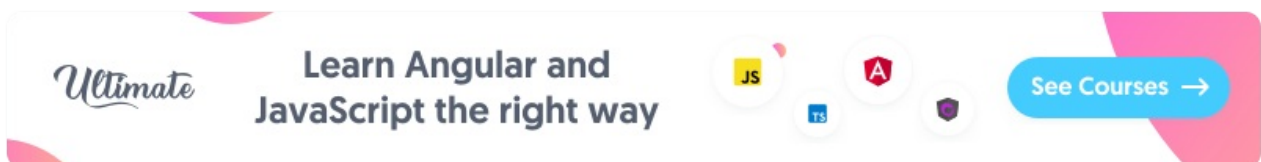
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/withLatestFrom.ts>

zip

signature: `zip(observables: *): Observable`

After all observables emit, emit values as an array

💡 Combined with [interval](#) or [timer](#), zip can be used to time output from another source!



Examples

Example 1: zip multiple observables emitting at alternate intervals

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { delay } from 'rxjs/operators';
import { of, zip } from 'rxjs';

const sourceOne = of('Hello');
const sourceTwo = of('World!');
const sourceThree = of('Goodbye');
const sourceFour = of('World!');
//wait until all observables have emitted a value then emit all
as an array
const example = zip(
  sourceOne,
  sourceTwo.pipe(delay(1000)),
  sourceThree.pipe(delay(2000)),
  sourceFour.pipe(delay(3000))
);
//output: ["Hello", "World!", "Goodbye", "World!"]
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: zip when 1 observable completes

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { take } from 'rxjs/operators';
import { interval, zip } from 'rxjs';

//emit every 1s
const source = interval(1000);
//when one observable completes no more values will be emitted
const example = zip(source, source.pipe(take(2)));
//output: [0,0]...[1,1]
const subscribe = example.subscribe(val => console.log(val));
```

Example 3: get X/Y coordinates of drag start/finish (mouse down/up)

([StackBlitz](#))


```
// RxJS v6+
import { fromEvent, zip } from 'rxjs';
import { map } from 'rxjs/operators';




const documentEvent = eventName => fromEvent(document, eventName)

  .pipe(map((e: MouseEvent) => ({x: e.clientX, y: e.clientY})));

zip(documentEvent('mousedown'), documentEvent('mouseup'))
  .subscribe(e => console.log(JSON.stringify(e)));
```



Additional Resources

- [zip](#)  - Official docs
- [Combination operator: zip](#)   - André Staltz

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/zip.ts>

Conditional Operators

For use-cases that depend on a specific condition to be met, these operators do the trick.

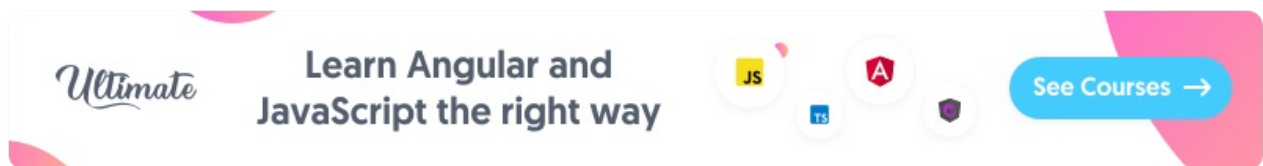
Contents

- [defaultIfEmpty](#)
- [every](#)
- [iif](#)

defaultIfEmpty

signature: `defaultIfEmpty(defaultValue: any): Observable`

Emit given value if nothing is emitted before completion.



Examples

Example 1: Default for empty value

([Stackblitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { defaultIfEmpty } from 'rxjs/operators';
import { of } from 'rxjs';

//emit 'Observable.of() Empty!' when empty, else any values from
source
const exampleOne = of().pipe(defaultIfEmpty('Observable.of() Empty!'));
//output: 'Observable.of() Empty!'
const subscribe = exampleOne.subscribe(val => console.log(val));
```

Example 2: Default for Observable.empty

([Stackblitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { defaultIfEmpty } from 'rxjs/operators';
import { empty } from 'rxjs';

//emit 'Observable.empty()!' when empty, else any values from source
const example = empty().pipe(defaultIfEmpty('Observable.empty()!'));
//output: 'Observable.empty()!'
const subscribe = example.subscribe(val => console.log(val));
```



Additional Resources

- [defaultIfEmpty](#)  - Official docs

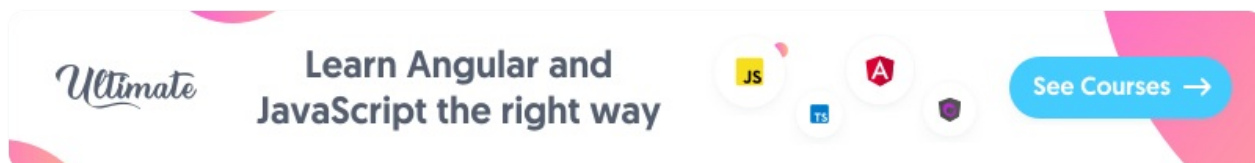
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/defaultIfEmpty.ts>

every

signature: `every(predicate: function, thisArg: any): Observable`

If all values pass predicate before completion emit true, else false.



Examples

Example 1: Some values false

([Stackblitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { every } from 'rxjs/operators';
import { of } from 'rxjs';

//emit 5 values
const source = of(1, 2, 3, 4, 5);
const example = source.pipe(
  //is every value even?
  every(val => val % 2 === 0)
);
//output: false
const subscribe = example.subscribe(val => console.log(val));
```


Example 2: All values true

([Stackblitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { every } from 'rxjs/operators';
import { of } from 'rxjs';

//emit 5 values
const allEvens = of(2, 4, 6, 8, 10);
const example = allEvens.pipe(
  //is every value even?
  every(val => val % 2 === 0)
);
//output: true
const subscribe = example.subscribe(val => console.log(val));
```

Additional Resources

- [every](#)  - Official docs

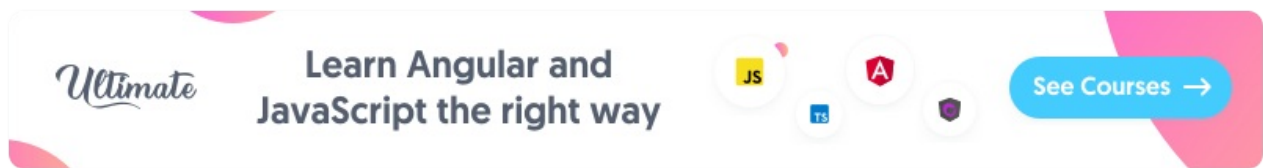
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/every.ts>

iif

signature: `iif<T, F>(condition: () => boolean, trueResult: SubscribableOrPromise<T> = EMPTY, falseResult: SubscribableOrPromise<F> = EMPTY): Observable<T | F>`

Decides at subscription time which Observable will actually be subscribed.



Examples

Example 1: simple iif

([Stackblitz](#))

```
// RxJS v6+
import { iif, of, interval } from 'rxjs';
import { mergeMap } from 'rxjs/operators';

const r$ = of('R');
const x$ = of('X');

interval(1000).pipe(
  mergeMap(v =>
    iif(
      () => v % 4 === 0,
      r$,
      x$
    )
  )
).subscribe(console.log);

//output: R, X, X, X, R, X, X, X, etc...
```

Example 2: iif with mouse moves

([Stackblitz](#))

```
// RxJS v6+
import { fromEvent, iif, of } from 'rxjs';
import { mergeMap, map, throttleTime, filter } from 'rxjs/operators';

const r$ = of(`I'm saying R!!`);
const x$ = of(`X's always win!!`);

fromEvent(document, 'mousemove').pipe(
  throttleTime(50),
  filter((move: MouseEvent) => move.clientY < 210),
  map((move: MouseEvent) => move.clientY),
  mergeMap(yCoord =>
    iif(
      () => yCoord < 110,
      r$,
      x$
    )
  )
).subscribe(console.log);
```

Additional Resources

- [iif](#)  - Official docs

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/observable/iif.ts>

Creation Operators

These operators allow the creation of an observable from nearly anything. From generic to specific use-cases you are free, and encouraged, to turn [everything into a stream](#).

Contents

- [create](#)
- [empty](#)
- [from](#) ★
- [fromEvent](#)
- [interval](#)
- [of](#) ★
- [range](#)
- [throw](#)
- [timer](#)

★ - *commonly used*

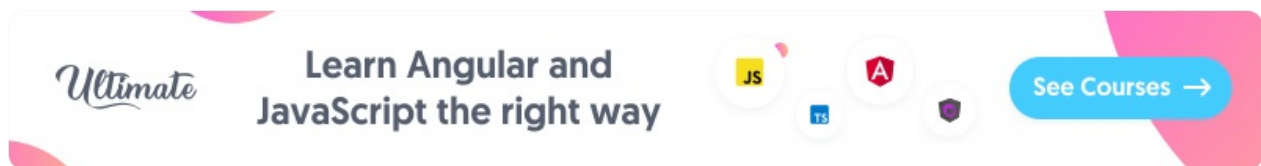
Additional Resources

- [Creating Observables From Scratch](#)   - André Staltz

create

signature: `create(subscribe: function)`

Create an observable with given subscription function.



Examples

Example 1: Observable that emits multiple values

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { Observable } from 'rxjs';
/*
  Create an observable that emits 'Hello' and 'World' on
  subscription.
*/
const hello = Observable.create(function(observer) {
  observer.next('Hello');
  observer.next('World');
});

//output: 'Hello'...'World'
const subscribe = hello.subscribe(val => console.log(val));
```

Example 2: Observable that emits even numbers on timer

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { Observable } from 'rxjs';

/*
  Increment value every 1s, emit even numbers.
*/
const evenNumbers = Observable.create(function(observer) {
  let value = 0;
  const interval = setInterval(() => {
    if (value % 2 === 0) {
      observer.next(value);
    }
    value++;
  }, 1000);

  return () => clearInterval(interval);
});
//output: 0...2...4...6...8
const subscribe = evenNumbers.subscribe(val => console.log(val));

//unsubscribe after 10 seconds
setTimeout(() => {
  subscribe.unsubscribe();
}, 10000);
```

Additional Resources

- [create](#) 📖 - Official docs
- [Creation operators: Create\(\)](#) 🖥️ 📄 - André Staltz
- [Using Observable.create for fine-grained control](#) 🖥️ 📄 - Shane Osbourne

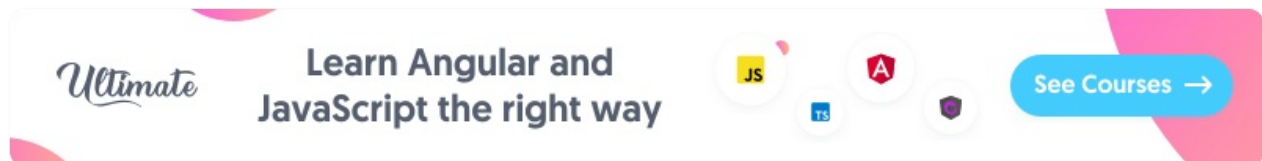
📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/observable/GenerateObservable.ts>

empty

signature: `empty(scheduler: Scheduler): Observable`

Observable that immediately completes.



Examples

Example 1: empty immediately completes

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { empty } from 'rxjs';

//output: 'Complete!'
const subscribe = empty().subscribe({
  next: () => console.log('Next'),
  complete: () => console.log('Complete!')
});
```

Example 2: empty with timer

([StackBlitz](#))




```
// RxJS v6+
import { interval, fromEvent, merge, empty } from 'rxjs';
import { switchMap, scan, takeWhile, startWith, mapTo } from 'rxjs/operators';

const countdownSeconds = 10;
const setHTML = id => val => (document.getElementById(id).innerHTML = val);
const pauseButton = document.getElementById('pause');
const resumeButton = document.getElementById('resume');
const interval$ = interval(1000).pipe(mapTo(-1));

const pause$ = fromEvent(pauseButton, 'click').pipe(mapTo(false));
const resume$ = fromEvent(resumeButton, 'click').pipe(mapTo(true));

const timer$ = merge(pause$, resume$)
  .pipe(
    startWith(true),
    // if timer is paused return empty observable
    switchMap(val => (val ? interval$ : empty())),
    scan((acc, curr) => (curr ? curr + acc : acc), countdownSeconds),
    takeWhile(v => v >= 0)
  )
  .subscribe(setHTML('remaining'));
```

Additional Resources

- [empty](#)  - Official docs
- [Creation operators: empty, never, and throw](#)   - André Staltz

 Source Code:

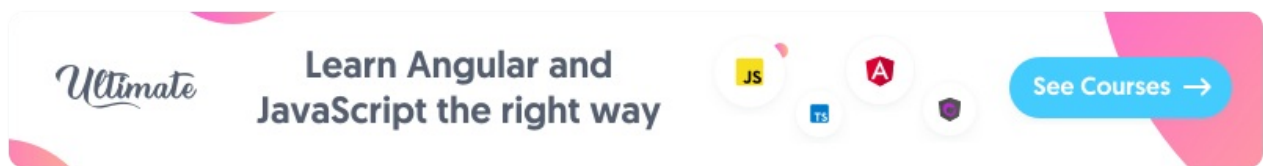
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/observable/EmptyObservable.ts>

from

signature: `from(ish: ObservableInput, mapFn: function, thisArg: any, scheduler: Scheduler): Observable`

Turn an array, promise, or iterable into an observable.

- 💡 This operator can be used to convert a promise to an observable!
 - 💡 For arrays and iterables, all contained values will be emitted as a sequence!
 - 💡 This operator can also be used to emit a string as a sequence of characters!
-



Examples

Example 1: Observable from array

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';

//emit array as a sequence of values
const arraySource = from([1, 2, 3, 4, 5]);
//output: 1,2,3,4,5
const subscribe = arraySource.subscribe(val => console.log(val));
```


Example 2: Observable from promise

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';

//emit result of promise
const promiseSource = from(new Promise(resolve => resolve('Hello
  World!')));
//output: 'Hello World'
const subscribe = promiseSource.subscribe(val => console.log(val)
);
```



Example 3: Observable from collection

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';

//works on js collections
const map = new Map();
map.set(1, 'Hi');
map.set(2, 'Bye');

const mapSource = from(map);
//output: [1, 'Hi'], [2, 'Bye']
const subscribe = mapSource.subscribe(val => console.log(val));
```

Example 4: Observable from string

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';

//emit string as a sequence
const source = from('Hello World');
//output: 'H','e','l','l','o',' ','W','o','r','l','d'
const subscribe = source.subscribe(val => console.log(val));
```

Related Recipes

- [Progress Bar](#)
- [HTTP Polling](#)

Additional Resources

- [from](#)  - Official docs
- [Creation operators: from, fromArray, fromPromise](#)   - André Staltz

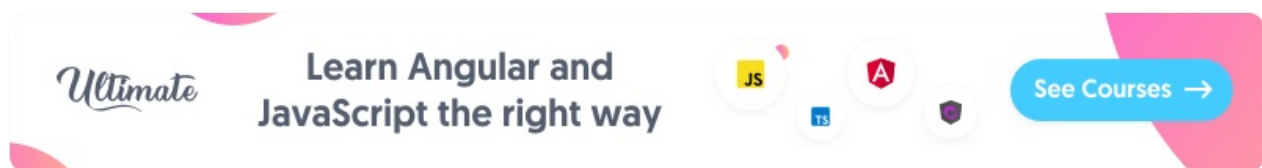
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/observable/from.ts>

fromEvent

signature: `fromEvent(target: EventTargetLike, eventName: string, selector: function): Observable`

Turn event into observable sequence.



Examples

Example 1: Observable from mouse clicks

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { fromEvent } from 'rxjs';
import { map } from 'rxjs/operators';

//create observable that emits click events
const source = fromEvent(document, 'click');
//map to string with given event timestamp
const example = source.pipe(map(event => `Event time: ${event.timeStamp}`));
//output (example): 'Event time: 7276.3900000000001'
const subscribe = example.subscribe(val => console.log(val));
```

Related Recipes

- [Smart Counter](#)
- [Progress Bar](#)
- [Game Loop](#)
- [HTTP Polling](#)

Additional Resources

- [fromEvent](#)  - Official docs



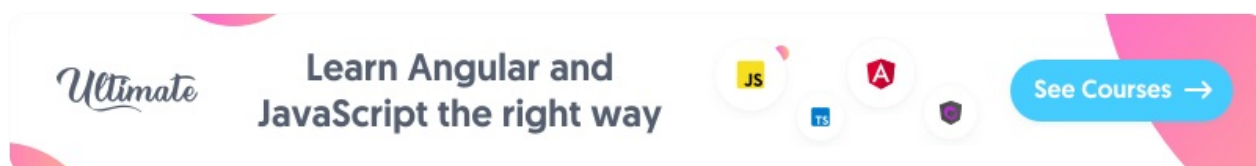
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/observable/FromEventObservable.ts>

interval

signature: `interval(period: number, scheduler: Scheduler): Observable`

Emit numbers in sequence based on provided timeframe.



Examples



Example 1: Emit sequence of values at 1 second interval

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';

//emit value in sequence every 1 second
const source = interval(1000);
//output: 0,1,2,3,4,5....
const subscribe = source.subscribe(val => console.log(val));
```

Additional Resources

- [interval](#)  - Official docs
- [Creation operators: interval and timer](#)   - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/observable/interval.ts>

of / just

signature: `of(...values, scheduler: Scheduler): Observable`

Emit variable amount of values in a sequence and then emits a complete notification.

Examples

Example 1: Emitting a sequence of numbers

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
//emits any number of provided values in sequence
const source = of(1, 2, 3, 4, 5);
//output: 1,2,3,4,5
const subscribe = source.subscribe(val => console.log(val));
```




Example 2: Emitting an object, array, and function

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
//emits values of any type
const source = of({ name: 'Brian' }, [1, 2, 3], function hello() {
  return 'Hello';
});
//output: {name: 'Brian'}, [1,2,3], function hello() { return 'Hello' }
const subscribe = source.subscribe(val => console.log(val));
```



Additional Resources

- [of](#)  - Official docs
- [Creation operators: of](#)   - André Staltz



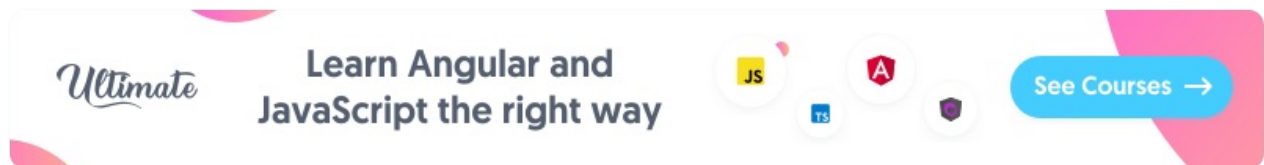
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/observable/of.ts>

range

signature: `range(start: number, count: number, scheduler: Scheduler): Observable`

Emit numbers in provided range in sequence.



Examples


Example 1: Emit range 1-10

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { range } from 'rxjs';

//emit 1-10 in sequence
const source = range(1, 10);
//output: 1,2,3,4,5,6,7,8,9,10
const example = source.subscribe(val => console.log(val));
```

Additional Resources

- [range](#)  - Official docs



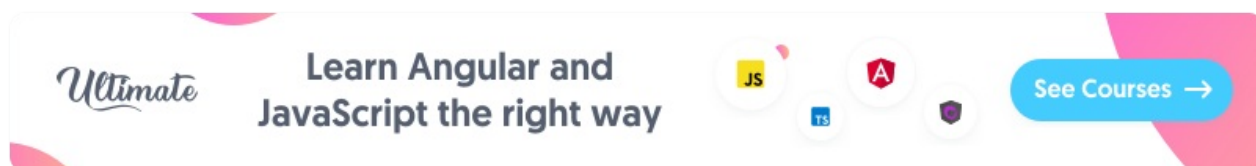
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/observable/range.ts>

throw

signature: `throw(error: any, scheduler: Scheduler): Observable`

Emit error on subscription.



Examples

Example 1: Throw error on subscription

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { throwError } from 'rxjs';

//emits an error with specified value on subscription
const source = throwError('This is an error!');
//output: 'Error: This is an error!'
const subscribe = source.subscribe({
  next: val => console.log(val),
  complete: () => console.log('Complete!'),
  error: val => console.log(`Error: ${val}`)
});
```

Related Examples

- [Throwing after 3 retries](#)

Additional Resources

- [throw](#) 📄 - Official docs
 - [Creation operators: empty, never, and throw](#) 🖥️ 📄 - André Staltz
-

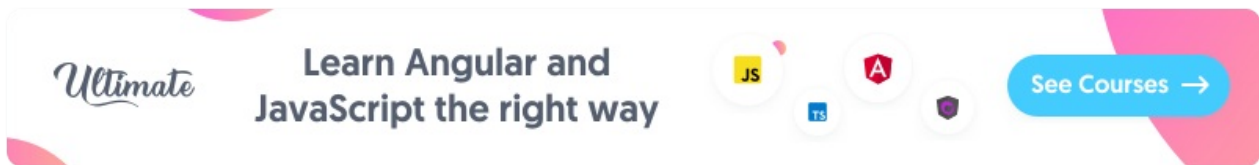
📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/observable/throwError.ts>

timer

signature: `timer(initialDelay: number | Date, period: number, scheduler: Scheduler): Observable`

After given duration, emit numbers in sequence every specified duration.



Examples

Example 1: timer emits 1 value then completes

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { timer } from 'rxjs';

//emit 0 after 1 second then complete, since no second argument
//is supplied
const source = timer(1000);
//output: 0
const subscribe = source.subscribe(val => console.log(val));
```

Example 2: timer emits after 1 second, then every 2 seconds

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { timer } from 'rxjs';

/*
  timer takes a second argument, how often to emit subsequent values
  in this case we will emit first value after 1 second and subsequent
  values every 2 seconds after
*/
const source = timer(1000, 2000);
//output: 0,1,2,3,4,5.....
const subscribe = source.subscribe(val => console.log(val));
```

Related Recipes

- [HTTP Polling](#)

Additional Resources

- [timer](#) 📖 - Official docs
- [Creation operators: interval and timer](#) 🖨️ 💡 - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/observable/timer.ts>

Error Handling Operators

Errors are an unfortunate side-effect of development. These operators provide effective ways to gracefully handle errors and retry logic, should they occur.

Contents

- [catch / catchError](#) ★
- [retry](#)
- [retryWhen](#)

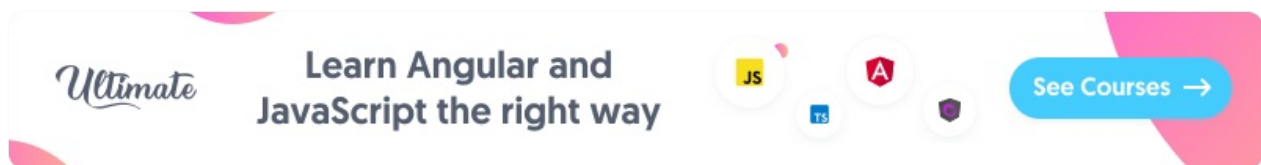
★ - *commonly used*

catch / catchError

signature: `catchError(project : function): Observable`

Gracefully handle errors in an observable sequence.

⚠ Remember to return an observable from the catchError function!



Examples

([example tests](#))

Example 1: Catching error from observable

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { throwError, of } from 'rxjs';
import { catchError } from 'rxjs/operators';
//emit error
const source = throwError('This is an error!');
//gracefully handle error, returning observable with error message
const example = source.pipe(catchError(val => of(`I caught: ${val}`)));
//output: 'I caught: This is an error'
const subscribe = example.subscribe(val => console.log(val));
```


Example 2: Catching rejected promise

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { timer, from, of } from 'rxjs';
import { mergeMap, catchError } from 'rxjs/operators';

//create promise that immediately rejects
const myBadPromise = () =>
  new Promise((resolve, reject) => reject('Rejected!'));
//emit single value after 1 second
const source = timer(1000);
//catch rejected promise, returning observable containing error
message
const example = source.pipe(
  mergeMap(_ =>
    from(myBadPromise()).pipe(catchError(error => of(`Bad Promise: ${error}`)))
  )
);
//output: 'Bad Promise: Rejected'
const subscribe = example.subscribe(val => console.log(val));
```

Additional Resources

- [Error handling operator: catch](#) 📺 💡 - André Staltz

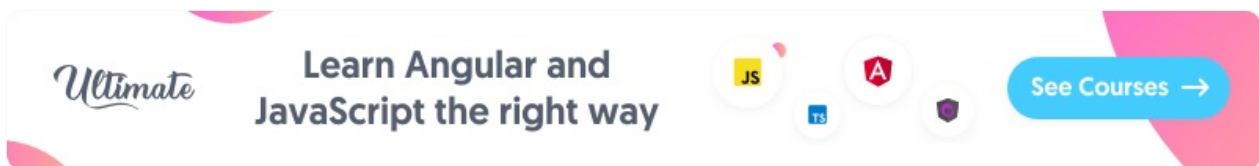
📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/catchError.ts>

retry

signature: `retry(number: number): Observable`

Retry an observable sequence a specific number of times should an error occur.



Examples

Example 1: Retry 2 times on error

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, of, throwError } from 'rxjs';
import { mergeMap, retry } from 'rxjs/operators';

//emit value every 1s
const source = interval(1000);
const example = source.pipe(
  mergeMap(val => {
    //throw error for demonstration
    if (val > 5) {
      return throwError('Error!');
    }
    return of(val);
  }),
  //retry 2 times on error
  retry(2)
);
/*
output:
0..1..2..3..4..5..
0..1..2..3..4..5..
0..1..2..3..4..5..
"Error!: Retried 2 times then quit!"
*/
const subscribe = example.subscribe({
  next: val => console.log(val),
  error: val => console.log(`${val}: Retried 2 times then quit!`)
});
```

Additional Resources

- [retry](#) 📖 - Official docs
- [Error handling operator: retry and retryWhen](#) 📖 💡 - André Staltz

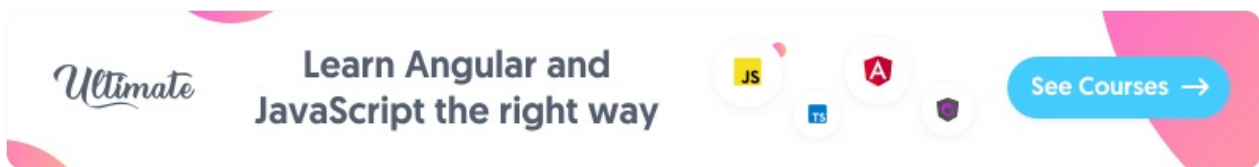
📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/retry.ts>

retryWhen

signature: `retryWhen(receives: (errors: Observable) => Observable, the: scheduler): Observable`

Retry an observable sequence on error based on custom criteria.



Examples

Example 1: Trigger retry after specified duration

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { timer, interval } from 'rxjs';
import { map, tap, retryWhen, delayWhen } from 'rxjs/operators';

//emit value every 1s
const source = interval(1000);
const example = source.pipe(
  map(val => {
    if (val > 5) {
      //error will be picked up by retryWhen
      throw val;
    }
    return val;
  }),
  retryWhen(errors =>
    errors.pipe(
      //log error message
      tap(val => console.log(`Value ${val} was too high!`)),
      //restart in 5 seconds
      delayWhen(val => timer(val * 1000))
    )
  )
);
/*
output:
0
1
2
3
4
5
"Value 6 was too high!"
--Wait 5 seconds then repeat
*/
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Customizable retry with increased duration

([StackBlitz](#))

```
import { Observable, _throw, timer } from 'rxjs';
import { mergeMap, finalize } from 'rxjs/operators';

export const genericRetryStrategy = ({
  maxRetryAttempts = 3,
  scalingDuration = 1000,
  excludedStatusCodes = []
}: {
  maxRetryAttempts?: number,
  scalingDuration?: number,
  excludedStatusCodes?: number[]
} = {}) => (attempts: Observable<any>) => {
  return attempts.pipe(
    mergeMap((error, i) => {
      const retryAttempt = i + 1;
      // if maximum number of retries have been met
      // or response is a status code we don't wish to retry, th
      row error
      if (
        retryAttempt > maxRetryAttempts ||
        excludedStatusCodes.find(e => e === error.status)
      ) {
        return _throw(error);
      }
      console.log(
        `Attempt ${retryAttempt}: retrying in ${retryAttempt *
          scalingDuration}ms`
      );
      // retry after 1s, 2s, etc...
      return timer(retryAttempt * scalingDuration);
    }),
    finalize(() => console.log('We are done!'))
  );
};
```

```
import { Component, OnInit } from '@angular/core';
import { catchError, retryWhen } from 'rxjs/operators';
import { of } from 'rxjs';
import { genericRetryStrategy } from './rxjs-utils';
import { AppService } from './app.service';



@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ]
})
export class AppComponent implements OnInit {
  constructor(private _appService: AppService) {}

  ngOnInit() {
    this._appService
      .getData(500)
      .pipe(
        retryWhen(genericRetryStrategy()),
        catchError(error => of(error))
      )
      .subscribe(console.log);

    // excluding status code, delay for logging clarity
    setTimeout(() => {
      this._appService
        .getData(500)
        .pipe(
          retryWhen(genericRetryStrategy({
            scalingDuration: 2000,
            excludedStatusCodes: [500]
          })),
          catchError(error => of(error))
        )
        .subscribe(e => console.log('Exluded code:', e.status));

    }, 8000);
  }
}
```


Additional Resources

- [retryWhen](#)  - Official docs
 - [Error handling operator: retry and retryWhen](#)   - André Staltz
-

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/retryWhen.ts>

Multicasting Operators

In RxJS observables are cold, or unicast by default. These operators can make an observable hot, or multicast, allowing side-effects to be shared among multiple subscribers.

Contents

- [publish](#)
- [multicast](#)
- [share](#) ★
- [shareReplay](#) ★

★ - *commonly used*

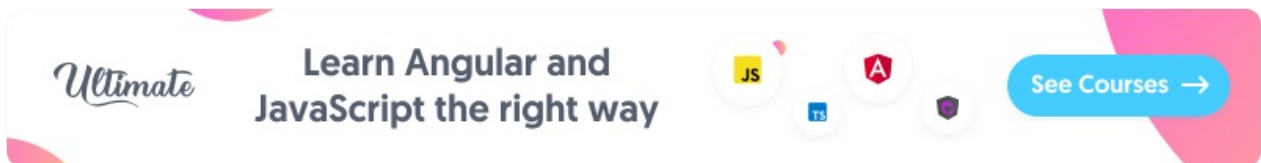
Additional Resources

- [Hot vs Cold Observables](#) 📄 - Ben Lesh
- [Unicast v Multicast](#) 📄 - GitHub Discussion
- [Demystifying Hot and Cold Observables](#) 📄 - André Staltz

publish

signature: `publish() : ConnectableObservable`

Share source and make hot by calling connect.



Examples

Example 1: Connect observable after subscribers

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { publish, tap } from 'rxjs/operators';

//emit value every 1 second
const source = interval(1000);
const example = source.pipe(
  //side effects will be executed once
  tap(_ => console.log('Do Something!')),
  //do nothing until connect() is called
  publish()
);

/*
  source will not emit values until connect() is called
  output: (after 5s)
  "Do Something!"
  "Subscriber One: 0"
  "Subscriber Two: 0"
  "Do Something!"
  "Subscriber One: 1"
  "Subscriber Two: 1"
*/
const subscribe = example.subscribe(val =>
  console.log(`Subscriber One: ${val}`)
);
const subscribeTwo = example.subscribe(val =>
  console.log(`Subscriber Two: ${val}`)
);

//call connect after 5 seconds, causing source to begin emitting
  items
setTimeout(() => {
  example.connect();
}, 5000);
```

Additional Resources

- [publish](#)  - Official docs



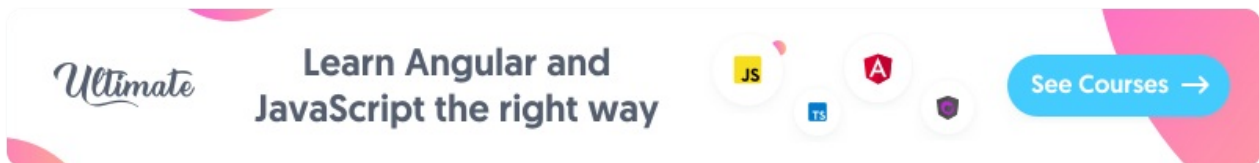
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/publish.ts>

multicast

signature: `multicast(selector: Function): Observable`

Share source utilizing the provided Subject.



Examples

Example 1: multicast with standard Subject

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { Subject, interval } from 'rxjs';
import { take, tap, multicast, mapTo } from 'rxjs/operators';

//emit every 2 seconds, take 5
const source = interval(2000).pipe(take(5));

const example = source.pipe(
  //since we are multicasting below, side effects will be executed once
  tap(() => console.log('Side Effect #1')),
  mapTo('Result!')
);

//subscribe subject to source upon connect()
const multi = example.pipe(multicast(() => new Subject()));
/*
  subscribers will share source
  output:
  "Side Effect #1"
  "Result!"
  "Result!"
  ...
*/
const subscriberOne = multi.subscribe(val => console.log(val));
const subscriberTwo = multi.subscribe(val => console.log(val));
//subscribe subject to source
multi.connect();
```

Example 2: multicast with ReplaySubject

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, ReplaySubject } from 'rxjs';
import { take, multicast, tap, mapTo } from 'rxjs/operators';

//emit every 2 seconds, take 5
const source = interval(2000).pipe(take(5));

//example with ReplaySubject
const example = source.pipe(
  //since we are multicasting below, side effects will be executed once
  tap(_ => console.log('Side Effect #2')),
  mapTo('Result Two!')
);
//can use any type of subject
const multi = example.pipe(multicast(() => new ReplaySubject(5)))
;
//subscribe subject to source
multi.connect();

setTimeout(() => {
  /*
    subscriber will receive all previous values on subscription
    because
    of ReplaySubject
    */
  const subscriber = multi.subscribe(val => console.group(val));
}, 5000);
```

Additional Resources

- [multicast](#)  - Official docs

 Source Code:

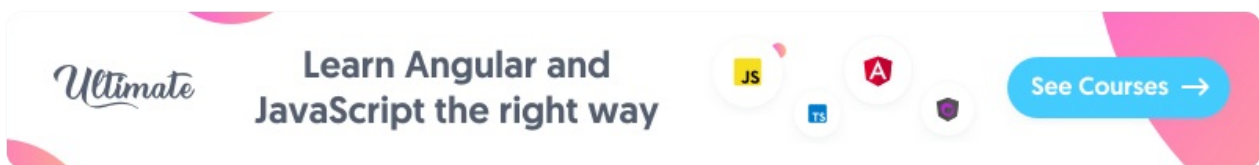
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/multicast.ts>

share

signature: `share(): Observable`

Share source among multiple subscribers.

💡 share is like [multicast](#) with a Subject and refCount!



Examples

Example 1: Multiple subscribers sharing source

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { timer } from 'rxjs';
import { tap, mapTo, share } from 'rxjs/operators';

//emit value in 1s
const source = timer(1000);
//log side effect, emit result
const example = source.pipe(
  tap(() => console.log('***SIDE EFFECT***')),
  mapTo('***RESULT***')
);




/*
  ***NOT SHARED, SIDE EFFECT WILL BE EXECUTED TWICE***
  output:
  ***SIDE EFFECT***
  ***RESULT***
  ***SIDE EFFECT***
  ***RESULT***
*/
const subscribe = example.subscribe(val => console.log(val));
const subscribeTwo = example.subscribe(val => console.log(val));

//share observable among subscribers
const sharedExample = example.pipe(share());
/*
  ***SHARED, SIDE EFFECT EXECUTED ONCE***
  output:
  ***SIDE EFFECT***
  ***RESULT***
  ***RESULT***
*/
const subscribeThree = sharedExample.subscribe(val => console.log(val));
const subscribeFour = sharedExample.subscribe(val => console.log(val));
```

Related Recipes

- [Progress Bar](#)
- [Game Loop](#)

Additional Resources

- [share](#)  - Official docs
- [Sharing streams with share](#)   - John Linquist



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/share.ts>

shareReplay

signature: `shareReplay(bufferSize?: number, windowTime?: number, scheduler?I IScheduler): Observable`

Share source and replay specified number of emissions on subscription.

Why use `shareReplay` ?

You generally want to use `shareReplay` when you have side-effects or taxing computations that you do not wish to be executed amongst multiple subscribers. It may also be valuable in situations where you know you will have late subscribers to a stream that need access to previously emitted values. This ability to *replay* values on subscription is what differentiates `share` and `shareReplay`.

For instance, suppose you have an observable that emits the last visited url. In the first example we are going to use `share` :

```
// simulate url change with subject
const routeEnd = new Subject<{data: any, url: string}>();

// grab url and share with subscribers
const lastUrl = routeEnd.pipe(
  pluck('url'),
  share()
);

// initial subscriber required
const initialSubscriber = lastUrl.subscribe(console.log);

// simulate route change
routeEnd.next({data: {}, url: 'my-path'});

// nothing logged
const lateSubscriber = lastUrl.subscribe(console.log);
```

In the above example nothing is logged as the `lateSubscriber` subscribes to the source. Now suppose instead we wanted to give access to the last emitted value on subscription, we can accomplish this with `shareReplay` :

```
import { Subject } from 'rxjs/Subject';
import { ReplaySubject } from 'rxjs/ReplaySubject';
import { pluck, share, shareReplay, tap } from 'rxjs/operators';

// simulate url change with subject
const routeEnd = new Subject<{data: any, url: string}>();

// grab url and share with subscribers
const lastUrl = routeEnd.pipe(
  tap(_ => console.log('executed')),
  pluck('url'),
  // defaults to all values so we set it to just keep and replay
  // last one
  shareReplay(1)
);

// requires initial subscription
const initialSubscriber = lastUrl.subscribe(console.log);

// simulate route change
// logged: 'executed', 'my-path'
routeEnd.next({data: {}, url: 'my-path'});

// logged: 'my-path'
const lateSubscriber = lastUrl.subscribe(console.log);
```

Note that this is similar behavior to what you would see if you subscribed a `ReplaySubject` to the `lastUrl` stream, then subscribed to that `Subject` :

```
// simulate url change with subject
const routeEnd = new Subject<{data: any, url: string}>();

// instead of using shareReplay, use ReplaySubject
const shareWithReplay = new ReplaySubject();

// grab url and share with subscribers
const lastUrl = routeEnd.pipe(
  pluck('url')
)
.subscribe(val => shareWithReplay.next(val));

// simulate route change
routeEnd.next({data: {}, url: 'my-path'});

// subscribe to ReplaySubject instead
// logged: 'my path'
shareWithReplay.subscribe(console.log);
```

In fact, if we dig into the source code we can see a very similar technique is being used. When a subscription is made, `shareReplay` will subscribe to the source, sending values through an internal `ReplaySubject` :

([source](#))


```





    return function shareReplayOperation(this: Subscriber<T>, source: Observable<T>) {
        refCount++;
        if (!subject || hasError) {
            hasError = false;
            subject = new ReplaySubject<T>(bufferSize, windowTime, scheduler);
            subscription = source.subscribe({
                next(value) { subject.next(value); },
                error(err) {
                    hasError = true;
                    subject.error(err);
                },
                complete() {
                    isComplete = true;
                    subject.complete();
                },
            });
        }

        const innerSub = subject.subscribe(this);

        return () => {
            refCount--;
            innerSub.unsubscribe();
            if (subscription && refCount === 0 && isComplete) {
                subscription.unsubscribe();
            }
        };
    };
}

```


Learn Angular and JavaScript the right way

[See Courses →](#)

Examples

Example 1: Multiple subscribers sharing source

([Stackblitz](#))

```
// RxJS v6+
import { Subject, ReplaySubject } from 'rxjs';
import { pluck, share, shareReplay, tap } from 'rxjs/operators';

// simulate url change with subject
const routeEnd = new Subject<{data: any, url: string}>();
// grab url and share with subscribers
const lastUrl = routeEnd.pipe(
  tap(_ => console.log('executed')),
  pluck('url'),
  // defaults to all values so we set it to just keep and replay
  last one
  shareReplay(1)
);
// requires initial subscription
const initialSubscriber = lastUrl.subscribe(console.log)
// simulate route change
// logged: 'executed', 'my-path'
routeEnd.next({data: {}, url: 'my-path'});
// logged: 'my-path'
const lateSubscriber = lastUrl.subscribe(console.log);
```

Additional Resources

- [shareReplay](#)  - Official docs

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/shareReplay.ts>

Filtering Operators

In a [push based approach](#), picking and choosing how and when to accept items is important. These operators provide techniques for accepting values from an observable source and dealing with [backpressure](#).

Contents

- [audit](#)
- [auditTime](#)
- [debounce](#)
- [debounceTime](#) ★
- [distinctUntilChanged](#) ★
- [filter](#) ★
- [first](#)
- [ignoreElements](#)
- [last](#)
- [sample](#)
- [single](#)
- [skip](#)
- [skipUntil](#)
- [skipWhile](#)
- [take](#) ★
- [takeUntil](#) ★
- [takeWhile](#)
- [throttle](#)
- [throttleTime](#)

★ - *commonly used*


audit

signature: `audit(durationSelector: (value) => Observable | Promise): Observable`

Ignore for time based on provided observable, then emit most recent value

[Examples Coming Soon!]

Additional Resources

- [audit](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/audit.ts>

auditTime

signature: `auditTime(duration: number, scheduler?: Scheduler): Observable`

Ignore for given time then emit most recent value

Why use `auditTime`

When you are interested in ignoring a source observable for a given amount of time, you can use `auditTime`. A possible use case is to only emit certain events (i.e. mouse clicks) at a maximum rate per second. After the specified duration has passed, the timer is disabled and the most recent source value is emitted on the output Observable, and this process repeats for the next source value.

💡 If you want the timer to reset whenever a new event occurs on the source observable, you can use [debounceTime](#)


Examples

Example 1: Emit clicks at a rate of at most one click per second

([stackBlitz](#))

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

// Create observable that emits click events
const source = fromEvent(document, 'click');
// Emit clicks at a rate of at most one click per second
const example = source.pipe(auditTime(1000))
// Output (example): '(1s) --- Clicked --- (1s) --- Clicked'
const subscribe = example.subscribe(val => console.log('Clicked'))
);
```



Additional Resources

- [auditTime](#)  - Official docs

 Source Code:

[https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/auditTime](https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/auditTime.ts)
.ts

debounce

signature: `debounce(durationSelector: function): Observable`

Discard emitted values that take less than the specified time, based on selector function, between output.

💡 Though not as widely used as [debounceTime](#), **debounce** is important when the debounce rate is variable!

Examples

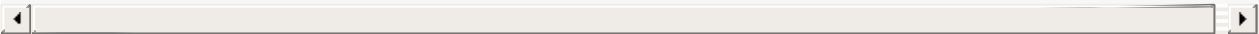
Example 1: Debounce on timer

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of, timer } from 'rxjs';
import { debounce } from 'rxjs/operators';

//emit four strings
const example = of('WAIT', 'ONE', 'SECOND', 'Last will display');

/*
    Only emit values after a second has passed between the last
    emission,
    throw away all other values
*/
const debouncedExample = example.pipe(debounce(() => timer(1000))
);
/*
    In this example, all values but the last will be omitted
    output: 'Last will display'
*/
const subscribe = debouncedExample.subscribe(val => console.log(
val));
```



Example 2: Debounce at increasing interval

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, timer } from 'rxjs';
import { debounce } from 'rxjs/operators';

//emit value every 1 second, ex. 0...1...2
const interval$ = interval(1000);
//raise the debounce time by 200ms each second
const debouncedInterval = interval$.pipe(debounce(val => timer(val * 200)));
/*
  After 5 seconds, debounce time will be greater than interval time,
  all future values will be thrown away
  output: 0...1...2...3...4.....(debounce time over 1s, no values emitted)
*/
const subscribe = debouncedInterval.subscribe(val =>
  console.log(`Example Two: ${val}`)
);
```

Additional Resources

- [debounce](#) 📖 - Official docs
- [Transformation operator: debounce and debounceTime](#) 📖 💡 - André Staltz

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/debounce.ts>

debounceTime

signature: `debounceTime(dueTime: number, scheduler: Scheduler): Observable`

Discard emitted values that take less than the specified time between output

💡 This operator is popular in scenarios such as type-ahead where the rate of user input must be controlled!

Examples

Example 1: Debouncing based on time between input

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { fromEvent, timer } from 'rxjs';
import { debounceTime, map } from 'rxjs/operators';

const input = document.getElementById('example');

//for every keyup, map to current input value
const example = fromEvent(input, 'keyup').pipe(map(i => i.currentTarget.value));

//wait .5s between keyups to emit current value
//throw away all other values
const debouncedInput = example.pipe(debounceTime(500));

//log values
const subscribe = debouncedInput.subscribe(val => {
  console.log(`Debounced Input: ${val}`);
});
```

Additional Resources

- [debounceTime](#) 📄 - Official docs
- [Transformation operator: debounce and debounceTime](#) 🖨️ 💡 - André Staltz

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/debounceTime.ts>

distinctUntilChanged

signature: `distinctUntilChanged(compare: function): Observable`

Only emit when the current value is different than the last.

💡 `distinctUntilChanged` uses `===` comparison by default, object references must match!

Examples

Example 1: `distinctUntilChanged` with basic values

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { distinctUntilChanged } from 'rxjs/operators';

//only output distinct values, based on the last emitted value
const myArrayWithDuplicatesInARow = from([1, 1, 2, 2, 3, 1, 2, 3]);

const distinctSub = myArrayWithDuplicatesInARow
  .pipe(distinctUntilChanged())
  //output: 1,2,3,1,2,3
  .subscribe(val => console.log('DISTINCT SUB:', val));

const nonDistinctSub = myArrayWithDuplicatesInARow
  //output: 1,1,2,2,3,1,2,3
  .subscribe(val => console.log('NON DISTINCT SUB:', val));
```




Example 2: distinctUntilChanged with objects

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { distinctUntilChanged } from 'rxjs/operators';

const sampleObject = { name: 'Test' };
//Objects must be same reference
const myArrayWithDuplicateObjects = from([
  sampleObject,
  sampleObject,
  sampleObject
]);
//only out distinct objects, based on last emitted value
const nonDistinctObjects = myArrayWithDuplicateObjects
  .pipe(distinctUntilChanged())
  //output: 'DISTINCT OBJECTS: {name: 'Test'}'
  .subscribe(val => console.log('DISTINCT OBJECTS:', val));
```

Additional Resources

- [distinctUntilChanged](#)  - Official docs
- [Filtering operator: distinct and distinctUntilChanged](#)   - André Staltz

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/distinctUntilChanged.ts>

filter

signature: `filter(select: Function, thisArg: any): Observable`

Emit values that pass the provided condition.

💡 If you would like to complete an observable when a condition fails, check out [takeWhile!](#)

Examples

Example 1: filter for even numbers

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { filter } from 'rxjs/operators';

//emit (1,2,3,4,5)
const source = from([1, 2, 3, 4, 5]);
//filter out non-even numbers
const example = source.pipe(filter(num => num % 2 === 0));
//output: "Even number: 2", "Even number: 4"
const subscribe = example.subscribe(val => console.log(`Even number: ${val}`));
```

Example 2: filter objects based on property

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { filter } from 'rxjs/operators';

//emit ({name: 'Joe', age: 31}, {name: 'Bob', age:25})
const source = from([{ name: 'Joe', age: 31 }, { name: 'Bob', age
: 25 }]);
//filter out people with age under 30
const example = source.pipe(filter(person => person.age >= 30));
//output: "Over 30: Joe"
const subscribe = example.subscribe(val => console.log(`Over 30:
${val.name}`));
```

Example 3: filter for number greater than specified value

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { filter } from 'rxjs/operators';

//emit every second
const source = interval(1000);
//filter out all values until interval is greater than 5
const example = source.pipe(filter(num => num > 5));
/*
  "Number greater than 5: 6"
  "Number greater than 5: 7"
  "Number greater than 5: 8"
  "Number greater than 5: 9"
*/
const subscribe = example.subscribe(val =>
  console.log(`Number greater than 5: ${val}`)
);
```

Related Recipes

- [HTTP Polling](#)

- [Game Loop](#)

Additional Resources

- [filter](#) 📄 - Official docs
- [Adding conditional logic with filter](#) 📄 💻 - John Linquist
- [Filtering operator: filter](#) 📄 💻 - André Staltz



Source Code:

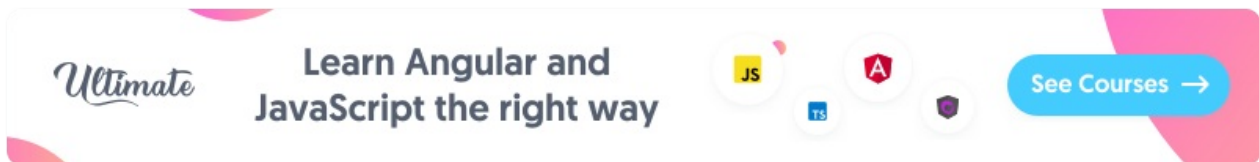
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/filter.ts>

first

signature: `first(predicate: function, select: function)`

Emit the first value or first to pass provided expression.

💡 The counterpart to first is [last!](#)



Examples

([example tests](#))

Example 1: First value from sequence

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { first } from 'rxjs/operators';

const source = from([1, 2, 3, 4, 5]);
//no arguments, emit first value
const example = source.pipe(first());
//output: "First value: 1"
const subscribe = example.subscribe(val => console.log(`First value: ${val}`));
```

Example 2: First value to pass predicate

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { first } from 'rxjs/operators';

const source = from([1, 2, 3, 4, 5]);
//emit first item to pass test
const example = source.pipe(first(num => num === 5));
//output: "First to pass test: 5"
const subscribe = example.subscribe(val =>
  console.log(`First to pass test: ${val}`)
);
```




Example 3: Utilizing default value

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { first } from 'rxjs/operators';

const source = from([1, 2, 3, 4, 5]);
//no value will pass, emit default
const example = source.pipe(first(val => val > 5, 'Nothing'));
//output: 'Nothing'
const subscribe = example.subscribe(val => console.log(val));
```

Additional Resources

- [first](#)  - Official docs
- [Filtering operator: take, first, skip](#)   - André Staltz

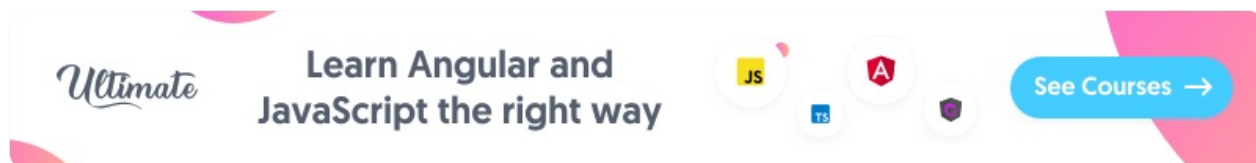
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/first.ts>

ignoreElements

signature: `ignoreElements(): Observable`

Ignore everything but complete and error.



Examples

Example 1: Ignore all elements from source

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { take, ignoreElements } from 'rxjs/operators';

//emit value every 100ms
const source = interval(100);
//ignore everything but complete
const example = source.pipe(
  take(5),
  ignoreElements()
);
//output: "COMPLETE!"
const subscribe = example.subscribe(
  val => console.log(`NEXT: ${val}`),
  val => console.log(`ERROR: ${val}`),
  () => console.log('COMPLETE!')
);
```

Example 2: Only displaying error

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, throwError, of } from 'rxjs';
import { mergeMap, ignoreElements } from 'rxjs/operators';

//emit value every 100ms
const source = interval(100);
//ignore everything but error
const error = source.pipe(
  mergeMap(val => {
    if (val === 4) {
      return throwError(`ERROR AT ${val}`);
    }
    return of(val);
  }),
  ignoreElements()
);
//output: "ERROR: ERROR AT 4"
const subscribe = error.subscribe(
  val => console.log(`NEXT: ${val}`),
  val => console.log(`ERROR: ${val}`),
  () => console.log('SECOND COMPLETE!')
);
```

Additional Resources

- [ignoreElements](#)  - Official docs

 Source Code:

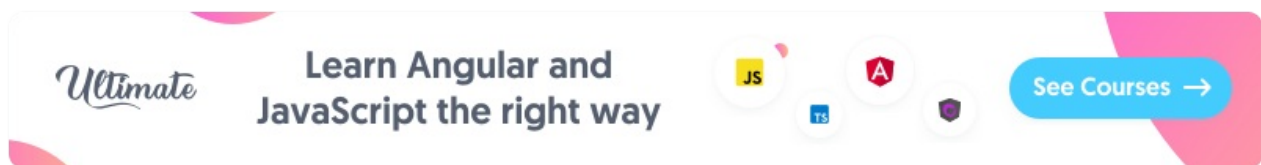
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/ignoreElements.ts>

last

signature: `last(predicate: function): Observable`

Emit the last value emitted from source on completion, based on provided expression.

💡 The counterpart to last is **first**!



Examples

Example 1: Last value in sequence

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { last } from 'rxjs/operators';

const source = from([1, 2, 3, 4, 5]);
//no arguments, emit last value
const example = source.pipe(last());
//output: "Last value: 5"
const subscribe = example.subscribe(val => console.log(`Last value: ${val}`));
```

Example 2: Last value to pass predicate

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { last } from 'rxjs/operators';

const source = from([1, 2, 3, 4, 5]);
//emit last even number
const exampleTwo = source.pipe(last(num => num % 2 === 0));
//output: "Last to pass test: 4"
const subscribeTwo = exampleTwo.subscribe(val =>
  console.log(`Last to pass test: ${val}`)
);
```

Example 3: Last with default value

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { last } from 'rxjs/operators';

const source = from([1, 2, 3, 4, 5]);
//no values will pass given predicate, emit default
const exampleTwo = source.pipe(last(v => v > 5, 'Nothing!'));
//output: 'Nothing!'
const subscribeTwo = exampleTwo.subscribe(val => console.log(val)
);
```

Additional Resources

- [last](#) 📖 - Official docs
- [Filtering operator: takeLast, last](#) 📖 💡 - André Staltz

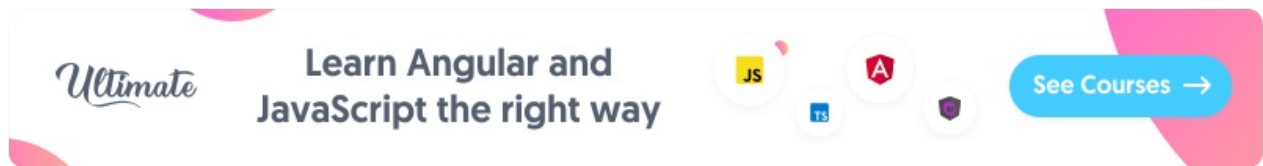
📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/last.ts>

sample

signature: `sample(sampler: Observable): Observable`

Sample from source when provided observable emits.



Examples

Example 1: Sample source every 2 seconds

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { sample } from 'rxjs/operators';

//emit value every 1s
const source = interval(1000);
//sample last emitted value from source every 2s
const example = source.pipe(sample(interval(2000)));
//output: 2..4..6..8..
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Sample source when interval emits

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))


```
// RxJS v6+
import { interval, zip, from } from 'rxjs';
import { sample } from 'rxjs/operators';

const source = zip(
  //emit 'Joe', 'Frank' and 'Bob' in sequence
  from(['Joe', 'Frank', 'Bob']),
  //emit value every 2s
  interval(2000)
);
//sample last emitted value from source every 2.5s
const example = source.pipe(sample(interval(2500)));
//output: ["Joe", 0]...["Frank", 1].....
const subscribe = example.subscribe(val => console.log(val));
```

Example 3: Distinguish between drag and click

From [Stack Overflow](#) By [Dorus](#)

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { fromEvent, merge } from 'rxjs';
import { sample, mapTo } from 'rxjs/operators';

const listener = merge(
  fromEvent(document, 'mousedown').pipe(mapTo(false)),
  fromEvent(document, 'mousemove').pipe(mapTo(true))
)
.pipe(sample(fromEvent(document, 'mouseup')))
.subscribe(isDragging => {
  console.log('Were you dragging?', isDragging);
});
```

Additional Resources

- [sample](#)  - Official docs



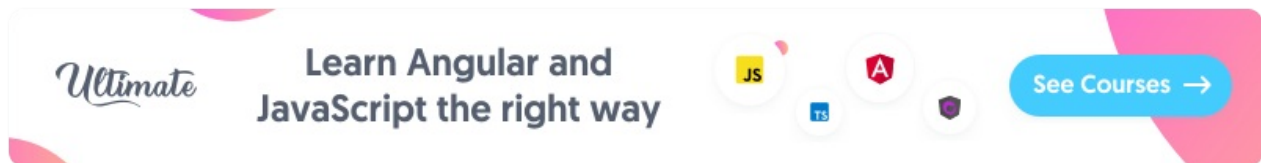
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/sample.ts>

single

signature: `single(a: Function): Observable`

Emit single item that passes expression.



Examples

Example 1: Emit first number passing predicate

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { single } from 'rxjs/operators';

//emit (1,2,3,4,5)
const source = from([1, 2, 3, 4, 5]);
//emit one item that matches predicate
const example = source.pipe(single(val => val === 4));
//output: 4
const subscribe = example.subscribe(val => console.log(val));
```

Additional Resources

- [single](#)  - Official docs

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/single.ts>

skip

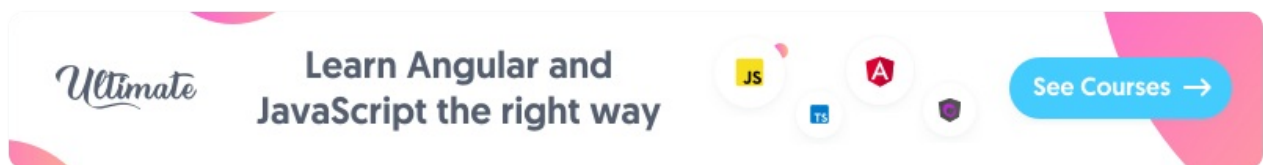
signature: `skip(the: Number): Observable`

Skip the provided number of emitted values.

Why use `skip` ?

Skip allows you to ignore the first x emissions from the source. Generally `skip` is used when you have an observable that always emits certain values on subscription that you wish to ignore. Perhaps those first few aren't needed or you are subscribing to a `Replay` or `BehaviorSubject` and do not need to act on the initial values. Reach for `skip` if you are only concerned about later emissions.

You could mimic `skip` by using `filter` with indexes. Ex. `.filter((val, index) => index > 1)`



Examples

Example 1: Skipping values before emission

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { skip } from 'rxjs/operators';

//emit every 1s
const source = interval(1000);
//skip the first 5 emitted values
const example = source.pipe(skip(5));
//output: 5...6...7...8.....
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Short hand for a specific filter use case

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { skip, filter } from 'rxjs/operators';




const numArrayObs = from([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

// 3,4,5...
const skipObs = numArrayObs.pipe(skip(2)).subscribe(console.log);

// 3,4,5...
const filterObs = numArrayObs
  .pipe(filter((val, index) => index > 1))
  .subscribe(console.log);

//Same output!
```

Additional Resources

- [skip](#)  - Official docs
- [Filtering operator: take, first, skip](#)   - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/skip.ts>

skipUntil

signature: `skipUntil(the: Observable): Observable`

Skip emitted values from source until provided observable emits.

Examples

Example 1: Skip until observable emits

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, timer } from 'rxjs';
import { skipUntil } from 'rxjs/operators';

//emit every 1s
const source = interval(1000);
//skip emitted values from source until inner observable emits (
6s)
const example = source.pipe(skipUntil(timer(6000)));
//output: 5...6...7...8.....
const subscribe = example.subscribe(val => console.log(val));
```

Additional Resources

- [skipUntil](#)  - Official docs

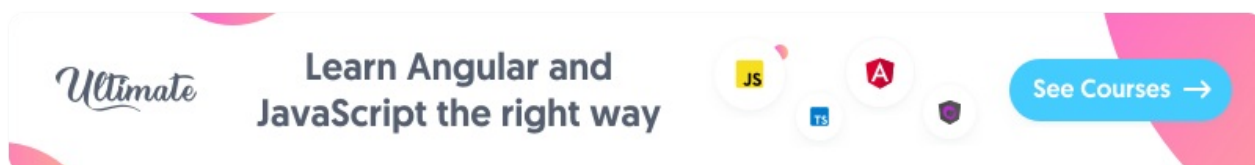
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/skipUntil.ts>

skipWhile

signature: `skipWhile(predicate: Function): Observable`

Skip emitted values from source until provided expression is false.



Examples

Example 1: Skip while values below threshold

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { skipWhile } from 'rxjs/operators';

//emit every 1s
const source = interval(1000);
//skip emitted values from source while value is less than 5
const example = source.pipe(skipWhile(val => val < 5));
//output: 5...6...7...8.....
const subscribe = example.subscribe(val => console.log(val));
```

Additional Resources

- [skipWhile](#)  - Official docs



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/skipWhile.ts>

take

signature: `take(count: number): Observable`

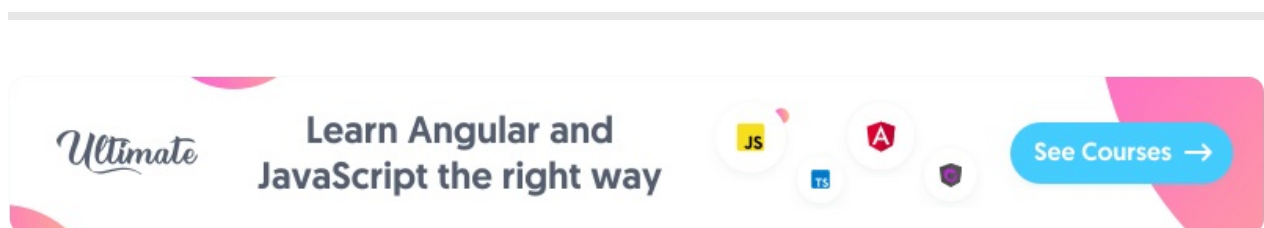
Emit provided number of values before completing.

Why use `take`

When you are interested in only the first set number of emission, you want to use `take`. Maybe you want to see what the user first clicked on when he/she first entered the page, you would want to subscribe to the click event and just take the first emission. There is a race and you want to observe the race, but you're only interested in the first who crosses the finish line. This operator is clear and straight forward, you just want to see the first n numbers of emission to do whatever it is you need.

💡 If you want to take a variable number of values based on some logic, or another observable, you can use [takeUntil](#) or [takeWhile](#)!

💡 `take` is the opposite of `skip` where `take` will take the first n number of emissions while `skip` will skip the first n number of emissions.



Examples

Example 1: Take 1 value from source

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
import { take } from 'rxjs/operators';

//emit 1,2,3,4,5
const source = of(1, 2, 3, 4, 5);
//take the first emitted value then complete
const example = source.pipe(take(1));
//output: 1
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Take the first 5 values from source

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';

//emit value every 1s
const interval$ = interval(1000);
//take the first 5 emitted values
const example = interval$.pipe(take(5));
//output: 0,1,2,3,4
const subscribe = example.subscribe(val => console.log(val));
```

Example 3: Taking first click location

([StackBlitz](#) | [jsFiddle](#))

```
<div id="locationDisplay">
  Where would you click first?
</div>
```

```
// RxJS v6+
import { fromEvent } from 'rxjs';
import { take, tap } from 'rxjs/operators';

const oneClickEvent = fromEvent(document, 'click').pipe(
  take(1),
  tap(v => {
    document.getElementById(
      'locationDisplay'
    ).innerHTML = `Your first click was on location ${v.screenX}:
    ${v.screenY}`;
  })
);

const subscribe = oneClickEvent.subscribe();
```

Additional Resources

- [take](#) 📄 - Official docs
- [Filtering operator: take, first, skip](#) 📺 💰 - André Staltz

📁 Source Code:

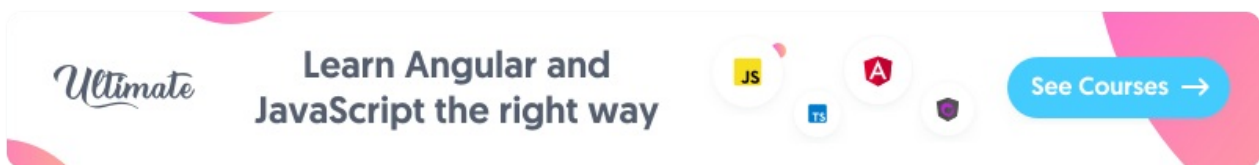
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/take.ts>

takeUntil

signature: `takeUntil(notifier: Observable): Observable`

Emit values until provided observable emits.

💡 If you only need a specific number of values, try [take](#)!



Examples

Example 1: Take values until timer emits

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, timer } from 'rxjs';
import { takeUntil } from 'rxjs/operators';

//emit value every 1s
const source = interval(1000);
//after 5 seconds, emit value
const timer$ = timer(5000);
//when timer emits after 5s, complete source
const example = source.pipe(takeUntil(timer$));
//output: 0,1,2,3
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Take the first 5 even numbers

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs/observable/interval';
import { takeUntil, filter, scan, map, withLatestFrom } from 'rxjs/operators';

//emit value every 1s
const source = interval(1000);
//is number even?
const isEven = val => val % 2 === 0;
//only allow values that are even
const evenSource = source.pipe(filter(isEven));
//keep a running total of the number of even numbers out
const evenNumberCount = evenSource.pipe(scan((acc, _) => acc + 1, 0));
//do not emit until 5 even numbers have been emitted
const fiveEvenNumbers = evenNumberCount.pipe(filter(val => val > 5));

const example = evenSource.pipe(
  //also give me the current even number count for display
  withLatestFrom(evenNumberCount),
  map(([val, count]) => `Even number (${count}) : ${val}`),
  //when five even numbers have been emitted, complete source observable
  takeUntil(fiveEvenNumbers)
);
/*
    Even number (1) : 0,
    Even number (2) : 2
    Even number (3) : 4
    Even number (4) : 6
    Even number (5) : 8
*/
const subscribe = example.subscribe(val => console.log(val));
```

Additional Resources

- [takeUntil](#)  - Official docs

- [Avoiding takeUntil leaks](#) - Angular in Depth
 - [Stopping a stream with takeUntil](#) 🖨️ 💡 - John Linquist
-

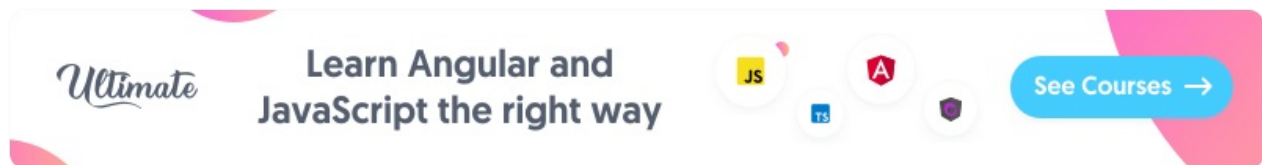
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/takeUntil.ts>

takeWhile

signature: `takeWhile(predicate: function(value, index): boolean): Observable`

Emit values until provided expression is false.



Examples

Example 1: Take values under limit

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
import { takeWhile } from 'rxjs/operators';

//emit 1,2,3,4,5
const source = of(1, 2, 3, 4, 5);
//allow values until value from source is greater than 4, then complete
const example = source.pipe(takeWhile(val => val <= 4));
//output: 1,2,3,4
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Difference between takeWhile() and filter()

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
import { takeWhile, filter } from 'rxjs/operators';

// emit 3, 3, 3, 9, 1, 4, 5, 8, 96, 3, 66, 3, 3, 3
const source = of(3, 3, 3, 9, 1, 4, 5, 8, 96, 3, 66, 3, 3, 3);

// allow values until value from source equals 3, then complete
// output: [3, 3, 3]
source
  .pipe(takeWhile(it => it === 3))
  .subscribe(val => console.log('takeWhile', val));

// output: [3, 3, 3, 3, 3, 3, 3]
source
  .pipe(filter(it => it === 3))
  .subscribe(val => console.log('filter', val));
```

Related Recipes

- [Smart Counter](#)

Additional Resources

- [takeWhile](#) 📖 - Official docs
- [Completing a stream with takeWhile](#) 📖 💡 - John Linquist

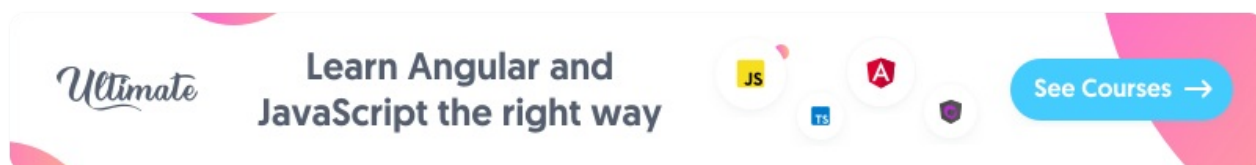
📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/takeWhile.ts>

throttle

signature: `throttle(durationSelector: function(value): Observable | Promise): Observable`

Emit value on the leading edge of an interval, but suppress new values until `durationSelector` has completed.



Examples

Example 1: Throttle for 2 seconds, based on second observable

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { throttle } from 'rxjs/operators';

//emit value every 1 second
const source = interval(1000);
//throttle for 2 seconds, emit latest value
const example = source.pipe(throttle(val => interval(2000)));
//output: 0...3...6...9
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Throttle with promise


([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { throttle, map } from 'rxjs/operators';

//emit value every 1 second
const source = interval(1000);
//incrementally increase the time to resolve based on source
const promise = val =>
  new Promise(resolve =>
    setTimeout(() => resolve(`Resolved: ${val}`), val * 100)
  );
//when promise resolves emit item from source
const example = source.pipe(
  throttle(promise),
  map(val => `Throttled off Promise: ${val}`)
);

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

Additional Resources

- [throttle](#)  - Official docs
- [Filtering operator: throttle and throttleTime](#)   - André Staltz

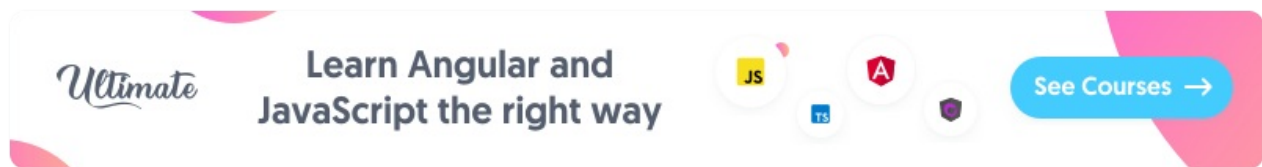
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/throttle.ts>

throttleTime

signature: `throttleTime(duration: number, scheduler: Scheduler): Observable`

Emit latest value when specified duration has passed.



Examples

Example 1: Receive latest value every 5 seconds

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { throttleTime } from 'rxjs/operators';

//emit value every 1 second
const source = interval(1000);
/*
  throttle for five seconds
  last value emitted before throttle ends will be emitted from s
  ource
*/
const example = source.pipe(throttleTime(5000));
//output: 0...6...12
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Throttle merged observable

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, merge } from 'rxjs';
import { throttleTime, ignoreElements } from 'rxjs/operators';

const source = merge(
  //emit every .75 seconds
  interval(750),
  //emit every 1 second
  interval(1000)
);
//throttle in middle of emitted values
const example = source.pipe(throttleTime(1200));
//output: 0...1...4...4...8...7
const subscribe = example.subscribe(val => console.log(val));
```

Additional Resources

- [throttleTime](#) 📖 - Official docs
- [Filtering operator: throttle and throttleTime](#) 📖 💡 - André Staltz

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/throttleTime.ts>

Transformation Operators

Transforming values as they pass through the operator chain is a common task. These operators provide transformation techniques for nearly any use-case you will encounter.

Contents

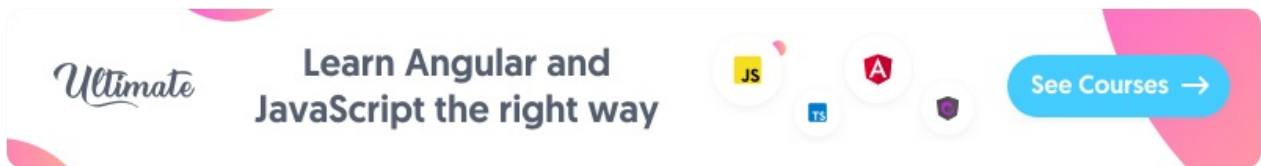
- [buffer](#)
- [bufferCount](#)
- [bufferTime](#) ★
- [bufferToggle](#)
- [bufferWhen](#)
- [concatMap](#) ★
- [concatMapTo](#)
- [exhaustMap](#)
- [expand](#)
- [groupBy](#)
- [map](#) ★
- [mapTo](#)
- [mergeMap / flatMap](#) ★
- [partition](#)
- [pluck](#)
- [reduce](#)
- [scan](#) ★
- [switchMap](#) ★
- [window](#)
- [windowCount](#)
- [windowTime](#)
- [windowToggle](#)
- [windowWhen](#)

★ - *commonly used*

buffer

signature: `buffer(closingNotifier: Observable): Observable`

Collect output values until provided observable emits, emit as array.



Examples

Example 1: Buffer until document click

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))




```
// RxJS v6+
import { interval, fromEvent } from 'rxjs';
import { buffer } from 'rxjs/operators';

//Create an observable that emits a value every second
const myInterval = interval(1000);
//Create an observable that emits every time document is clicked
const bufferBy = fromEvent(document, 'click');
/*
Collect all values emitted by our interval observable until we c
lick document. This will cause the bufferBy Observable to emit a
value, satisfying the buffer. Pass us all collected values sinc
e last buffer as an array.
*/
const myBufferedInterval = myInterval.pipe(buffer(bufferBy));
//Print values to console
//ex. output: [1,2,3] ... [4,5,6,7,8]
const subscribe = myBufferedInterval.subscribe(val =>
  console.log(' Buffered Values:', val)
);
```

Related Recipes

- [Game Loop](#)

Additional Resources

- [buffer](#)  - Official docs
- [Transformation operator: buffer](#)   - André Staltz

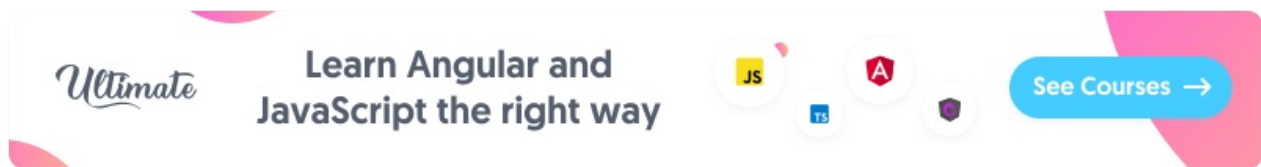
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/buffer.ts>

bufferCount

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

Collect emitted values until provided number is fulfilled, emit as array.



Examples

Example 1: Collect buffer and emit after specified number of values

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { bufferCount } from 'rxjs/operators';

//Create an observable that emits a value every second
const source = interval(1000);
//After three values are emitted, pass on as an array of buffered values
const bufferThree = source.pipe(bufferCount(3));
//Print values to console
//ex. output [0,1,2]...[3,4,5]
const subscribe = bufferThree.subscribe(val =>
  console.log('Buffered Values:', val)
);
```

Example 2: Overlapping buffers

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { bufferCount } from 'rxjs/operators';

//Create an observable that emits a value every second
const source = interval(1000);
/*
bufferCount also takes second argument, when to start the next buffer
for instance, if we have a bufferCount of 3 but second argument
(startBufferEvery) of 1:
1st interval value:
buffer 1: [0]
2nd interval value:
buffer 1: [0,1]
buffer 2: [1]
3rd interval value:
buffer 1: [0,1,2] Buffer of 3, emit buffer
buffer 2: [1,2]
buffer 3: [2]
4th interval value:
buffer 2: [1,2,3] Buffer of 3, emit buffer
buffer 3: [2, 3]
buffer 4: [3]
*/
const bufferEveryOne = source.pipe(bufferCount(3, 1));
//Print values to console
const subscribe = bufferEveryOne.subscribe(val =>
  console.log('Start Buffer Every 1:', val)
);
```

Additional Resources

- [bufferCount](#)  - Official docs

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/bufferCount.ts>

bufferTime

signature: `bufferTime(bufferTimeSpan: number, bufferCreationInterval: number, scheduler: Scheduler): Observable`

Collect emitted values until provided time has passed, emit as array.

Examples

Example 1: Buffer for 2 seconds

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { bufferTime } from 'rxjs/operators';

//Create an observable that emits a value every 500ms
const source = interval(500);
//After 2 seconds have passed, emit buffered values as an array
const example = source.pipe(bufferTime(2000));
//Print values to console
//ex. output [0,1,2]...[3,4,5,6]
const subscribe = example.subscribe(val =>
  console.log('Buffered with Time:', val)
);
```

Example 2: Multiple active buffers

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { bufferTime } from 'rxjs/operators';

//Create an observable that emits a value every 500ms
const source = interval(500);
/*
bufferTime also takes second argument, when to start the next bu
ffer (time in ms)
for instance, if we have a bufferTime of 2 seconds but second ar
gument (bufferCreationInterval) of 1 second:
ex. output: [0,1,2]...[1,2,3,4,5]...[3,4,5,6,7]
*/
const example = source.pipe(bufferTime(2000, 1000));
//Print values to console
const subscribe = example.subscribe(val =>
  console.log('Start Buffer Every 1s:', val)
);
```

Additional Resources

- [bufferTime](#)  - Official docs

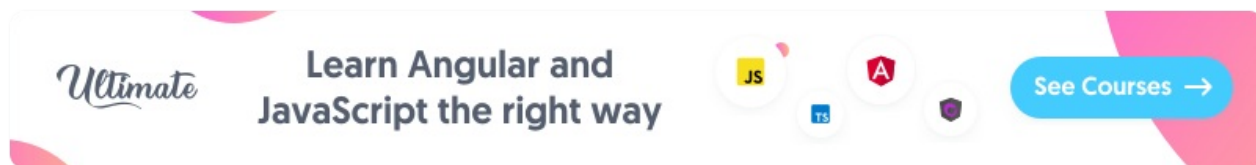
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/bufferTime.ts>

bufferToggle

signature: `bufferToggle(openings: Observable, closingSelector: Function): Observable`

Toggle on to catch emitted values from source, toggle off to emit buffered values as array.



Examples

Example 1: Toggle buffer on and off at interval

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { bufferToggle } from 'rxjs/operators';

//emit value every second
const sourceInterval = interval(1000);
//start first buffer after 5s, and every 5s after
const startInterval = interval(5000);
//emit value after 3s, closing corresponding buffer
const closingInterval = val => {
  console.log(`Value ${val} emitted, starting buffer! Closing in 3s!`);
  return interval(3000);
};
//every 5s a new buffer will start, collecting emitted values for 3s then emitting buffered values
const bufferToggleInterval = sourceInterval.pipe(
  bufferToggle(
    startInterval,
    closingInterval
  )
);
//log to console
//ex. emitted buffers [4,5,6]...[9,10,11]
const subscribe = bufferToggleInterval.subscribe(val =>
  console.log('Emitted Buffer:', val)
);
```

Example 2: Toggle buffer on and off on mouse down/up

([StackBlitz](#))

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

fromEvent(document, 'mousemove')
  .pipe(
    bufferToggle(
      fromEvent(document, 'mousedown'),
      _ => fromEvent(document, 'mouseup')
    )
  )
  .subscribe(console.log)
```

Additional Resources

- [bufferToggle](#)  - Official docs



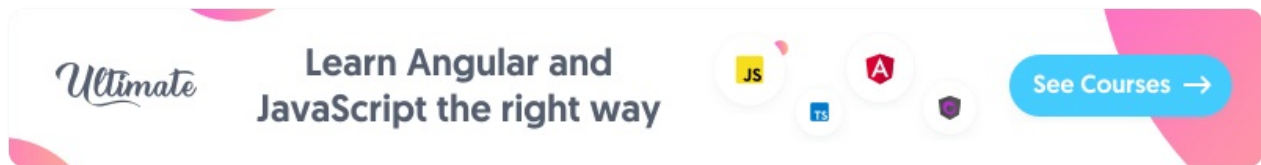
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/bufferToggle.ts>

bufferWhen

signature: `bufferWhen(closingSelector: function): Observable`

Collect all values until closing selector emits, emit buffered values.



Examples

Example 1: Emit buffer based on interval

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { bufferWhen } from 'rxjs/operators';

//emit value every 1 second
const oneSecondInterval = interval(1000);
//return an observable that emits value every 5 seconds
const fiveSecondInterval = () => interval(5000);
//every five seconds, emit buffered values
const bufferWhenExample = oneSecondInterval.pipe(bufferWhen(fiveSecondInterval));
//log values
//ex. output: [0,1,2,3]...[4,5,6,7,8]
const subscribe = bufferWhenExample.subscribe(val =>
  console.log('Emitted Buffer: ', val)
);
```

Additional Resources

- [bufferWhen](#)  - Official docs

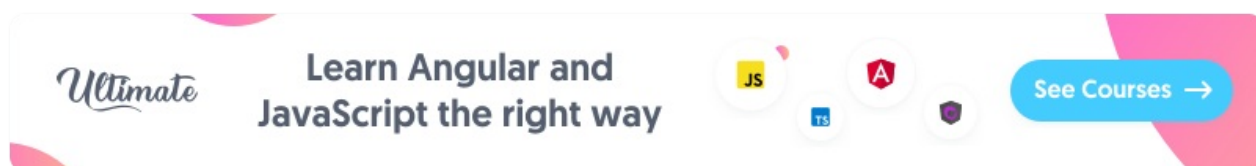
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/bufferWhen.ts>

concatMap

signature: `concatMap(project: function, resultSelector: function): Observable`

Map values to inner observable, subscribe and emit in order.



Examples

Example 1: Demonstrating the difference between `concatMap` and `mergeMap`

([StackBlitz](#))

💡 Note the difference between `concatMap` and `mergeMap`. Because `concatMap` does not subscribe to the next observable until the previous completes, the value from the source delayed by 2000ms will be emitted first. Contrast this with `mergeMap` which subscribes immediately to inner observables, the observable with the lesser delay (1000ms) will emit, followed by the observable which takes 2000ms to complete.

```
// RxJS v6+
import { of } from 'rxjs';
import { concatMap, delay, mergeMap } from 'rxjs/operators';

//emit delay value
const source = of(2000, 1000);
// map value from source into inner observable, when complete emit result and move to next
const example = source.pipe(
  concatMap(val => of(`Delayed by: ${val}ms`).pipe(delay(val)))
);
//output: With concatMap: Delayed by: 2000ms, With concatMap: Delayed by: 1000ms
const subscribe = example.subscribe(val =>
  console.log(`With concatMap: ${val}`)
);

// showing the difference between concatMap and mergeMap
const mergeMapExample = source
  .pipe(
    // just so we can log this after the first example has run
    delay(5000),
    mergeMap(val => of(`Delayed by: ${val}ms`).pipe(delay(val)))
  )
  .subscribe(val => console.log(`With mergeMap: ${val}`));
```

Example 2: Map to promise

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
import { concatMap } from 'rxjs/operators';

//emit 'Hello' and 'Goodbye'
const source = of('Hello', 'Goodbye');
//example with promise
const examplePromise = val => new Promise(resolve => resolve(`${
val} World!`));
// map value from source into inner observable, when complete em
it result and move to next
const example = source.pipe(concatMap(val => examplePromise(val))
);
//output: 'Example w/ Promise: 'Hello World', Example w/ Promise
: 'Goodbye World'
const subscribe = example.subscribe(val =>
  console.log('Example w/ Promise:', val)
);
```




Example 3: Supplying a projection function

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))


```
// RxJS v6+
import { of } from 'rxjs';
import { concatMap } from 'rxjs/operators';

//emit 'Hello' and 'Goodbye'
const source = of('Hello', 'Goodbye');
//example with promise
const examplePromise = val => new Promise(resolve => resolve(`${
val} World!`));
//result of first param passed to second param selector function
before being returned
const example = source.pipe(
  concatMap(val => examplePromise(val), result => `${result} w/
selector!`)
);
//output: 'Example w/ Selector: 'Hello w/ Selector', Example w/
Selector: 'Goodbye w/ Selector'
const subscribe = example.subscribe(val =>
  console.log('Example w/ Selector:', val)
);
```

Additional Resources

- [concatMap](#)  - Official docs
- [Use RxJS concatMap to map and concat higher order observables](#)   - André Staltz

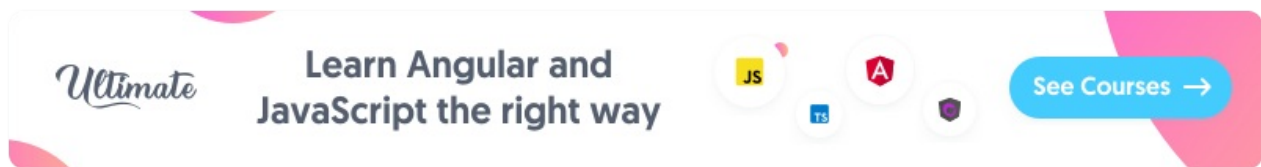
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/concatMap.ts>

concatMapTo

signature: `concatMapTo(observable: Observable, resultSelector: function): Observable`

Subscribe to provided observable when previous completes, emit values.



Examples

Example 1: Map to basic observable (simulating request)

([StackBlitz](#))

```
// RxJS v6+
import { of, interval } from 'rxjs';
import { concatMapTo, delay, take } from 'rxjs/operators';

//emit value every 2 seconds
const sampleInterval = interval(500).pipe(take(5));
const fakeRequest = of('Network request complete').pipe(delay(3000));
//wait for first to complete before next is subscribed
const example = sampleInterval.pipe(concatMapTo(fakeRequest));
//result
//output: Network request complete...3s...Network request complete'
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Using projection with `concatMap`

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { concatMapTo, take } from 'rxjs/operators';
//emit value every 2 seconds
const interval$ = interval(2000);
//emit value every second for 5 seconds
const source = interval(1000).pipe(take(5));
/*
    ***Be Careful***: In situations like this where the source emits
    at a faster pace
    than the inner observable completes, memory issues can arise.
    (interval emits every 1 second, basicTimer completes every 5)
*/
// basicTimer will complete after 5 seconds, emitting 0,1,2,3,4
const example = interval$.pipe(
    concatMapTo(
        source,
        (firstInterval, secondInterval) => `${firstInterval} ${secondInterval}`
    )
);
/*
    output: 0 0
           0 1
           0 2
           0 3
           0 4
           1 0
           1 1
           continued...

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

Additional Resources

- [concatMapTo](#)  - Official docs



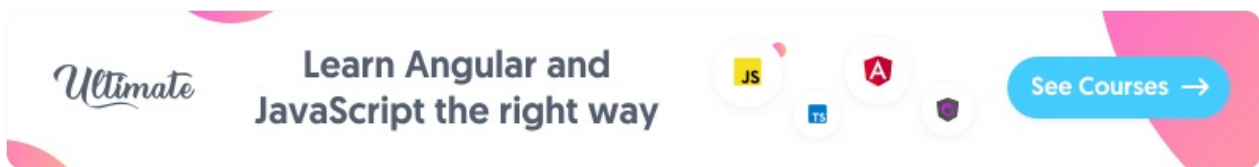
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/concatMapTo.ts>

exhaustMap

signature: `exhaustMap(project: function, resultSelector: function): Observable`

Map to inner observable, ignore other values until that observable completes.



Examples

Example 1: exhaustMap with interval

([Stackblitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, merge, of } from 'rxjs';
import { delay, take, exhaustMap } from 'rxjs/operators';

const sourceInterval = interval(1000);
const delayedInterval = sourceInterval.pipe(delay(10), take(4));

const exhaustSub = merge(
  // delay 10ms, then start interval emitting 4 values
  delayedInterval,
  // emit immediately
  of(true)
)
.pipe(exhaustMap(_ => sourceInterval.pipe(take(5))))
/*
    * The first emitted value (of(true)) will be mapped
    * to an interval observable emitting 1 value every
    * second, completing after 5.
    * Because the emissions from the delayed interval
    * fall while this observable is still active they will be ignored.
    *
    * Contrast this with concatMap which would queue,
    * switchMap which would switch to a new inner observable each emission,
    * and mergeMap which would maintain a new subscription for each emitted value.
    */
// output: 0, 1, 2, 3, 4
.subscribe(val => console.log(val));
```

Example 2: Another exhaustMap with interval

([Stackblitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { exhaustMap, tap, take } from 'rxjs/operators';

const firstInterval = interval(1000).pipe(take(10));
const secondInterval = interval(1000).pipe(take(2));

const exhaustSub = firstInterval
  .pipe(
    exhaustMap(f => {
      console.log(`Emission Corrected of first interval: ${f}`);
      return secondInterval;
    })
  )
  /*
```

When we subscribed to the first interval, it starts to emit a values (starting 0).

This value is mapped to the second interval which then begins to emit (starting 0).

While the second interval is active, values from the first interval are ignored.

We can see this when firstInterval emits number 3,6, and so on...

Output:

Emission of first interval: 0

0

1

Emission of first interval: 3

0

1

Emission of first interval: 6

0

1

Emission of first interval: 9

0

1

*/

```
.subscribe(s => console.log(s));
```

Outside Examples

`exhaustMap` for login effect in [@ngrx example app](#)

([Source](#))

```
@Effect()
login$ = this.actions$.pipe(
  ofType(AuthActionTypes.Login),
  map((action: Login) => action.payload),
  exhaustMap((auth: Authenticate) =>
    this.authService
      .login(auth)
      .pipe(
        map(user => new LoginSuccess({ user })),
        catchError(error => of(new LoginFailure(error)))
      )
  )
);
```

Additional Resources

- [exhaustMap](#)  - Official docs



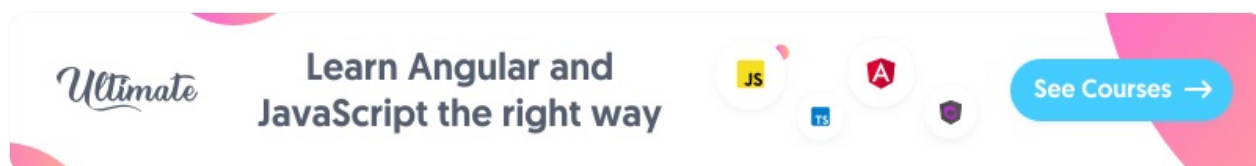
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/exhaustMap.ts>

expand

signature: `expand(project: function, concurrent: number, scheduler: Scheduler): Observable`

Recursively call provided function.



Examples

Example 1: Add one for each invocation

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, of } from 'rxjs';
import { expand, take } from 'rxjs/operators';

//emit 2
const source = of(2);
const example = source.pipe(
  //recursively call supplied function
  expand(val => {
    //2,3,4,5,6
    console.log(`Passed value: ${val}`);
    //3,4,5,6
    return of(1 + val);
  }),
  //call 5 times
  take(5)
);
/*
  "RESULT: 2"
  "Passed value: 2"
  "RESULT: 3"
  "Passed value: 3"
  "RESULT: 4"
  "Passed value: 4"
  "RESULT: 5"
  "Passed value: 5"
  "RESULT: 6"
  "Passed value: 6"
*/
//output: 2,3,4,5,6
const subscribe = example.subscribe(val => console.log(`RESULT:
${val}`));
```

Related Recipes

- [Game Loop](#)

Additional Resources

- [expand](#)  - Official docs

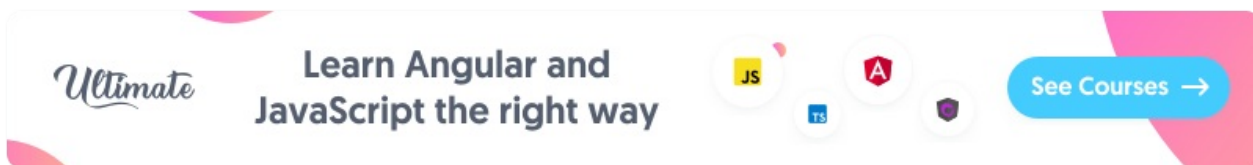
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/expand.ts>

groupBy

signature: `groupBy(keySelector: Function, elementSelector: Function): Observable`

Group into observables based on provided value.



Examples

Example 1: Group by property

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { groupBy, mergeMap, toArray } from 'rxjs/operators';

const people = [
  { name: 'Sue', age: 25 },
  { name: 'Joe', age: 30 },
  { name: 'Frank', age: 25 },
  { name: 'Sarah', age: 35 }
];
//emit each person
const source = from(people);
//group by age
const example = source.pipe(
  groupBy(person => person.age),
  // return each item in group as array
  mergeMap(group => group.pipe(toArray()))
);
/*
  output:
  [{age: 25, name: "Sue"},{age: 25, name: "Frank"}]
  [{age: 30, name: "Joe"}]
  [{age: 35, name: "Sarah"}]
*/
const subscribe = example.subscribe(val => console.log(val));
```

Additional Resources

- [groupBy](#) 📖 - Official docs
- [Group higher order observables with RxJS groupBy](#) 🖥️ 💡 - André Staltz
- [Use groupBy in real RxJS applications](#) 🖥️ 💡 - André Staltz

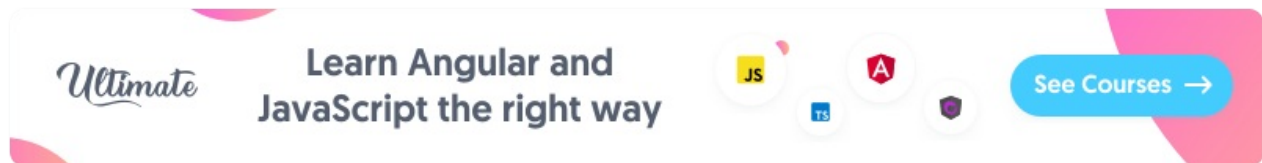
📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/groupBy.ts>

map

signature: `map(project: Function, thisArg: any): Observable`

Apply projection with each value from source.



Examples

Example 1: Add 10 to each number

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { map } from 'rxjs/operators';

//emit (1,2,3,4,5)
const source = from([1, 2, 3, 4, 5]);
//add 10 to each value
const example = source.pipe(map(val => val + 10));
//output: 11,12,13,14,15
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Map to single property

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { map } from 'rxjs/operators';

//emit ({name: 'Joe', age: 30}, {name: 'Frank', age: 20},{name:
'Ryan', age: 50})
const source = from([
  { name: 'Joe', age: 30 },
  { name: 'Frank', age: 20 },
  { name: 'Ryan', age: 50 }
]);
//grab each persons name, could also use pluck for this scenario
const example = source.pipe(map(({ name }) => name));
//output: "Joe","Frank","Ryan"
const subscribe = example.subscribe(val => console.log(val));
```

Related Recipes

- [Smart Counter](#)
- [Game Loop](#)
- [HTTP Polling](#)

Additional Resources

- [map](#) 📖 - Official docs
- [map vs flatMap](#) 📺 - Ben Lesh
- [Transformation operator: map and mapTo](#) 📺 💰 - André Staltz

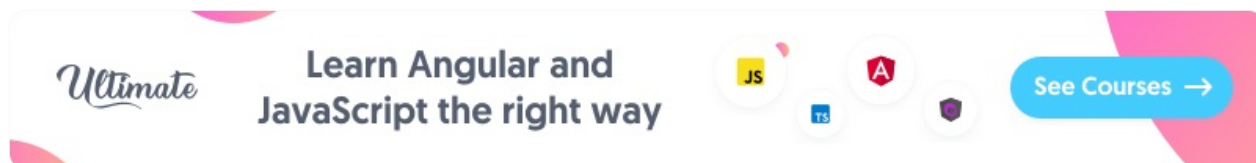
📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/map.ts>

mapTo

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

Map emissions to constant value.



Examples

Example 1: Map every emission to string

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { mapTo } from 'rxjs/operators';

//emit value every two seconds
const source = interval(2000);
//map all emissions to one value
const example = source.pipe(mapTo('HELLO WORLD!'));
//output: 'HELLO WORLD!'...'HELLO WORLD!'...'HELLO WORLD!'...
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Mapping clicks to string

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { fromEvent } from 'rxjs';
import { mapTo } from 'rxjs/operators';

//emit every click on document
const source = fromEvent(document, 'click');
//map all emissions to one value
const example = source.pipe(mapTo('GOODBYE WORLD!'));
//output: (click)'GOODBYE WORLD!'...
const subscribe = example.subscribe(val => console.log(val));
```

Related Recipes

- [HTTP Polling](#)
- [Smart Counter](#)

Additional Resources

- [mapTo](#) 📖 - Official docs
- [Changing behavior with mapTo](#) 🖥️ 💰 - John Linquist
- [Transformation operator: map and mapTo](#) 🖥️ 💰 - André Staltz

📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/mapTo.ts>

mergeMap

signature: `mergeMap(project: function: Observable, resultSelector: function: any, concurrent: number): Observable`

Map to observable, emit values.

💡 `flatMap` is an alias for `mergeMap`!

💡 If only one inner subscription should be active at a time, try `switchMap` !

💡 If the order of emission and subscription of inner observables is important, try `concatMap` !

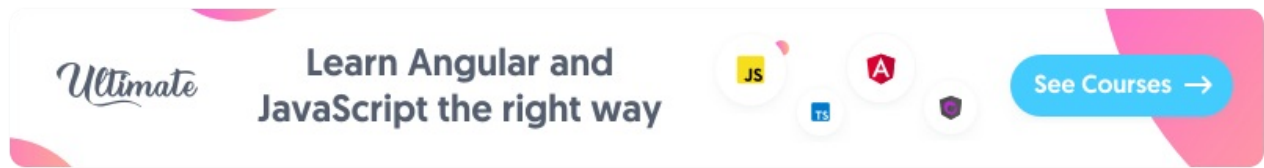
Why use `mergeMap` ?

This operator is best used when you wish to flatten an inner observable but want to manually control the number of inner subscriptions.

For instance, when using `switchMap` each inner subscription is completed when the source emits, allowing only one active inner subscription. In contrast, `mergeMap` allows for multiple inner subscriptions to be active at a time. Because of this, one of the most common use-case for `mergeMap` is requests that should not be canceled, think writes rather than reads. Note that if order must be maintained `concatMap` is a better option.

Be aware that because `mergeMap` maintains multiple active inner subscriptions at once it's possible to create a memory leak through long-lived inner subscriptions. A basic example would be if you were mapping to an observable with an inner timer, or a stream of dom events. In these cases, if you still wish to utilize `mergeMap` you may want to take advantage of another operator to

manage the completion of the inner subscription, think `take` or `takeUntil` . You can also limit the number of active inner subscriptions at a time with the `concurrent` parameter, seen in [example 4](#).



Examples

Example 1: mergeMap with observable

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
import { mergeMap } from 'rxjs/operators';

//emit 'Hello'
const source = of('Hello');
//map to inner observable and flatten
const example = source.pipe(mergeMap(val => of(`${val} World!`)))
;
//output: 'Hello World!'
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: mergeMap with promise

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
import { mergeMap } from 'rxjs/operators';

//emit 'Hello'
const source = of('Hello');
//mergeMap also emits result of promise
const myPromise = val =>
  new Promise(resolve => resolve(`${val} World From Promise!`));
//map to promise and emit result
const example = source.pipe(mergeMap(val => myPromise(val)));
//output: 'Hello World From Promise'
const subscribe = example.subscribe(val => console.log(val));
```

Example 3: mergeMap with resultSelector

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
import { mergeMap } from 'rxjs/operators';

/*
  you can also supply a second argument which receives the source
  value and emitted
  value of inner observable or promise
*/
//emit 'Hello'
const source = of('Hello');
//mergeMap also emits result of promise
const myPromise = val =>
  new Promise(resolve => resolve(`${val} World From Promise!`));
const example = source.pipe(
  mergeMap(
    val => myPromise(val),
    (valueFromSource, valueFromPromise) => {
      return `Source: ${valueFromSource}, Promise: ${valueFromPromise}`;
    }
  )
);
//output: "Source: Hello, Promise: Hello World From Promise!"
const subscribe = example.subscribe(val => console.log(val));
```

Example 4: mergeMap with concurrent value

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { mergeMap, take } from 'rxjs/operators';

//emit value every 1s
const source = interval(1000);

const example = source.pipe(
  mergeMap(
    //project
    val => interval(5000).pipe(take(2)),
    //resultSelector
    (oVal, iVal, oIndex, iIndex) => [oIndex, oVal, iIndex, iVal],





    //concurrent
    2
  )
);
/*
    Output:
    [0, 0, 0, 0] <--1st inner observable
    [1, 1, 0, 0] <--2nd inner observable
    [0, 0, 1, 1] <--1st inner observable
    [1, 1, 1, 1] <--2nd inner observable
    [2, 2, 0, 0] <--3rd inner observable
    [3, 3, 0, 0] <--4th inner observable
*/
const subscribe = example.subscribe(val => console.log(val));
```

Related Recipes

- [HTTP Polling](#)

Additional Resources

- [mergeMap](#) 📖 - Official docs
- [map vs flatMap](#) 📖 💡 - Ben Lesh
- [Async requests and responses in RxJS](#) 📖 💡 - André Staltz

- [Use RxJS mergeMap to map and merge higher order observables](#)   - André Staltz
 - [Use RxJS mergeMap for fine grain custom behavior](#)   - André Staltz
-

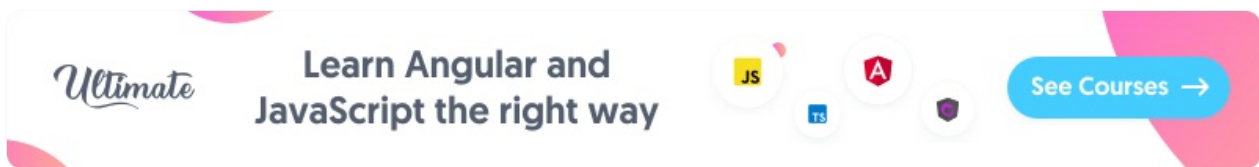
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/mergeMap.ts>

partition

signature: `partition(predicate: function: boolean, thisArg: any): [Observable, Observable]`

Split one observable into two based on provided predicate.



Examples

Example 1: Split even and odd numbers

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from, merge } from 'rxjs';
import { partition, map } from 'rxjs/operators';

const source = from([1, 2, 3, 4, 5, 6]);
//first value is true, second false
const [evens, odds] = source.pipe(partition(val => val % 2 === 0))
);
/*
  Output:
  "Even: 2"
  "Even: 4"
  "Even: 6"
  "Odd: 1"
  "Odd: 3"
  "Odd: 5"
*/
const subscribe = merge(
  evens.pipe(map(val => `Even: ${val}`)),
  odds.pipe(map(val => `Odd: ${val}`))
).subscribe(val => console.log(val));
```



Example 2: Split success and errors

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { merge, of, from } from 'rxjs';
import { map, partition, catchError } from 'rxjs/operators';

const source = from([1, 2, 3, 4, 5, 6]);
//if greater than 3 throw
const example = source.pipe(
  map(val => {
    if (val > 3) {
      throw `${val} greater than 3!`;
    }
    return { success: val };
  }),
  catchError(val => of({ error: val }))
);
//split on success or error
const [success, error] = example.pipe(partition(res => res.success));
/*
  Output:
  "Success! 1"
  "Success! 2"
  "Success! 3"
  "Error! 4 greater than 3!"
*/
const subscribe = merge(
  success.pipe(map(val => `Success! ${val.success}`)),
  error.pipe(map(val => `Error! ${val.error}`))
).subscribe(val => console.log(val));
```

Additional Resources

- [partition](#)  - Official docs



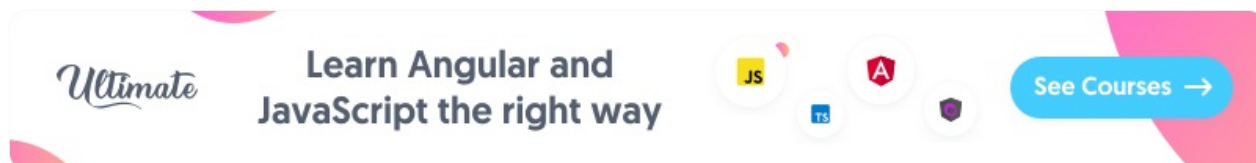
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/partition.ts>

pluck

signature: `pluck(properties: ...args): Observable`

Select properties to emit.



Examples

Example 1: Pluck object property

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { pluck } from 'rxjs/operators';

const source = from([ { name: 'Joe', age: 30 }, { name: 'Sarah',
age: 35 } ]);
//grab names
const example = source.pipe(pluck('name'));
//output: "Joe", "Sarah"
const subscribe = example.subscribe(val => console.log(val));
```


Example 2: Pluck nested properties

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from } from 'rxjs';
import { pluck } from 'rxjs/operators';

const source = from([
  { name: 'Joe', age: 30, job: { title: 'Developer', language: 'JavaScript' } },
  //will return undefined when no job is found
  { name: 'Sarah', age: 35 }
]);
//grab title property under job
const example = source.pipe(pluck('job', 'title'));
//output: "Developer" , undefined
const subscribe = example.subscribe(val => console.log(val));
```

Additional Resources

- [pluck](#)  - Official docs

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/pluck.ts>

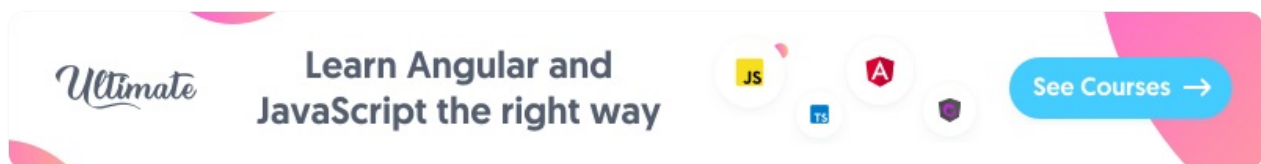
reduce

signature: `reduce(accumulator: function, seed: any): Observable`

Reduces the values from source observable to a single value that's emitted when the source completes.

💡 Just like `Array.prototype.reduce()`

💡 If you need the current accumulated value on each emission, try [scan](#)!



Examples

Example 1: Sum a stream of numbers

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
import { reduce } from 'rxjs/operators';

const source = of(1, 2, 3, 4);
const example = source.pipe(reduce((acc, val) => acc + val));
//output: Sum: 10'
const subscribe = example.subscribe(val => console.log('Sum:', val));
```

Additional Resources

- [reduce](#)  - Official docs

- [Scan\(\) vs reduce\(\) | RxJS TUTORIAL 📄](#) - Academind
-

 Source Code:

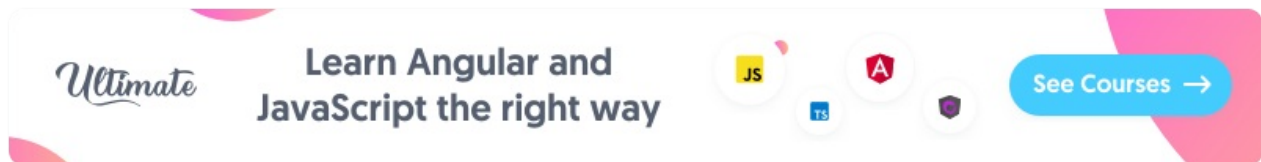
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/reduce.ts>

scan

signature: `scan(accumulator: function, seed: any): Observable`

Reduce over time.

💡 You can create [Redux](#)-like state management with scan!



Examples

Example 1: Sum over time

([StackBlitz](#))

```
// RxJS v6+
import { of } from 'rxjs';
import { scan } from 'rxjs/operators';

const source = of(1, 2, 3);
// basic scan example, sum over time starting with zero
const example = source.pipe(scan((acc, curr) => acc + curr, 0));
// log accumulated values
// output: 1,3,6
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Accumulating an object

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { Subject } from 'rxjs';
import { scan } from 'rxjs/operators';

const subject = new Subject();
//scan example building an object over time
const example = subject.pipe(
  scan((acc, curr) => Object.assign({}, acc, curr), {})
);
//log accumulated values
const subscribe = example.subscribe(val =>
  console.log('Accumulated object:', val)
);
//next values into subject, adding properties to object
// {name: 'Joe'}
subject.next({ name: 'Joe' });
// {name: 'Joe', age: 30}
subject.next({ age: 30 });
// {name: 'Joe', age: 30, favoriteLanguage: 'JavaScript'}
subject.next({ favoriteLanguage: 'JavaScript' });
```

Example 3: Emitting random values from the accumulated array.

([StackBlitz](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { scan, map, distinctUntilChanged } from 'rxjs/operators';

// Accumulate values in an array, emit random values from this array.
const scanObs = interval(1000)
  .pipe(
    scan((a, c) => [...a, c], []),
    map(r => r[Math.floor(Math.random() * r.length)]),
    distinctUntilChanged()
  )
  .subscribe(console.log);
```

Example 4: Accumulating http responses over time

([StackBlitz](#))

```
// RxJS v6+
import { interval, of } from 'rxjs';
import { scan, delay, repeat, mergeMap } from 'rxjs/operators';

const fakeRequest = of('response').pipe(delay(2000));







// output:
// ['response'],
// ['response', 'response'],
// ['response', 'response', 'response'],
// etc...

interval(1000)
  .pipe(
    mergeMap(_ => fakeRequest),
    scan<string>((allResponses, currentResponse) =>
      [...allResponses, currentResponse], []),
  )
  .subscribe(console.log);
```

Related Recipes

- [Smart Counter](#)
- [Progress Bar](#)

Additional Resources

- [scan](#)  - Official docs
- [Aggregating streams with reduce and scan using RxJS](#)  - Ben Lesh
- [Updating data with scan](#)   - John Linquist
- [Transformation operator: scan](#)   - André Staltz



Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/scan.ts>

switchMap

signature: `switchMap(project: function: Observable, resultSelector: function(outerValue, innerValue, outerIndex, innerIndex): any): Observable`

Map to observable, complete previous inner observable, emit values.

💡 If you would like more than one inner subscription to be maintained, try `mergeMap` !

💡 This operator is generally considered a safer default to `mergeMap` !

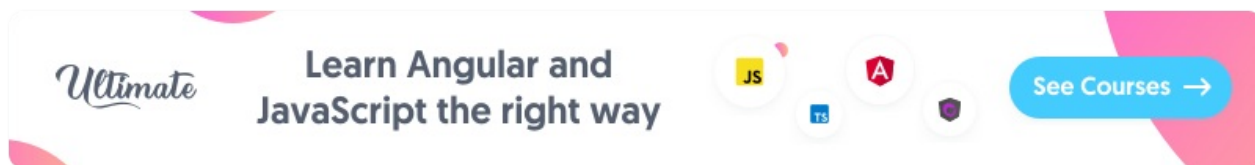
💡 This operator can cancel in-flight network requests!

Why use `switchMap` ?

The main difference between `switchMap` and other flattening operators is the cancelling effect. On each emission the previous inner observable (the result of the function you supplied) is cancelled and the new observable is subscribed. You can remember this by the phrase **switch to a new observable**.

This works perfectly for scenarios like `typeaheads` where you are no longer concerned with the response of the previous request when a new input arrives. This also is a safe option in situations where a long lived inner observable could cause memory leaks, for instance if you used `mergeMap` with an interval and forgot to properly dispose of inner subscriptions. Remember, `switchMap` maintains only one inner subscription at a time, this can be seen clearly in the [first example](#).

Be careful though, you probably want to avoid `switchMap` in scenarios where every request needs to complete, think writes to a database. `switchMap` could cancel a request if the source emits quickly enough. In these scenarios `mergeMap` is the correct option.



Examples

Example 1: Restart interval every 5 seconds

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { timer, interval } from 'rxjs';
import { switchMap } from 'rxjs/operators';

//emit immediately, then every 5s
const source = timer(0, 5000);
//switch to new inner observable when source emits, emit items that are emitted
const example = source.pipe(switchMap(() => interval(500)));
//output: 0,1,2,3,4,5,6,7,8,9...0,1,2,3,4,5,6,7,8
const subscribe = example.subscribe(val => console.log(val));
```

Example 2: Reset on every click

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, fromEvent } from 'rxjs';
import { switchMap, mapTo } from 'rxjs/operators';

//emit every click
const source = fromEvent(document, 'click');
//if another click comes within 3s, message will not be emitted
const example = source.pipe(
  switchMap(val => interval(3000).pipe(mapTo('Hello, I made it!'))
);
// (click)...3s...'Hello I made it!'...(click)...2s(click)...
const subscribe = example.subscribe(val => console.log(val));
```

Example 3: Using a `resultSelector` function

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { timer, interval } from 'rxjs';
import { switchMap } from 'rxjs/operators';

//emit immediately, then every 5s
const source = timer(0, 5000);
//switch to new inner observable when source emits, invoke project function and emit values
const example = source.pipe(
  switchMap(
    _ => interval(2000),
    (outerValue, innerValue, outerIndex, innerIndex) => ({
      outerValue,
      innerValue,
      outerIndex,
      innerIndex
    })
  )
);
/*
  Output:
  {outerValue: 0, innerValue: 0, outerIndex: 0, innerIndex: 0}
  {outerValue: 0, innerValue: 1, outerIndex: 0, innerIndex: 1}
  {outerValue: 1, innerValue: 0, outerIndex: 1, innerIndex: 0}
  {outerValue: 1, innerValue: 1, outerIndex: 1, innerIndex: 1}
*/
const subscribe = example.subscribe(val => console.log(val));
```

Example 4: Countdown timer with switchMap

([StackBlitz](#))


```
// RxJS v6+
import { interval, fromEvent, merge, empty } from 'rxjs';
import { switchMap, scan, takeWhile, startWith, mapTo } from 'rxjs/operators';

const countdownSeconds = 10;
const setHTML = id => val => (document.getElementById(id).innerHTML = val);
const pauseButton = document.getElementById('pause');
const resumeButton = document.getElementById('resume');
const interval$ = interval(1000).pipe(mapTo(-1));

const pause$ = fromEvent(pauseButton, 'click').pipe(mapTo(false));
const resume$ = fromEvent(resumeButton, 'click').pipe(mapTo(true));

const timer$ = merge(pause$, resume$)
  .pipe(
    startWith(true),
    switchMap(val => (val ? interval$ : empty())),
    scan((acc, curr) => (curr ? curr + acc : acc), countdownSeconds),
    takeWhile(v => v >= 0)
  )
  .subscribe(setHTML('remaining'));
```

HTML

```
<h4>
Time remaining: <span id="remaining"></span>
</h4>
<button id="pause">
Pause Timer
</button>
<button id="resume">
Resume Timer
</button>
```

Related Recipes

- [Smart Counter](#)
- [Progress Bar](#)
- [HTTP Polling](#)

Additional Resources

- [switchMap](#)  - Official docs
- [Avoiding switchMap-Related Bugs](#) - Nicholas Jamieson
- [Starting a stream with switchMap](#)   - John Linquist
- [Use RxJS switchMap to map and flatten higher order observables](#)   - André Staltz
- [Use switchMap as a safe default to flatten observables in RxJS](#)   - André Staltz

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/switchMap.ts>

window

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

Observable of values for window of time.

Examples




Example 1: Open window specified by inner observable

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { timer, interval } from 'rxjs';
import { window, scan, mergeAll } from 'rxjs/operators';

//emit immediately then every 1s
const source = timer(0, 1000);
const example = source.pipe(window(interval(3000)));
const count = example.pipe(scan((acc, curr) => acc + 1, 0));
/*
  "Window 1:"
  0
  1
  2
  "Window 2:"
  3
  4
  5
  ...
*/
const subscribe = count.subscribe(val => console.log(`Window ${val}:`));
const subscribeTwo = example
  .pipe(mergeAll())
  .subscribe(val => console.log(val));
```

Additional Resources

- [window](#)  - Official docs
 - [Split an RxJS observable with window](#)   - André Staltz
-



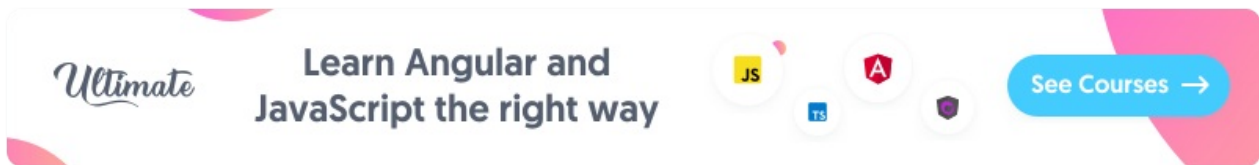
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/window.ts>

windowCount

signature: `windowCount(windowSize: number, startWindowEvery: number): Observable`

Observable of values from source, emitted each time provided count is fulfilled.



Examples

Example 1: Start new window every x items emitted

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval } from 'rxjs';
import { windowCount, mergeAll, tap } from 'rxjs/operators';

//emit every 1s
const source = interval(1000);
const example = source.pipe(
  //start new window every 4 emitted values
  windowCount(4),
  tap(_ => console.log('NEW WINDOW!'))
);

const subscribeTwo = example
  .pipe(
    //window emits nested observable
    mergeAll()
    /*
        output:
        "NEW WINDOW!"
        0
        1
        2
        3
        "NEW WINDOW!"
        4
        5
        6
        7
    */
  )
  .subscribe(val => console.log(val));
```

Additional Resources

- [windowCount](#)  - Official docs

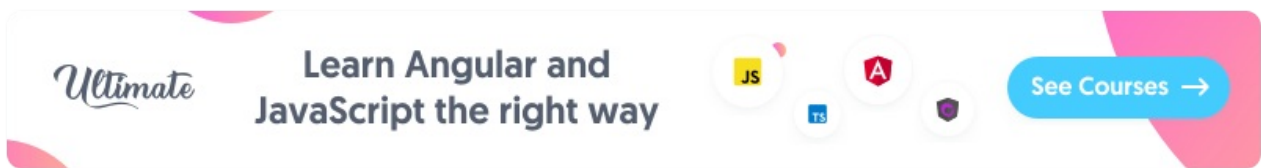
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/windowCount.ts>

windowTime

signature: `windowTime(windowTimeSpan: number, windowCreationInterval: number, scheduler: Scheduler): Observable`

Observable of values collected from source for each provided time span.



Examples

Example 1: Open new window every specified duration

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))


```
// RxJS v6+
import { timer } from 'rxjs';
import { windowTime, tap, mergeAll } from 'rxjs/operators';

//emit immediately then every 1s
const source = timer(0, 1000);
const example = source.pipe(
  //start new window every 3s
  windowTime(3000),
  tap(_ => console.log('NEW WINDOW!'))
);

const subscribeTwo = example
  .pipe(
    //window emits nested observable
    mergeAll()
    /*
      output:
      "NEW WINDOW!"
      0
      1
      2
      "NEW WINDOW!"
      3
      4
      5
    */
  )
  .subscribe(val => console.log(val));
```

Additional Resources

- [windowTime](#)  - Official docs

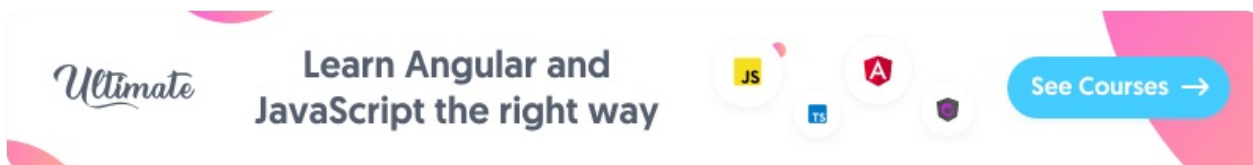
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/windowTime.ts>

windowToggle

signature: `windowToggle(openings: Observable,
closingSelector: function(value): Observable): Observable`

Collect and emit observable of values from source between opening and closing emission.



Examples

Example 1: Toggle window at increasing interval


([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { timer, interval } from 'rxjs';
import { tap, windowToggle, mergeAll } from 'rxjs/operators';

//emit immediately then every 1s
const source = timer(0, 1000);
//toggle window on every 5
const toggle = interval(5000);
const example = source.pipe(
  //turn window on every 5s
  windowToggle(toggle, val => interval(val * 1000)),
  tap(_ => console.log('NEW WINDOW!'))
);

const subscribeTwo = example
  .pipe(
    //window emits nested observable
    mergeAll()
    /*
      output:
      "NEW WINDOW!"
      5
      "NEW WINDOW!"
      10
      11
      "NEW WINDOW!"
      15
      16
      "NEW WINDOW!"
      20
      21
      22
    */
  )
  .subscribe(val => console.log(val));
```

Additional Resources

- [windowToggle](#)  - Official docs
- [Split an RxJS observable conditionally with windowToggle](#)   - André

Staltz



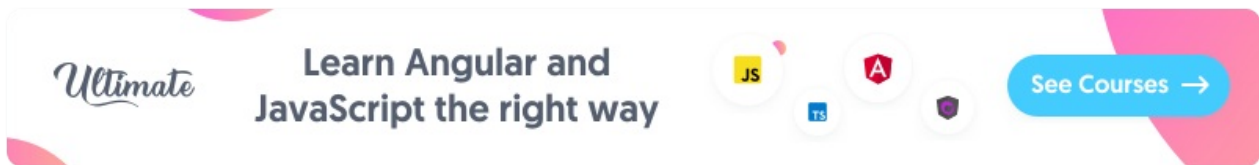
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/windowToggle.ts>

windowWhen

signature: `windowWhen(closingSelector: function(): Observable): Observable`

Close window at provided time frame emitting observable of collected values from source.



Examples

Example 1: Open and close window at interval

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, timer } from 'rxjs';
import { windowWhen, tap, mergeAll } from 'rxjs/operators';

//emit immediately then every 1s
const source = timer(0, 1000);
const example = source.pipe(
  //close window every 5s and emit observable of collected values from source
  windowWhen(() => interval(5000)),
  tap(_ => console.log('NEW WINDOW!'))
);

const subscribeTwo = example
  .pipe(
    //window emits nested observable
    mergeAll()
  )
  /*
    output:
    "NEW WINDOW!"
    0
    1
    2
    3
    4
    "NEW WINDOW!"
    5
    6
    7
    8
    9
  */
  .subscribe(val => console.log(val));
```

Additional Resources

- [windowWhen](#)  - Official docs

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/windowWhen.ts>

Utility Operators

From logging, handling notifications, to setting up schedulers, these operators provide helpful utilities in your observable toolkit.

Contents

- [do / tap](#) ★
- [delay](#) ★
- [delayWhen](#)
- [dematerialize](#)
- [finalize / finally](#)
- [let](#)
- [repeat](#)
- [timeout](#)
- [toPromise](#)

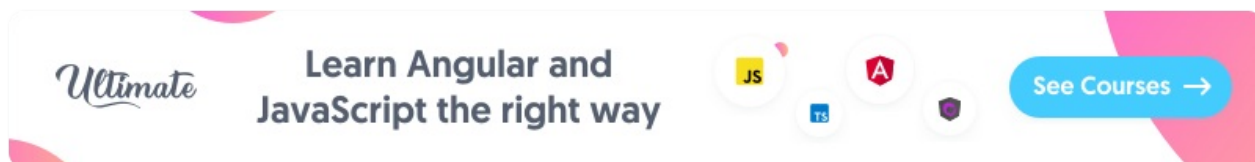
★ - *commonly used*

do / tap

signature: `do(nextOrObserver: function, error: function, complete: function): Observable`

Transparently perform actions or side-effects, such as logging.

💡 If you are using as a pipeable operator, `do` is known as `tap` !



Examples

Example 1: Logging with do

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
import { tap, map } from 'rxjs/operators';

const source = of(1, 2, 3, 4, 5);
//transparently log values from source with 'do'
const example = source.pipe(
  tap(val => console.log(`BEFORE MAP: ${val}`)),
  map(val => val + 10),
  tap(val => console.log(`AFTER MAP: ${val}`))
);

//'do' does not transform values
//output: 11...12...13...14...15
const subscribe = example.subscribe(val => console.log(val));
```

Additional Resources

- [do](#) 📖 - Official docs
- [Logging a stream with do](#) 📖 💡 - John Linquist
- [Utility operator: do](#) 📖 💡 - André Staltz

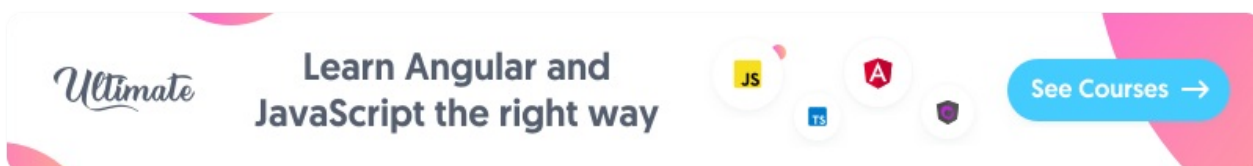
📁 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/do.ts>

delay

signature: `delay(delay: number | Date, scheduler: Scheduler): Observable`

Delay emitted values by given time.



Examples

Example 1: Delay for increasing durations

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of, merge } from 'rxjs';
import { mapTo, delay } from 'rxjs/operators';

//emit one item
const example = of(null);
//delay output of each by an extra second
const message = merge(
  example.pipe(mapTo('Hello')),
  example.pipe(
    mapTo('World!'),
    delay(1000)
  ),
  example.pipe(
    mapTo('Goodbye'),
    delay(2000)
  ),
  example.pipe(
    mapTo('World!'),
    delay(3000)
  )
);
//output: 'Hello'...'World!'...'Goodbye'...'World!'
const subscribe = message.subscribe(val => console.log(val));
```

Related Recipes

- [Progress Bar](#)

Additional Resources

- [delay](#) 📖 - Official docs
- [Transformation operator: delay and delayWhen](#) 🖨️ 💡 - André Staltz

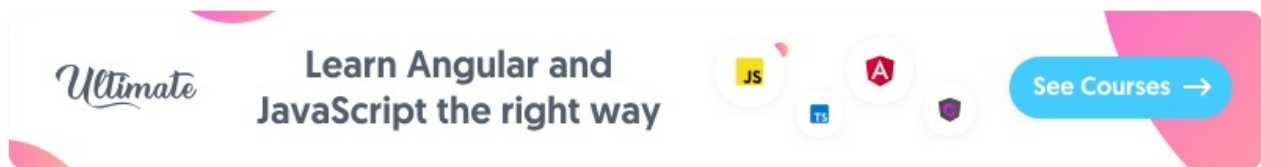
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/delay.ts>

delayWhen

signature: `delayWhen(selector: Function, sequence: Observable): Observable`

Delay emitted values determined by provided function.



Examples

Example 1: Delay based on observable

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { interval, timer } from 'rxjs';
import { delayWhen } from 'rxjs/operators';

//emit value every second
const message = interval(1000);
//emit value after five seconds
const delayForFiveSeconds = () => timer(5000);
//after 5 seconds, start emitting delayed interval values
const delayWhenExample = message.pipe(delayWhen(delayForFiveSeconds));
//log values, delayed for 5 seconds
//ex. output: 5s....1...2...3
const subscribe = delayWhenExample.subscribe(val => console.log(val));
```

Additional Resources

- [delayWhen](#) 📄 - Official docs
 - [Transformation operator: delay and delayWhen](#) 📄 📄 - André Staltz
-



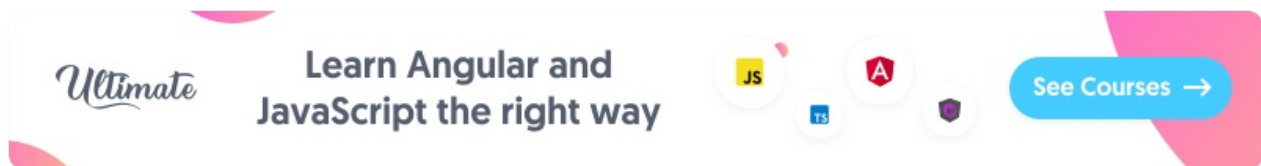
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/delayWhen.ts>

dematerialize

signature: `dematerialize(): Observable`

Turn notification objects into notification values.



Examples

Example 1: Converting notifications to values

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { from, Notification } from 'rxjs';
import { dematerialize } from 'rxjs/operators';

//emit next and error notifications
const source = from([
  Notification.createNext('SUCCESS!'),
  Notification.createError('ERROR!')
]).pipe(
  //turn notification objects into notification values
  dematerialize()
);

//output: 'NEXT VALUE: SUCCESS' 'ERROR VALUE: 'ERROR!'
const subscription = source.subscribe({
  next: val => console.log(`NEXT VALUE: ${val}`),
  error: val => console.log(`ERROR VALUE: ${val}`)
});
```

Additional Resources

- [dematerialize](#)  - Official docs

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/dematerialize.ts>

finalize / finally

signature: `finalize(callback: () => void)`

Call a function when observable completes or errors

Examples

Example 1: Execute callback function when the observable completes

([StackBlitz](#))

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

//emit value in sequence every 1 second
const source = interval(1000);
//output: 0,1,2,3,4,5....
const example = source.pipe(
  take(5), //take only the first 5 values
  finalize(() => console.log('Sequence complete')) // Execute wh
  en the observable completes
)
const subscribe = example.subscribe(val => console.log(val));
```

Related Recipes

- [HTTP Polling](#)

Additional Resources

- [finalize](#)  - Official docs



Source Code:

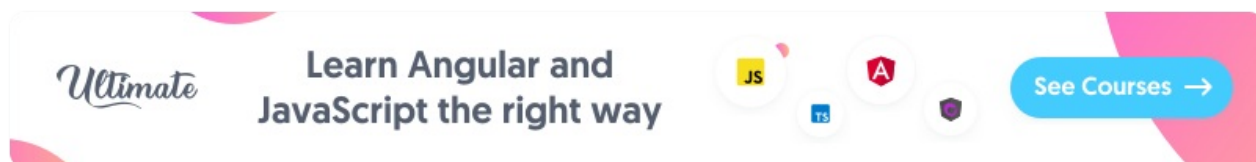
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/finalize.ts>

let

signature: `let(function): Observable`

Let me have the whole observable.

⚠ `let` is no longer available, or necessary, with pipeable operators! (RxJS 5.5+)



Examples

Example 1: Reusing error handling logic with let

([jsBin](#) | [jsFiddle](#))

```
// custom error handling logic
const retryThreeTimes = obs =>
  obs.retry(3).catch(_ => Rx.Observable.of('ERROR!'));
const examplePromise = val =>
  new Promise(resolve => resolve(`Complete: ${val}`));

//faking request
const subscribe = Rx.Observable.of('some_url')
  .mergeMap(url => examplePromise(url))
  // could reuse error handling logic in multiple places with let

  .let(retryThreeTimes)
  //output: Complete: some_url
  .subscribe(result => console.log(result));

const customizableRetry = retryTimes => obs =>
  obs.retry(retryTimes).catch(_ => Rx.Observable.of('ERROR!'));

//faking request
const secondSubscribe = Rx.Observable.of('some_url')
  .mergeMap(url => examplePromise(url))
  // could reuse error handling logic in multiple places with let

  .let(customizableRetry(3))
  //output: Complete: some_url
  .subscribe(result => console.log(result));
```

Example 2: Applying map with let

([jsBin](#) | [jsFiddle](#))

```
//emit array as a sequence
const source = Rx.Observable.from([1, 2, 3, 4, 5]);
//demonstrating the difference between let and other operators
const test = source
  .map(val => val + 1)
  /*
    this would fail, let behaves differently than most operators
    val in this case is an observable
  */
  //.let(val => val + 2)
  .subscribe(val => console.log('VALUE FROM ARRAY: ', val));

const subscribe = source
  .map(val => val + 1)
  //'let' me have the entire observable
  .let(obs => obs.map(val => val + 2))
  //output: 4,5,6,7,8
  .subscribe(val => console.log('VALUE FROM ARRAY WITH let: ', val));
```

Example 3: Applying multiple operators with let

([jsBin](#) | [jsFiddle](#))

```
//emit array as a sequence
const source = Rx.Observable.from([1, 2, 3, 4, 5]);

//let provides flexibility to add multiple operators to source observable then return
const subscribeTwo = source
  .map(val => val + 1)
  .let(obs =>
    obs
      .map(val => val + 2)
      //also, just return evens
      .filter(val => val % 2 === 0)
  )
  //output: 4,6,8
  .subscribe(val => console.log('let WITH MULTIPLE OPERATORS: ', val));
```

Example 4: Applying operators through function

([jsBin](#) | [jsFiddle](#))

```
//emit array as a sequence
const source = Rx.Observable.from([1, 2, 3, 4, 5]);

//pass in your own function to add operators to observable
const obsArrayPlusYourOperators = yourAppliedOperators => {
  return source.map(val => val + 1).let(yourAppliedOperators);
};
const addTenThenTwenty = obs => obs.map(val => val + 10).map(val => val + 20);
const subscribe = obsArrayPlusYourOperators(addTenThenTwenty)
  //output: 32, 33, 34, 35, 36
  .subscribe(val => console.log('let FROM FUNCTION:', val));
```

Additional Resources

- [let](#)  - Official docs



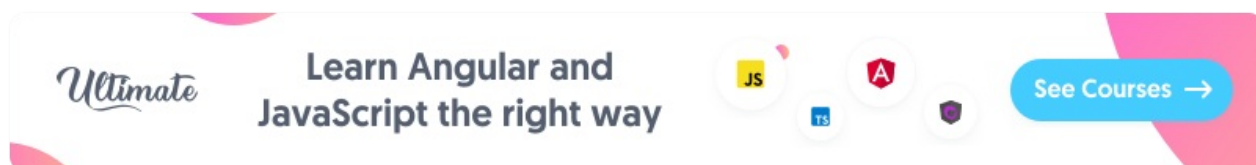
Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/let.ts>

repeat

signature: `repeat(count: number): Observable`

Repeats the stream of items emitted by the source Observable at most count times.



Examples

Example 1: Repeat 3 times


([StackBlitz](#))

```
// RxJS v6+
import { repeat, delay } from 'rxjs/operators';
import { of } from 'rxjs';

const delayedThing = of('delayed value').pipe(delay(2000));

delayedThing.pipe(
  repeat(3)
)
.subscribe(console.log)
```

Additional Resources

- [repeat](#)  - Official docs

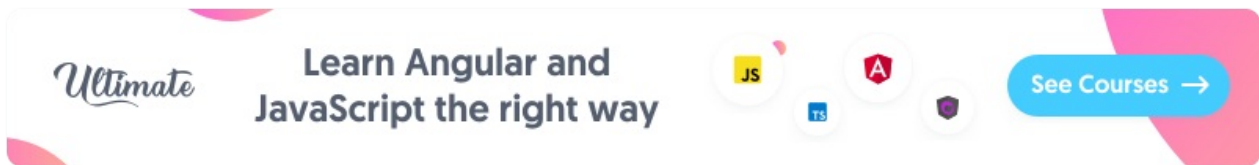
 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/repeat.ts>

timeout

signature: `timeout(due: number, scheduler: Scheduler): Observable`

Error if no value is emitted before specified duration



Examples

Example 1: Timeout after 2.5 seconds

([StackBlitz](#) | [jsBin](#) | [jsFiddle](#))

```
// RxJS v6+
import { of } from 'rxjs';
import { concatMap, timeout, catchError, delay } from 'rxjs/operators';

// simulate request
function makeRequest(timeToDelay) {
  return of('Request Complete!').pipe(delay(timeToDelay));
}

of(4000, 3000, 2000)
  .pipe(
    concatMap(duration =>
      makeRequest(duration).pipe(
        timeout(2500),
        catchError(error => of(`Request timed out after: ${duration}`))
      )
    )
  )
  /*
   * "Request timed out after: 4000"
   * "Request timed out after: 3000"
   * "Request Complete!"
   */
  .subscribe(val => console.log(val));
```

Additional Resources

- [timeout](#)  - Official Docs

 Source Code:

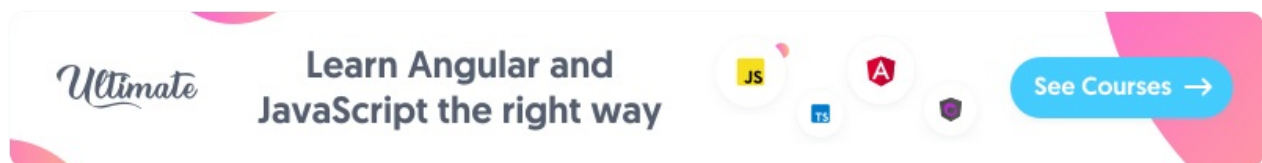
<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/timeout.ts>

toPromise

signature: `toPromise() : Promise`

Convert observable to promise.

⚠️ `toPromise` is not a pipable operator, as it does not return an observable.



Examples

Example 1: Basic Promise

([jsBin](#) | [jsFiddle](#))

```
//return basic observable
const sample = val => Rx.Observable.of(val).delay(5000);
//convert basic observable to promise
const example = sample('First Example')
  .toPromise()
  //output: 'First Example'
  .then(result => {
    console.log('From Promise:', result);
  });
```

Example 2: Using Promise.all

([jsBin](#) | [jsFiddle](#))

```
//return basic observable
const sample = val => Rx.Observable.of(val).delay(5000);
/*
  convert each to promise and use Promise.all
  to wait for all to resolve
*/
const example = () => {
  return Promise.all([
    sample('Promise 1').toPromise(),
    sample('Promise 2').toPromise()
  ]);
};
//output: ["Promise 1", "Promise 2"]
example().then(val => {
  console.log('Promise.all Result:', val);
});
```

Additional Resources

- [toPromise](#)  - Official Docs

 Source Code:

<https://github.com/ReactiveX/rxjs/blob/master/src/internal/operators/toPromise.ts>

RxJS Operators By Example

A complete list of RxJS operators with clear explanations, relevant resources, and executable examples.

Prefer a split by operator type?

Contents (In Alphabetical Order)

- [audit](#)
- [auditTime](#)
- [buffer](#)
- [bufferCount](#)
- [bufferTime](#) ★
- [bufferToggle](#)
- [bufferWhen](#)
- [catch / catchError](#) ★
- [combineAll](#)
- [combineLatest](#) ★
- [concat](#) ★
- [concatAll](#)
- [concatMap](#) ★
- [concatMapTo](#)
- [create](#)
- [debounce](#)
- [debounceTime](#) ★
- [defaultIfEmpty](#)
- [delay](#)
- [delayWhen](#)
- [distinctUntilChanged](#) ★
- [do / tap](#) ★
- [empty](#)
- [every](#)
- [exhaustMap](#)
- [expand](#)

- [filter](#) ★
- [finalize / finally](#)
- [first](#)
- [forkJoin](#)
- [from](#) ★
- [fromEvent](#)
- [groupBy](#)
- [iif](#)
- [ignoreElements](#)
- [interval](#)
- [last](#)
- [let](#)
- [map](#) ★
- [mapTo](#)
- [merge](#) ★
- [mergeAll](#)
- [mergeMap / flatMap](#) ★
- [multicast](#)
- [of](#) ★
- [partition](#)
- [pluck](#)
- [publish](#)
- [race](#)
- [range](#)
- [repeat](#)
- [retry](#)
- [retryWhen](#)
- [sample](#)
- [scan](#) ★
- [share](#) ★
- [shareReplay](#) ★
- [single](#)
- [skip](#)
- [skipUntil](#)
- [skipWhile](#)
- [startWith](#) ★

- [switchMap](#) ★
- [take](#) ★
- [takeUntil](#) ★
- [takeWhile](#)
- [throttle](#)
- [throttleTime](#)
- [throw](#)
- [timeout](#)
- [timer](#)
- [toPromise](#)
- [window](#)
- [windowCount](#)
- [windowTime](#)
- [windowToggle](#)
- [windowWhen](#)
- [withLatestFrom](#) ★
- [zip](#)

★ - *commonly used*

Additional Resources

- [What Are Operators?](#) 📄 - Official Docs
- [What Operators Are](#) 🖥️ 💻 - André Staltz

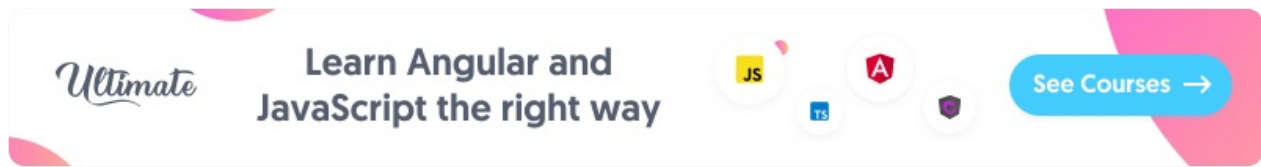
Recipes

Common use-cases and interesting recipes to help learn RxJS.

Contents

- [Progress Bar](#)
- [Smart Counter](#)
- [Game Loop](#)
- [HTTP Polling](#)

HTTP Polling



Examples

Example 1

By [@barryrowe](#)

This recipe demonstrates one way you can achieve polling an HTTP endpoint on an interval. This is a common task in web applications, and one that RxJS tends to handle really well as the continuous series of HTTP requests and responses is easy to reason about as a stream of data.

([StackBlitz](#))

```
// Import stylesheets
import './style.css';

import { Observable, Subscription, of, fromEvent, from, empty, merge, timer } from 'rxjs';
import { map, mapTo, switchMap, tap, mergeMap, takeUntil, filter, finalize } from 'rxjs/operators';

declare type RequestCategory = 'cats' | 'meats';

// Constants for Cat Requests
const CATS_URL = "https://placekitten.com/g/{w}/{h}";
function mapCats(response): Observable<string> {

  return from(new Promise((resolve, reject) => {
    var blob = new Blob([response], {type: "image/png"});
    let reader = new FileReader();
    reader.onload = (data: any) => {
      resolve(data.target.result);
    }
  }));
}
```

```
    };
    reader.readAsDataURL(blob);
  }));
}

// Constants for Meat Requests
const MEATS_URL = "https://baconipsum.com/api/?type=meat-and-filler";
function mapMeats(response): Observable<string> {
  const parsedData = JSON.parse(response);
  return of(parsedData ? parsedData[0] : '');
}

/*****
 * Our Operating State
 *****/

// Which type of data we are requesting
let requestCategory: RequestCategory = 'cats';
// Current Polling Subscription
let pollingSub: Subscription;
/*****/

/**
 * This function will make an AJAX request to the given Url, map
 * the
 * JSON parsed response with the provided mapper function, and emit
 * the result onto the returned observable.
 */
function requestData(url: string, mapFunc: (any) => Observable<string>): Observable<string> {
  console.log(url)
  const xhr = new XMLHttpRequest();
  return from(new Promise<string>((resolve, reject) => {

    // This is generating a random size for a placekitten image
    // so that we get new cats each request.
    const w = Math.round(Math.random() * 400);
    const h = Math.round(Math.random() * 400);
    const targetUrl = url
      .replace('{w}', w.toString())
      .replace('{h}', h.toString());
```

```

    xhr.addEventListener("load", () => {
      resolve(xhr.response);
    });
    xhr.open("GET", targetUrl);
    if(requestCategory === 'cats') {
      // Our cats urls return binary payloads
      // so we need to respond as such.
      xhr.responseType = "arraybuffer";
    }
    xhr.send();
  )))
  .pipe(
    switchMap((data) => mapFunc(xhr.response)),
    tap((data) => console.log('Request result: ', data))
  );
}

/**
 * This function will begin our polling for the given state, and
 * on the provided interval (defaulting to 5 seconds)
 */
function startPolling(category: RequestCategory, interval: number = 5000): Observable<string> {
  const url = category === 'cats' ? CATS_URL : MEATS_URL;
  const mapper = category === 'cats' ? mapCats : mapMeats;

  return timer(0, interval)
    .pipe(
      switchMap(_ => requestData(url, mapper))
    );
}

// Gather our DOM Elements to wire up events
const startButton = document.getElementById('start');
const stopButton = document.getElementById('stop');
const text = document.getElementById('text');
const pollingStatus = document.getElementById('polling-status');
const catsRadio = document.getElementById('catsCheckbox');
const meatsRadio = document.getElementById('meatsCheckbox');
const catsClick$ = fromEvent(catsRadio, 'click').pipe(mapTo('cat

```

```
s')));
const meatsClick$ = fromEvent(meatsRadio, 'click').pipe(mapTo('meats'));
const catImage: HTMLImageElement = <HTMLImageElement>document.getElementById('cat');
// Stop polling
let stopPolling$ = fromEvent(stopButton, 'click');

function updateDom(result) {
  if (requestCategory === 'cats') {
    catImage.src = result;
    console.log(catImage);
  } else {
    text.innerHTML = result;
  }
}

function watchForData(category: RequestCategory) {
  // Start new Poll
  return startPolling(category, 5000).pipe(
    tap(updateDom),
    takeUntil(
      // stop polling on either button click or change of categories
      merge(
        stopPolling$,
        merge(catsClick$, meatsClick$).pipe(filter(c => c !== category))
      )
    ),
    // for demo purposes only
    finalize(() => pollingStatus.innerHTML = 'Stopped')
  )
}

// Handle Form Updates
catsClick$
  .subscribe((category: RequestCategory) => {
    requestCategory = category;
    catImage.style.display = 'block';
    text.style.display = 'none';
  });
```

```
meatsClick$
  .subscribe((category: RequestCategory) => {
    requestCategory = category;
    catImage.style.display = 'none';
    text.style.display = 'block';
  });

// Start Polling
fromEvent(startButton, 'click')
  .pipe(
    // for demo purposes only
    tap(_ => pollingStatus.innerHTML = 'Started'),
    mergeMap(_ => watchForData(requestCategory))
  )
  .subscribe();
```

Operators Used

- [filter](#)
- [fromEvent](#)
- [from](#)
- [map](#)
- [mapTo](#)
- [merge](#)
- [mergeMap](#)
- [switchMap](#)
- [timer](#)

Example 2: Simple http polling

By [@adamlubek](#)

This recipe demonstrates polling an HTTP endpoint using `repeat`. It waits for 3 seconds following the response to poll again. Code below is simplified to demonstrate bare bones of solution but link below contains verbose logging and error handling.

([StackBlitz](#))


```
// RxJS v6+
import { of } from 'rxjs';
import { delay, tap, mergeMap, repeat } from 'rxjs/operators';

const fakeDelayedRequest = () => of(new Date()).pipe(delay(1000))
;

const display = response => {
  document.open();
  document.write(response);
};

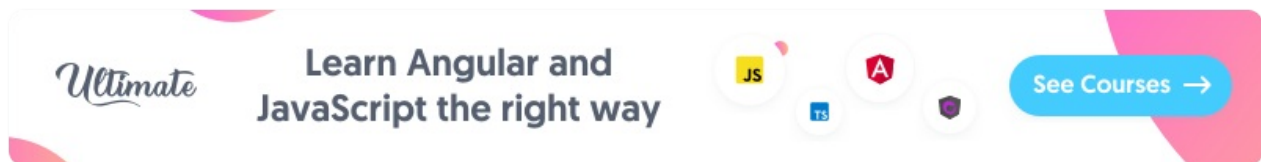
const poll = of({}).pipe(
  mergeMap(_ => fakeDelayedRequest()),
  tap(display),
  delay(3000),
  repeat()
);

poll.subscribe();
```

Game Loop

By [@barryrowe](#)

This recipe demonstrates one way you might create a Game Loop as a combined set of streams. The recipe is intended to highlight how you might re-think existing problems with a reactive approach. In this recipe we provide the overall loop as a stream of frames and their deltaTimes since the previous frames. Combined with this is a stream of user inputs, and the current gameState, which we can use to update our objects, and render to to the screen on each frame emission.



Example Code

([StackBlitz](#))

```
import { BehaviorSubject, Observable, of, fromEvent } from 'rxjs'
;
import { buffer, bufferCount, expand, filter, map, share, tap,
withLatestFrom } from 'rxjs/operators';

import { IFrameData } from './frame.interface';
import { KeyUtil } from './keys.util';
import { clampMag, runBoundaryCheck, clampTo30FPS } from './game
.util';

const boundaries = {
  left: 0,
  top: 0,
  bottom: 300,
  right: 400
};

const bounceRateChanges = {
  left: 1.1,
  top: 1.2,
```

```
    bottom: 1.3,
    right: 1.4
  }
const baseObjectVelocity = {
  x: 30,
  y: 40,
  maxX: 250,
  maxY: 200
};

const gameArea: HTMLElement = document.getElementById('game');
const fps: HTMLElement = document.getElementById('fps');

/**
 * This is our core game loop logic. We update our objects and gameState here
 * each frame. The deltaTime passed in is in seconds, we are given our current state,
 * and any inputStates. Returns the updated Game State
 */
const update = (deltaTime: number, state: any, inputState: any): any => {
  //console.log("Input State: ", inputState);
  if(state['objects'] === undefined) {
    state['objects'] = [
      {
        // Transformation Props
        x: 10, y: 10, width: 20, height: 30,
        // State Props
        isPaused: false, toggleColor: '#FF0000', color: '#000000'
      },
      {
        // Movement Props
        velocity: baseObjectVelocity
      },
      {
        // Transformation Props
        x: 200, y: 249, width: 50, height: 20,
        // State Props
        isPaused: false, toggleColor: '#00FF00', color: '#0000FF'
      },
      {
        // Movement Props
        velocity: {x: -baseObjectVelocity.x, y: 2*baseObjectVelocity.y}
```

```
city.y} }  
  ];  
  } else {  
  
    state['objects'].forEach((obj) => {  
      // Process Inputs  
      if (inputState['spacebar']) {  
        obj.isPaused = !obj.isPaused;  
        let newColor = obj.toggleColor;  
        obj.toggleColor = obj.color;  
        obj.color = newColor;  
      }  
  
      // Process GameLoop Updates  
      if(!obj.isPaused) {  
  
        // Apply Velocity Movements  
        obj.x = obj.x += obj.velocity.x*deltaTime;  
        obj.y = obj.y += obj.velocity.y*deltaTime;  
  
        // Check if we exceeded our boundaries  
        const didHit = runBoundaryCheck(obj, boundaries);  
        // Handle boundary adjustments  
        if(didHit){  
          if(didHit === 'right' || didHit === 'left') {  
            obj.velocity.x *= -bounceRateChanges[didHit];  
          } else {  
            obj.velocity.y *= -bounceRateChanges[didHit];  
          }  
        }  
      }  
  
      // Clamp Velocities in case our boundary bounces have gott  
      en  
      // us going toooooo fast.  
      obj.velocity.x = clampMag(obj.velocity.x, 0, baseObjectVelocity.maxX);  
      obj.velocity.y = clampMag(obj.velocity.y, 0, baseObjectVelocity.maxY);  
    });  
  }  
}
```

```
    return state;
}

/**
 * This is our rendering function. We take the given game state
 * and render the items
 * based on their latest properties.
 */
const render = (state: any) => {
    const ctx: CanvasRenderingContext2D = (<HTMLCanvasElement>game
Area).getContext('2d');
    // Clear the canvas
    ctx.clearRect(0, 0, gameArea.clientWidth, gameArea.clientHeight);

    // Render all of our objects (simple rectangles for simplicity)

    state['objects'].forEach((obj) => {
        ctx.fillStyle = obj.color;
        ctx.fillRect(obj.x, obj.y, obj.width, obj.height);
    });
};

/**
 * This function returns an observable that will emit the next f
rame once the
 * browser has returned an animation frame step. Given the previ
ous frame it calculates
 * the delta time, and we also clamp it to 30FPS in case we get
long frames.
 */
const calculateStep: (prevFrame: IFrameData) => Observable<IFram
eData> = (prevFrame: IFrameData) => {
    return Observable.create((observer) => {

        requestAnimationFrame((frameStartTime) => {
            // Millis to seconds
            const deltaTime = prevFrame ? (frameStartTime - prevFrame.
frameStartTime)/1000 : 0;
            observer.next({
```

```
        frameStartTime,
        deltaTime
    });
    })
  })
  .pipe(
    map(clampTo30FPS)
  )
};

// This is our core stream of frames. We use expand to recursive
// ly call the
// `calculateStep` function above that will give us each new Fr
// ame based on the
// window.requestAnimationFrame calls. Expand emits the value o
// f the called functions
// returned observable, as well as recursively calling the func
// tion with that same
// emitted value. This works perfectly for calculating our fram
// e steps because each step
// needs to know the lastStepFrameTime to calculate the next. W
// e also only want to request
// a new frame once the currently requested frame has returned.
const frames$ = of(undefined)
  .pipe(
    expand((val) => calculateStep(val)),
    // Expand emits the first value provided to it, and in this
    // case we just want to ignore the undefined input frame
    filter(frame => frame !== undefined),
    map((frame: IFrameData) => frame.deltaTime),
    share()
  )

// This is our core stream of keyDown input events. It emits an
// object like `{"spacebar": 32}`
// each time a key is pressed down.
const keysDown$ = fromEvent(document, 'keydown')
  .pipe(
    map((event: KeyboardEvent) => {
      const name = KeyUtil.codeToKey(''+event.keyCode);
      if (name !== ''){
        let keyMap = {};

```

```

        keyMap[name] = event.code;
        return keyMap;
    } else {
        return undefined;
    }
    })),
    filter((keyMap) => keyMap !== undefined)
);

// Here we buffer our keyDown stream until we get a new frame emission. This
// gives us a set of all the keyDown events that have triggered since the previous
// frame. We reduce these all down to a single dictionary of keys that were pressed.
const keysDownPerFrame$ = keysDown$
    .pipe(
        buffer(frames$),
        map((frames: Array<any>) => {
            return frames.reduce((acc, curr) => {
                return Object.assign(acc, curr);
            }, {});
        })
    );

// Since we will be updating our gamestate each frame we can use an Observable
// to track that as a series of states with the latest emission being the current
// state of our game.
const gameState$ = new BehaviorSubject({});

// This is where we run our game!
// We subscribe to our frames$ stream to kick it off, and make sure to
// combine in the latest emission from our inputs stream to get the data
// we need to perform our gameState updates.
frames$
    .pipe(
        withLatestFrom(keysDownPerFrame$, gameState$),
        // HOMEWORK_OPPORTUNITY: Handle Key-up, and map to a true Ke

```

```
yState change object
  map(([deltaTime, keysDown, gameState]) => update(deltaTime,
gameState, keysDown)),
  tap((gameState) => gameState$.next(gameState))

)
.subscribe((gameState) => {
  render(gameState);
});

// Average every 10 Frames to calculate our FPS
frames$
  .pipe(
    bufferCount(10),
    map((frames) => {
      const total = frames
        .reduce((acc, curr) => {
          acc += curr;
          return acc;
        }, 0);

      return 1/(total/frames.length);
    })
  ).subscribe((avg) => {
    fps.innerHTML = Math.round(avg) + ' ';
  })
```

supporting js

- [game.util.ts](#)
- [keys.util.ts](#)
- [frame.interface.ts](#)

html


```
<canvas width="400px" height="300px" id="game"></canvas>
<div id="fps"></div>
<p class="instructions">
  Each time a block hits a wall, it gets faster. You can hit SPA
  CE to pause the boxes. They will change colors to show they are
  paused.
</p>
```

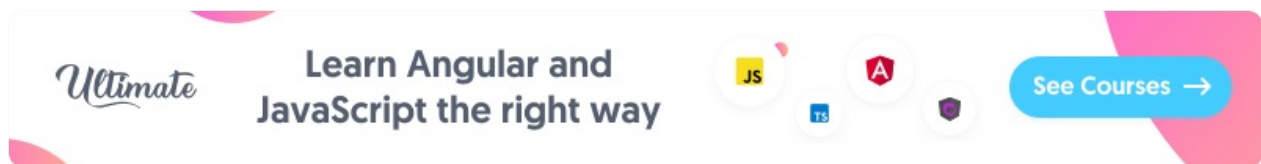
Operators Used

- `buffer`
- `bufferCount`
- `expand`
- `filter`
- `fromEvent`
- `map`
- `share`
- `tap`
- `withLatestFrom`

Progress Bar

By [@barryrowe](#)

This recipe demonstrates the creation of an animated progress bar, simulating the management of multiple requests, and updating overall progress as each completes.



Example Code

([StackBlitz](#))

```
import './style.css';

import { Observable, of, empty, fromEvent, from } from 'rxjs';
import {
  delay,
  switchMapTo,
  concatAll,
  count,
  scan,
  withLatestFrom,
  share
} from 'rxjs/operators';

const requestOne = of('first').pipe(delay(500));
const requestTwo = of('second').pipe(delay(800));
const requestThree = of('third').pipe(delay(1100));
const requestFour = of('fourth').pipe(delay(1400));
const requestFive = of('fifth').pipe(delay(1700));

const loadButton = document.getElementById('load');
const progressBar = document.getElementById('progress');
const content = document.getElementById('data');
```

```
// update progress bar as requests complete
const updateProgress = progressRatio => {
  console.log('Progress Ratio: ', progressRatio);
  progressBar.style.width = 100 * progressRatio + '%';
  if (progressRatio === 1) {
    progressBar.className += ' finished';
  } else {
    progressBar.className = progressBar.className.replace(' fini
shed', '');
  }
};

// simple helper to log updates
const updateContent = newContent => {
  content.innerHTML += newContent;
};

const displayData = data => {
  updateContent(`<div class="content-item">${data}</div>`);
};

// simulate 5 separate requests that complete at variable length
const observables: Array<Observable<string>> = [
  requestOne,
  requestTwo,
  requestThree,
  requestFour,
  requestFive
];

const array$ = from(observables);
const requests$ = array$.pipe(concatAll());
const clicks$ = fromEvent(loadButton, 'click');

const progress$ = clicks$.pipe(
  switchMapTo(requests$),
  share()
);

const count$ = array$.pipe(count());

const ratio$ = progress$.pipe(
```

```
scan(current => current + 1, 0),
withLatestFrom(count$, (current, count) => current / count)
);

clicks$.pipe(switchMapTo(ratio$)).subscribe(updateProgress);

progress$.subscribe(displayData);
```

html

```
<div class="progress-container">
  <div class="progress" id="progress"></div>
</div>

<button id="load">
Load Data
</button>

<div id="data">

</div>
```

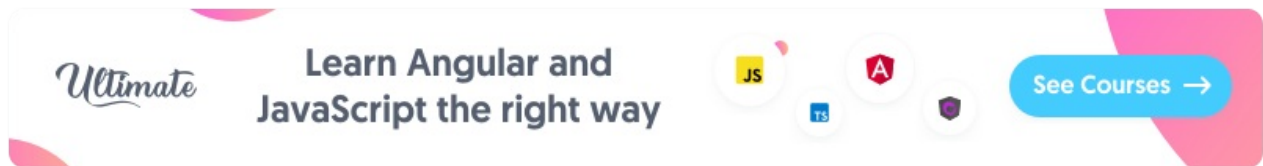
Thanks to [@johnlinquist](#) for the additional help with example!

Operators Used

- [concatAll](#)
- [delay](#)
- [fromEvent](#)
- [from](#)
- [scan](#)
- [share](#)
- [switchMap](#)
- [withLatestFrom](#)

Smart Counter

An interesting element on interfaces which involve dynamically updating numbers is a smart counter, or odometer effect. Instead of jumping a number up and down, quickly counting to the desired number can achieve a cool effect. An example of a popular library that accomplishes this is [odometer](#) by [Hubspot](#). Let's see how we can accomplish something similar with just a few lines of RxJS.



Vanilla JS

([JSBin](#) | [JSFiddle](#))

```
// utility functions
const takeUntilFunc = (endRange, currentNumber) => {
  return endRange > currentNumber
    ? val => val <= endRange
    : val => val >= endRange;
};

const positiveOrNegative = (endRange, currentNumber) => {
  return endRange > currentNumber ? 1 : -1;
};

const updateHTML = id => val => (document.getElementById(id).innerHTML = val);
// display
const input = document.getElementById('range');
const updateButton = document.getElementById('update');

const subscription = (function(currentNumber) {
  return fromEvent(updateButton, 'click').pipe(
    map(_ => parseInt(input.value)),
    switchMap(endRange => {
      return timer(0, 20).pipe(
        mapTo(positiveOrNegative(endRange, currentNumber)),
        startWith(currentNumber),
        scan((acc, curr) => acc + curr),
        takeWhile(takeUntilFunc(endRange, currentNumber));
      )
    }
  ),
  tap(v => (currentNumber = v)),
  startWith(currentNumber)
)
.subscribe(updateHTML('display'));
})(0);
```

HTML

```
<input id="range" type="number">
<button id="update">Update</button>
<h3 id="display">0</h3>
```

We can easily take our vanilla smart counter and wrap it in any popular component based UI library. Below is an example of an Angular smart counter component which takes an `Input` of the updated end ranges and performs the appropriate transition.

Angular Version

([StackBlitz](#))

```
import { Component, Input, OnDestroy } from '@angular/core';
import { Subject } from 'rxjs/Subject';
import { timer } from 'rxjs/observable/timer';
import { switchMap, startWith, scan, takeWhile, takeUntil, mapTo
} from 'rxjs/operators';

@Component({
  selector: 'number-tracker',
  template: `
    <h3> {{ currentNumber }}</h3>
  `
})
export class NumberTrackerComponent implements OnDestroy {
  @Input()
  set end(endRange: number) {
    this._counterSub$.next(endRange);
  }
  @Input() countInterval = 20;
  public currentNumber = 0;
  private _counterSub$ = new Subject();
  private _onDestroy$ = new Subject();

  constructor() {
    this._counterSub$
      .pipe(
        switchMap(endRange => {
          return timer(0, this.countInterval).pipe(
            mapTo(this.positiveOrNegative(endRange, this.current
Number)),
            startWith(this.currentNumber),
            scan((acc: number, curr: number) => acc + curr),
            takeWhile(this.isApproachingRange(endRange, this.cur
```

```

rentNumber))
    )
    }},
    takeUntil(this._onDestroy$)
)
.subscribe((val: number) => this.currentNumber = val);
}

private positiveOrNegative(endRange, currentNumber) {
    return endRange > currentNumber ? 1 : -1;
}

private isApproachingRange(endRange, currentNumber) {
    return endRange > currentNumber
        ? val => val <= endRange
        : val => val >= endRange;
}

ngOnDestroy() {
    this._onDestroy$.next();
    this._onDestroy$.complete();
}
}

```

HTML

```

<p>
  <input type="number"
    (keyup.enter)="counterNumber = vanillaInput.value"
    #vanillaInput>
  <button
    (click)="counterNumber = vanillaInput.value">
    Update number
  </button>
</p>
<number-tracker [end]="counterNumber"></number-tracker>

```

Operators Used

- [fromEvent](#)
- [map](#)
- [mapTo](#)
- [scan](#)
- [startWith](#)
- [switchMap](#)
- [takeWhile](#)

Concepts

Short explanations of common RxJS scenarios and use-cases.

Contents

- [RxJS v5 -> v6 Upgrade](#)
- [Understanding Operator Imports](#)

RxJS v5 -> v6 Upgrade

Preparing to upgrade from RxJS v5 to v6? Here are some resources you might find handy:

RxJS-TsLint

TsLint rules for migration to RxJS 6. Auto-update project for new import paths and transition to pipeable operators.

RxJS v5.x to v6 Update Guide

Comprehensive guide for updating your project from RxJS v5 to 6

Interactive Comparison of RxJS v5 and v6 Code

Demonstrates differences between v5 and v6 code, as well as RxJS examples utilizing the experimental [pipeline operator](#).

Pipeable Operators

Explanation and examples of utilizing pipeable operators.

Understanding Operator Imports

A problem you may have run into in the past when consuming or creating a public library that depends on RxJS is handling operator inclusion. The most predominant way to include operators in your project is to import them like below:

```
import 'rxjs/add/operator/take';
```

This adds the imported operator to the `Observable` prototype for use throughout your project:

[\(Source\)](#)

```
import { Observable } from '../Observable';
import { take } from '../operator/take';

Observable.prototype.take = take;

declare module '../Observable' {
  interface Observable<T> {
    take: typeof take;
  }
}
```

This method is generally *OK* for private projects and modules, the issue arises when you are using these imports in say, an [npm](#) package or library to be consumed throughout your organization.

A Quick Example

To see where a problem can spring up, let's imagine **Person A** is creating a public Angular component library. In this library you need a few operators so you add the typical imports:

some-public-library.ts

```
import 'rxjs/add/operator/take';  
import 'rxjs/add/operator/concatMap';  
import 'rxjs/add/operator/switchMap';
```

Person B comes along and includes your library. They now have access to these operators even though they did not personally import them. *Probably not a huge deal but it can be confusing.* You use the library and operators, life goes on...

A month later **Person A** decides to update their library. They no longer need

`switchMap` or `concatMap` so they remove the imports:

some-public-library.ts

```
import 'rxjs/add/operator/take';
```

Person B upgrades the dependency, builds their project, which now fails. They never included `switchMap` or `concatMap` themselves, it was **just working** based on the inclusion of a 3rd party dependency. If you were not aware this could be an issue it may take a bit to track down.

The Solution

Instead of importing operators like:

```
import 'rxjs/add/operator/take';
```

We can instead import them like:

```
import { take } from 'rxjs/operator/take';
```

This keeps them off the `Observable` prototype and let's us call them directly:

```
import { take } from 'rxjs/operator/take';
import { of } from 'rxjs/observable/of';

take.call(
  of(1, 2, 3),
  2
);
```

This quickly gets **ugly** however, imagine we have a longer chain:

```
import { take } from 'rxjs/operator/take';
import { map } from 'rxjs/operator/map';
import { of } from 'rxjs/observable/of';

map.call(
  take.call(
    of(1, 2, 3),
    2
  ),
  val => val + 2
);
```

Pretty soon we have a block of code that is near impossible to understand. How can we get the best of both worlds?

Pipeable Operators

RxJS now comes with a `pipe` helper on `Observable` that alleviates the pain of not having operators on the prototype. We can take the ugly block of code from above:

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

map.call(
  take.call(
    of(1, 2, 3),
    2
  ),
  val => val + 2
);
```

And transform it into:

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

of(1, 2, 3)
  .pipe(
    take(2),
    map(val => val + 2)
  );
```

Much easier to read, right? This also has the benefit of greatly reducing the RxJS bundle size in your application. For more on this, check out [Ashwin Sureshkumar's](#) excellent article [Reduce Angular app bundle size using lettable operators](#).