20 RxJS Operators

You Absolutely Need to Know

www.rxjs.live

combineLatest

Combines the values from all supplied observables. Note the previous pipeable operator is deprecated with the suggestion of using the static version. Use this when you need to support one observable being dependent upon another.

```
obs$.pipe(combineLatest(obs2$)); // [obs$Value, latest obs2$Value]
```

concat

Emit values from provided observable after the first observable completes. Use this when you need to handle multiple sets of data in the same way in order.

```
obs$.pipe(concat(obs2$));
// all values from obs$ then values from obs2$
```

concatMap

Allows for emitting values from an operation that creates a separate observable, it will emit values from the created observable. Original values will be emitted in order after each of their created observables have completed. Use this when you need to flatten an observable of observables when you want to handle each parent emit after the child observable completes.

```
obs$.pipe( // 0
  concatMap(x => {
    return getLetter(x); // Observable which emits one value (‘a’)
  }) // ‘a’
```

count

Tells how many values were emitted, when the stream completes. Use this when you need to find out how many items were in an observable stream. This example shows an operation via count that doesn't effect the values.

```
obs$.pipe( count() ); // grab the count
```

Looking to step up your RxJS Skills?
Attend the RxJS Live Conference!
https://rxjs.live

20 RxJS Operators

You Absolutely Need to Know

www.rxjs.live

filter

Allows you to prevent values from being emitted based upon a supplied function. Use this when you want to control what values are emitted for further processing.

```
obs$.pipe(
  filter(x => {
    return x % 2 === 0; // Only allow even numbers through.
  })
);
```

first

Only emit the first value, after the first value the observable will complete. Use this if you only care about the first value.

```
obs$.pipe(first());
```

last

Only emit the last value prior to completion. Use this if you only care about the last value.

```
obs$.pipe(last());
```

startWith

Provides the ability to specify a value which will be the first value emitted by the observable. Use this to seed your observable with a specific value.

```
obs$.pipe(startWith(1000));
```

debounceTime

Adds a time buffer to only emit when no other values have been emitted in the timeframe specified. Use this to limit longer running processes that can be requested multiple times when you only care about the most recent value. i.e. making HTTP calls for autocomplete

```
obs$.pipe(debounceTime(250));
```

distinct

A given value is not emitted more than once. Use this when you do not want to reprocess the same information more than once.

```
obs$.pipe(distinct());
```

distinctUntilChanged

Emits only when the current value is different than the previous value. Use this when you are only interested in doing something when there is a new value. i.e. don't validate a textbox value if the user pasted the same value over itself.

```
obs$.pipe(distinctUntilChanged());
```

scan

Allows for accumulation of data as values emit, like a running total for numbers. The accumulated value will emit when the source observable emits. Use this to aggregate data as values emit.

```
obs$.pipe(
  scan((acc, x) => {
    return acc + x;
  }, seed)
);
```

switchMap

Allows for emitting values from an operation that creates a separate observable, it will emit values from the created observable. Values from the created observable will be emitted only from the most recent emitted source observable. Use this when you only care about the most recent parent emit value's child values, because a new parent emit will cancel the previous child observable.

```
obs$.pipe(
  switchMap(x => {
    return getRemoteData(x);
  })
);
```

take

Allows specification of the number of emits that occur before the observable completes. Use this when you want to limit the number of values.

```
obs$.pipe(take(2)); // will emit the first two values and then complete
);
```

takeUntil

Allows specifying when an observable will complete based upon the emission of separate observable. Use this when you want one event to signal the completion of another. i.e. only take mouse move events until a mouse up event.

```
obs$.pipe(takeUntil(otherObs$));
```

zip

Pairs values from multiple observables into a single array value. When one of the "zipped" observable completes, the resulting observable completes. Use this when when the values of multiple observables are matched and their values are needed together.

```
obs$.pipe(zip(obs2$));
```

tap

Allows for side-effects based upon the source observable, but does not have an effect on the values being emitted. Use this to use the emit of an observable to trigger something outside the scope of the observable. A common use case is to place debugging statements such as logging.

```
obs$.pipe(
  tap(value => {
    log(`my value ${value}`);
  })
);
```

withLatestFrom

Allows “pulling” latest value from another observable when the source observable emits. The source value is combined with the other observable in an array. Use this when you need information from another observable, but may not care when that observable emits.

```
obs$.pipe(withLatestFrom(obs2$));
```

map

Allows the values to be modified to a new value. Use this when you want to change values being emitted.

```
obs$.pipe(
  map(x => {
    return x * 2;
  })
);
```

endWith

Allows you to specify the last value to be emitted before completion.

```
obs$.pipe(endWith(1000));
```