



The RxJS library

Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change ([Wikipedia](#)). RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code ([RxJS Docs](#)).

RxJS provides an implementation of the `Observable` type, which is needed until the type becomes part of the language and until browsers support it. The library also provides utility functions for creating and working with observables. These utility functions can be used for:

- Converting existing code for async operations into observables
- Iterating through the values in a stream
- Mapping values to different types
- Filtering streams
- Composing multiple streams

Observable creation functions

RxJS offers a number of functions that can be used to create new observables. These functions can simplify the process of creating observables from things such as events, timers, promises, and so on. For example:

Create an observable from a promise

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

// Create an Observable out of a promise
const data = from(fetch('/api/endpoint'));
// Subscribe to begin listening for async result
data.subscribe({
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
  complete() { console.log('Completed'); }
});
```

Create an observable from a counter

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

// Create an Observable that will publish a value on an interval
const secondsCounter = interval(1000);
// Subscribe to begin publishing values
secondsCounter.subscribe(n =>
  console.log(`It's been ${n} seconds since subscribing!`));
```

Create an observable from an event

```
1. import { fromEvent } from 'rxjs';
2.
3. const el = document.getElementById('my-element');
4.
5. // Create an Observable that will publish mouse movements
6. const mouseMoves = fromEvent(el, 'mousemove');
7.
8. // Subscribe to start listening for mouse-move events
9. const subscription = mouseMoves.subscribe((evt: MouseEvent) =>
  {
10.   // Log coords of mouse movements
11.   console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);
12.
13.   // When the mouse is over the upper-left of the screen,
14.   // unsubscribe to stop listening for mouse movements
15.   if (evt.clientX < 40 && evt.clientY < 40) {
16.     subscription.unsubscribe();
17.   }
18. });
```

Create an observable that creates an AJAX request

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

// Create an Observable that will create an AJAX request
const apiData = ajax('/api/data');
// Subscribe to create the request
apiData.subscribe(res => console.log(res.status, res.response));
```

Operators

Operators are functions that build on the observables foundation to enable sophisticated manipulation of collections. For example, RxJS defines operators such as `map()`, `filter()`, `concat()`, and `flatMap()`.

Operators take configuration options, and they return a function that takes a source observable. When executing this returned function, the operator observes the source observable's emitted values, transforms them, and returns a new observable of those transformed values. Here is a simple example:

Map operator

```
1. import { map } from 'rxjs/operators';
2.
3. const nums = of(1, 2, 3);
4.
5. const squareValues = map((val: number) => val *
    val);
6. const squaredNums = squareValues(nums);
7.
8. squaredNums.subscribe(x => console.log(x));
9.
10. // Logs
11. // 1
12. // 4
13. // 9
```

You can use *pipes* to link operators together. Pipes let you combine multiple functions into a single function. The `pipe()` function takes as its arguments the functions you want to combine, and returns a new function that, when executed, runs the composed functions in sequence.

A set of operators applied to an observable is a recipe—that is, a set of instructions for producing the values you're interested in. By itself, the recipe doesn't do anything. You need to call `subscribe()` to produce a result through the recipe.

Here's an example:

Standalone pipe function

```
1. import { filter, map } from 'rxjs/operators';
2.
3. const nums = of(1, 2, 3, 4, 5);
4.
5. // Create a function that accepts an Observable.
6. const squareOddVals = pipe(
7.   filter((n: number) => n % 2 !== 0),
8.   map(n => n * n)
9. );
10.
11. // Create an Observable that will run the filter and map
   functions
12. const squareOdd = squareOddVals(nums);
13.
14. // Subscribe to run the combined functions
15. squareOdd.subscribe(x => console.log(x));
```

The `pipe()` function is also a method on the RxJS [Observable](#), so you use this shorter form to define the same operation:

Observable.pipe function

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

const squareOdd = of(1, 2, 3, 4, 5)
  .pipe(
    filter(n => n % 2 !== 0),
    map(n => n * n)
  );

// Subscribe to get values
squareOdd.subscribe(x => console.log(x));
```

Common operators

RxJS provides many operators, but only a handful are used frequently. For a list of operators and usage samples, visit the [RxJS API Documentation](#).

Note that, for Angular apps, we prefer combining operators with pipes, rather than chaining. Chaining is used in many RxJS examples.

AREA	OPERATORS
Creation	<code>from</code> , <code>fromEvent</code> , <code>of</code>
Combination	<code>combineLatest</code> , <code>concat</code> , <code>merge</code> , <code>startWith</code> , <code>withLatestFrom</code> , <code>zip</code>
Filtering	<code>debounceTime</code> , <code>distinctUntilChanged</code> , <code>filter</code> , <code>take</code> , <code>takeUntil</code>
Transformation	<code>bufferTime</code> , <code>concatMap</code> , <code>map</code> , <code>mergeMap</code> , <code>scan</code> , <code>switchMap</code>
Utility	<code>tap</code>
Multicasting	<code>share</code>

Error handling

In addition to the `error()` handler that you provide on subscription, RxJS provides the `catchError` operator that lets you handle known errors in the observable recipe.

For instance, suppose you have an observable that makes an API request and maps to the response from the server. If the server returns an error or the value doesn't exist, an error is produced. If you catch this error and supply a default value, your stream continues to process values rather than erroring out.

Here's an example of using the `catchError` operator to do this:

catchError operator

```
1. import { ajax } from 'rxjs/ajax';
2. import { map, catchError } from 'rxjs/operators';
3. // Return "response" from the API. If an error happens,
4. // return an empty array.
5. const apiData = ajax('/api/data').pipe(
6.   map(res => {
7.     if (!res.response) {
8.       throw new Error('Value expected!');
9.     }
10.    return res.response;
11.  }),
12.  catchError(err => of([]))
13. );
14.
15. apiData.subscribe({
16.   next(x) { console.log('data: ', x); },
17.   error(err) { console.log('errors already caught... will not run'); }
18. });
```

Retry failed observable


Where the `catchError` operator provides a simple path of recovery, the `retry` operator lets you retry a failed request.

Use the `retry` operator before the `catchError` operator. It resubscribes to the original source observable, which can then re-run the full sequence of actions that resulted in the error. If this includes an HTTP request, it will retry that HTTP request.

The following converts the previous example to retry the request before catching the error:

retry operator

```
1. import { ajax } from 'rxjs/ajax';
2. import { map, retry, catchError } from 'rxjs/operators';
3.
4. const apiData = ajax('/api/data').pipe(
5.   retry(3), // Retry up to 3 times before failing
6.   map(res => {
7.     if (!res.response) {
8.       throw new Error('Value expected!');
9.     }
10.    return res.response;
11.  }),
12.  catchError(err => of([]))
13. );
14.
15. apiData.subscribe({
16.   next(x) { console.log('data: ', x); },
17.   error(err) { console.log('errors already caught... will not run'); }
18. });
```



Do not retry **authentication** requests, since these should only be initiated by user action. We don't want to lock out user accounts with repeated login requests that the user has not initiated.

Naming conventions for observables

Because Angular applications are mostly written in TypeScript, you will typically know when a variable is an observable. Although the Angular framework does not enforce a naming convention for observables, you will often see observables named with a trailing “\$” sign.

This can be useful when scanning through code and looking for observable values. Also, if you want a property to store the most recent value from an observable, it can be convenient to simply use the same name with or without the “\$”.

For example:

Naming observables

```
1. import { Component } from
   '@angular/core';
2. import { Observable } from 'rxjs';
3.
4. @Component({
5.   selector: 'app-stopwatch',
6.   templateUrl:
7.     './stopwatch.component.html'
8. })
9.
10. export class StopwatchComponent {
11.
12.   stopwatchValue: number;
13.   stopwatchValue$: Observable<number>;
14.
15.   start() {
16.     this.stopwatchValue$.subscribe(num =>
17.       this.stopwatchValue = num
18.     );
19.   }
20. }
```