How to RxJS in Angular

Matthias Meier Follow
Apr 8 · 4 min read

Angular and RxJS

After a few years of Angular frontend development with heavy use of RxJS I decided to put some of my personal key learnings into a concise writeup. In this article, I'm assuming that you already have some basic understanding of how Observable-streams, as well as the different Subject-types work. If so, this may hopefully help you master the most common use-cases of RxJS in Angular.

Component Inputs

As the values of your component inputs change over time, you may want to do something with that data inside your component. So what you usually do is implement the NgOnChanges method to react to those changes. Now this gets a bit fuzzy as soon as you depend on other asynchronous data, which is exactly where RxJS comes in. Unfortunately, it is currently not natively supported for component inputs to be streamed.

Therefore, whenever I need the power of RxJS on changing component inputs, I use this pattern:

```
@Input() public amount: number;
public amount$ = new Subject();
```

```
public ngOnChanges(changes: SimpleChanges): void {
  if (changes.amount && changes.amount.currentValue !==
```

```
  undefined) {
      this.amount$.next(changes.amount.currentValue);
    }
  }
```

The `!== undefined` is only needed in case the new input value could be evaluated to `false` according to the laws of JavaScript.

Button Clicks

Let's say you had some RxJS-based counter going and you wanted to reset this counter by clicking a reset-button. The most straightforward way (I know) to accomplish this, would be, to create a new `onReset$` Subject which you can then weave into your counter-stream setup.

```
<button (click)="onReset$.next()">Reset</button>
```

Http Requests

For the basic setup of data services, you can follow the Angular Guide.
But usually, there are use cases where retrieving data is not as simple as that. Many times you rely on other asynchronous data. So the mental model for that could be:

> *As soon as I get data X, I want to load data Y (which relates X)*

Many times you could extend the above statement with

> *and if there still is an open Y-request (related to a **previous** X), I want to cancel that one.*

If that's what you want to achieve, the switchMap-operator is what you're looking for. Consider the following example for retrieving the bookmarks of our current user.

```
public bookmarks$ = this.currentUser$.pipe(
  switchMap(user =>
this.bookmarksService.getBookmarks(user.id)),
);
```

Async Pipe

Angular includes a very useful async-pipe, which makes it easy to consume your observable streams in-template. So instead of subscribing within your TypeScript component and assigning the value to a public property, you can just use the async-pipe to retrieve your asynchronous values.

```
<ul>
  <li *ngFor="let bookmark of bookmarks$|async">{{bookmark}}
</li>
</ul>
```

Sharing Expensive Data

Using the techniques mentioned above, you may go ahead and subscribe multiple times to your bookmarks$ stream, using the async-pipe. When having a look at your network console, you will soon realize, that multiple requests are being done. That's not what we want. Instead, we want to **share** the response with whomever is interested in it.

There are multiple ways to achieve this. One of them being the publishReplay-operator, which publishes the source stream as a ReplaySubject, followed by the refCount-operator, which handles (un-) subscribing as long as there's at least one listener.

```
public bookmarks$ = this.currentUser$.pipe(
  switchMap(user =>
this.bookmarksService.getBookmarks(user.id)),
  publishReplay(1),
  refCount(),
);
```

Unsubscribing

On to the last point of this post. If you always use the async-pipe, there's no need to worry about unsubscribing, as the pipe handles this by itself. So, of course, it makes a lot of sense to use it whenever

possible. But there may be cases where you need to manually subscribe to observable-streams. In those cases, it is important, that you properly unsubscribe, or you will experience **memory leaks**. Using the takeUntil-operator was the cleanest way I found, to accomplish this.

The preceding setup in your component looks like this:

```
private unsubscribe$ = new Subject();
```

```
public ngOnDestroy(): void {
  this.unsubscribe$.next();
  this.unsubscribe$.complete();
}
```

So on any stream that needs to be closed, you can just use the takeUntil-operator right before calling `subscribe` and your subscriptions will be closed as soon as your component is about to be destroyed.

```
this.bookmarks$
  .pipe(takeUntil(this.unsubscribe$))
  .subscribe(bookmarks => {
    // do something
  });
```

• • •

I really hope this helped some of you to get started in the world of Angular and RxJS, as it can be quite a bit intimidating at first.

Any questions and suggestions are very welcome. More content will follow.

• • •

Originally published at https://www.matthiasmeier.io/blog/how-to-rxjs-in-angular/.