The Best Way To Unsubscribe RxJS Observables In The Angular Applications!

Tomas Trajan Follow

Jan 8 · 13 min read

An epic journey to RxJS .subscribe()-less Angular applications by 🐻 at Mount Cook (New Zealand)

A Short Introduction To RxJS Observables

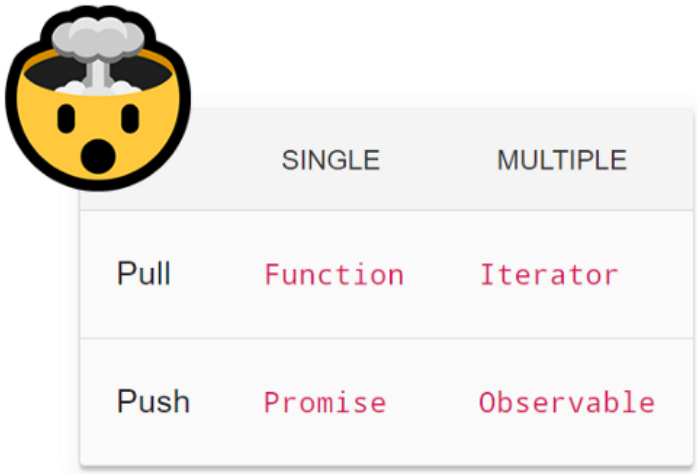
(feel free to skip this part and get straight to the main course🍔)

The *RxJS* (aka `Observable` -s) is a rather new-ish technology in the frontend engineering space. Made popular mostly by its inclusion in the core Angular APIs. It proved to be a very powerful tool when dealing with the collections of asynchronous events.

> *We can think of RxJS Observable as an potentially infinite async Array with all the conveniences we're used to like filter or map and much more...*

Evolution Of Async Concepts

In the beginning, we started with simple callbacks. A callback is a function we pass as an argument to some other function. The callback will be then executed later, when something is done.



One of the most information-dense diagram ever created! Make sure to take your time to understand and internalize this!

Later, we were mostly working with promises. Promises always guaranteed to return single value, be it result or error. Once the value was resolved, handlers got executed and that was it. Everything was completed, cleaned up and we could move on.

But then, everything changed forever. The frontend sages discovered the next piece of the push / pull puzzle…

> *With Observables, we're now dealing with **zero to many** values over time.*

Observables behavior necessitates a new way of consuming of the incoming values. We have to **subscribe** to the observable stream so that our handler gets called every time a new value is emitted.

We can't really know how many values will be coming beforehand. More so, some streams are potentially infinite (eg *user clicks*, *websocket messages*). This leads us to the realization that we have to manage the state of a subscriptions on our own.

> *There are many different ways how to handle RxJS subscriptions in Angular applications*

They provide different trade-offs in terms of verbosity, robustness or simplicity. In this article we're going to explore many of this approaches . In general we will try to optimize our solution so that it is:

- **concise**—minimize the amount of code we have to write and read
- **robust**—minimize the possibility of introducing bugs
- **simple**—easy to understand whats going on

What Are We Going To Learn?

On our journey we will go through various possible solutions to subscribing to RxJs Observable.

1. The .subscribe() aka "*The Memory Leak*"
2. The .unsubscribe()
3. Let's get declarative with `takeUntil`
4. Use the `take(1)` for initialization
5. The fabled `| async` pipe

6. The detour— `| async` pipe taken too far 😵
7. The final destination—NgRx Effects

1. The .subscribe() aka “The Memory Leak”

Let’s start with the simplest example. We have a `timer` which is a infinite cold observable.

> *Cold Observable is an Observable which will do nothing by itself. Somebody has to subscribe to it to start its execution. Infinite means that once subscribed, observable will never* `complete` .

We subscribe to the timer in `ngOnInit` method of a component and call `console.log` every time timer emits a new value.

```
@Component({/* ... */})
export class SomeComponent implements OnInit {

    everySecond$: Observable<number> = timer(0, 1000);

    ngOnInit() {
        this.everySecond$.subscribe(second => console.log(second));
    }

}
```

Why Is This A Memory Leak

Implementation looks good and does exactly what we expect. What would happen if we navigated to some other screen which is implemented using different components?

> *The component will get destroyed but the subscription will live on*

More logs will keep getting added to the browser console. More so, if we navigated back to the original route. The component would get recreated together with a new subscription.

> *We could repeat this process multiple times and the console output would get very very busy*😉

When Is It OK To Just Subscribe

Some components (eg `AppComponent`) and most of the services (with exception of services from lazy loaded modules and services provided in `@Component` decorator) in our Angular application will be instantiated **only once** during the application startup.

> *If we know that we're dealing with such a case it is OK to subscribe to an Observable without providing any unsubscription logic.*

These components and services will live for the whole duration of the application lifetime so they will not produce any memory leaks.

Eventually, these subscriptions will get cleaned up when we navigate away from application to some other website.

2. The .unsubscribe() Method

OK, we figured out that we have probably implemented couple of accidental memory leaks and we're eager to get rid of them ASAP!

> *The memory leaks are created when we destroy and recreate our components but we don't clean up existing subscriptions. As we re-create our components we keep adding more and more subscriptions, hence the memory leak...*

Subscribing to an observable yields us `Subscription` object which has an `unsubscribe()` method. This method can be used to remove the subscription when we no longer need it.

```
@Component({/* ... */})
export class AppComponent implements OnInit, OnDestroy {
    private subscription: Subscription;

    timer$: Observable<number>;

    /* ... */

    ngOnInit() {
       this.subscription = this.timer$
          .subscribe(secondsElapsed => console.log(secondsElapse));
    }

    ngOnDestroy() {
        this.subscription.unsubscribe();
    }
}
```

Store reference to the original subscription and use imperative unsubscribe inside of ngOnDestory which gets called during the destruction of the component

Or we can get a bit more fancy with multiple subscriptions...

```
@Component({/* ... */})
export class AppComponent implements OnInit, OnDestroy {
    private subscriptions: Subscription[] = [];

    everySecond$: Observable<number> = timer(0, 1000);
    everyMinute$: Observable<number> = timer(0, 60000);

    ngOnInit() {
        this.subscriptions.push(this.everySecond$.subscribe(second => console.log(second)));
        this.subscriptions.push(this.everyMinute$.subscribe(minute => console.log(minute)));
    }

    ngOnDestroy() {
        this.subscriptions.forEach(subscription => subscription.unsubscribe());
    }
}
```

Store multiple subscription references in the subscriptions array and unsubscribe all of them during ngOnDestroy

So is this really wrong? No, it works perfectly fine. The problem with this approach is that we're mixing observable streams with plain old imperative logic.

> *In my experience, developers who are learning RxJS for the first time need to really be able to switch their perspective from imperative world view to **thinking in streams**. Handling stuff using an imperative approach when declarative "Observable friendly" alternative is available tends to slow down that learning process and therefore should be avoided!*

Thanks to Wojciech Trawiński for enhancing this point by showing that there is a built in mechanism in the `Subscription` itself to make this happen. That being said I would still recommend to use more declarative approach to unsubscribing described later...

```
@Component({ /* ... */})
export class AppComponent implements OnInit, OnDestroy {
    private subscription: Subscription = new Subscription();

    everySecond$: Observable<number> = timer(0, 1000);
    everyMinute$: Observable<number> = timer(0, 60000);

    ngOnInit() {
        this.subscription.add(this.everySecond$.subscribe(second => console.log(second)));
        this.subscription.add(this.everyMinute$.subscribe(minute => console.log(minute)));
    }

    ngOnDestroy() {
        this.subscription.unsubscribe();
    }
}
```

Collect subscriptions using **subscription.add()** method instead in array of subscriptions.

3. Let's Get Declarative With takeUntil

So let's move on and make our applications better with a help of the `takeUntil` RxJS operator (this will also make Ben Lesh happy as a side-effect).

> *Official Docs:* `takeUntil(notifier: Observable<any>)` *—Emits the values emitted by the source Observable* ***until*** *a* `notifier` *Observable emits a value.*

```
@Component({/* ... */})
export class AppComponent implements OnInit, OnDestroy {
    private unsubscribe$ = new Subject<void>;

    routeActivationEndEvent$: Observable<ActivationEnd>;

    /* ... */

    ngOnInit() {
       this.routeActivationEndEvent$
          .pipe(takeUntil(unsubscribe$))    // usubscribe original observable on unsubscribe$ emmission
          .subscribe(search => this.titleService.updateTitle());
    }

    ngOnDestroy() {
        this.unsubscribe$.next();           // these two lines are easy to forget
        this.unsubscribe$.complete();       // because forgeting them wouldn't lead to any error
    }                                       // ngOnDestroy() method is enforced by interface though
}                                           // but we may forget to add interface too 😃
```

Note that we are using .pipe() method of the observable to add operators, in our case takeUntil to the observable chain

This solution is **declarative**! This means that we declare our Observable chain before hand with everything that it needs to accommodate for the whole life cycle from start to end.

> *The takeUntil() solution is great but unfortunately it comes also with a couple of disadvantages*

Most obviously, it's quite verbose ! We have to create additional `Subject` and correctly implement `OnDestroy` interface in every component of our application which is quite a lot!

Even bigger problem is that it is a quite error prone process. It is **VERY** easy to forget to implement `OnDestroy` interface. And this itself can go wrong in two different ways...

1. Forgetting to implement the `OnDestroy` interface itself

2. Forgetting to call `.next()` and `.complete()` methods in the **ngOnDestroy** implementation (leaving it empty)

> *In case you're saying that you will just always check for it, sure, I was thinking the same until I discovered couple of memory leaks in one of my applications with exactly this issue!*

The largest problem with this is that these two things will **NOT** result in any obvious errors whatsoever so they are very easy to miss!

A possible solution would be to implement (or find if it exists) a custom `tslint` rule which will check for missing (or empty) `ngOnDestroy()` methods in every component which can also be problematic because not every component uses subscriptions...

> *Many thanks to Brian Love for feedback. We should not forget about the fact that the `takeUntil` operator has to be last operator in the `pipe` (usually) to prevent situation when subsequent operator return additional observables which can prevent clean up.*

4. The take(1) For Initialization

Some subscriptions only have to happen **once** during the application startup. They might be needed to kick-start some processing or fire the first request to load the initial data.

In such scenarios we can use RxJS `take(1)` operator which is great because it automatically unsubscribes after the first execution.

```
@Component({/* ... */})
export class SearchComponent {
    search$: Observable<SearchState>;

    ngOnInit() {

       this.search$          // initialize search based on last stored query
          .pipe(take(1))     // will unsubscribe automatically after first execution, great!
          .subscribe(search => this.searchResults(search.query));
    }

    searchResults(query: string) {
        this.searchService.search(query);
    }
}

export interface SearchState {
    query: string;
    results: SearchResults[]
}

export interface SearchResult {
    id: string;
    /* ... */
}
```

We're triggering **initial search** for results based on stored query and **every additional search** will be result of user interaction. (For example there will be a (onchange)="searchResult($event.target.value)" in the component template)

> *The operator itself is* `take(n: number)` *so we could pass any number, but for our scenario the number 1 is all what we need!*

Please note that the `take(1)` will not fire (and complete the observable stream) in case the original observable never emits. We have to make sure we only use it in cases where this can't happen or provide additional unsubscription handling! Thanks Brian Love for feedback!

Also it might be worth using `first()` operator which does exactly how it sounds. Additionally, the operators supports passing of a predicate so its kinda like a combination of `filter` and `take(1)`.

> *Great feedback from Rokas Brazdzionis: Just keep in mind that* `take(1)` *still doesn't unsubscribe when component is being destroyed. The subscription remains active until first value is emitted no matter if component is active or destroyed. So if we do something more crazy, like accessing the DOM, in our subscription —we might end up with an error in the console.*

Sidenote — The <ng-container> element

Before we venture further, let's talk a bit about the `<ng-container>` element. The element is special in that it doesn't produce any corresponding `DOM` element. This makes it a perfect tool for implementation of the conditional parts of a template which will come very handy in our next scenario.

```
@Component({
    selector: 'some',
    template: `
        <ng-container *ngIf="condition">
            <p>There will be only this element without any wrapper</p>
        </ng-container>
        <ng-container *ngIf="todos$ | async as todos">
            <Todo *ngFor="let todo of todos" [todo]="todo"></Todo>
            <!-- there will be only list of <Todo> elements -->
        </ng-container>
    `
})
export class SomeComponent {
    condition = true;
    todos$: Observable<Todo[]>;
}
```

Example of using <ng-container> element to unwrap observable stream and story it in a variable

5. The Fabled | async Pipe

Angular comes with built in support for pipes. A pipe is neat little abstraction and corresponding syntax which enables us to decouple implementation of various data transforms which then can be used in templates of multiple components.

> *One useful example would be* `| json` *pipe which is provided out of the box and enables us to display content of Javascript objects. We can use it in a template like this* `{{ someObject | json }}` .

This brings us to the `| async` pipe which subscribes to the provided Observable behind the scenes and gives us unwrapped plain old Javascript value. This value then can be used in the template as per usual.

In addition to that, all the subscriptions initiated by the `| async` pipe are automatically unsubscribed when the component is destroyed. That's a perfect situation and we can easily consume async data without any possibility to introduce memory leaks!

> *The | async pipes automatically unsubscribes all active subscriptions when component is destroyed*

```
@Component({
    selector: 'todos',
    template: `
        <ng-container *ngIf="todoState$ | async as todoState">
            <input [value]="todoState.newTodo" />
            <Todo *ngFor="let todo of todoState.items" [todo]="todo"></Todo>
            <!-- there will be only list of <Todo> elements -->
        </ng-container>
    `
})
export class TodosComponent {
    todoState$: Observable<TodoState>;
}
```

Example of using | async pipe to unwrap stream of todoState to use it in the template

Another big advantage of using `| async` pipe together with `*ngIf` directive is that we can guarantee that the unwrapped value will be available to all child components at the time they are rendered.

Such an approach helps us to prevent excessive use of “elvis” operator (`?.`)in our templates which is used to get rid `prop of undefined` errors... Without `<ng-container>` it would look more like this...

```
@Component({
    selector: 'todo-editor',
    template: `
        <h1>{{(todo$ | async)?.name}}</h1>
        <input type="checkbox" [value]="(todo$ | async).?done" />
    `
})
export class TodoEditorComponent {
    todo$: Observable<Todo>;
}
```

Using of a “elvis”—?. operator to prevent “prop of undefined” errors.

6. The Detour—| async Pipe Taken Too Far 😵

As goes one funny saying...

> *The scientists were so focused on whether they could make it work that they forget to ask themselves if they should...*

> *The same situation happened to me while working on the Angular NgRx Material Starter on my quest to remove every single* `OnDestroy` / `takeUntil` *occurrence. I came up with a funny working solution, but I would not really recommend it, but who knows? Let's have a look anyway*😂

The Context

The previous solution with `| async` pipe works perfectly for any use case when we need to get hold of the data which is available as a Observable with the intention of displaying it in our UI.

This works really well and the unwrapped data is available in the template so we can use it freely to display it and also to pass it to the component methods. The only missing thing is the triggering (calling) of said methods.

Usually this will be the responsibility of our users and their interaction with our component. Let's say we want to toggle our todo item...

```
@Component({
    template: `
        <ng-container *ngIf="todos$ | async as todos">

            <h1>Todos {{todos.length}}</h1>

            <ul>
                <li *ngFor="let todo of todos" (click)="toggle(todo.id)">{{todo.name}}</li>
            </ul>

        </ng-container>
    `
})
export class TodosComponent {
    todos$: Observable<Todo[]>;

    /* ... */

    toggle(id: string) {
       this.todoService.toggleTodo(id);
    }
}
```

The toggle() method is triggered by user interaction

The unwrapped data is available in the template and it will be passed to the `todoService` as a result of user interaction.

What about cases when we need to trigger something as a reaction to the data itself so we can't rely on users to do it for us? That would be a perfect fit for using `.subscribe()` , right?

```
@Component({ /* ... */ })
export class AppComponent implements OnInit, OnDestroy {

    private unsubscribe$ = new Subject<void>();

    routeActivationEndEvent$: Observable<ActivationEnd>;

    /* ... */

    ngOnInit() {
        // update page title after navigation has ended
        this.routeActivationEndEvent$
          .pipe(takeUntil(unsubscribe$))
          .subscribe(event => this.titleService.updateTitle())
    }

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

Using subscribe to set new browser title as a reaction to navigation

But our goal was to **NOT** use `.subscribe()` or at least to remove the need to manually unsubscribe...

Psst... there is a way!

```
// don't do this at home
@Component({
    template: `
        <ng-container *ngIf="routeActivationEndEvent$ | async">{{updateTitle()}}</ng-container>
    `
})
export class AppComponent {
    routeActivationEndEvent$: Observable<ActivationEnd>;

    /* ... */

    updateTitle() {
        // update page title after navigation has ended
        this.titleService.updateTitle())
    }
}
```

Enter the | async pipe based side effects 😹

So what do we have here?

We're using `| async` pipe to subscribe to the Observable and instead of displaying any content we're evaluating (executing) `{{ }}` our `updateTitle()` component method every time a new value is pushed by the Observable stream.

> *In the example above, we are not passing any value to the called method but it is possible... We could do something like this:* `<ng-container *ngIf="someStream$ | async as value">{{doStuff(value)}}</ng-container>` .

Advantages

- less code and concepts (no `Subject` with `.next()` and `.complete()`, no `OnDestroy`, no `takeUntil`)
- can't forget to implement (or make mistake) in `OnDestroy` / `takeUntil`
- subscriptions always happen in the template (locality)

Disadvantages

- has to be used with `OnPush` change detection strategy (or else it will call function on every change detection cycle)
- it's weird (subjective)
- probably was not meant to be used like that
- there is a better way... 😉

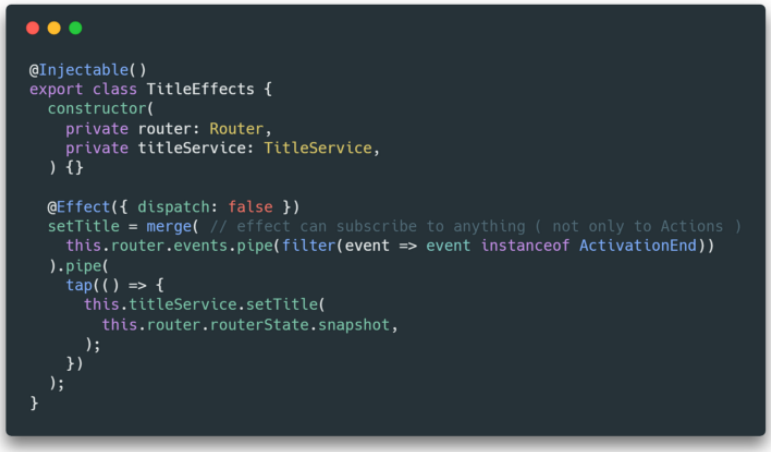
7. The Final Destination — NgRx Effects

In the previous solution, we were trying to make something happen outside of the components template with the help of an `| async` pipe. Stuff happening outside or let's say "on the side" sounds very much like a hint pointing to the concept of **side-effects**.

> *In this post, we are dealing mostly with the plain RxJS but Angular ecosystem contains also NgRx, a state management library based on RxJS primitives which implements one way data flow (Flux / Redux pattern)*

NgRx Effects can help us to remove last explicit `.subscribe()` calls from our apps (without the need for template based side-effects)! Effects are implemented in isolation and are subscribed automatically by the library.

Let's have a look on example of how would such an implementation look like...



```
@Injectable()
export class TitleEffects {
  constructor(
    private router: Router,
    private titleService: TitleService,
  ) {}

  @Effect({ dispatch: false })
  setTitle = merge( // effect can subscribe to anything ( not only to Actions )
    this.router.events.pipe(filter(event => event instanceof ActivationEnd))
  ).pipe(
    tap(() => {
      this.titleService.setTitle(
        this.router.routerState.snapshot,
      );
    })
  );
}
```

Implementation of NgRx effect. This would still needed to be imported in some NgModule with EffectsModule.forFeature([TitleEffects, ...])

The Observable stream of actions (or any other stream) will be subscribed and managed by the library so we don't have to implement any unsubscribe logic. Yaay 🎉 !

> *Side-effects implemented with the help of NgRx Effects are independent from the component life-cycle which prevents memory leaks and host of other problems!*

As a bonus, using NgRx Effects means we are dealing with the side-effects as well defined concept which leads to cleaner architecture, promotes maintainability and it's much easier to test!

. . .

> *Do you think that NgRx or Redux are overkill for your needs? Looking for something simpler? Check out @angular-extensions/model library!*

Try out @angular-extensions/model library! Check out docs & Github

Recapitulation

1. RxJS is a powerful tool to manage collections of async events.
2. We subscribe to event streams which can emit zero to many values and can be potentially infinite.
3. This leaves us with the need to manage unsubscription manually
4. Memory leaks are result of incorrect unsubscription implementation
5. There are many ways to unsubscribe from Observable streams in Angular
6. Different approaches provide us with different trade-offs
7. In general it is best to use `| async` pipe to subscribe and unwrap values in the component templates (with help of `<ng-container>` element)
8. The `| async` pipe can be used also to trigger side effects but there is a better way
9. Use NgRx Effects to implement side-effects which should be triggered in response to Observable streams!

That's It For Today!

> *I hope you enjoyed this article and will now be able to handle subscriptions in your Angular applications with ease!*

Please support this guide with your 👏👏👏 using the clap button on the upper left side and help it spread to a wider audience 🙏 Also, don't hesitate to ping me if you have any questions using the article responses or Twitter DMs @tomastrajan.

> *And never forget, future is bright*

Obviously the bright future! (📸 by Vitalii Ustymenko)

> *Starting an Angular project? Check out Angular NgRx Material Starter!*

Angular NgRx Material Starter with built in best practices, theming and much more!

If you made it this far, feel free to check out some of my other articles about Angular and frontend software development in general...

Total Guide To Angular 6+ Dependency 

Injection—providedIn vs providers:[ ] 💉

Let’s learn when and how to use new better Angular 6+ dependency injection...

medium.com

The Ultimate Answer To The Very Common Angular Question: subscribe()...

Most of the popular Angular state management libraries like NgRx expose...

blog.angularindepth.com

Total Guide To Dynamic Angular Animations That Can Be Customized At...

Animations make projects look much better

blog.angularindepth.com

How Did Angular CLI Budgets Save My Day And How They Can Save Yours

Budgets is one of the less known features of the Angular CLI which helps you to kee...

medium.com